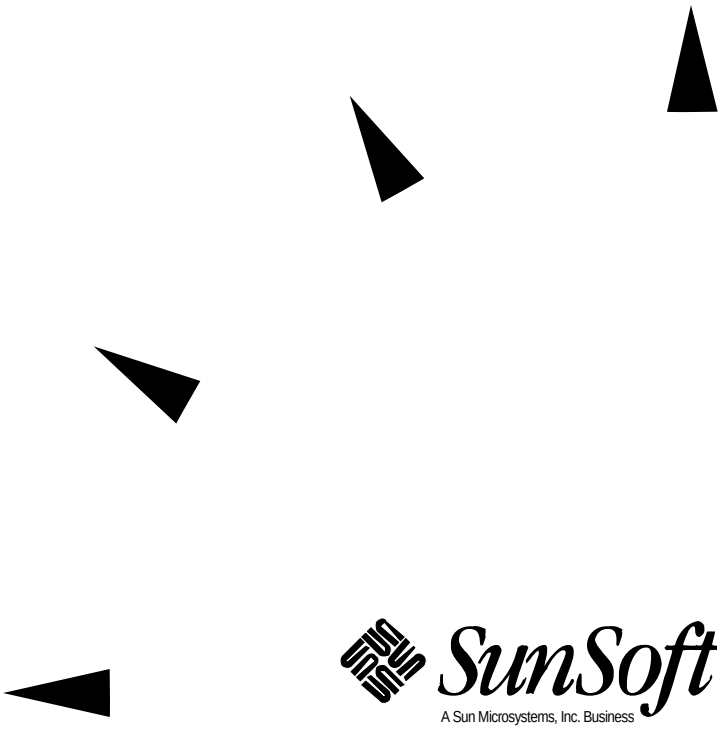
x86 Assembly Language Reference Manual



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

SunSoft
A Sun Microsystems, Inc. Business

Please
Recycle

Adobe PostScript

Contents

Tables

The SunOS Assembler for x86

This section contains a brief description of the SunOS assembler that runs on x86 and also includes a list of documents that can be used for reference.

The SunOS assembler that runs on x86, referred to as the “SunOS x86” in this manual, translates source files that are in assembly language format into object files in linking format.

In the program development process, the assembler is a tool to use in producing program modules intended to exploit features of the Intel® architecture in ways that cannot be easily done using high level languages and their compilers. More precisely, the assembler is the tool of choice when assembly language is the language of choice.

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

The assembly language described in this manual offers full direct access to the x86 instruction set. The assembler may also be used in connection with SunOS™ 5.1 macro preprocessors to achieve full macro-assembler capability. Further more, the assembler responds to directives that allow the programmer a great deal of direct control over the contents of the relocatable object file into which it translates the input source files.

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.

1.1 References

Use the following documents as references:

- *Intel 80386 Programmer's Reference Manual*
- *i486 Microprocessor Programmer Reference Manual (1990)*
- *Intel 80387 Programmer's Reference Manual (1987)*
- *System V Application Binary Interface Intel 386 Processor Supplement*
- *System V Application Binary Interface*
- *SVID System V Interface Definition*

You should also become familiar with the following:

- Manual pages: `as`(1), `ld`(1), `cpp`(1), `mn(4)`, `cof2elf`(1), `elf`(3E), `dis`(1), `a.out`(5).
- ELF-related sections of the *Programming Utilities* manual.

Assembler Input

The SunOS x86 assembler translates source files in the assembly language format specified in this document into relocatable object files for processing by the link editor. This translation process is called *assembly*. The main input required to assemble a source file in assembly language format is that source file itself.

Such a source file may be produced by one of the following:

- a human programmer using a text editor
- a compiler as an intermediate step in the process of translating from a high-level language to executable code
- an automatic program generator
- some other mechanism.

In whatever manner it is produced, the source input file must have a certain structure and content. The specification of this structure and content constitutes the syntax of the assembly language.

The assembler may also allow ancillary input incidental to the translation process. For example, there are several invocation options available. Each such option exercised constitutes information input to the assembler. However, this ancillary input has little direct connection to the translation process, so it is not properly a subject for this manual. Information about invoking the assembler and the available options appears in the as(1) man pages.

2

This chapter describes the overall structure required by the assembler for input source files. This structure is relatively simple: the input source file must be a sequence of assembly language statements. This chapter also begins the specification of the contents of the input source file by describing assembly language statements as textual objects of a certain form.

This document completes the specification by presenting detailed assembly language statements that correspond to the Intel instruction set and are intended for use on machines that run SunOS x86 architecture. For more information on assembly language instruction sets, please refer to the the product documentation from Intel Corporation.

2.1 Source Files in Assembly Language Format

This section details the following:

- file organization
- statements
- values and symbols
- expressions
- machine instruction syntax

File Organization

The input to the assembler is a text file consisting of a sequence of statements. Each statement ends with the first occurrence of a newline character (ASCII LF), or of a semi-colon (;) that is not within a string operand or between a slash and a newline character. Thus, it is possible to have several statements on one line.

To make programs easy to read, understand and maintain, however, it is good programming practice not to have more than one statement per line. As indicated above, a line may contain one or more statements. If several statements appear on a line, they must be separated by semicolons (;).

Statements

This section outlines the types of statements that apply to assembly language. Each statement must be one of the following types:

- An *empty* statement is one that contains nothing other than spaces, tabs, or formfeed characters.

  Empty statements have no meaning to the assembler. They can be inserted freely to improve the appearance of a source file or of a listing generated from it.

- An *assignment* statement is one that gives a value to a symbol. It consists of a symbol, followed by an equal sign (=), followed by an expression.

  The expression is evaluated and the result is assigned to the symbol. Assignment statements do not generate any code. They are used only to assign assembly time values to symbols.

- A *pseudo operation* statement is a directive to the assembler that does not necessarily generate any code. It consists of a pseudo operation code, optionally followed by operands. Every pseudo operation code begins with a period (.).

- A *machine operation* statement is a mnemonic representation of an executable machine language instruction to which it is translated by the assembler. It consists of an operation code, optionally followed by operands.

Furthermore, any statement remains a statement even if it is modified in either or both of the following ways:

- Prefixing a label at the beginning of the statement.

  A label consists of a symbol followed by a colon (:). When the assembler encounters a label, it assigns the value of the location counter to the label.

- Appending a comment at the end of the statement by preceding the comment with a slash (/).

  The assembler ignores all characters following a slash up to the next occurrence of newline. This facility allows insertion of internal program documentation into the source file for a program.

2

Values and Symbol Types

This section presents the values and symbol types that the assembler uses.

Values

Values are represented in the assembler by numerals which can be faithfully represented in standard two's complement binary positional notation using 32 bits. All integer arithmetic is performed using 32 bits of precision. Note, however, that the values used in an x86 instruction may require 8, 16, or 32 bits.

Symbols

A symbol has a value and a symbol type, each of which is either specified explicitly by an assignment statement or implicitly from context. Refer to the next section for the regular definition of the expressions of a symbol.

The following symbols are reserved by the assembler:

`.` Commonly referred to as dot. This is the location counter while assembling a program. It takes on the current location in the `text`, `data`, or `bss` section.

`.text` This symbol is of type text. It is used to label the beginning of a `.text` section in the program being assembled.

`.data` This symbol is of type data. It is used to label the beginning of a .`data` section in the program being assembled.

`.bss` This symbol is of type `bss`. It is used to label the beginning of a `.bss` section in the program being assembled.

`.init` This is used with C++ programs which require constuctors.

`.fini` This is used with C++ programs which require denstuctors.

