x86 Assembly Language Reference Manual

Sun Microsystems, Inc.
901 N. San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 805-4693-10
October 1998

Contents

Preface

This preface is a brief description of the SunOS™ assembler that runs on x86. This preface 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.

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. Furthermore, the assembler responds to directives that allow the programmer direct control over the contents of the relocatable object file.

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

Before You Read This Book

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:

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

How This Book Is Organized

This document is organized into the following chapters:

Chapter 1 describes the overall structure required by the assembler for input source files.

Chapter 2 describes the instruction set mappings for the SunOS x86 processor.

Chapter 3 provides an overview of ELF (*Executable and Linking Format*) for the relocatable object files produced by the assembler.

Appendix A describes the assembler command line options.

What Typographic Changes Mean

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

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

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

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

CHAPTER 1

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.

This chapter has the following organization:

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.

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.

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 Translations and their Associations

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.

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
&	Bitwise logical and
|	Bitwise logical or
>>	Right shift
<<	Left shift
\%	Remainder operator
!	Bitwise logical and not
^	Bitwise logical XOR

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
 | 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*. 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

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.

CHAPTER 2

Instruction-Set Mapping

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.

This chapter is organized as follows:

Introduction

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 (`[]`) denote choices, but at least one is required.
- 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
```

- `imm[8|16|32|48]` — an immediate value. You define immediate values using the regular expression syntax previously described (see also Expressions and Immediate Values on page 210). If there is a choice between operand sizes, the assembler will choose the smallest representation.
- `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.
- `creg` — a control register; the control registers are: `%cr0`, `%cr2`, `%cr3`, or `%cr4`.
- `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.
- `dreg` is a debug register; the debug registers are: `%db0`, `%db1`, `%db2`, `%db3`, `%db6`, `%db7`.
- `sreg` is a segment register. The 16-bit 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(0)), %st(1) - %st(7).
- An instruction can act on zero or more operands. An operand can be any of the following:
  - an immediate operand (in the instruction itself)
  - a register (32-bit genera, segment, or status/instruction register), (16-bit word register), and (8-bit byte register).
  - a pointer to a memory location.
  - an I/O port
- Instruction syntax is:

  operand1 → operand2

  where operand1 and operand2 are operated on and the result stored in operand2. The → arrow shows the direction. The direction is opposite of that described in the Intel Corporation i486 Microprocessor Programmer's Reference Manual.
- 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.
- cc — condition codes; the 30 condition codes are:

TABLE 2–1 Condition Codes

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

TABLE 2–1 Condition Codes *(continued)*

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

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 the Intel Corporation *i486 Microprocessor Programmer's Reference Manual.*

Segment Register Instructions

The 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]
```

Operation

```
mem[32|48] → reg[16|32]
```

Description

Reads a full pointer from memory and stores it in the specified segment register (DS, ES, FS, GS or SS) with a 16- or 32-bit offset value.

Example

Load a 16-bit pointer from memory location 0x44444444 into the DX register:

```
ldsw 0x44444444, %dx
```

Load a 32-bit pointer from memory location 0x33333333 into the EDX register:

```
ldsl 0x33333333, %edx
```

Pop Stack into Word (pop)

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

Operation

stack → r/m[16|32]

stack → segment register

Description

Replaces the previous contents of the register or memory operand with a word or long from the top of the stack.

Replaces the previous contents of the segment register operand with a long.

For a word, SP + 2; for a long, SP + 4.

Example

Replace the contents of the memory location pointed to by the EDI register, plus an offset of 4, with the word from the top of the stack:

```
popw 4(edi)
```

Replace the contents of the memory location pointed to by the EAX register with the long from the top of the stack:

```
popl %eax
```

Push Word/Long onto Stack (`push`)

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

Operation

r/m[16|32] → stack segment register → stack

Description

For a word, SP - 2; for a long, SP - 4. Replaces the new top of stack, pointed to by SP, with the register, memory, immediate, or segment register operand.

Example

Replaces the new top of stack with the 16-bit immediate value, -126:

```
pushw $-126
```

Replaces the new top of stack with the 32-bit immediate value, 23456789:

```
pushl $23456789
```

Replaces the new top of stack with the content of the AX register:

```
pushw %ax
```

Replaces the new top of stack with the content of the EBX register:

```
pushl %ebx
```

I/O Instructions

Input from Port (in, ins)

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

Operation

imm[8|16|32] → [AL|AX|EAX]

DX → [AL|AX|EAX] DX → ES:(E)DI

Description

`in` transfers a byte, word, or long from the immediate port into the byte, word, or long memory address pointed to by the AL, AX, or EAX register, respectively.

The second form of the `in` instruction transfers a byte, word, or long from a port (0 to 65535), specified in the DX register, into the byte, word, or long memory address pointed to by the AL, AX, or EAX register, respectively.

When an 8-bit port is specified, the upper-eight bits of the port address will be 0.

The `ins` instruction transfers a string from a port specified in the DX register to the memory byte or word pointed to by the ES:destination index. Load the desired port number into the DX register and the desired destination address into the DI or EDI index register before executing the `ins` instruction. After a transfer occurs, the destination-index register is automatically incremented or decremented as determined by the value of the direction flag (DF). The index register is incremented if DF = 0 (DF cleared by `a cld` instruction); it is decremented if DF = 1 (DF set by a `std` instruction). The increment or decrement count is 1 for a byte transfer, 2 for a word, and 4 for a long. Use the `rep` prefix with the `ins` instruction for a block transfer of CX bytes or words.

Example

Transfer an immediate 8-bit port address into the AL register:

```
inb $0xff
```

Transfer a 16-bit port address, specified in the DX register, into the AX register:

```
inw (%dx)
```

Transfer a string from the port address, specified in the DX register, into the ES:destination index register:

```
insl
```

Output from Port (out, outs)

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

Operation

[AL|AX|EAX] → imm[8|16|32]

[AL|AX|EAX] → DX

ES:(E)DI → DX

Description

Transfers a byte, word, or long from the memory address pointed to by the content of the AL, AX, or EAX register to the immediate 8-, 16-, or 32-bit port address.

The second form of the `out` instruction transfers a byte, word, or long from the AL, AX, or EAX registers respectively to a port (0 to 65535), specified by the DX register.

The `outs` instruction transfers a string from the memory byte or word pointed to by the ES:source index to the port addressed in the DX register. Load the desired port number into the DX register and the desired source address into the SI or ESI index register before executing the `outs` instruction. After a transfer occurs, the destination-index register is automatically incremented or decremented as determined by the value of the direction flag (DF). The index register is incremented if DF = 0 (DF cleared by a `cld` instruction); it is decremented if `DF = 1` (DF set by a `std` instruction). The increment or decrement count is 1 for a byte transfer, 2 for a word, and 4 for a long. Use the `rep` prefix with the `outs` instruction for a block transfer of CX bytes or words.

Example

Transfer a word from the AX register into the 16-bit port address, 0xff:

```
outw $0xff
```

Transfer a long from the EAX register into the 32-bit port address specified by the DX register:

```
outl (%dx)
```

Transfer a string from the memory byte or word pointed to by the ES:source index to the port addressed in the DX register:

```
outsl
```

Flag Instructions

Load Flags into AH Register (lahf)

```
lahf
```

Operation

SF:ZF:xx:AF:xx:PF:xx:CF → AH

Description

Transfers the low byte of the flags word to the AH register. The bits (lsb to msb) are: sign, zero, indeterminate, auxiliary carry, indeterminate, parity, indeterminate, and carry.

Example

Transfer the flags word into the AH register:

```
lahf
```

Store AH into Flags (sahf)

```
sahf
```

Operation

AH → SF:ZF:xx:AF:xx:PF:xx:CF

Description

Loads flags (sign, zero, indeterminate, auxiliary carry, indeterminate, parity, indeterminate, and carry) with values from the AH register.

Example

Load values from the AH register into the flags word:

```
sahf
```

Pop Stack into Flag (`popf`)

```
popf{wl}
```

Operation

stack → flags register

Description

Pops the word or long from the top of the stack and stores the value in the flags register. Stores a word in FLAGS; stores a long in EFLAGS.

Example

Pops the word from the top of the stack and stores it in the flags register:

```
popfw
```

Pops the long from the top of the stack and stores it in the eflags register:

```
popfl
```

Push Flag Register Onto Stack (`pushf`)

```
pushf{wl}
```

Operation

flags register → stack

Description

For a word, SP - 2 and copies FLAGS to the new top of stack pointed to by SP. For a long, SP - 4 and copies EFLAGS to the new top of stack pointed to by SS:eSP.

Example

Pushes the flags register onto the top of the stack:

```
pushfw
```

Pushes the eflags register onto the top of the stack:

```
pushfl
```

Complement Carry Flag (cmc)

```
cmc
```

Operation

not CF → CF

Description

Reverses the setting of the carry flag; affects no other flags.

Example

Reverse the setting of the carry flag:

```
cmc
```

Clear Carry Flag (clc)

```
clc
```

Operation

0 → CF

Description

Sets the carry flag to zero; affects no other flags.

Example

Clear the carry flag:

```
clc
```

Set Carry Flag (stc)

```
stc
```

Operation

1 → CF

Description

Sets the carry flag to 1.

Example

Set the carry flag:

```
stc
```

Clear Interrupt Flag (cli)

```
cli
```

Operation

0 → IF

Description

Clears the interrupt flag if the current privilege level is at least as privileged as IOPL; affects no other flags. External interrupts disabled at the end of the `cli` instruction or from that point on until the interrupt flag is set.

Example

Clear the interrupt flag:

```
cli
```

Set Interrupt Flag (sti)

```
sti
```

Operation

1 → IF

Description

Sets the interrupt flag to 1.

Example

Set the interrupt flag:

```
sti
```

Clear Direction Flag (cld)

```
cld
```

Operation

0 → DF

Description

Clears the direction flag; affects no other flags or registers. Causes all subsequent string operations to increment the index registers, (E)SI and/or (E)DI, used during the operation.

Example

Clear the direction flag:

```
cld
```

Set Direction Flag (std)

```
std
```

Operation

1 → DF

Description

Sets the direction flag to 1, causing all subsequent string operations to decrement the index registers, (E)SI and/or (E)DI, used during the operation.

Example

Set the direction flag:

```
std
```

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]
```

Operation

reg[8|16|32] + r/m[8|16|32] → r/m[8|16|32]

r/m[8|16|32] + reg[8|16|32] → reg[8|16|32]

imm[8|16|32] + r/m[8|16|32] → r/m[8|16|32]

Description

Integer adds operand1 to operand2 and stores the result in operand2.

When an immediate byte is added to a word or long, the immediate value is sign-extended to the size of the word or long operand.

If you wish to decimal adjust (`daa`) or ASCII adjust (`aaa`) the `add` result, use the form of `add` that stores the result in AL.

Example

Integer adds the 8-bit constant, -126, to the content of the AL register:

```
addb $-126,%al
```

Integer adds the word contained in the effective address (addressed by the EDI register plus an offset of 4) to the content of the DX register:

```
addw 4(%edi),%dx
```

Integer adds the content of the EDX register to the effective address (addressed by the EDI register plus an offset of 4):

```
addl %edx, 4(%edi)
```

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]
```

Operation

(reg[8|16|32] + CF) + r/m[8|16|32] → r/m[8|16|32]

(r/m[8|16|32] + CF) + reg[8|16|32] → reg[8|16|32]

(imm[8|16|32] + CF) + r/m[8|16|32] → r/m[8|16|32]

Description

Integer adds operand1 and the carry flag to operand2 and stores the result in operand2. `adc` is typically executed as part of a multi-byte or multi-word add operation. When an immediate byte is added to a word or long, the immediate value is sign-extended to the size of the word or long operand.

