Assembler Input 1

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.

This chapter has the following organization:

Introduction	*page 1*
Source Files in Assembly Language Format	*page 2*
Pseudo Operations	*page 14*

Introduction

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. A source file may be produced by one of the following:

- A 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.

1

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.

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 product documentation from Intel Corporation.

Source Files in Assembly Language Format

This section details the following:

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

File Organization

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 semicolon (;) 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.

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 constructors.

`.fini`

This is used with C++ programs which require destructors.

Symbol Types

Symbol type is one of the following:

`undefined`

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.

1

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

Table 1-1 Object File Sections

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 3, "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 1-2 Operators Supported by the Assembler

Operator	Action
+	Addition
–	Subtraction
*	Multiplication
\/	Division

Table 1-2 Operators Supported by the Assembler

Operator	Action
`&`	Bitwise logical and
`|`	Bitwise logical or
`>>`	Right shift
`<<`	Left shift
`\%`	Remainder operator
`!`	Bitwise logical and not

Expression Syntax

Table 1-3 shows 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.

Table 1-3 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
        ;
```

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.

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 describes 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 operating environment 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`, where 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 "Notational Conventions" on page 23. 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 also listed.

Table 1-4 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

Table 1-5 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

1

Table 1-6 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 1-7 Description of 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.

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.

Pseudo Operations

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

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 is 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 satisfies 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 occurs.

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 is 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.

Because 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 satisfies 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`. `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`.