Symbol Types

Symbol type is one of the following:

*undefine*d

A value is of undefined symbol type if it has not yet been defined. Example instances of undefined symbol types are forward references and externals.

absolute
A value is of absolute symbol type it does not change with relocation. Example instances of absolute symbol types are numeric constants and expressions whose proper sub-expressions are themselves all absolute.

text
A value is of text symbol type if it is relative to the `.text` section.

data
A value is of data symbol type if it is relative to the `.data` section.

bss
A value is of `bss` symbol type if it is relative to the `.bss` section.

You can give any of these symbol types the attribute `EXTERNAL`.

Sections

Five of the symbol types are defined with respect to certain sections of the object file into which the assembler translates the source file. This section describes symbol types.

If the assembler translates a particular assembly language statement into a machine language instruction or into a data allocation, the translation will be associated with one of the following five sections of the object file into which the assembler is translating the source file:

Section	Purpose
text	This is an initialized section. Normally, it is read-only and contains code from a program. It may also contain read-only tables
data	This is an initialized section. Normally, it is readable and writable. It contains initialized data. These can be scalars or tables.
bss	This is an initialized section. Space is not allocated for this segment in the object file.
init	This is used with C++ programs that require constructors.
fini	This is used by C++ programs that require destructors.

An optional section, `.comment`, may also be produced (see *Chapter 4, Assembler Output*).

The section associated with the translated statement is `.text` unless the original statement occurs after a section control pseudo operation has directed the assembler to associate the statement with another section.

Expressions

The expressions accepted by the x86 assembler are defined by their syntax and semantics. The following are the operators supported by the assembler:

Table 2-1 Operators Supported by the Assembler

Operator	***Action***
+	Addition
–	Subtraction
*	Multiplication
\/	Division
&	Bit-wise logical and
|	Bit-wise logical or
>>	Right shift
<<	Left shift
\%	Remainder operator
!	Bit-wise logical and not

Expression Syntax

In the following table that includes syntactic rules, the non terminals are represented by lowercase letters, the terminal symbols are represented by uppercase letters, and the symbols enclosed in double quotes are terminal symbols. There is no precedence assigned to the operators. You must use square brackets to establish precedence.

The terminal nodes are given by the following regular expressions:

```
LABEL   = [a-zA-Z_][a-zA-Z0-9_]*:
DEC_VAL = [1-9][0-9]*
HEX_VAL = 0[Xx][0-9a-fA-F][0-9a-fA-F]*
```

```
OCT_VAL = 0[0-7]*
BIN_VAL = 0[Bb][0-1][0-1]*
```

In the above regular expressions, choices are enclosed in square brackets; a range of choices is indicated by letters or numbers separated by a dash (-); and the asterisk (*) indicates zero or more instances of the previous character.

Table 2-2 Syntactical Rules of Expressions

```
expr            : term
                | expr "+" term
                | expr "-" term
                | expr "\*" term
                | expr "\/" term
                | expr "&" term
                | expr "|" term
                | expr ">>" term
                | expr "<<" term
                | expr "\%" term
                | expr "!" term
                ;

term            : id
                | number
                | "-" term
                | "[" expr "]"
                | "<o>" term
                | "<s>" term
                ;

id              : LABEL
                ;

number          : DEC_VAL
                | HEX_VAL
                | OCT_VAL
                | BIN_VAL
                ;
```

Expression Semantics (Absolute vs. Relocatable)

Semantically, the expressions fall into two groups, absolute and relocatable. The equations later in this section show the legal combinations of absolute and relocatable operands for the addition and subtraction operators. All other operations are only legal on absolute-valued expressions.

All numbers have the absolute attribute. Symbols used to reference storage, text, or data are relocatable. In an assignment statement, symbols on the left side inherit their relocation attributes from the right side.

In the equations below, `a` is an absolute-valued expression and `r` is a relocatable-valued expression. The resulting type of the operation is shown to the right of the equal sign.

```
a + a = a
r + a = r
a - a = a
r - a = r
r - r = a
```

In the last example, you must declare the relocatable expressions before taking their difference.

Following are some examples of valid expressions:

```
label
$label
[label + 0x100]
[label1 - label2]
$[label1 - label2]
```

Following are some examples of invalid expressions:

```
[$label - $label]
[label1 * 5]
(label + 0x20)
```

Machine Instruction Syntax

This section describes the instructions that the assembler accepts. The detailed specification of how the particular instructions operate is not included; for this, see Intel's *80386 Programmer's Reference Manual*.

The following list delineates the three main aspects of the SunOS x86 assembler:

- All register names use the percent sign (%) as a prefix to distinguish them from symbol names.
- Instructions with two operands use the left one as the source and the right one as the destination. This follows the SunOS system's assembler convention, and is reversed from Intel's notation.
- Most instructions that can operate on a byte, word, or long may have `b`, `w`, or `l` appended to them. When an `opcode` is specified with no type suffix, it usually defaults to long. In general, the SunOS assembler derives its type information from the `opcode`, whereas the Intel assembler can derive its type information from the operand types. Where the type information is derived motivates the `b`, `w`, and `l` suffixes used in the SunOS assembler. For example, in the instruction `movw $1,%eax` the `w` suffix indicates the operand is a word.

Operands

Three kinds of operands are generally available to the instructions: *register*, *memory*, and *immediate*. Full descriptions of each type appear in the "Notational Conventions" section. Indirect operands are available **only** to jump and call instructions.

The assembler always assumes it is generating code for a 32-bit segment. When 16-bit data is called for (e.g., `movw %ax, %bx`), the assembler automatically generates the 16-bit data prefix byte.

Byte, word, and long registers are available on the x86 processor. The instruction pointer (`%eip`) and flag register (`%efl`) are not available as explicit operands to the instructions. The code segment (`%cs`) may be used as a source operand but not as a destination operand.

The names of the byte, word, and long registers available as operands and a brief description of each follow; the segment registers are listed also.

Table 2-3 8-Bit (byte), 16-Bit (word), and 32-Bit (long) General Registers

8-Bit (byte) General Registers

`%al`	Low byte of `%ax` register
`%ah`	High byte of `%ax` register
`%cl`	Low byte of `%cx` register
`%ch`	High byte of `%cx` register
`%dl`	Low byte of `%dx` register
`%dh`	High byte of `%dx` register
`%bl`	Low byte of `%bx` register
`%bh`	High byte of `%bx` register

16-Bit (word) General Registers

`%ax`	Low 16-bits of `%eax` register
`%cx`	Low 16-bits of `%ecx` register
`%dx`	Low 16-bits of `%edx` register
`%bx`	Low 16-bits of `%ebx` register
`%sp`	Low 16-bits of the stack pointer
`%bp`	Low 16-bits of the frame pointer
`%si`	Low 16-bits of the source index register
`%di`	Low 16-bits of the destination index register

32-Bit (long) General Registers

`%eax`	32-bit general register
`%ecx`	32-bit general register
`%edx`	32-bit general register
`%ebx`	32-bit general register
`%esp`	32-bit stack pointer
`%ebp`	32-bit frame pointer
`%esi`	32-bit source index register
`%edi`	32-bit destination index register

Table 2-4 Description of Segment Registers

Segment Registers	
`%cs`	Code segment register; all references to the instruction space use this register
`%ds`	Data segment register, the default segment register for most references to memory operands
`%ss`	Stack segment register, the default segment register for memory operands in the stack (i.e., default segment register for `%bp`, `%sp`, `%esp`, and `%ebp`)
`%es`	General-purpose segment register; some string instructions use this extra segment as their default segment
`%fs`	General-purpose segment register
`%gs`	General-purpose segment register

Instruction Description

This section describes the SunOS x86 instruction syntax.

The assembler assumes it is generating code for a 32-bit segment, therefore, it also assumes a 32-bit address and automatically precedes word operations with a 16-bit data prefix byte.