Example

Integer add the 8-bit content of the effective memory address (ESI register plus an offset of 1) and the carry flag to the content of the address in the CL register:

```
adcb 1(%esi), %cl
```

Integer add the 16-bit content of the effective memory address (EDI register plus an offset of 4) and the carry flag to the content of the address in the DX register:

```
adcw 4(%edi), %dx
```

Integer add the 32-bit content of the address in the EDX register and the carry flag to the effective memory address (EDI register plus an offset of 4):

```
adcl %edx, 4(%edi)
```

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]
```

Operation

r/m[8|16|32] - reg[8|16|32] → r/m[8|16|32]

reg[8|16|32] - r/m[8|16|32] → reg[8|16|32]

r/m[8|16|32] - imm[8|16|32] → r/m[8|16|32]

Description

Subtracts operand1 from operand2 and stores the result in operand2. When an immediate byte value is subtracted from a word, the immediate value is sign-extended to the size of the word operand before the subtract operation is executed.

If you wish to decimal adjust (`das`) or ASCII adjust (`aas`) the `sub` result, use the form of `sub` that stores the result in AL.

Example

Integer subtract the 8-bit constant, -126, from the content of the effective address (addressed by the ESI register plus an offset of 1):

```
subb $-126, 1(%esi)
```

Integer subtract the 16-bit constant, 1234, from the content of the effective address (addressed by the EDI register plus an offset of 4):

```
subw $1234, 4(%edi)
```

Integer subtract the 32-bit content of the EDX register from the effective address (addressed by the EDI register plus an offset of 4):

```
subl %edx, 4(%edi)
```

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]
```

Operation

r/m[8|16|32] - (reg[8|16|32] + CF) → r/m[8|16|32]

reg[8|16|32] - (r/m[8|16|32] + CF) → reg[8|16|32]

r/m[8|16|32] - (imm[8|16|32] + CF) → r/m[8|16|32]

Description

Subtracts (operand1 and the carry flag) from operand2 and stores the result in operand2. When an immediate byte value is subtracted from a word, the immediate value is sign-extended to the size of the word operand before the subtract operation is executed.

Example

Integer subtract the 8-bit content of the CL register plus the carry flag from the effective address (addressed by the ESI register plus an offset of 1):

```
sbbb %cl, 1(%esi)
```

Integer subtract the 16-bit constant, -126, plus the carry flag from the AL register:

```
sbbw $-126, %al
```

Integer subtract the 32-bit constant, 12345678, plus the carry flag from the effective address (addressed by the EDI register plus an offset of 4):

```
sbbl $12345678, 4(%edi)
```

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]
```

Operation

r/m[8|16|32] - reg[8|16|32]

reg[8|16|32] - r/m[8|16|32]

r/m[8|16|32] - imm[8|16|32]

Description

Subtracts operand1 from operand2, but does not store the result; only changes the flags. `cmp` is typically executed in conjunction with conditional jumps and the `set`*cc* instruction. If an operand greater than one byte is compared to an immediate byte, the immediate byte value is first sign-extended.

Example

Compare the 8-bit constant, 0xff, with the content of the AL register:

```
cmpb $0xff, %al
```

Compare the 16-bit content of the DX register with the effective address (addressed by the EDI register plus an offset of 4):

```
cmpw %dx, 4(%edi)
```

Compare the 32-bit content of the effective address (addressed by the EDI register plus an offset of 4) to the EDX register:

```
cmpl 4(%edi), %edx
```

Increment by 1 (inc)

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

Operation

r / m[8 | 16 | 32] + 1 → r / m[8 | 16 | 32]

Description

Adds 1 to the operand and does not change the carry flag. Use the `add` instruction with an immediate value of 1 to change the carry flag,.

Example

Add 1 to the contents of the byte at the effective address (addressed by the ESI register plus an offset of 1):

```
incb 1(%esi)
```

Add 1 to the 16-bit contents of the AX register:

```
incw %ax
```

Add 1 to the 32-bit contents at the effective address (addressed by the EDI register):

```
incl 4(%edi)
```

Decrease by 1 (dec)

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

Operation

r/m[8|16|32] - 1 → r/m[8|16|32]

Description

Subtracts 1 from the operand. Does not change the carry flag. To change the carry flag, use the `sub` instruction with an immediate value of 1.

Example

Subtract 1 from the 8-bit contents of the effective address (addressed by the ESI register plus an offset of 1):

```
decb 1(%esi)
```

Subtract 1 from the 16-bit contents of the BX register:

```
decw %bx
```

Subtract 1 from the 32-bit contents of the effective address (addressed by the EDI register plus an offset of 4):

```
decl 4(%edi)
```

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]
```

Operation

reg[8|16|32] and r/m[8|16|32] → r/m[8|16|32]

r/m[8|16|32] and reg[8|16|32] → reg[8|16|32]

imm[8|16|32] and r/m[8|16|32] → r/m[8|16|32]

Description

Performs a bit-wise logical AND of the two operands. The result of a bit-wise logical AND is 1 if the value of that bit in both operands is 1; otherwise, the result is 0. `test` discards the results and modifies the flags. The OF and CF flags are cleared; SF, ZF and PF flags are set according to the result.

Example

Perform a logical AND of the constant, 0xff, and the 8-bit contents of the effective address (addressed by the ESI register plus an offset of 1):

```
testb $0xff, 1(%esi)
```

Perform a logical AND of the 16-bit contents of the DX register and the contents of the effective address (addressed by the EDI register plus an offset of 4):

```
testw %dx, 4(%edi)
```

Perform a logical AND of the constant, 0xffeeddcc, and the 32-bit contents of the effective address (addressed by the EDI register plus an offset of 4):

```
testl $0xffeeddcc, 4(%edi)
```

Shift (sal, shl, sar, shr)

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]

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

Operation

shift-left r/m[8|16|32] by imm8 → r/m[8|16|32]

shift-left r/m[8|16|32] by %cl → r/m[8|16|32]

shift-right r/m[8|16|32] by imm8 → r/m[8|16|32]

shift-right r/m[8|16|32] by %cl → r/m[8|16|32]

Description

`sal` (or its synonym `shl`) left shifts (multiplies) a byte, word, or long value for a count specified by an immediate value and stores the product in that byte, word, or long respectively. The second variation left shifts by a count value specified in the CL register. The high-order bit is shifted into the carry flag; the low-order bit is set to 0.

`sar` right shifts (signed divides) a byte, word, or long value for a count specified by an immediate value and stores the quotient in that byte, word, or long respectively. The second variation right shifts by a count value specified in the CL register. `sar` rounds toward negative infinity; the high-order bit remains unchanged.

`shr` right shifts (unsigned divides) a byte, word, or long value for a count specified by an immediate value and stores the quotient in that byte, word, or long respectively. The second variation divides by a count value specified in the CL register. `shr` sets the high-order bit to 0.

Example

Right shift, count specified by the constant (253), the 8-bit contents of the effective address (addressed by the ESI register plus an offset of 1):

```
sarb $253, 1(%esi)
```

Right shift, count specified by the contents of the CL register, the 16-bit contents of the effective address (addressed by the EDI register plus an offset of 4):

```
shrw %cl, 4(%edi)
```

Left shift, count specified by the constant (253), the 32-bit contents of the effective address (addressed by the EDI register plus an offset of 4):

```
shll $253, 4(%edi)
```

Double Precision Shift Left (shld)

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

Operation

by imm8 shift-left r/m[16|32] bits reg[16|32] → r/m[16|32]

by reg[16|32] shift-left r/m[16|32] bits r/m[16|32] → r/m[16|32]

Description

`shld` double-precision left shifts a 16- or 32-bit register value into a word or long for the count specified by an immediate value, MODULO 32 (0 to 31). The result is stored in that particular word or long.

The second variation of `shld` double-precision left shifts a 16- or 32-bit register or memory value into a word or long for the count specified by register CL MODULO 32 (0 to 31).The result is stored in that particular word or long.

`shld` sets the SF, ZF, and PF flags according to the value of the result; CS is set to the value of the last bit shifted out; OF and AF are undefined.

Example

Use the count specified by the constant, 253, to double-precision left shift a 16-bit register value from the DX register to the effective address (addressed by the EDI register plus an offset of 4):

```
shldw $253, %dx, 4(%edi)
```

Use the count specified (%CL MOD 32) by the 32-bit EDX register to double-precision left shift a 32-bit memory value at the effective address (addressed by the EDI register plus an offset of 4):

```
shldl %cl,%edx, 4(%edi)
```

Double Precision Shift Right (shrd)

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

Operation

by imm8 shift-right r / m[16 | 32] bits reg[16 | 32] → r / m[16 | 32]

by reg[16 | 32] shift-right r / m[16 | 32] bits r / m[16 | 32] → r / m[16 | 32]

Description

`shrd` double-precision right shifts a 16- or 32-bit register value into a word or long for the count specified by an immediate value MODULO 32 (0 to 31). The result is stored in that particular word or long.

The second variation of `shrd` double-precision right shifts a 16- or 32-bit register or memory value into a word or long for the count specified by register CL MODULO 32 (0 to 31).The result is stored in that particular word or long.

`shrd` sets the SF, ZF, and PF flags according to the value of the result; CS is set to the value of the last bit shifted out; OF and AF are undefined.

Example

Use the count specified by the constant, 253, to double-precision right shift a 16-bit register value from the DX register to the effective address (addressed by the EDI register plus an offset of 4):

```
shrdw $253, %dx, 4(%edi)
```

Use the count specified (%CL MOD 32) by the 32-bit EDX register to double-precision right shift a 32-bit memory value at the effective address (addressed by the EDI register plus an offset of 4)

```
shrdl %cl,%edx, 4(%edi)
```

One’s Complement Negation (not)

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

Operation

not r/m[8|16|32] → r/m[8|16|32]

Description

Inverts each bit value of the byte, word, or long; that is, every 1 becomes a 0 and every 0 becomes a 1.

Example

Invert each of the 8-bit values at the effective address (addressed by the ESI register plus an offset of 1):

```
notb 1(%esi)
```

Invert each of the 16-bit values at the effective address (addressed by the EDI register plus an offset of 4):

```
notw 4(%edi)
```

Invert each of the 32-bit values at the effective address (addressed by the EDI register plus an offset of 4):

```
notl 4(%edi)
```

Two's Complement Negation (neg)

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

Operation

two's-complement r/m[8|16|32] → r/m[8|16|32]

Description

Replace the value of the byte, word, or long with its two's complement; that is, neg subtracts the byte, word, or long value from 0, and puts the result in the byte, word, or long respectively.

neg sets the carry flag to 1, unless initial value of the byte, word, or long is 0. In this case neg clears the carry flag to 0.