Notational Conventions

This manual uses the following notational conventions:

- The mnemonics are expressed in a regular expression-type syntax.
  - Alternatives separated by a vertical bar (`|`) and enclosed within square brackets (`[]`) denote that you must choose one of them.
  - Alternatives enclosed within curly braces (`{}`) denote that you can use one or none of them.
  - The vertical bar separates different suffixes for operators or operands. For example, `imm[8|16|32]` indicates that an 8-, 16-, or 32-bit immediate value is permitted in an instruction.
- `imm[8|16|32|48]` — an immediate value. You define immediate values using the regular expression syntax previously described. If there is a choice between operand sizes, the assembler will choose the smallest representation.
- `reg[8|16|32]` — a general-purpose register, where each number indicates one of the following:

`32:`	`%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`, `%ebp`, `%esp`
`16:`	`%ax`, `%cx`, `%dx`, `%bx`, `%si`, `%di`, `%bp`, `%sp`
`8:`	`%al`, `%ah`, `%cl`, `%ch`, `%dl`, `%dh`, `%bl`, `%bh`

- `mem[8|16|32|48|64|80]` — a memory operand; the 8, 16, 32, 48, 64, and 80 suffixes represent byte, word, long (or float), inter-segment, double, and long double memory address quantities, respectively.
- `r/m[8|16|32]` — a general-purpose register or memory operand; the operand type is determined from the suffix. They are: 8 = byte, 16 = word, and 32 = long. The registers for each operand size are the same as `reg[8|16|32]` above.
- `creg` — a control register; the control registers are: `%cr0`, `%cr2`, `%cr3`, or `%cr4`.
- `dreg` — a debug register; the debug registers are: `%db0`, `%db1`, `%db2`, `%db3`, `%db6`, and `%db7`.
- `sreg` — a segment register; the segment registers are: `%cs`, `%ds`, `%ss`, `%es`, `%fs`, and `%gs`.
- `treg` — a test register; the test registers are: `%tr6` and `%tr7`.

- `freg` — floating-point registers; these registers are as follows:

`%st, %st(1), %st(2), %st(3) %st(4), %st(5), %st(6), %st(7)`

Note – `%st` is the same as `%st(0)`.

- `cc` — condition codes; the 30 condition codes are:

`a`	above
`ae`	above or equal
`b`	below
`be`	below or equal
`c`	carry
`e`	equal
`g`	greater
`ge`	greater than or equal to
`l`	less than
`le`	less than or equal to
`na`	not above
`nae`	not above or equal to
`nb`	not below
`nbe`	not below or equal to
`nc`	not carry
`ne`	not equal
`ng`	not greater than
`nge`	not greater than or equal to
`nl`	not less than
`nle`	not less than or equal to
`no`	not overflow
`np`	not parity
`ns`	not sign
`nz`	not zero
`o`	overflow
`p`	parity
`pe`	parity even
`po`	parity odd
`s`	sign
`z`	zero

- `disp[8|32]` — the number of bits used to define the distance of a relative jump; because the assembler only supports a 32-bit address space, only 8-bit sign extended and 32-bit addresses are supported.
- `immPtr` — an immediate pointer; when the immediate form of a long call or a long jump is used, the selector and offset are encoded as an immediate pointer. An immediate pointer consists of `$imm16`, `$imm32` where the first immediate value represents the segment and the second represents the offset.

Addressing Modes

Addressing modes are represented by the following:

```
[sreg:][offset][([base][,index][,scale])]
```

- All the items in the square brackets are optional, but at least one is necessary. If you use any of the items inside the parentheses, the parentheses are mandatory.
- `sreg` is a segment register override prefix. It may be any segment register. If a segment override prefix is present, you must follow it by a colon before the offset component of the address. `sreg` does not represent an address by itself. An address must contain an offset component.
- `offset` is a displacement from a segment base. It may be absolute or relocatable. A label is an example of a relocatable offset. A number is an example of an absolute offset.
- `base` and `index` can be any 32-bit register. `scale` is a multiplication factor for the `index` register field. Its value may be 1, 2, 4, 8 to indicate the number to multiply by. The multiplication then occurs by 1, 2, 4, and 8.

  Refer to Intel's *80386 Programmer's Reference Manual* for more details on x86 addressing modes.

Following are some examples of addresses:

```
movl var, %eax
```

Move the contents of memory location `var` into `%eax`.

```
movl %cs:var, %eax
```

Move the contents of the memory location `var` in the code segment into `%eax`.

`movl $var, %eax`

Move the address of `var` into `%eax`.

`movl array_base(%esi), %eax`

Add the address of memory location `array_base` to the contents of `%esi` to get an address in memory. Move the contents of this address into `%eax`.

`movl (%ebx, %esi, 4), %eax`

Multiply the contents of `%esi` by 4 and add this to the contents of `%ebx` to produce a memory reference. Move the contents of this memory location into `%eax`.

`movl struct_base(%ebx, %esi, 4), %eax`

Multiply the contents of `%esi` by 4, add this to the contents of `%ebx`, and add this to the address of `struct_base` to produce an address. Move the contents of this address into `%eax`.

Expressions and Immediate Values

An immediate value is an expression preceded by a dollar sign:

`immediate: "$" expr`

Immediate values carry the absolute or relocatable attributes of their expression component. Immediate values cannot be used in an expression, and should be considered as another form of address, i.e., the immediate form of address.

`immediate: "$" expr "," "$" expr`

The first `expr` is 16 bits of segment. The second `expr` is 32 bits of offset.

2.2 Pseudo Operations

The pseudo-operations listed in this section are supported by the x86 ssembler.

General Pseudo Operations

Below is a list of the pseudo operations supported by the assembler. This is followed by a separate listing of pseudo operations included for the benefit of the debuggers `dbx`(1).

`.align val`

The align pseudo op causes the next data generated to be aligned `modulo val`. `val` should be a positive integer value.

`.bcd val`

The `.bcd` pseudo op generates a packed decimal (80-bit) value into the current section. This is not valid for the `.bss` section. `val` is a nonfloating-point constant.

`.bss`

The `.bss` pseudo op changes the current section to `.bss`.

`.bss tag, bytes`

Define symbol `tag` in the `.bss` section and add `bytes` to the value of dot for `.bss`. This does not change the current section to `.bss`. `bytes` must be a positive integer value.

`.byte val [, val]`

The `.byte` pseudo op generates initialized bytes into the current section. This is not valid for `.bss`. Each `val` must be an 8-bit value.

`.comm name, expr [, alignment]`

The `.comm` pseudo op allocates storage in the `.data` section. The storage is referenced by the symbol `name`, and has a size in bytes of `expr`. `expr` must be a positive integer. `name` cannot be predefined. If the alignment is given, the address of the name will be aligned to a multiple of alignments.

`.data`

The `data` pseudo op changes the current section to `.data`.

`.double val`

The `.double` pseudo op generates an 80387 64 bit floating-point constant (IEEE 754) into the current section. Not valid in the `.bss` section. `val` is a floating-point constant. `val` 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.

`.even`

The `.even` pseudo op aligns the current program counter (`.`) to an even boundary.

`.file` "*string*"

The .file op 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.

`.float val`

The `.float` pseudo op generates an 80387 32 bit floating-point constant (IEEE 754) into the current section. This is not valid in the `.bss` section. `val` is a floating-point constant. `val` 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.

`.globl` *symbol* [, *symbol*]*

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

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

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

`.ident “string”`

The `.ident` pseudo op creates an entry in the comment section containing `string. string` is any sequence of characters, not including the double quote (`"`).

`.lcomm name, expr`

The `.lcomm` pseudo op allocates storage in the `.bss` section. The storage is referenced by the symbol `name`, and has a size of `expr`. `name` cannot be predefined, and `expr` must be a positive integer type. If the alignment is given, the address of name will be aligned to a multiple of alignment.

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

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

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

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

`.long val`

The `.long` pseudo op generates a long integer (32-bit, two's complement value) into the current section. This pseudo op is not valid for the `.bss` section. `val` is a nonfloating-point constant.

`.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

`.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.

`.set name, expr`

The `.set` pseudo op sets the value of symbol `name` to `expr`. This is equivalent to an assignment.

`.string “str”`

This pseudo op places the characters in `str` into the object module at the current location and terminates the string with a null. The string must be enclosed in double quotes (""). This pseudo op is not valid for the `.bss` section.

`.text`

The `.text` pseudo op defines the current section as `.text`.

`.value expr [,expr]`

The `.value` pseudo op is used to generate an initialized word (16-bit, two’s complement value) into the current section. This pseudo op is not valid in the `.bss` section. Each `expr` must be a 16-bit value.

`.version string`

The `.version` pseudo op puts the C compiler version number into the .comment section.

`.volatile`

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

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

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

- 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.

`symbol =expr`

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

Symbol Definition Pseudo Operations

`.def name`
The `.def` pseudo op starts a symbolic description for symbol name. See `endef` (above). `name` is a symbol name.

`.dim expr [,expr]`
The `.dim` pseudo op is used with the `.def` pseudo op. If the `name` of a `.def` is an array, the expressions give the dimensions; up to four dimensions are accepted. The type of each expression should be positive.

`.endef`
The `.endef` pseudo op is the ending bracket for a `.def`.

`.file name`
The `.file` pseudo op is the source file name. Only one is allowed per source file. This **must** be the first line in an assembly file.

`.line expr`
The `.line` pseudo op is used with the `.def` pseudo op. It defines the source line number of the definition of symbol `name` in the `.def`. `expr` should yield a positive value.

`.ln line [,addr]`
This pseudo op provides the relative source line number to the beginning of a function. It is used to pass information through to `sdb`.

`.scl expr`
The `.scl` pseudo op is used with the `.def` pseudo op. Within the `.def` it gives `name` the storage class of `expr`. The type of `expr` should be positive.

`.size expr`
The `.size` pseudo op is used with the `.def` pseudo op. If the `name` of a `.def` is an object such as a structure or an array, this gives it a total size of `expr`. `expr` must be a positive integer.

`.stabs` *name type* `0` *desc value*

`.stabn` *type* `0` *desc value*
The `.stabs` and `.stabn` pseudo ops are debugger directives generated by the C compiler when the `-g` option are used. *name* provides the symbol table name and type structure. *type* identifies the type of symbolic information (i.e., source file, global symbol, or source line). *desc* specifies the number of bytes occupied by a variable or type, or the nesting level for a scope symbol. *value* specifies an address or an offset.

`.tag str`
The `.tag` pseudo op is used in conjunction with a previously defined `.def` pseudo op. If the `name` of a `.def` is a structure or a union, `str` should be the `name` of that structure or union tag defined in a previous `.def-.endef` pair.

`.type expr`
The `.type` pseudo op is used within a `.def-.endef` pair. It gives `name` the C compiler type representation `expr`.

`.val expr`
The `.val` pseudo op is used with a `.def-.endef` pair. It gives `name` (in the `.def`) the value of `expr`. The type of `expr` determines the section for `name`.

Instruction-Set Mapping

3.1 Introduction

This chapter describes the instruction set mappings for the SunOS x86 processor. For more details of the operation and a summary of the exceptions, please refer to the *i486 Microprocessor Programmer's Reference Manual* from Intel Corporation.

Although the Intel processor supports address-size attributes of either 16 or 32 bits, the x86 assembler only supports address-size attributes of 32 bits. The operand-size is either 16 or 32 bits. An instruction that accesses 16-bit words or 32-bit longs has an operand-size attribute of either 16 or 32 bits.

Notational Conventions

The notational conventions used in the instructions included in this chapter are described below:

- The mnemonics are expressed in a regular expression-type syntax.
- When a group of letters is separated from other letters by a bar (|) within square brackets or curly braces, then the group of letters between the bars or between a bar and a closing bracket or brace is considered an atomic unit.

  For example, `fld[lst]` means `fldl`, `flds`, or `fldt`; `fst{ls}` means `fst`, `fstl`, or `fsts`; and `fild{l|ll}` means `fild`, `fildl`, or `fildll`.
-
  - Square brackets ([]) denotes choices, but at least one irequired.

- Alternatives enclosed within curly braces ({}) denote that you can use one or none of them
- The vertical bar separates different suffixes for operators or operands. For example, the following indicates that an 8-, 16-, or 32-bit immediate value is permitted in an instruction:

`imm[8|16|32]`

- The SunOS operators are built from the Intel operators by adding suffixes to them. The 80387, 80486 deals with three data types: integer, packed decimal, and real.

  The SunOS assembler is not typed; the operator has to carry with it the type of data item it is operating on. If the operation is on an integer, the following suffixes apply: `none` for Intel's `short`(16 bits), `l` for Intel's `long` (32 bits), and `ll` for Intel's `longlong`(64 bits). If the operator applies to reals, then: `s` is `short` (32 bits), `l` is `long` (64 bits), and `t` is `temporary real`(80 bits).

- `reg[8|16|32]` defines a general-purpose register, where each number indicates one of the following:

  `32:%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`, `%ebp`, `%esp`

  `16:%ax`, `%cx`, `%dx`, `%bx`, `%si`, `%di`, `%bp`, `%sp`

  `8: %al`, `%ah`, `%cl`, `%ch`, `%dl`, `%dh`, `%bl`, `%bh`

- `mem[8|16|32|48]`stands for a memory operand, which is one of the following: the 8, 16, 32, and 48 suffixes represent byte, word, long, and inter-segment memory address quantities, respectively.
- `r/m[8|16|32]` is a general-purpose register or memory operand; the operand type is determined from the suffix. They are: 8 = byte, 16 = word, and 32 = long. The registers for each operand size are the same as `reg[8|16|32]` above.
- `creg` is a control register; the control registers are: `%cr0`, `%cr2`, or `%cr3`.
- `dreg` is a debug register; the debug registers are: `%db0`, `%db1`, `%db2`, `%db3`, `%db6`, `%db7`.
- `sreg` is a segment register. The segment registers are: `%cs`, `%ds`, `%ss`, `%es`, `%fs`, and `%gs`.
- `treg` is a test register. The test registers are: `%tr6` and `%tr7`.
- `.freg` is floating point registers %st, %st(1) - %st(7).

- cc represent condition codes. There are 30 condition codes. For more information on condition codes, refer to Chapter 2, “Assembler Input,” in this manual.

References

This document presumes that you are familiar with the manner in which the Intel instruction sets function. For more information on specific instruction descriptions, please refer to x86 product documentation from Intel Corporation.

3.2 Segment Register Instructions

Following are the segment register instructions supported by the x86 processor.