Example

Replace the 8-bit contents of the effective address (addressed by the ESI register plus an offset of 1) with its two's complement:

```
negb 1(%esi)
```

Replace the 16-bit contents of the effective address (addressed by the EDI register plus an offset of 4) with its two’s complement:

```
negw 4(%edi)
```

Replace the 32-bit contents of the effective address (addressed by the EDI register plus an offset of 4) with its two’s complement:

```
negl 4(%edi)
```

Check Array Index Against Bounds (bound)

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

Operation

r/m[16|32] bound reg[16|32] → CC is unchanged

Description

Ensures that a signed array index (16- or 32-bit register) value falls within the upper and lower bounds of a block of memory. The upper and lower bounds are specified by a 16- or 32-bit register or memory value. If the signed array index value is not within the bounds, an Interrupt 5 occurs; the return EIP points to the `bound` instruction.

Example

Check the 16-bit signed array index value in the AX register against the doubleword with the upper and lower bounds specified by DX:

```
boundw %ax, %dx
```

Check the 32-bit signed array index value in the EAX register against the doubleword with the upper and lower bounds specified by EDX:

```
boundl %eax, %edx
```

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]
```

Operation

reg[8|16|32] land r/m[8|16|32] → r/m[8|16|32] r/m[8|16|32] land reg[8|16|32] → reg[8|16|32]

imm[8|16|32] land r/m[8|16|32] → r/m[8|16|32]

Description

Performs a logical AND of each bit in the values specified by the two operands and stores the result in the second operand.

TABLE 2–2 Logical AND

Values	**Result**
0 LAND 0	0
0 LAND 1	0
1 LAND 0	0
1 LAND 1	1

Example

Perform an 8-bit logical AND of the CL register and the contents of the effective address (addressed by the ESI register plus an offset of 1):

```
andb %cl, 1(%esi)
```

Perform a 16-bit logical AND of the constant, 0xffee, and the contents of the effective address (addressed by the AX register):

```
andw $0xffee, %ax
```

Perform a 32-bit logical AND of the contents of the effective address (addressed by the EDI register plus an offset of 4) and the EDX register:

```
andl 4(%edi), %edx
```

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]
```

Operation

reg[8|16|32] LOR r/m[8|16|32] → r/m[8|16|32]

r/m[8|16|32] LOR reg[8|16|32] → reg[8|16|32]

imm[8|16|32] LOR r/m[8|16|32] → r/m[8|16|32]

Description

Performs a logical OR of each bit in the values specified by the two operands and stores the result in the second operand.

TABLE 2–3 Inclusive OR

Values	Result
0 LOR 0	0
0 LOR 1	1
1 LOR 0	1
1 LOR 1	1

Example

Perform an 8-bit logical OR of the constant, 0xff, and the AL register:

```
orb $0xff, %al
```

Perform a 16-bit logical OR of the constant, 0xff83, and the contents of the effective address (addressed by the EDI register plus an offset of 4):

```
orw $0xff83, 4(%edi)
```

Perform a 32-bit logical OR of the EDX register and the contents of the effective address (addressed by the EDI register plus an offset of 4):

```
orl %edx, 4(%edi)
```

Logical Exclusive 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]
```

Operation

reg[8|16|32] XOR r/m[8|16|32] → r/m[8|16|32]

r/m[8|16|32] XOR reg[8|16|32] → reg[8|16|32]

imm[8|16|32] XOR r/m[8|16|32] → r/m[8|16|32]

Description

Performs an exclusive OR of each bit in the values specified by the two operands and stores the result in the second operand.

TABLE 2–4 Exclusive XOR

Values	**Result**
0 XOR 0	0
0 XOR 1	1
1 XOR 0	1
1 XOR 1	0

Example

Perform a 8-bit exclusive OR of the constant, 0xff, and the AL register:

```
xorb $0xff, %al
```

Perform a 16-bit exclusive OR of the constant, 0xff83, and the contents of the effective address (addressed by the EDI register plus an offset of 4):

```
xorw $0xff83, 4(%edi)
```

Perform a 32-bit exclusive OR of the EDX register and the contents of the effective address (addressed by the EDI register plus an offset of 4):

```
xorl %edx, 4(%edi)
```

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]
```

Operation

r/m8 × AL → AX r/m16 × AX → DX:AX

r/m32 × EAX → EDX:EAX r/m[16|32] × reg[16|32] → reg|16|32]

imm[16|32] × r/m[16|32] → reg|16|32]

Description

The single-operand form of `imul` executes a signed multiply of a byte, word, or long by the contents of the AL, AX, or EAX register and stores the product in the AX, DX:AX or EDX:EAX register respectively.

The two-operand form of `imul` executes a signed multiply of a register or memory word or long by a register word or long and stores the product in that register word or long.

The three-operand form of `imul` executes a signed multiply of a 16- or 32-bit immediate by a register or memory word or long and stores the product in a specified register word or long.

`imul` clears the overflow and carry flags under the following conditions:

TABLE 2–5 Clearing OR and CF Flags — `imul`

Instruction Form	**Condition for Clearing OF and CF**
r/m8 × AL → AX	AL = sign-extend of AL to 16 bits
r/m16 × AX → DX:AX	AX= sign-extend of AX to 32 bits

TABLE 2–5 Clearing OR and CF Flags — `imul` *(continued)*

Instruction Form	Condition for Clearing OF and CF
r/m32 × EAX → EDX:EAX	EDX:EAX= sign-extend of EAX to 32 bits
r/m[16|32] × reg[16|32] → reg|16|32]	Product fits exactly within reg[16|32]
imm[16|32] × r/m[16|32] → reg|16|32]	Product fits exactly within reg[16|32]

Example

Perform an 8-bit signed multiply of the AL register and the contents of the effective address (addressed by the ESI register plus an offset of 1):

```
imulb 1(%esi)
```

Perform a 16-bit signed multiply of the constant, -126, and the contents of the effective address (addressed by the EDI register plus an offset of 4). Store the result in the DX register:

```
imulw $-126, 4(%edi), %dx
```

Perform a 32-bit signed multiply of the constant, 12345678, and the contents of the effective address (addressed by the EDI register plus an offset of 4). Store the result in the EDX register:

```
imull $12345678, 4(%edi), %edx
```

Unsigned Multiplication of AL, AX or EAX(mul)

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

Operation

r/m8 × AL → AX

r/m16 × AX → DX:AX

r/m32 × EAX → EDX:EAX

Description

`mul` executes a unsigned multiply of a byte, word, or long by the contents of the AL, AX, or EAX register and stores the product in the AX, DX:AX or EDX:EAX register respectively.

`mul` clears the overflow and carry flags under the following conditions:

TABLE 2–6 Clearing OF and CF Flags — `mul`

Instruction Form	**Condition for Clearing OF and CF**
r/m8 × AL → AX	clear to 0 if AH is 0; otherwise, set to 1
r/m16 × AX → DX:AX	clear to 0 if DX is 0; otherwise, set to 1
r/m32 × EAX → EDX:EAX	clear to 0 if EDX is 0; otherwise, set to 1

Example

Perform an 8-bit unsigned multiply of the AL register and the contents of the effective address (addressed by the ESI register plus an offset of 1):

```
mulb 1(%esi)
```

Perform a 16-bit unsigned multiply of the AL register and the contents of the effective address (addressed by the EDI register plus an offset of 4):

```
mulw 4(%edi)
```

Perform a 32-bit unsigned multiply of the AL register and the contents of the effective address (addressed by the EDI register plus an offset of 1):

```
mull 1(%edi)
```

Unsigned Divide (div)

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

Operation

AX r/m8 → AL DX:AX

r/m16 → AX EDX:EAX

r/m32 → EAX

Description

`div` executes unsigned division. `div` divides a 16-, 32-, or 64-bit register value (dividend) by a register or memory byte, word, or long (divisor). The quotient is stored in the AL, AX, or EAX register respectively.

The remainder is stored in AH, Dx, or EDX. The size of the divisor (8-, 16- or 32-bit operand) determines the particular register used as the dividend.

The OF, SF, ZF, AR, PF and CF flags are undefined.

Example

Perform an 8-bit unsigned divide of the AX register by the contents of the effective address (addressed by the ESI register plus an offset of 1) and store the quotient in the AL register, and the remainder in AH:

```
divb 1(%esi)
```

Perform a 16-bit unsigned divide of the DX:AX register by the contents of the effective address (addressed by the EDI register plus an offset of 4) and store the quotient in the AX register, and the remainder in DX:

```
divw 4(%edi)
```

Perform a 32-bit unsigned divide of the EDX:EAX register by the contents of the effective address (addressed by the EDI register plus an offset of 4) and store the quotient in the EAX register, and the remainder in EDX:

```
divl 4(%edi)
```

Signed Divide (idiv)

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

Operation

AX r/m8 → AL

DX:AX r/m16 → AX

EDX:EAX r/m32 → EAX

Description

`idiv` executes signed division. `idiv` divides a 16-, 32-, or 64-bit register value (dividend) by a register or memory byte, word, or long (divisor). The size of the divisor (8-, 16- or 32-bit operand) determines the particular register used as the dividend, quotient, and remainder.

TABLE 2–7 `idiv` Register Assignment

Divisor Operand Size	Dividend	Quotient	Remainder
byte	AX	AL	AH
word	DX:AX	AX	DX
long	EDX:EAX	EAX	EDX

If the resulting quotient is too large to fit in the destination, or if the divisor is 0, an Interrupt 0 is generated. Non-integral quotients are truncated toward 0. The remainder has the same sign as the dividend; the absolute value of the remainder is always less than the absolute value of the divisor.

Example

Perform a 16-bit signed divide of the DX:AX register by the contents of the effective address (addressed by the EDI register plus an offset of 4) and store the quotient in the AX register

```
divw 4(%edi)
```

Conversion Instructions

Convert Byte to Word (cbtw)

```
cbtw
```

Operation

sign-extend AL → AX

Description

`cbtw` converts the signed byte in AL to a signed word in AX by extending the most-significant bit (sign bit) of AL into all bits of AH.

Example

```
cbtw
```

Convert Word to Long (cwtl)

```
cwtl
```

Operation

sign-extend AX → EAX

Description

`cwtl` converts the signed word in AX to a signed long in EAX by extending the most-significant bit (sign bit) of AX into two most-significant bytes of EAX.

Example

```
cwtl
```

Convert Signed Word to Signed Double Word (cwtd)

```
cwtd
```

Operation

sign-extend AX → DX:AX

Description

`cwtd` converts the signed word in AX to a signed double word in DX:AX by extending the most-significant bit (sign bit) of AX into all bits of DX.

Example

```
cwtd
```

Convert Signed Long to Signed Double Long (cltd)

```
cltd
```

Operation

sign-extend EAX → EDX:EAX

Description

`cltd` converts the signed long in EAX to a signed double long in EDX:EAX by extending the most-significant bit (sign bit) of EAX into all bits of EDX.

Example

```
cltd
```

Decimal Arithmetic Instructions

Decimal Adjust AL after Addition (daa)

```
daa
```

Operation

decimal-adjust AL → AL

Description

Use `daa` only after executing the form of an `add` instruction that stores a two-BCD-digit byte result in the AL register. `daa` then adjusts AL to a two-digit packed decimal result.

Example

Decimal adjust the two-BCD-digit in the AL register:

```
daa
```

Decimal Adjust AL after Subtraction (das)

```
das
```

Operation

decimal-adjust AL → AL

Description

Use das only after executing the form of a sub instruction that stores a two-BCD-digit byte result in the AL register. das then adjusts AL to a two-digit packed decimal result.

Example

Decimal adjust the two-BCD-digit in the AL register:

```
das
```

ASCII Adjust after Addition (aaa)

```
aaa
```

Operation

ASCII-adjust AL → AL

Description

You use aaa only after executing the form of an add instruction that stores a two-BCD-digit byte result in the AL register. aaa then adjusts AL to contain the correct decimal result. The top nibble of AL is set to 0. To convert AL to an ASCII result, follow the aaa instruction with:

```
or %al, 0x30
```

The following table shows how aaa handles a carry.

TABLE 2–8 aaa Handling a Carry

Carry	Action
decimal carry	AH + 1; CF and AF set to 1
no decimal carry	AH unchanged; CF and AF cleared to 0

TABLE 2–8 `aaa` Handling a Carry *(continued)*

Example

Adjust the AL register to contain the correct decimal result after an `add` instruction that stores a two-BCD-digit byte.

```
aaa
```

ASCII Adjust after Subtraction (aas)

```
aas
```

Operation

ASCII-adjust AL → AL

Description

Use `aas` only after executing the form of an `add` instruction that stores a two-BCD-digit byte result in the AL register. `aas` then adjusts AL to contain the correct decimal result. The top nibble of AL is set to 0. To convert AL to an ASCII result, follow the `aas` instruction with:

```
or %al, 0x30
```

Table 2–9 shows how `aas` handles a carry.

TABLE 2–9 How `aas` Handles a Carry

Carry	Action
decimal carry	AH - 1; CF and AF set to 1
no decimal carry	AH unchanged; CF and AF cleared to 0

Example

Adjust the AL register to contain the correct decimal result after a `sub` instruction that stores a two-BCD-digit byte

```
aas
```

ASCII Adjust AX after Multiply (aam)

```
aam
```

Operation

AL 10 → AH mod 10 AL → AL

Description

You use `aam` only after executing a `mul` instruction between two BCD digits (unpacked). `mul` stores the result in the AX register. The result is less than 100 so it can be contained in the AL register (the low byte of the AX register). `aam` unpacks the AL result by dividing AL by 10, stores the quotient (most-significant digit) in AH, and stores the remainder (least-significant digit) in AL.

Example

Adjust the AL register to contain the correct decimal result after a `mul` instruction between two BCD digits:

```
aam
```

ASCII Adjust AX before Division (aad)

```
aad
```

Operation

AL + (AH × 10) → AL 0 → AH

Description

aad prepares two unpacked BCD digits for a division operation that yields an unpacked result. The least-significant digit is in AL; the most-significant in AH.

aad prepares the AL and AH registers:

```
AL + (AH × 10) → AL
 0 → AH
```

AX is then equal to the binary equivalent of the original unpacked two-digit BCD number.

Example

Adjust the AL and AH registers for a division operation by setting the AX register equal to the original unpacked two-digit number:

```
aad
```

Coprocessor Instructions

fwait

Wait (wait, fwait)

```
wait
```

Description

wait — processor suspends instruction execution until the BUSY # pin is inactive (high).

fwait — processor checks for pending unmasked numeric exceptions before proceeding.

Example

Suspend instruction execution until not BUSY and check for exceptions:

```
wait
```

String Instructions

All Intel string op mnemonics default to long.

Move Data from String to String (movs)

```
movs{bwl}
movs{bwl} m[8|16|32], reg[16|32]
```

Operation

move {bwl} [(E)SI] → ES: (E)DI]

move {bwl} DS: [(E)SI] → ES: [(E)DI]

Description

Copies the byte, word, or long in [(E)SI] to the byte, word, or long in ES:[(E)DI}. Before executing the move instruction, load the index values into the SI source- and DI destination-index registers.

The destination operand must be addressable from the ES register; it cannot span segments. A source operand, however, can span segments; the default is DS.

After the data is moved, both the source- and destination-index registers are automatically incremented or decremented as determined by the value of the direction flag (DF). The index registers are incremented if DF = 0 (DF cleared by a `cld` instruction); they are decremented if DF = 1 (DF set by a `std` instruction). The increment/decrement count is 1 for a byte move, 2 for a word, and 4 for a long.

For a block move of CX bytes or words, precede a `movs` instruction with a `rep` prefix.

Example

Copy the 8-bit byte from the DS:[(E)SI] to the ES:[(E)DI] register.

```
movsb
```

Compare String Operands (cmps)

```
cmps{bwl}
```

Operation

compare DS:[(E)SI] with ES:[(E)DI]

Description

Compares the byte, word, or long in DS:[(E)SI] with the byte, word, or long in ES:[(E)DI}. Before executing the `cmps` instruction, load the index values into the SI source- and DI destination-index registers.

`cmps` subtracts the operand indexed by the destination-index from the operand indexed by the source-index register.

After the data is compared, both the source- and destination-index registers are automatically incremented or decremented as determined by the value of the direction flag (DF). The index registers are incremented if DF = 0 (DF cleared by a `cld` instruction); they are decremented if DF = 1 (DF set by a `std` instruction). The increment/decrement count is 1 for a byte move, 2 for a word, and 4 for a long.

For a block compare of CX or ECX bytes, words or longs, precede a `cmps` instruction with a `repz` or `repnz` prefix.

Example

Compare the 8-bit byte in the DS:[(E)SI] register to the ES:[(E)DI] register.

```
cmpsb
```

Compare the 16-bit word in the DS:[(E)SI] register to the ES:[(E)DI] register.

```
cmpsw
```

Compare the 32-bit word in the DS:[(E)SI] register to the ES:[(E)DI] register.

```
cmpsl
```

Store String Data (stos)

```
stos{bwl}
```

Operation

store [AL|AX|EAX] → ES:[(E)DI]

Description

Transfers the contents of the AL, AX, or EAX register to the memory byte or word addressed in the destination register relative to the ES segment. Before executing the move instruction, load the index values into the DI destination-index register.

The destination operand must be addressable from the ES register; it cannot span segments.

After the data is transferred, the destination-index register is automatically incremented or decremented as determined by the value of the direction flag (DF). The index registers are incremented if DF = 0 (DF cleared by `a cld` instruction); they are decremented if DF = 1 (DF set by a `std` instruction). The increment/decrement count is 1 for a byte move, 2 for a word, and 4 for a long.

For a block transfer of CX bytes, words or longs, precede a `stos` instruction with a `rep` prefix.

Example

Transfer the contents of the AL register to the memory byte addressed in the destination register, relative to the ES segment.

```
stosb
```

Transfer the contents of the AX register to the memory word addressed in the destination register, relative to the ES segment

```
stosw
```

Transfer the contents of the EAX register to the memory double-word addressed in the destination register, relative to the ES segment

```
stosl
```

The Load String Operand (lods)

```
lods{bwl}
```

Operation

load ES:[(E)DI] → [AL|AX|EAX]

Description

Loads the memory byte or word addressed in the destination register into the AL, AX, or EAX register. Before executing the `lods` instruction, load the index values into the SI source-index register.

After the data is loaded, the source-index register is automatically incremented or decremented as determined by the value of the direction flag (DF). The index register is incremented if DF = 0 (DF cleared by `a cld` instruction); it is decremented if DF = 1 (DF set by a `std` instruction). The increment/decrement count is 1 for a byte move, 2 for a word, and 4 for a long.

For a block transfer of CX bytes, words or longs, precede a `lods` instruction with a `rep` prefix; however, `lods` is used more typically within a loop construct where further processing of the data moved into AL, AX, or EAX is usually required.

Example

Load the memory byte addressed in the destination register, relative to the ES segment register, into the AL register.

```
lodsb
```

Load the memory word addressed in the destination register, relative to the ES segment register, into the AX register.

```
lodsw
```

Load the memory double-word addressed in the destination register, relative to the ES segment register, into the EAX register.

```
lodsl
```

Compare String Data (scas)

```
scas{bwl}
```

Operation

compare ES:[(E)DI] with [AL|AX|EAX]

Description

Compares the memory byte or word addressed in the destination register relative to the ES segment with the contents of the AL, AX, or EAX register. The result is discarded; only the flags are set.

Before executing the scas instruction, load the index values into the DI destination-index register. The destination operand must be addressable from the ES register; it cannot span segments.

After the data is transferred, the destination-index register is automatically incremented or decremented as determined by the value of the direction flag (DF). The index registers are incremented if DF = 0 (DF cleared by `a cld` instruction); they are decremented if DF = 1 (DF set by a `std` instruction). The increment/decrement count is 1 for a byte move, 2 for a word, and 4 for a long.

For a block search of CX or ECX bytes, words or longs, precede a `scas` instruction with a `repz` or `repnz` prefix.

Example

Compare the memory byte addressed in the destination register, relative to the ES segment, with the contents of the AL register.

```
scasb
```

Compare the memory word addressed in the destination register, relative to the ES segment, with the contents of the AX register

```
scasw
```

Compare the memory byte double-word addressed in the destination register, relative to the ES segment, with the contents of the EAX register

```
scasl
```

Look-Up Translation Table (xlat)

```
xlat
```

Operation

set AL to DS:[(E)BX + unsigned AL]

Description

Changes the AL register from the table index to the table entry. AL should be the unsigned index into a table addressed by DS:BX (16-bit address) or DS:EBX (32-bit address).

Example

Change the AL register from the table index to the table entry.

```
xlat
```

Repeat String Operation (rep, repnz, repz)

```
rep;
repnz;
repz;
```

Operation

repeat string-operation until tested-condition

Description

Use the `rep` (repeat while equal), `repnz` (repeat while nonzero) or `repz` (repeat while zero) prefixes in conjunction with string operations. Each prefix causes the associated string instruction to repeat until the count register (CX) or the zero flag (ZF) matches a tested condition.

Example

Repeat while equal: Copy the 8-bit byte from the DS:[(E)SI] to the ES:[(E)DI] register.

```
rep; movsb
```

Repeat while not zero: Compare the memory byte double-word addressed in the destination register EDL, relative to the ES segment, with the contents of the EAX register.

```
repnz; scasl
```

Repeat while zero:Transfer the contents of the EAX register to the memory double-word addressed in the destination register EDL, relative to the ES segment.

```
repz; stosl
```

Procedure Call and Return Instructions

Far Call — Procedure Call (lcall)

```
lcall    immptr
lcall    *mem48
```

Operation

far call ptr16:{16|32} far call m16:{16|32}

Description

The `lcall` instruction calls intersegment (far) procedures using a full pointer. `lcall` causes the procedure named in the operand to be executed. When the called procedure completes, execution flow resumes at the instruction following the `lcall` instruction (see the `return` instruction).

`lcall` ptr16:{16|32} uses a four-byte or six-byte operand as a long pointer to the called procedure.

`lcall` m16:{16|32} fetches the long pointer from the specified memory location.

In Real Address Mode or Virtual 8086 Mode, the long pointer provides 16 bits for the CS register and 16 or 32 bits for the EIP register. Both forms of the lcall instruction push the CS and IP or EIP registers as a return address.

Example

Use a four-byte operand as a long pointer to the called procedure.

```
lcall $0xfebc, $0x12345678
```

Fetch a long pointer from the memory location addressed by the `edx` register, offset by 3.

```
lcall *3(%edx)
```

Near Call — Procedure Call (call)

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

Operation

near call rel{16|32}

near call r/m{16|32}

Description

The `call` instruction calls near procedures using a full pointer. `call` causes the procedure named in the operand to be executed. When the called procedure completes, execution flow resumes at the instruction following the `call` instruction (see the `return` instruction).