Load Full Pointer (`lds`, `les`, `lfs`, `lgs`, and `lss`)

```
lds{wl}       mem[32|48], reg[16|32]
les{wl}       mem[32|48], reg[16|32]
lfs{wl}       mem[32|48], reg[16|32]
lgs{wl}       mem[32|48], reg[16|32]
lss{wl}       mem[32|48], reg[16|32]
```

*Pop Stack into Word (*pop*)*

```
pop{wl}   r/m[16|32]
pop{l}    [%ds|%ss|%es|%fs|%gs]
```

Push Stack into Word(push)

```
push{wl}  r/m[16|32]
push{wl}  imm[8|16|32]
push{l}   [%cs|%ds|%ss|%es|%fs|%gs]
```

3.3 I/O Instructions

*Input from Port (*in, ins*)*

```
in{bwl}  imm8
in{bwl}  (%dx)
ins{bwl}
```

*Output from Port (*out, outs*)*

```
out{bwl} imm8
out{bwl} (%dx)
outs{bwl}
```

3.4 Flag Instructions

*Load Flags into AH Register (*lahf*)*

```
lahf
```

*Store AH into Flags (*sahf*)*

```
sahf
```

*Pop Stack into Flag (*popf*)*

```
popf{wl}
```

*Push Stack into Flag (*pushf*)*

```
pushf{wl}
```

*Complement Carry Flag (*cmc*)*

```
cmc
```

Clear Carry Flag (`clc`*)*

```
clc
```

Set Carry Flag (`stc`*)*

```
stc
```

Clear Interrupt Flag (`cli`*)*

```
cli
```

Set Interrupt Flag (`sti`*)*

```
sti
```

Clear Direction Flag (`cld`*)*

```
cld
```

Set Direction Flag (`std`*)*

```
std
```

3.5 Arithmetic Logical Instructions

Integer Addition (`add`*)*

```
add{bwl} reg[8|16|32], r/m[8|16|32]
add{bwl} r/m[8|16|32], reg[8|16|32]
add{bwl} imm[8|16|32], r/m[8|16|32]
```

Integer Add With Carry (`adc`*)*

```
adc{bwl} reg[8|16|32], r/m[8|16|32]
adc{bwl} r/m[8|16|32], reg[8|16|32]
adc{bwl} imm[8|16|32], r/m[8|16|32]
```

*Integer Subtraction (*sub*)*

```
sub{bwl} reg[8|16|32], r/m[8|16|32]
sub{bwl} r/m[8|16|32], reg[8|16|32]
sub{bwl} imm[8|16|32], r/m[8|16|32]
```

*Integer Subtraction With Borrow (*sbb*)*

```
sbb{bwl} reg[8|16|32], r/m[8|16|32]
sbb{bwl} r/m[8|16|32], reg[8|16|32]
sbb{bwl} imm[8|16|32], r/m[8|16|32]
```

*Compare Two Operands (*cmp*)*

```
cmp{bwl}   reg[8|16|32], r/m[8|16|32]
cmp{bwl}   r/m[8|16|32], reg[8|16|32]
cmp{bwl}   imm[8|16|32], r/m[8|16|32]
```

*Increment by 1 (*inc*)*

```
inc{bwl}   r/m[8|16|32]
```

*Decrease by 1 (*dec*)*

```
dec{bwl}   r/m[8|16|32]
```

*Logical Comparison or Test (*test*)*

```
test{bwl}  reg[8|16|32], r/m[8|16|32]
test{bwl}  r/m[8|16|32], reg[8|16|32]
test{bwl}  imm[8|16|32], r/m[8|16|32]
```

*Shift (*sal*, *shl*, *sar*, *shr*)*

```
sal{bwl} imm8, r/m[8|16|32]
sal{bwl} %cl, r/m[8|16|32]
shl{bwl} imm8, r/m[8|16|32]
shl{bwl} %cl, r/m[8|16|32]
```

```
sar{bwl} imm8, r/m[8|16|32]
sar{bwl} %cl, r/m[8|16|32]
shr{bwl} imm8, r/m[8|16|32]
shr{bwl} %cl, r/m[8|16|32]
```

*Double Precision Shift Left (*shld*)*

```
shld{wl} imm8, reg[16|32], r/m[16,32]
shld{wl} reg[16|32], r/m[16,32], r/m[16,32]
```

*Double Precision Shift Right (*shrd*)*

```
shrd{wl} imm8, reg[16|32]
shrd{wl} reg[16|32], r/m[16,32]
```

*One's Complement Negation (*not*)*

```
not{bwl}    r/m[8|16|32]
```

*Two's Complement Negation (*neg*)*

```
neg{bwl}    r/m[8|16|32]
```

*Check Array Index Against Bounds (*bound*)*

```
bound{wl}   r/m[16|32], reg[16|32]
```

*Logical And (*and*)*

```
and{bwl} reg[8|16|32], r/m[8|16|32]
and{bwl} r/m[8|16|32], reg[8|16|32]
and{bwl} imm[8|16|32], r/m[8|16|32]
```

*Logical Inclusive OR (*or*)*

```
or{bwl}  reg[8|16|32], r/m[8|16|32]
or{bwl}  r/m[8|16|32], reg[8|16|32]
```

```
or{bwl}  imm[8|16|32], r/m[8|16|32]
```

*Logical Exclusion OR (*xor*)*

```
xor{bwl} reg[8|16|32], r/m[8|16|32]
xor{bwl} r/m[8|16|32], reg[8|16|32]
xor{bwl} imm[8|16|32], r/m[8|16|32]
```

3.6 Multiply and Divide Instructions

When the type suffix is not included in a multiply or divide instruction, it defaults to a `long`.

*Signed Multiply (*imul*)*

```
imulb       r/m8
imulw       r/m16
imul{l}     r/m32
imul{wl}    r/m[16|32], reg[16|32]
imul{bwl}   imm[16|32], r/m[16|32], reg[16|32]
```

Unsigned Multiplication of AL, AX *or* EAX*(*mul*)*

```
mul{bwl} r/m[8|16|32]
```

*Unsigned Divide (*div*)*

```
div{bwl}      r/m[8|16|32]
```

*Signed Divide (*idiv*)*

```
idiv{bwl}     r/m[8|16|32]
```

3.7 Conversion Instructions

*Convert Byte to Word (*cbtw*)*

```
cbtw
```

Convert Word to Long (cwtl)

```
cwtl
```

*Convert Signed Word to Signed Double Word (*cwtd*)*

```
cwtd
```

Convert Signed Long to Signed Double Long (cltd)

```
cltd
```

3.8 Decimal Arithmetic Instructions

*Decimal Adjust AL after Addition (*daa*)*

```
daa
```

*Decimal Adjust AL after Subtraction (*das*)*

```
das
```

*ASCII Adjust after Addition (*aaa*)*

```
aaa
```

*ASCII Adjust after Subtraction (*aas*)*

```
aas
```

*ASCII Adjust AX after Multiply (*aam*)*

```
aam
```

*ASCII Adjust AX before Division (*aad*)*

```
aad
```

3.9 Coprocessor Instructions

*Wait (*wait, fwait*)*

```
wait
fwait
```

3.10 String Instructions

All Intel string op mnemonics default to *long.*

*Move Data from String to String (*movs, smov*)*

```
movs{bwl}
smov{bwl}
```

Compare String Operands (cmps, scmp)

```
cmps{bwl}
scmp{bwl}
```

*Store String Data (*stos, ssto*)*

```
stos{bwl}
ssto{bwl}
```

*The Load String Operand (*lods, slod*)*

```
lods{bwl}
slod{bwl}
```

Compare String Data (`scas`, `ssca`*)*

```
scas{bwl}
ssca{bwl}
```

Look-Up Translation Table (`xlat`*)*

```
xlat
```

Repeat Following String Operation (`rep`, `repnz`, `repz`*)*

```
rep
repnz
repz
```

3.11 Procedure Call and Return Instructions

```
lcall   immptr
lcall   *mem48
```

Call Procedure (`call`*)*

```
call    disp32
call    *r/m32
```

Return from Procedure (`ret`*)*

```
ret
ret     imm16
```

Long Return (`lret`*)*

```
lret
lret    imm16
```

Enter or Make a Stack Frame for Procedure Parameters (`enter`*)*

```
enter   imm16, imm8
```

*High Level Procedure Exit (*leave*)*

```
leave
```

3.12 Jump Instructions

*Jump if ECX is Zero (*jcxz*)*

```
jcxz    disp8
```

*Loop Control with CX Counter (*loop, loopnz, loopz*)*

```
loop    disp8
loopnz  disp8
loopne  disp8
loopz   disp8
loope   disp8
```

Jump (jmp, ljmp)

```
jmp        disp[8|32]
ljmp       immPtr
jmp        *r/m32
ljmp       *mem48
jcc        disp[8|32]
```