`call` rel{16|32} adds a signed offset to address of the instruction following the `call` instruction to determine the destination; that is, the displacement is relative to the next instruction. The displacement value is stored in the EIP register. For rel16, the upper 16 bits of EIP are cleared to zero resulting in an offset value that does not exceed 16 bits.

`call` r/m{16|32} specifies a register or memory location from which the absolute segment offset is fetched. The offset of the instruction following the `call` instruction is pushed onto the stack. After the procedure completes, the offset is popped by a near `ret` instruction within the procedure.

Both forms of the `call` instruction have no affect on the CS register.

Example

Program counter minus 0x11111111.

```
call .-0x11111111
```

Add a signed offset value to the address of the next instruction.

```
call *4(%edi)
```

Return from Procedure (ret)

```
ret
ret  imm16
```

Operation

return to caller

Description

The `ret` instruction transfers control to the return address located on the stack. This address is usually placed on the stack by a `call` instruction. Issue the `ret` instruction within the called procedure to resume execution flow at the instruction following the `call`.

The optional numeric (16- or 32-bit) parameter to `ret` specifies the number of stack bytes or words to be released after the return address is popped from the stack. Typically, these bytes or words are used as input parameters to the called procedure.

For an intersegment (near) return, the address on the stack is a segment offset that is popped onto the instruction pointer. The CS register remains unchanged.

Example

Transfer control to the return address located on the stack.

```
ret
```

Transfer control to the return address located on the stack. Release the next 16-bytes of parameters.

```
ret $-32767
```

Long Return (lret)

```
lret
lret     imm16
```

Operation

return to caller

Description

The `lret` instruction transfers control to a return address located on the stack. This address is usually placed on the stack by an `lcall` instruction. Issue the `lret` instruction within the called procedure to resume execution flow at the instruction following the `call`.

The optional numeric (16- or 32-bit) parameter to `lret` specifies the number of stack bytes or words to be released after the return address is popped from the stack. Typically, these bytes or words are used as input parameters to the called procedure.

For an intersegment (far) return, the address on the stack is a long pointer. The offset is popped first, followed by the selector.

In Real Mode, CS and IP are loaded directly. In Protected mode, an intersegment return causes the processor to check the descriptor addressed by the return selector. The AR byte of the descriptor must indicate a code segment of equal or lesser privilege (or greater or equal numeric value) than the current privilege level. Returns to a lesser privilege level cause the stack to be reloaded from the value saved beyond the parameter block.

Example

Transfer control to the return address located on the stack.

```
lret
```

Transfer control to the return address located on the stack. Release the next 16-bytes of parameters.

```
lret $-32767
```

Enter/Make Stack Frame for Procedure Parameters (enter)

```
enter    imm16, imm8
```

Operation

make stack frame for procedure parameters

Description

Create the stack frame required by most block-structured high-level languages. The imm16 operand specifies the number of bytes of dynamic storage allocated on the stack for the routine being entered. The imm8 operand specifies the lexical nesting level (0 to 31) of the routine within the high-level language source code. The nesting level determines the number of stack frame pointers copied into the new stack frame from the preceding frame.

Example

Create a stack frame with 0xfecd bytes of dynamic storage on the stack and a nesting level of 0xff.

```
enter $0xfecd, $0xff
```

High Level Procedure Exit (leave)

```
leave
```

Operation

set (E)SP to (E)BP, then pop (E)BP

Description

The `leave` instruction reverses the actions of an `enter` instruction. `leave` copies the frame pointer to the stack point and releases the stack space formerly used by a procedure for its local variables. `leave` pops the old frame pointer into (E)BP, thus restoring the caller’s frame. A subsequent `ret` *nn* instruction removes any arguments pushed onto the stack of the exiting procedure.

Example

Copy the frame pointer to the stack pointer and release the stack space.

```
leave
```

Jump Instructions

Jump if ECX is Zero (jcxz)

```
jcxz     disp8
```

Operation

jump to disp8 if (E)CX is 0

Description

The jcxz instruction tests the contents of the CX or ECX register for 0. jcxz differs from other conditional jumps that it tests the flags, rather than (E)CX.

jcxz is useful at the beginning of a loop that terminates with a conditional loop instruction; such as:

```
loopne .-126
```

In this case, jcxz tests CX or ECX for 0 prior to entering the loop, thus executing 0 times:

Example

```
jcxz .-126
 ...
loopne .-126
```

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

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

Operation

decrement count; jump to disp8 if count not equal 0

decrement count; jump to disp8 if count not equal 0 and ZF = 0 decrement count; jump to disp8 if count not equal 0 and ZF = 1

Description

`loop` decrements the count register; the flags register remains unchanged. Conditions are checked for by the particular form of `loop` you used. If the conditions match, a short jump is made to the address specified by the disp8 operand. The range of the disp8 operand, relative to the current instruction, is +127 decimal bytes to -128 decimal bytes.

`loop` instructions provide iteration control and combine loop index management with conditional branching. Prior to using the `loop` instruction, load the count register with an unsigned iteration count. Then, add the `loop` instruction at the end of a series of instructions to be iterated. The disp8 operand points to the beginning of the iterative loop.

Example

Decrement the count register and when the count is not equal to zero, jump short to the disp8 location.

```
loopne .-126
```

Jump (jmp, ljmp)

```
jmp disp{8|16|32}
jmp *r/m{16|32}
ljmp immPtr
ljmp *mem48
jcc disp{8|32}
```

Operation

jump short or near; displacement relative to next instruction

jump far (intersegment; 4- or 6-byte immediate address

jump if condition is met; displacement relative to next instruction

Description

The jmp instruction transfers execution control to a different point in the instruction stream; records no return information.

Jumps with destinations of disp[8|16|32] or r/m[16|32] are near jumps and do not require changes to the segment register value.

`jmp` rel{16|32} adds a signed offset to the address of the instruction following the `jmp` instruction to determine the destination; that is, the displacement is relative to the next instruction. The displacement value is stored in the EIP register. For rel16, the upper 16 bits of EIP are cleared to zero resulting in an offset value not to exceed 16 bits.

`ljmp` ImmPtr or *mem48 use a four- or six-byte operand as a long pointer to the destination. In Real Address Mode or Virtual 8086 mode, the long pointer provides 16 bits for the CS register and 16 or 32 bits for the EIP register. In Protected mode, both long pointer forms consult the AR (Access Rights) byte of the descriptor indexed by the selector part of the long pointer. The `jmp` performs one of the following control transfers depending on the value of the AR byte:

- A jump to a code segment at the same privilege level
- A task switch

Example

Jump to the relative effective address (addressed by the `EDI` register plus an offset of 4):

```
jmp *4(%edi)
```

Long jump, use 0xfebc for the CS register and 0x12345678 for the EIP register:

```
ljmp $0xfebc, $0x12345678
```

Jump if not equal:

```
jne .+10
```

Interrupt Instructions

Call to Interrupt Procedure (int, into)

```
int 3
int imm8
  into
```

Operation

interrupt 3 — trap to debugger

interrupt numbered by immediate byte

interrupt 4 — if overflow flag is 1

Description

The `int` instruction generates a software call to an interrupt handler. The imm8 (0 to 255) operand specifies an index number into the IDT (Interrupt Descriptor Table) of the interrupt routine to be called. In Protect Mode, the IDT consists of an array of 8-byte descriptors; the descriptor for the interrupt invoked must indicate an interrupt, trap, or task gate. In Real Address Mode, the IDT is an array of four byte-long pointers. In Protected and Real Address Modes, the base linear address of the IDT is defined by the contents of the IDTR.

The `into` form of the `int` instruction implies interrupt 4. The interrupt occurs only if the overflow flag is set.

The first 32 interrupts are reserved for system use. Some of these interrupts are used for internally generated exceptions.

The `int` imm8 form of the interrupt instruction behaves like a far call except that the flags register is pushed onto the stack before the return address. Interrupt procedures return via the `iret` instruction, which pops the flags and return address from the stack.

In Real Address Mode, the `int imm8` pushes the flags, CS, and the return IP onto the stack, in that order, then jumps to the long pointer indexed by the interrupt number.

Example

Trap to debugger:

```
int $3
```

Trap to interrupt 0xff:

```
int $0xff
```

Trap to interrupt 4:

```
into
```

Interrupt Return (iret)

```
iret
```

Operation

return → routine

Description

In Real Address Mode, `iret` pops CS, the flags register, and the instruction pointer from the stack and resumes the routine that was interrupted. In Protected Mode, the setting of the nested task flag (NT) determines the action of `iret`. The IOPL flag register bits are changed when CPL equals 0 and the new flag image is popped from the stack.

`iret` returns from an interrupt procedure without a task switch if NT equals 0. Returned code must be equally or less privileged than the interrupt routine as indicated CS selector RPL bits popped from the stack. If the returned code is less privileged, `iret` pops SS and the stack pointer from the stack.

`iret` reverses the operation of an INT or CALL that caused the task switch if NT equals 1.The task executing `iret` is updated and saved in its task segment. The code that follows `iret` is executed if the task is re-entered.

Example

Resume the interrupted routine:

```
iret
```

Protection Model Instructions

Store Local Descriptor Table Register (sldt)

```
sldt r/m16
```

Operation

LDTR → r / m[16]

Description

The Local Descriptor Table Register (LDTR) is stored by `sldt` as indicated by the effective address operand. LDTR is stored into the two-byte register or the memory location.

`sldt` is not used in application programs. It is used only in operating systems.

Example

Store the LDTR in the effective address (addressed by the EBX register plus and offset of 5):

```
sldt 5(%ebx)
```

Store Task Register (str)

```
str r/m16
```

Operation

STR → r / m(16

Description

The contents of the task register is stored by `sldt` as indicated by the effective address operand. STR is stored into the two-byte register or the memory location.

Example

Store str in the effective address (addressed by the EBX register plus an offset of 5):

```
str 5(%ebx)
```

Load Local Descriptor Table Register (lldt)

```
lldt     r/m16
```

Operation

SELECTOR → LDTR

Description

LDTR is loaded by LLDT. The operand (word) contains a selector to a local GDT (Global Descriptor Table). The descriptor registers are not affected.The task state segment LDT field does not change.

The LDTR is marked invalid if the selector operand is 0. A #GP fault is caused by all descriptor references (except LSL VERR, VERW, or LAR instructions).

LLDT is not used in application programs. It is used in operating systems.

Example

Load the LLDT register from the effective address (addressed by the EBX register plus and offset of 5):

```
lldt 5(%ebx)
```

Load Task Register (ltr)

```
ltr r/m16
```

Operation

r/m16 → Task Register

Description

The task register is loaded by LTR from the source register or memory location specified by the operand. The loaded task state segment is tagged busy. A task switch does not occur.

Example

Load the TASK register from the effective address (addressed by the EBX register plus and offset of 5):

```
ltr 5(%ebx)
```

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

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

Operation

1 → ZF (if segment can be read or written)

Description

VERR and VERW contains the value of a selector in the two-byte register or memory operand. VERR and VERW determine if the indicated segment can be reached in the current privilege level and whether it is readable (VERR) or writable (VERW). If the segment can be accessed, the zero flag (ZF) is set to 1, otherwise the zero flag is set to 0. For the zero flag to be set these conditions must be met:

- The selector denotes a descriptor; the selector is “defined”.
- The selector is a code or data segment; not a task statement, LDT or a gate.
- For VERR, the segment must be readable, for VERW, writable.
- The descriptor privilege level (DPL) can be any value for VERR. otherwise the DPL must have the same or less privilege as the current level and the DPL of the selector.

Validation is performed as if the segment were loaded into DS, ES, FS, or GS and the indicated write or read performed. The validation results are indicated by the zero flag. The value of the selector cannot result in an exception.

Example

Determine if the segment indicated by the effective address (addressed by the EBX register plus an offset of 5) can be reached in the current privilege level and whether it is readable (VERR):

```
verr 5(%ebx)
```

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

```
sgdt     mem48
sidt     mem48
```

Operation

DTR → mem48

Description

The contents of the descriptor table register is copied by `sgdt/sidt` to the six bytes of memory specified by the operand. The first word at the effective address is assigned the LIMIT field of the register. If the operand-size attribute is 32-bits:

- The base field of the register is assigned to the next three bytes.
- The fourth byte is written as zero.
- The last byte is undefined.

If the operand-size attribute is 16-bits, the 32-bit BASEfield of the register is assigned to the next four bytes.

`sgdt/sldt` are not used in application programs, they are used in operating systems.

Example

Copy the contents of the Global Descriptor Table Register to the specified memory location:

```
sgdt 0x55555555
```

Copy the contents of the Interrupt Descriptor Table Register to the effective address (addressed by the EBX register plus an offset of 5):

```
sidt 5 (%ebx)
```

Load Global/Interrupt Descriptor Table (lgdt, lidt)

```
lgdt     mem48
lidt     mem48
```

Operation

MEM48 → GDTR MEM48 → IDTR

Description

The GDTR and IDTR are loaded with a linear base address and limit value from a six-byte operand in memory by the `lgdt/lidt` instructions. For a 16-bit operand:

- Load the register with a 16-bit limit and a 24-bit base.
- The six-byte data operand high-order eight bits are not used.

For a 32-bit operand:

- Load the register with a 16-bit limit and a 32-bit base.
- The six-byte data operand high-order eight bits are used as the high-order base address bits.

All 48-bits of the six-byte data operand are always stored into by the `sgdt/sidt` instructions. For a 16-bit and a 32-bit operand, the upper eight-bits are written with the high-order eight address bits. lgdt or lidt, when used with a 16-bit operand to load the register stored by `sgdt` or `sidt`, stores the upper eight-bits as zeros.

`lgdt` and `lidt` are not used in application programs; they are used in operation system. `lgdt` and `lidt` are the only instructions that load a linear address directly in 80386 Protected Mode.

Example

Load the Global/Interrupt Descriptor Table Register from memory address 0x55555555:

```
lgdt 0x55555555
lidt 0x55555555
```

Store Machine Status Word (smsw)

```
smsw     r/m16
```

Operation

MSW → r/m16

Description

The machine status word is stored by `smsw` in the two-byte register of memory location pointed to by the effective address operand.

80386 machines should use MOV ..., CR0.

Example

Store the machine status word in the effective address (addressed by the EBX register plus an offset of 5):

```
smsw 5(%ebx)
```

Load Machine Status Word (lmsw)

```
lmsw     r/m16
```

Operation

r/m16 → MSW

Description

The machine status word (part of CR0) is loaded by `lmsw` from the source operand. `lmsw` can be used to switch to Protected Mode if followed by an intersegment jump to clear the instruction queue. `lmsw` cannot switch back to Real Address Mode.

`lmsw` is not used in application programs. It is used in operating systems.

Example

Load the machine status word from the contents of the effective address (addressed by the EBX register plus an offset of 5):

```
lmsw 5(%ebx)
```

Load Access Rights (lar)

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

Operation

r / m16 (masked by FF00) → r16

r / m32 (masked by 00FxFF00) → r32

Description

If the selector is visible at the CPL (modified by the RPL) and is a valid descriptor type, `lar` stores a form of the second doubleword of the descriptor for the source selector. The designated register is loaded with the double-word (high-order) of the descriptor masked by 00FxFF00, and the zero flag is set to 1. The x in 00Fx ... indicates that these four bits loaded by `lar` are undefined. The zero flag is cleared if the selector is invisible or of the wrong type.

The 32-bit value is stored in the 32-bit destination register if the 32-bit operand size is specified. If the 16-bit operand size is specified, the lower 16-bits of this value are stored in the 16-bit destination register.

For `lar`, all data segment descriptors and code are valid.

Example

Load access rights from the contents of the effective address (addressed by the EBX register plus an offset of 5) into the EDX register:

```
lar 5(%ebx),  %edx
```

Load Segment Limit (lsl)

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

Operation

Selector rm16 (byte) → r16

Selector rm32 (byte) → r32

Selector rm16 (page) → r16

Selector rm32 (page) → r32

Description

`lsl` loads a register with a segment limit (unscrambled). The descriptor type must be accepted by `lsl`, and the source selector must be visible at the CPL weakened by

RPL. ZF is then set to 1. Otherwise, ZF is set to 0 and the destination register is unchanged.

The segment limit is loaded as a byte value. A page value limit in the descriptor is translated by `lsl` to a byte limit before `lsl` loads it in the destination register (the 20-bit limit from the descriptor is shifted left 12 and OR’d with 00000FFFH).

`lsl` stores the 32-bit granular limit in the 16-bit destination register.

For `lsl`, code and data segment descriptors are valid.

Example

Load a segment limit from the contents of the effective address (addressed by the EBX register plus an offset of 5) into the EDX register.

```
lsl 5(%ebx), %edx
```

Clear Task-Switched (clts)

```
clts
```

Operation

0 → TS Flag in CR0

Description

The task-switched flag in register CR0 is cleared by `clta`. The TS Flag is set by the 80386 for each task switch. The TS Flag is used as follows:

- If the TS Flag is set, each execution of the ESC instruction is trapped.
- If the TS Flag and the MP Flag are both set, execution of a Wait instruction is trapped.

If a task switch is made after an ESC instruction is started, save the processor extension context before a new ESC instruction can be run. The fault handler resets the TS Flag and saves the context.

`clts` is not used in application program, it is used in operating systems.

`clts` can only be executed at privilege level 0.

Example

Clear the TS flag:

```
clts
```

Adjust RPL Field of Selector (arpl)

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

Operation

If RPL 1 < RPL 2, 1 → ZF

Description

`arpl` has two operands. The first operand is a 16-bit word register or memory variable that contains the value of a selector. The second operand is a word register. If the RPL field of the second operand is greater than the RPL field of the first operand, ZF is set to 1 and the RPL field of the first operand is increased to match the RPL field of the second operand. Otherwise, no change is made to the first operand and the ZF is set to 0.

`arpl` is not used in application programs, it is used in operating systems.

`arpl` guarantees that a selector to a subroutine does not request a privilege greater than allowed. Normally, the second operand of `arpl` is a register that contains the CS selector value of the caller.

Example

```
arpl %sp, 5(%ebx)
```

Bit Instructions

Bit Scan Forward (bsf)

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

Operation

(r/m = 0) 0 → ZF (r/m ≠ 0) 0 → ZF

Description

bsf scans the bits, starting at bit 0, in the doubleword operand or the second word. If the bits are all zero, ZF is cleared. Otherwise, ZF is set and the bit index of the first set bit, found while scanning in the forward direction, is loaded into the destination register.

Example

```
bsf 4(%edi), %edx
```

Bit Scan Reverse (bsr)

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

Operation

(r/m = 0) 0 → ZF (r/m ≠ 0) 0 → ZF

Description

bsr scans the bits, starting at the most significant bit, in the doubleword operand or the second word. If the bits are all zero, ZF is cleared. Otherwise, ZF is set and the bit index of the first set bit found, while scanning in the reverse direction, is loaded into the destination register

Example

```
bsr 4(%edi), %edx
```

Bit Test (bt)

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

Operation

BIT [LeftSRC, RightSRC] → CF

Description

The bit indicated by the first operand (base) and the second operand (offset) are saved by *bt* into CF (carry flag).

Example

```
btl $253, 4(%edi)
btl %edx, 4(%edi)
```

Bit Test And Complement (btc)

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

Operation

BIT [LeftSRC, RightSRC] → CF

NOT BIT [LeftSRC, RightSRC] → BIT[LeftSRC, RightSRC]

Description

The bit indicated by the first operand (base) and the second operand (offset) are saved by *btc* into CF (carry flag) and complements the bit.

Example

```
btl $253, 4(%edi)
btl %edx, 4(%edi)
```

Bit Test And Reset (btr)

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

Operation

BIT[LeftSRC, RightSRC] → CF

0 → BIT[LeftSRC, RightSRC]

Description

The value of the first operand (base) and the second operand (bit offset) are saved by `btr` into the carry flag and then it stores 0 in the bit.

Example

```
btrl $253, 4(%edi)
 btrl $edx, 4(%edi)
```

Bit Test And Set (bts)

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

Operation

BIT[LeftSRC, RightSRC] → CF

0 → BIT[LeftSRC, RightSRC]

Description

The value of the first operand (base) and the second operand (bit offset) are saved by `bts` into the carry flag and then it stores 1 in the bit.

Example

```
btsl $253, 4(%edi)
 btsl $edx, 4(%edi)
```

Exchange Instructions

Compare and Exchange (cmpxchg)[486]

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

Example

```
cmpxchgb %cl, 1(%esi)
 cmpxchgl %edx, 4(%edi)
```

Floating-Point Transcendental Instructions

Floating-Point Sine (fsin)

```
fsin
```

Example

Replace the contents of the top of the stack with its sine.

```
fsin
```

Floating-Point Cosine (fcos)

```
fcos
```

Example

Replace the contents of the top of the stack with its cos.

```
fcos
```

Floating-Point Sine and Cosine (fsincos)

```
fsincos
```

Example

Replace the contents of the top of the stack with its sine and then push the cosine onto the FPU stack.

```
fsincos
```

Floating-Point Constant Instructions

Floating-Point Load One (fld)

```
fld1
fld12+
fld12e
fldpi
fldlg2
```

```
fldln2
fldz
```

Example

Use these constant instructions to push often-used values onto the FPU stack.

```
fldl 2(%ecx)
```

Processor Control Floating-Point Instructions

Floating-Point Load Control Word (fldcw)

```
fldcw r/m16
```

Example

Load the FPU control word with the value in the specified memory address.

```
fldcw 2(%ecx)
```

Floating-Point Load Environment (fldenv)

```
fldenv mem
```

Example

Reload the FPU environment from the source-operand specified memory space.

```
fldenv 2(%ecx)
```

Miscellaneous Floating-Point Instructions

Floating-Point Different Reminder (fprem)

```
fprem1
```

Example

Divide stack element 0 by stack element 1 and leave the remainder in stack element 0.

```
fprem
```

Floating-Point Comparison Instructions

Floating-Point Unsigned Compare (fucom)

```
fucom freg
```

Description:

Compare stack element 0 with stack element (i). Use condition codes:

No compare: 111

(i) < stack 0: 000

(i) > stack 0: 001

(i) = stack 0: 100

Example

Compare stack element 0 with stack element 7.

```
fucom %st(7)
```

Floating-Point Unsigned Compare And Pop (fucomp)

```
fucomp freg
```

Description

Compare stack element 0 with stack element (i). Use condition codes shown for fucom. Then pop the stack.

Example

```
fucomp %st(7)
```

Floating-Point Unsigned Compare And Pop Two (fucompp)

```
fucompp
```

Description

Compare stack element 0 with stack element (i). Use condition codes shown for fucom. Then pop the stack twice.