3.13 Interrupt Instructions

*Call to Interrupt Procedure (*int, into*)*

```
int     imm8
into
```

*Interrupt Return (*iret*)*

```
iret
```

3.14 Protection Model Instructions

Store Local Descriptor Table Register (`sldt`*)*

```
sldt    r/m16
```

Store Task Register (`str`*)*

```
str     r/m16
```

Load Local Descriptor Table Register (`lldt`*)*

```
lldt    r/m16
```

Load Task Register (`ltr`*)*

```
ltr     r/m16
```

Verify a Segment for Reading or Writing (`verr`, `verw`*)*

```
verr    r/m16
verw    r/m16
```

Store Global/Interrupt Descriptor Table Register (`sgdt`, `sidt`*)*

```
sgdt    mem48
sidt    mem48
```

Load Global/Interrupt Descriptor Table (`lgdt`, `lidt`*)*

```
lgdt    mem48
lidt    mem48
```

Store Machine Status Word (`smsw`*)*

```
smsw    r/m16
```

*Load Machine Status Word (*lmsw*)*

```
lmsw    r/m16
```

*Load Access Rights (*lar*)*

```
lar    r/m32, reg32
```

*Load Segment Limit (*lsl*)*

```
lsl    r/m32, reg32
```

*Clear Task-Switched (*clts*)*

```
clts
```

*Adjust RPL Field of Selector (*arpl*)*

```
arpl r16, r/m16
```

3.15 Bit Instructions

Bit Scan Forward

```
bsf{wl}      r/m[16|32], reg[16|32]
```

Bit Scan Reverse

```
bsr{wl}      r/m[16|32], reg[16|32]
```

Bit Test

```
bt{wl}       imm8, r/m[16|32]
bt{wl}       reg[16|32], r/m[16|32]
```

Bit Test And Complement

```
btc{wl}      imm8, r/m[16|32]
btc{wl}      reg[16|32], r/m[16|32]
```

Bit Test And Reset

```
btr{wl}      imm8, r/m[16|32]
btr{wl}      reg[16|32], r/m[16|32]
```

Bit Test And Set

```
bts{wl}      imm8, r/m[16|32]
bts{wl}      reg[16|32], r/m[16|32]
```

3.16 Exchange Instructions

Compare and Exchange [486]

```
cmpxchg{bwl}  reg[8|16|32], r/m[8|16|32]
```

3.17 Floating Point Transcendental

Floating Point Sine

```
fsin
```

Floating Point Cosine

```
fcos
```

Floating Point Sine and Cosine

```
fsincos
```

3.18 Floating Point Constant

Floating Point Load One

```
fld1
fld12+
fld12e
```

```
fldpi
fldlg2
fldln2
fldz
```

3.19 Processor Control Floating Point

Floating Point Load Control Word

```
fldcw       r/m16
```

Floating Point Load Environment

```
fldenv      mem
```

3.20 Other Floating Point

Floating Point Different Reminder

```
fprem1
```

3.21 Floating Point Comparison

Floating Point Unsigned Compare

```
fucom       freg
```

Floating Point Unsigned Compare And Pop

```
fucomp      freg
```

Floating Point Unsigned Compare And Pop Two

```
fucompp
```

3.22 Load and Move Instructions

Load Effective Address

```
lea{wl}      r/m[16|32], reg[16|32]
```

Move

```
mov{bwl}     imm[8|16|32], r/m[8|16|32]
mov{bwl}     reg[8|16|32], r/m[8|16|32]
mov{bwl}     r/m[8|16|32], reg[8|16|32]
```

Move Segment Registers

```
movw         sreg,r/m16
movw         r/m16, sreg
```

Move Control Registers

```
mov{l}       creg, reg32
mov{l}       reg32, creg
```

Move Debug Registers

```
mov{l}       dreg, reg32
mov{l}       reg32, dreg
```

Move Test Registers

```
mov{l}       treg, reg32
mov{l}       reg32, treg
```

Move With Sign Extend

```
movsb{wl}     r/m8, reg[16|32]
movsbwl       r/m16, reg32
```

Move With Zero Extend

```
movzb[wl]    r/m8, reg[16|32]
movzwl       r/m16, reg32
```

3.23 Pop Instructions

Pop All General Registers

```
popa{wl}
```

3.24 Push Instructions

Push All General Registers

```
pusha{wl}
```

3.25 Rotate Instructions

Rotate With Carry Left

```
rcl{bwl}     imm8, r/m[8|16|32]
rcl{bwl}     %cl, r/m[8|16|32]
```

Rotate With Carry Right

```
rcr{bwl}     imm8, r/m[8|16|32]
rcr{bwl}     %cl, r/m[8|16|32]
```

Rotate Left

```
rol{bwl}     imm8, r/m[8|16|32]
rol{bwl}     %cl, r/m[8|16|32]
```

Rotate Right

```
ror{bwl}     imm8, r/m[8|16|32]
```

```
ror{bwl}     %cl, r/m[8|16|32]
```

3.26 Byte Instructions

Byte Set On Condition

```
setcc        r/m8
```

Byte Swap [486]

```
bswap        reg[16|32]
```

3.27 Exchange Instructions

Exchange And Add [486]

```
xadd{bwl}    reg[8|16|32], r/m[8|16|32]
```

Exchange Register / Memory With Register

```
xchg{bwl}    reg[8|16|32], r/m[8|16|32]
```

3.28 Miscellaneous Instructions

Write Back and Invalidate Cache [486 only]

```
wbinvd
```

Invalidate [486 only]

```
invd
```

Invalidate Page [486 only]

```
invlpg       mem32
```

*LOCK Prefix (*lock*)*

```
lock
```

*No Operation (*nop*)*

```
nop
```

*Halt (*hlt*)*

```
hlt
Address Prefix
addr16
Data Prefix
data16
```

3.29 Real Transfers

Load real

```
fld{lst}
```

Store real

```
fst{ls}
```

Store real and pop

```
fstp{lst}
```

Exchange registers

```
fxch
```

3.30 Integer Transfers

Integer load

```
fild{l|ll}
```

Integer store

```
fist{l}
```

Integer store and pop

```
fistp{l|ll}
```

3.31 Packed Decimal Transfers

Packed decimal (BCD) load

```
fbld
```

Packed decimal (BCD) store and pop

```
fbstp
```

3.32 Additions

Real add

```
fadd{ls}
```

Real add and pop

```
faddp
```

Integer add

```
fiadd{l}
```

3.33 Subtractions

Subtract real and pop

```
fsub{ls}
```

Subtract real

```
subp
```

Subtract real reversed

```
fsubr{ls}
```

Subtract real reversed and pop

```
fsubrp
```

Integer subtract

```
fsubrp
```

Integer subtract reverse

```
fisubr{l}
```

3.34 Multiplications

Multiply real

```
fmul{ls}
```

Multiply real and pop

```
fmulp
```

Integer multiply

```
fimul{l}
```

3.35 Divisions

Divide real

```
fdiv{ls}
```

Divide real and pop

```
divp
```

Divide real reversed

```
fdivr{ls}
```

Divide real reversed and pop

```
fdivrp
```

Integer divide

```
fidiv{l}
```

Integer divide reversed

```
fidivr{l}
```

3.36 Floating Point Opcode Errors

Warning – The SunOS x86 assembler generates the wrong object code for some of the floating point opcodes *`fsub`*, *`fsubr`*, *`fdiv`*, and *`fdivr`* when there are two floating register operands, and the second op destination is not the zeroth floating point register. This error has been made to many versions of the USL UNIX® system and would probably cause problems if it were fixed.