Example

```
fucompp %st(7)
```

Load and Move Instructions

Load Effective Address (lea)

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

Operation

Addr(m) → r16 Addr(m) → r32

Truncate to 16 bits(Addr(m)) → r16

Truncate to 16 bits(Addr(m)) → r32

Description

The offset part of the effective address is calculated by `lea` and stored in the specified register. The specified register determines the operand-size attribute if the instruction. The USE attribute of the segment containing the second operand determines the address-size attribute.

Example

```
leal 0x33333333, %edx
```

Move (mov)

```
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]
```

Operation

SRC → DEST

Description

`mov` stores or loads the following special registers in or from a general purpose register.

- Control registers CR0, CR2, and CR3
- Debug registers DR0, DR1, DR2, DR3, DR6, and DR7
- Test registers TR6 and TR7

These instructions always use 32-bit operands.

Example

```
movl %cr3, %ebp
 movl %db7, %ebp
 movl %ebp, %cr3
 movl %ebp, %db7
 movl %tr7, %ebp
 movl %ebp, %tr7
```

Move Segment Registers (movw)

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

Operation

r/m16 → Sreg

Sreg → r/m16

Description

`movw` copies the first operand to the second operand, including data from a descriptor. The descriptor table entry for the selector contains the data for the register. The DS and ES registers can be loaded with a null selector without causing an exception. Use of DS or ES however, causes a #GP(0), and no memory reference occurs.

All interrupts are inhibited until after the execution of the next instruction, after a `movw` into SS. Special actions and checks result from loading a segment register under Protected Mode.

Example

```
movw %CS, 5(%ebx)
 movw %(%ebx), %CS
```

Move Control Registers (mov)

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

Operation

SRC → DEST

Description

This form of mov stores or loads the Control Register CR0, CR2, or CR4 to or from a general purpose register.

These instructions are always used with 32-bit operands.

Example

```
movl %cr3, %ebp
 movl %ebp, %cr3
```

Move Debug Registers (mov)

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

Operation

SRC → DEST

Description

This form of `mov` stores or loads the Debug Register DR1, DR2, or DR3, DR6, and DR7 to or from a general purpose register.

These instructions are always used with 32-bit operands.

Example

```
movl %db7, %ebp
 movl %ebp, %db7
```

Move Test Registers (mov)

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

Operation

SRC → DEST

Description

This form of `mov` stores or loads the Test Register TR6 or TR7 to or from a general purpose register.

These instructions are always used with 32-bit operands.

Example

```
movl %tr7, %ebp
 movl %ebp, %tr7
```

Move With Sign Extend (movsx)

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

Operation

SignExtend(SRC) → DEST

Description

`movsx` reads the contents of the register or effective address as a word or byte. `movsx` then sign-extends the 16- or 32-bit value to the operand-size attribute of the instruction. The result is stored in the destination register by `movsx`.

Example

```
movsxbl 1(%esi), %edx
 movsxwl 5(%ebx), %edx
```

Move With Zero Extend (movzb)

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

Operation

SignExtend(SRC) → DEST

Description

movzx reads the contents of the register or effective address as a word or byte. movzx then sign-extends the 16- or 32-bit value to the operand-size attribute of the instruction. The result is stored in the destination register by movzx.

Pop Instructions

Pop All General Registers (popa)

```
popa{wl}
```

Operation

POP → r16 POP → r32

Description

The eight 16-bit general registers are popped by popa. However, the SP value is not loaded into SP, It is discarded. popa restores the general registers to their values before a previous pusha was executed. DI is the first register popped.

The eight 32-bit registers are popped by popad. However, the ESP value is not loaded into ESP, it is discarded. popad restores the general registers to their values before a previous pushad was executed. EDI is the first register popped.

Example

```
popal
```

Push Instructions

Push All General Registers (pusha)

```
pusha{wl}
```

Operation

SP → r16

SP → r32

Description

The 16-bit or 32-bit general registers are saved by `pusha` and `pushad`, respectively. The stack pointer is decremented by 16 by `pusha` to hold the eight word values. The stack pointer is decremented by 32 by `pushad` to hold the eight doubleword values. The registers are pushed onto the stack in the order received; the stack bytes appear in reverse order. DI or EDI is the last stack pushed.

Example

```
pushal
```

Rotate Instructions

Rotate With Carry Left (rcl)

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

Operation

r/m high-order bit → CF

CF → r/m low-order bit

r/m → ShiftLeft

Description

The left rotate instruction shifts all bits in the register or memory operand specified. The carry flag (CF) is included in the rotation. The most significant bit is rotated to the carry flag, the carry flag is rotated to the least significant bit position, all other bits are shifted to the left. The result includes the original value of the carry flag.

The first operand value indicates how many times the rotate takes place. The value is either the contents of the CL register or an immediate number. For a single rotate, where the first operand is one, the overflow flag (OF) is defined. For all other cases, OF is undefined. After the shift, the carry flag bit is XORed with the most significant result bit.

Example

```
rclb $1, 1(%esi)
 rclb $253, 1(%esi)
 rclb %cl, 1(%esi)
 rcll $1, 4(%edi)
 rcll $253, 4(%edi)
 rcll %cl, 4(%edi)
```

Rotate With Carry Right (rcr)

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

Operation

r/m high-order bit → CF

CF → r/m low-order bit

r/m → ShiftRight

Description

The right rotate instruction shifts all bits in the register or memory operand specified. The carry flag (CF) is included in the rotation. The least significant bit is rotated to the carry flag, the carry flag is rotated to the most significant bit position, all other bits are shifted to the right. The result includes the original value of the carry flag.

The first operand value indicates how many times the rotate takes place. The value is either the contents of the CL register or an immediate number. For a single rotate, where the first operand is one, the overflow flag (OF) is defined. For all other cases, OF is undefined. After the shift, the carry flag bit is XORed with the two most significant result bits.

Example

```
rcrb $1, 1(%esi)
 rcrb $253, 1(%esi)
 rcrb %cl, 1(%esi)
 rcrl $1, 4(%edi)
 rcrl $253, 4(%edi)
 rcrl %cl, 4(%edi)
```

Rotate Left (rol)

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

Operation

r/m high-order bit → CF

CF → r/m low-order bit

r/m → ShiftLeft

Description

The left rotate instruction shifts all bits in the register or memory operand specified. The most significant bit is rotated to the carry flag, the carry flag is rotated to the least significant bit position, all other bits are shifted to the left. The result *does not* include the original value of the carry flag.

The first operand value indicates how many times the rotate takes place. The value is either the contents of the CL register or an immediate number. For a single rotate, where the first operand is one, the overflow flag (OF) is defined. For all other cases, OF is undefined. After the shift, the carry flag bit is XORed with the most significant result bit.

Example

```
rclb $1, 1(%esi)
 rclb $253, 1(%esi)
 rclb %cl, 1(%esi)
 rcll $1, 4(%edi)
 rcll $253, 4(%edi)
 rcll %cl, 4(%edi)
```

Rotate Right (ror)

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

Operation

r/m high-order bit → CF

CF → r/m low-order bit

r/m → ShiftRight

Description

The right rotate instruction shifts all bits in the register or memory operand specified. The least significant bit is rotated to the carry flag, the carry flag is rotated to the most significant bit position, all other bits are shifted to the right. The result *does not* include the original value of the carry flag. The first operand value indicates how many times the rotate takes place. The value is either the contents of the CL register or an immediate number. For a single rotate, where the first operand is one, the overflow flag (OF) is defined. For all other cases, OF is undefined. After the shift, the carry flag bit is XORed with the two most significant result bits.

Example

```
rcrb $1, 1(%esi)
 rcrb $253, 1(%esi)
 rcrb %cl, 1(%esi)
 rcrl $1, 4(%edi)
 rcrl $253, 4(%edi)
 rcrl %cl, 4(%edi)
```

Byte Instructions

Byte Set On Condition (setcc)

```
setcc  r/m8
```

Operation

ConditionTrue: 1 → r/m8

ConditionFalse: 0 → rm/8

Description

If the condition is met, `set`*cc* stores a one byte at the destination specified by the effective address. If the condition is not met, `set`*cc* stores a zero byte. The following table lists the `set`*cc* condition options. Similar condition options are separated by commas, followed by the flag condition.

TABLE 2–10 `set`*cc* Condition List

Instruction (`set+cc`)	**Set Byte If:**
`seta, setnbe`	greater, not equal or less than, CF=0 & ZF=0
`setae, setnc, setnb`	equal or greater, not carry, not less than, CF=0
`setb, setc, setnae`	less than carry, carry = 1, not equal or greater than, CF=1
`setbe, setna`	equal or less than, not greater than carry, CF=1 or ZF=1
`sete, setz`	equal, zero, ZF=1
`setg, setnle`	greater, ZF=0 or SF=OF. not equal or less, ZF=1 or SF ≠ OF
`setge, setnl`	equal or greater, not less, SF = OF
`setl, setnge`	less, not equal or greater, SF ≠ OF
`setle, setng`	equal or less, not greater, ZF = 1 and SF ≠ OF
`setne, setnz`	not equal, not zero, ZF = 0
`setno`	not overflow, OF = 0
`setns`	not sign, SF=0
`seto`	overflow, OF = 1
`setpe, setp`	parity even, parity, PF = 1
`setpo, setnp`	parity odd, not parity, PF = 0
`sets`	sign, SF = 1

Example

```
set(cc) 1(%esi)
```

Byte Swap (bswap) [486]

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

Example

Convert little/big endian to big/little endian by swapping bytes.

```
bswap %ebx
```

Exchange Instructions

Exchange And Add (xadd) [486]

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

Example

Exchange the byte contents of the ESI register with the byte register and load the sum into the ESI register.

```
xaddb %cl, 1(%esi)
```

Exchange Register / Memory With Register (xchg)

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

Operation

DEST → temp

SRC → DEST

temp → SRC

Description

Two operands, in either order, are exchanged by `xchg`. During the exchange, BUS LOCK is asserted (regardless of the value of IOPL or the LOCK prefix) if a memory operand is part of the exchange.

Example

```
xchgb %cl, 1(%esi)  /*exchange byte register with EA byte */
 xchgl %ebp, %eax
 xchgl %ebx, %eax
 xchgl %ecx, %eax
 xchgl %edi, %eax
 xchgl %edx, %eax
 xchgl %edx, 4(%edi)  /*exchange word register with EA word */
 xchgl %esi, %eax
 xchgl %esp, %eax
```

Miscellaneous Instructions

Write Back and Invalidate Cache (wbinvd) [486 only]

```
wbinvd
```

Example

Write back and invalidate the cache.

```
wbinvd
```

Invalidate (invd) [486 only]

```
invd
```

Example

Invalidate the entire cache.

```
invd
```

Invalidate Page (invlpg) [486 only]

```
invlpg mem32
```

Example

Invalidate a single entry in the translation lookaside buffer.

```
invlpg 5(%ebx)
```

LOCK Prefix (lock)

```
lock
```

Operation

LOCK# → NEXT Instruction

Description

The LOCK # signal is asserted during execution of the instruction following the `lock` prefix. This signal can be used in a multiprocessor system to ensure exclusive use of shared memory while LOCK # is asserted. The `bts` instruction is the read-modify-write sequence used to implement test-and-run.

The `lock` prefix works only with the instructions listed here. If a `lock` prefix is used with any other instructions, an undefined opcode trap is generated.