Replace the following instructions, in column 1, with their substitutions, in column 2, for x86 platforms:

`fsub %st,%st(n)`	`fsubr %st, %st(n)`
`fsubp %st,%st(n)`	`fsubrp %st, %st(n)`
`fsub`	`fsubr`
`fsubr %st,%st(n)`	`fsub %st, %st(n)`
`fsubrp %st,%st(n)`	`fsubp %st, %st(n)`
`fsubr`	`fsub`
`fdiv %st,%st(n)`	`fdivr %st,%st(n)`
`fdivp %st,%st(n)`	`fdivrp %st,%st(n)`
`fdiv`	`fdivr`
`fdivr %st,%st(n)`	`fdiv %st,%st(n)`
`fdivrp %st,%st(n)`	`fdivp %st,%st(n)`
`fdivr`	`fdiv`

3.37 Other Arithmetic Operations

Square root

```
fsqrt
```

Scale

```
fscale
```

Partial remainder

```
fprem
```

Round to integer

```
frndint
```

Extract exponent and significand

```
fxtract
```

Absolute value

```
fabs
```

Change sign

```
fchs
```

3.38 Comparison Instructions

Compare real

```
fcom{ls}
```

Compare real and pop

```
fcomp{ls}
```

Compare real and pop twice

```
fcompp
```

Integer compare

```
ficom{l}
```

Integer compare and pop

```
ficomp{l}
```

Test

```
ftst
```

Examine

```
fxam
```

3.39 Transcendental Instructions

Partial tangent

```
fptan
```

Partial arctangent

```
fptan
```

$2^x - 1$

```
f2xm1
```

$Y * log2\ X$

```
fyl2x
```

$Y * log_2(X+1)$

```
fyl2xp1
```

3.40 Constant Instructions

Load $log_2 E$

```
fldl2e
```

Load $log_2 10$

```
fldl2t
```

Load $\log_{10} 2$

```
fldlg2
```

Load $\log_e 2$

```
fldln2
```

Load pi

```
fldpi
```

Load + 0

```
fldz
```

3.41 Processor Control Instructions

Initialize processor

```
finit/fninit
```

No operation

```
fnop
```

Save state

```
fsave/fnsave
```

Store control word

```
fstcw/fnstcw
```

Store environment

```
fstenv/fnstenv
```

Store status word

```
fstsw/fnstsw
```

Restore state

```
frstor
```

Set protected mode

```
fsetpm
```

CPU wait

```
fwait/wait
```

Clear exceptions

```
fclex/fnclex
```

Decrement stack pointer

```
fdecstp
```

Free registers

```
ffree
```

Increment stack pointer

```
fincstp
```

Assembler Output

4.1 Introduction to Assembler Output

The main output produced by assembling an input assembly language source file is the translation of that file into an object file in *Extensible and Linking Format* (ELF). ELF files thus produced by the assembler are relocatable files that hold code and/or data. They are input files for the linker, which combines these relocatable files with other ELF object files to create an executable file or a shared object file in the next stage of program building, after translation from source files into object files.

The three main kinds of ELF files are relocatable, executable and shared object files. The assembler may also produce ancillary output incidental to the translation process. For example, if the assembler is invoked with the `-V` option, it may write information to standard output and to standard error.

The assembler also creates a default output file when standard input or multiple input files are used. Ancillary output has little direct connection to the translation process, so it is not properly a subject for this manual. Information about such output appears in `as`(1) manual page.

Certain assembly language statements are directives to the assembler regarding the organization or content of the object file to be generated. Therefore, they have a direct effect on the translation performed by the assembler. To understand these directives, which are presented in Chapter 3, “Instruction-Set Mapping,” it is helpful to have some working knowledge of ELF, at least for relocatable files.

Hence, this chapter presents an overview of ELF for the relocatable object files produced by the assembler. The fully detailed definition of ELF appears in the *System V Application Binary Interface* and the *Intel 386 Processor Supplement*.

4.2 Object Files in Extensible and Linking Format (ELF)

Relocatable ELF files produced by the assembler consist of:

- an ELF header
- a section header table
- sections

The ELF header is always the first part of an ELF file. It is essentally a structure of fixed size and format. The fields, or members, of this structure describe the nature, organization and contents of the rest of the file. In particular, the ELF header has a field which specifies the location within the file at which the section header table begins.

The section header table is an array of section headers, which are structures of fixed size and format. The section headers are thus the elements of the array, or the entries in the table. The section header table has exactly one entry for each section in the ELF file. However, the table may also have entries (section headers) that do not correspond to any section in the file. Such entries and their array indices are reserved. The members of each section header constitute information useful to the linker about the contents of the corresponding section, if any.

All of a relocatable file's information that does not lie within its ELF header or its section header table lies within its sections. Sections contain most of the information needed to combine relocatable files with other ELF files to produce shared object files or executable files. Sections also contain the material to be combined. For example, sections may hold:

- Relocation tables
- Symbol tables
- String tables

Each section in an ELF file fills a contiguous (possibly empty) sequence of that file's bytes. Sections never overlap. However, the (set theoretic) union of a relocatable file's ELF header, the file's section header table, and all the file's

sections may omit some of the file's bytes. Bytes of a relocatable file that are not in the file's ELF header, or in the file's section header table, or in any of the file's sections constitute the file's inactive space. The contents of a file's inactive space, if any, are unspecified.

ELF Header

The *ELF header* is always located at the beginning of the ELF file. It describes the ELF file organization and contains the actual sizes of the object file control structures.

The ELF header consists of the following fields, or members, some of which have the value 0 for relocatable files:

e_ident – This is a byte array consisting of the EI_NIDENT initial bytes of the ELF header, where EI_NIDENT is a name for 16. The elements of this array mark the file as an ELF object file and provide machine-independent data which may be used to decode and interpret the file's contents.

e_type – Identifies the object file type. A value of 1, which has the name ET_REL, specifies a relocatable file. Table 4-1 describes all the object file types.

e_machine – Specifies the required architecture for an individual file. A value of 3, which has the name EM_386, specifies Intel 80386. EM_486, specifies Intel 80486.

e_version – Identifies the version of this object file's format. This field should have the current version number, named EV_CURRENT.

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

e_phoff – Program header table's file offset, in bytes. The value of 0 indicates no program header. (Relocatable files do not need a program header table.)

e_shoff – Section header table's file offset, in bytes. The value of 0 indicates no section header table. (Relocatable files must have a section header table.)

e_flag – Processor-specific flags associated with the file. For the Intel 80386, this field has value 0.

e_ehsize – ELF header's size, in bytes.

e_phentsize – Size, in bytes, of entries in the program header table. All entries are the same size. (Relocatable files do not need a program header table.)

e_phnum – Number of entries in program header table. A value of 0 indicates the file has no program header table. (Relocatable files do not need a program header table.)

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

e_shnum – Number of entries in section header table. A value of 0 indicates the file has no section header table. (Relocatable files must have a section header table.)

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

Table 4-1 Object File Types

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

Section Header

The *section header* table has all of the information necessary to locate and isolate each of the file's sections. A section header entry in a section header table contains information characterizing the contents of the corresponding section, if the file has such a section.

Each entry in the section header table is a section header, which is a structure of fixed size and format, consisting of the following fields, or members:

sh_name – Specifies the section name. The value of this field is an index into the section header string table section, wherein it indicates the beginning of a null-terminated string that names the section.

sh_type – Categorizes the section's contents and semantics. Table 4-3 describes the section types.

*sh_flag*s – One-bit descriptions of section attributes. Table 4-2 describes the section attribute flags.

sh_addr – Address at which the first byte resides if the section appears in the memory image of a process; a value of 0 indicates the section will not appear in the memory image of a process.

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

Note – If the section type is SHT_NOBITS, the corresponding section occupies no space in the file. In this case, *sh_offset* specifies the location at which the section would have begun if it did occupy space within the file.

---

sh_size – Specifies the size, in byte units, of the section.

Note – Even if the section type is SHT_NOBITS, *sh_size* may be non-zero; however, the corresponding section still occupies no space in the file.

---

sh_link – Section header table index link. The interpretation of this information depends on the section type, as described in Table 4-3.

sh_info – Extra information. The interpretation of this information depends on the section type, as described in Table 4-3.

sh_addralign – If a section has an address alignment constraint, the value in this field is the modulus, in byte units, by which the value of `sh_addr` must be congruent to 0; i.e., `sh_addr` = 0 (`mod sh_addralign`).

For example, if a section contains a long (32 bits), the entire section must be ensured long alignment, so `sh_addralign` would have the value 4. Only 0 and positive integral powers of 2 are currently allowed as values for this field. A value of 0 or 1 indicates no address alignment constraints.

sh_entsize – Size, in byte units, for entries in a section which is a table of fixed-size entries, such as a symbol table. Has the value 0 if the section is not a table of fixed-size entries.

Table 4-2 Section Attribute Flags

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

Table 4-3 Section Types

Name	Value	Description	Interpretation by	
			sh_info	sh_link
SHT_NULL	0	Marks section header as inactive; file has no corresponding section.	0	SHN_UNDEF
SHT_PROGBITS	1	Contains information defined by the program, and in a format and with a meaning determined solely by the program.	0	SHN_UNDEF

Table 4-3 Section Types (Continued)

Name	Value	Description	Interpretation by	
			sh_info	sh_link
SHT_SYMTAB	2	Is a complete symbol table, usually for link editing. This table may also be used for dynamic linking; however, it may contain many unnecessary symbols. **Note**: ***Only one section of this type is allowed in a file***	One greater than the symbol table index of the last local symbol.	The section header index of the associated string table.
SHT_STRTAB	3	Is a string table. A file may have multiple string table sections.	0	SHN_UNDEF
SHT_RELA	4	Contains relocation entries with explicit addends. A file may have multiple relocation sections.	The section header index of the section to which the relocation applies.	The section header index of the associated symbol table.
SHT_HASH	5	Is a symbol rehash table. **Note**: ***Only one section of this type is allowed in a file***	0	The section header index of the symbol table to which the hash table applies.
SHT_DYNAMIC	6	Contains dynamic linking information. **Note**: ***Only one section of this type is allowed in a file***	0	The section header index of the string table used by entries in the section.
SHT_NOTE	7	Contains information that marks the file.	0	SHN_UNDEF

Table 4-3 Section Types (Continued)

Name	Value	Description	Interpretation by sh_info	Interpretation by sh_link
SHT_NOBITS	8	Contains information defined by the program, and in a format and with a meaning determined solely by the program. However, a section of this type occupies no space in the file, but the section header's *offset* field specifies the location at which the section would have begun if it did occupy space within the file.	0	SHN_UNDEF
SHT_REL	9	Contains relocation entries without explicit addends. A file may have multiple relocation sections.	The section header index of the section to which the relocation applies.	The section header index of the associated symbol table.
SHT_SHLIB	10	Reserved.	0	SHN_UNDEF
SHT_DYNSYM	11	Is a symbol table with a minimal set of symbols for dynamic linking. **Note**: ***Only one section of this type is allowed in a file***	One greater than the symbol table index of the last local symbol.	The section header index of the associated string table.
SHT_LOPROC SHT_HIPROC	0x70000000 0x7fffffff	Lower and upper bounds of range of section types reserved for processor-specific semantics.	0	SHN_UNDEF
SHT_LOUSER SHT_HIUSER	0x80000000 0xffffffff	Lower and upper bounds of range of section types reserved for application programs. **Note**: ***Section types in this range may be used by an application without conflicting with system-defined section types.***	0	SHN_UNDEF

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

Sections

A section is the smallest unit of an object file that can be relocated. Sections containing the following material usually appear in relocatable ELF files:

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

Sections do not need to occur in any particular order within the object file. The sections of a relocatable ELF file contain all of that file's information which is not contained in the ELF header or in the section header table. The sections in any ELF file must satisfy several conditions:

1. Every section in the file must have exactly one section header entry in the section header table to describe the section. However, the section header table may have section header entries which correspond to no section in the file.
2. Each section occupies one contiguous sequence of bytes within a file. The section may be empty (even so, its section header entry in the section header table may have a non-zero value for the field `sh_size`).
3. A byte in a file can reside in at most one section. Sections in a file cannot overlap.
4. An object file may have inactive space, which is the set of all bytes in the file which are not part of the ELF header, or of the section header table, or of the program header table (for executable files), or of any section in the file. The contents of the inactive space are unspecified.

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

Note – Not all of the sections for which there are entries in the file's section header table need to be present.

Predefined Sections

Sections having certain names beginning with "." (dot) are predefined, with their types and attributes already assigned. These special sections are of two kinds: predefined user sections and predefined non-user sections.

Predefined User Sections

Sections that an assembly language programmer can manipulate by issuing section control directives in the source file are *user sections*. The predefined user sections are those predefined sections that are also user sections. Table 4-4 lists the names of the predefined user sections and briefly describes each.

Table 4-4 Predefined User Sections

Section Name	Description
".bss"	Uninitialized read-write data.
".comment"	Version control information.
".data" & ".data1"	Initialized read-write data.
".debug"	Debugging information.
".fini"	Runtime finalization instructions.
".init"	Runtime initialization instructions.
".rodata" & ".rodata1"	Read-only data.
".text"	Executable instructions.
".line"	Line # info for symbolic debugging.
".note"	Special information from vendors or system builders.

Predefined Non-User Sections

Table 4-5 lists and briefly describes the predefined sections that are not user sections, because assembly language programmers can not manipulate them by issuing section control directives in the source file.

Table 4-5 Predefined Non-User Sections

Section Name	**Description**
".dynamic"	Dynamic linking information.
".dynstr"	Strings needed for dynamic linking.
".dynsym"	Dynamic linking symbol table.
".got"	Global offset table.
".hash"	A symbol hash table.
".interp"	The path name of a program interpreter.
".plt"	The procedure linking table.
"rel*name"* & ".rela*name"*	Relocation information. *name* is the section to which the relocations apply. e.g. ".rel.text", ".rela.text".
".shstrtab"	String table for the section header table names.
".strtab"	The string table.
".symtab"	The symbol table.

Relocation Tables

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 link editor.

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

Symbol Tables

The *symbol table* contains information to locate and relocate symbolic definitions and references. The assembler creates the 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 link editor during relocation. The symbol table's section header contains the symbol table index for the first non-local symbol.

The symbol table contains the following information:

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

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

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

st_info – Specifies the symbol type and binding attributes. Table 4-6 and Table 4-7 describe the symbol types and binding attributes.

st_other – Undefined meaning. Current value is 0.

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

Table 4-6 Symbol Types

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

Table 4-7 Symbol Bindings

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

String Tables

A *string table* is a section which contains null-terminated variable-length character sequences, or strings. The object file uses these strings to represent symbol names and file names. The strings are referenced by indices into the string table section. The first and last bytes of a string table must be the null character.

- A string table index may refer to any byte in the section.
- Empty string table sections are permitted if zero is the value of `sh_size` in the section header entry for the string table in the section header table.

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

Using the Assembler Command Line

A.1 Assembler Command Line

Invoke the assembler command line as follows:

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

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

The `as` command translates the assembly language source files, *inputfile*, into an executable object file, *objfile*. The Intel assembler recognizes the file name 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 Intel assembler prints the error (including a synopsis of the command line syntax and options) to standard error output, and then terminates.

The Intel assembler supports `#define` 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

-D*name*
-D*name=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.

-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.

-m

This new option runs `m4` macro preprocessing on input. The `m4` preprocessor is more powerful than the C preprocessor (invoked by the `-P` option), so it is more useful for complex preprocessing. See the *SunOS 5.1 Reference Manual for x86* for a detailed description of the `m4` macro-processor.

-o *outfile*

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

-P

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

-Q[y|n]

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

-s

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

-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.

-V

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

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) manual page.

Index