`bt, bts, btr, btc`	m, r/imm
`xchg`	r, m
`xchg`	m, r
`add, or, adc, sbb, and, sub, xor`	m, r/imm
`not, neg, inc, dec`	m

Memory field alignment does not affect the integrity of `lock`.

If a different 80386 processor is concurrently executing an instruction that has a characteristic listed here, locked access is not guaranteed. The previous instruction:

- Does not follow a `lock` prefix
- Is not on the previous list of acceptable instructions
- A memory operand specified has a partial overlap with the destination operand.

Example

```
lock
```

No Operation (nop)

```
nop
```

Operation

NO OPERATION

Description

No operations are performed by `nop`. The `xchgl %eax, %eax` instruction is an alias for the `nop` instruction.

Example

```
nop
```

Halt (hlt)

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

Operation

HLT → ENTER HALT STATE

Description

`halt` puts the 80386 in a HALT state by stopping instruction execution. Execution is resumed by an `nmi` or an enabled interrupt. After a `halt`, if an interrupt is used to continue execution, the saved CS:EIP or CS:IP value points to the next instruction (after the `halt`).

The `halt` instruction is privileged.

Example

```
hlt
```

Real Transfer Instructions

Load Real (fld)

```
fld{lst}
```

Operation

SRC → STACK ELEMENT 0

Description

The source operand is pushed onto the stack by `fld`. The register used before the stack top-pointer is decremented, is the register number used if the source is a register.

Example

Load stack element 7 onto stack element 0.

```
fld %st (7)
```

Store Real (fst)

```
fst{ls}
```

Operation

STACK ELEMENT 0 → DESTINATION

Description

The current value of stack element 0 is copied to the destination. The destination can be a single- or double-real memory operand or another register.

Example

Store the contents of stack element 7 onto stack element 0.

```
%fst (7)
```

Store Real and Pop (fstp)

```
fstp{lst}
```

Operation

STACK ELEMENT 0 → DESTINATION THEN POP

Description

The current value of stack element 0 is copied to the destination. The destination can be a single-, double-, or extended-real memory operand, or another register. Then pop the stack register.

Example

Copy the contents of stack element 0 onto stack element 7 and pop stack element 0.

```
%fstp (7)
```

Exchange Registers (fxch)

```
fxch
```

Example

Exchange the contents of stack element 0 and stack element 7.

```
fxch %st(7)
```

Integer Transfer Instructions

Integer Load (fild)

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

Example

Convert the integer operand (signed) into extended-real and load it onto the floating-point stack.

```
fild 2(%eax)
```

Integer Store (fist)

```
fist{l}
```

Example

Convert the value in stack element 0 into a signed integer and transfer the result to register ECX with an offset of 2.

```
fist 2(%ecx)
```

Integer Store and Pop (fistp)

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

Example

Convert the value in stack element 0 into a signed integer and transfer the result to register ECX with an offset of 2, then pop the stack.

```
fistp 2(%ecx)
```

Packed Decimal Transfer Instructions

Packed Decimal (BCD) Load (fbld)

```
fbld
```

Example

Convert the source operand (BCD) into extended-real and push it onto the floating-point stack.

```
fbld 2(%ecx)
```

Packed Decimal (BCD) Store and Pop (fbstp)

```
fbstp
```

Example

Convert the value in stack element 0 to a packed decimal integer and store the result in register ECX with an offset of 2, and pop the stack.

```
fbstp 2(%ecx)
```

Addition Instructions

Real Add (fadd)

```
fadd{ls}
```

Example

Add stack element 7 to stack element 0 and return the sum to stack element 0.

```
fadd %st(7), %st
```

Real Add and Pop (faddp)

```
faddp
```

Example

Add stack element 0 to stack element 7 and return the sum to stack element 7, then pop the stack.

```
faddp %st, %st(7)
```

Integer Add (fiadd)

```
fiadd{l}
```

Example

Add the integer contents of register ECX to stack element 0.

```
fiadd 2(%ecx)
```

Subtraction Instructions

Subtract Real and Pop (fsub)

```
fsub{ls}
```

Example

Subtract stack element 7 from stack element 0 and return the difference to stack element 0.

```
fsub %st(7), %st
```

Subtract Real (fsubp)

```
fsubp
```

Example

Subtract stack element 7 from stack element 0 and return the difference to stack element 7, then pop the stack.

```
fsubp %st, %st(7)
```

Subtract Real Reversed (fsubr)

```
fsubr{ls}
```

Example

Subtract stack element 0 from stack element 7 and return the difference to stack element 0.

```
fsubr %st(7), %st
```

Subtract Real Reversed and Pop (fsubrp)

```
fsubrp
```

Example

Subtract stack element 0 from stack element 7 and return the difference to stack element 7, then pop the stack.

```
fsubrp %st, %st(7)
```

Integer Subtract (fisubrp)

```
fisubrp
```

Example

Subtract stack element 0 from the integer contents of register ECX (with an offset of 2) and return the difference to register ECX, then pop the stack.

```
fisubrp 2(%ecx)
```

Integer Subtract Reverse (fisubr)

```
fisubr{l}
```

Example

Subtract stack element 0 from the integer contents of register ECX (with an offset of 2) and return the difference to stack element 0.

```
fisubr 2(%ecx)
```

Multiplication Instructions

Multiply Real (fmul)

```
fmul{ls}
```

Example

Multiply stack element 7 by stack element 0 and return the product to stack element 0.

```
fmul %st(7), %st
```

Multiply Real and Pop (fmulp)

```
fmulp
```

Example

Multiply stack element 0 by stack element 7 and return the product to stack element 7, then pop the stack.

```
fmulp %st, %st(7)
```

Integer Multiply (fimul)

```
fimul{l}
```

Example

Multiply the integer contents of register ECX by stack element 0, return the product to register ECX.

```
fimul 2(%ecx)
```

Division Instructions

Divide Real (fdiv)

```
fdiv{ls}
```

Example

Divide stack element 0 by stack element 7 and return the result to stack element 0.

```
fdiv %st(7), %st
```

Divide Real and Pop (fdivp)

```
fdivp
```

Example

Divide stack element 7 by stack element 0 and return the result to stack element 7, then pop the stack.

```
fdivp %st, %st(7)
```

Divide Real Reversed (fdivr)

```
fdivr{ls}
```

Example

Divide stack element 0 by stack element 7 and return the result to stack element 7.

```
fdivr %st, %st(7)
```

Divide Real Reversed and Pop (fdivrp)

```
fdivrp
```

Example

Divide stack element 0 by stack element 7 and return the result to stack element 7, then pop the stack.

```
fdivrp %st, %st(7)
```

Integer Divide (fidiv)

```
fidiv{l}
```

Example

Divide stack element 0 by the integer contents of register ECX, with an offset of 2, and return the result to register ECX.

```
fidiv 2(%ecx)
```

Integer Divide Reversed (fidivr)

```
fidivr{l}
```

Example

Divide the integer contents of register ECX, with an offset of 2, by stack element 0 and return the result to stack element 0.

```
fidivr 2(%ecx)
```

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:

TABLE 2–11 Floating-point Opcodes

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)

TABLE 2–11 Floating-point Opcodes *(continued)*

fsub %st,%st(n)	**fsubr %st, %st(n)**
fdiv	fdivr
fdivr %st, %st(n)	fdvir %st, %st(n)
fdivrp %st, %st(n)	fdivp %st, %st(n)
fdivr	fdiv

Miscellaneous Arithmetic Operations

Square Root (fsqrt)

```
fsqrt
```

Example

Replace stack element 0 with the square root of its value.

```
fsqrt
```

Scale (fscale)

fscale

Example

Add the integer value in stack element 1 to the exponent of stack element 0 (multiplication and division by powers of 2).

```
fscale
```

Partial Remainder (fprem)

```
fprem
```

Example

Divide stack element 0 by stack element 1 and return the (partial) remainder to stack element 0.

```
fprem
```

Round to Integer (frndint)

```
frndint
```

Example

Round the value in stack element 0 to an integer according to the FPU control word RC field.

```
frndint
```

Extract Exponent and Significand (fxtract)

```
fxtract
```

Example

Separate stack element 0 into its exponent and significand and return the exponent to stack element 0, then push the significand onto the FPU stack.

```
fxtract
```

Absolute Value (fabs)

```
fabs
```

Example

Replace stack element 0 with its absolute value.

```
fabs
```

Change Sign (fchs)

```
fchs
```

Example

Replace the sign of stack element 0 with the opposite sign.

```
fchs
```

Comparison Instructions

Compare Real (fcom)

```
fcom{ls}
```

Example

Compare stack element 0 with stack element 7. Condition codes contain the result: No compare=111, st 0 greater than st 7=000, st 0 less than st 7=001, equal compare=100.

```
fcom %st(7)
```

Compare Real and Pop (fcomp)

```
fcomp{ls}
```

Example

Compare stack element 0 with stack element 7. Condition codes contain the result: No compare=111, st 0 greater than st 7=000, st 0 less than st 7=001, equal compare=100, then pop the stack.

```
fcomp %st(7)
```

Compare Real and Pop Twice (fcompp)

```
fcompp
```

Example

Compare stack element 0 with stack element 1. Condition codes contain the result: No compare=111, st 0 greater than st 7=000, st 0 less than st 7=001, equal compare=100, then pop the stack twice.

```
fcompp
```

Integer Compare (ficom)

```
ficom{l}
```

Example

Integer compare stack element 0 with the contents of register ECX (with an offset of 2). Condition codes contain the result: No compare=111, st 0 greater than st 7=000, st 0 less than st 7=001, equal compare=100,

```
ficom 2(%ecx)
```

Integer Compare and Pop (ficomp)

```
ficomp{l}
```

Example

Integer compare stack element 0 with the contents of register ECX (with an offset of 2). Condition codes contain the result: No compare=111, st 0 greater than st 7=000, st 0 less than st 7=001, equal compare=100, then pop the stack.

```
ficomp 2(%ecx)
```

Test (ftst)

```
ftst
```

Example

Compare stack element 0 with the value 0.0. Condition codes contain the result: No compare=111, st 0 greater than st 7=000, st 0 less than st 7=001, equal compare=100,

```
ftst
```

Examine (fxam)

```
fxam
```

Example

Report the type of object in stack element 0. FPU flags C3, C2, and C0 return the type:

`Unsupported`	`000`
`NaN`	`001`
`Normal`	`010`
`Infinity`	`011`
`Zero`	`100`
`Empty`	`101`
`Denormal`	`110`

```
fxam
```

Transcendental Instructions

Partial Tangent (fptan)

```
fptan
```

Example

Replace stack element 0 with its tangent and push a value of 1 onto the FPU stack.

```
fptan
```

Partial Arctangent (fpatan)

```
fpatan
```

Example

Divide stack element 1 by stack element 0, compute the arctangent and return the result in radians to stack element 1, then pop the stack.

```
fpatan
```

2^x - 1 (f2xm1)

```
f2xm1
```

Example

Replace the contents of stack element 0 (st) with the value of (2^{st}-1).

```
f2xm1
```

Y * log2 X (fyl2x)

```
fyl2x
```

Example

Compute the logarithm (base-2) of stack element 0 and multiply the result by stack element 1 and return the result to stack element 1, then pop the stack.

```
fy12x
```

Y * log $_2$ (X+1) (fyl2xp1)

```
fyl2xp1
```

Example

Compute the logarithm (base-2) of stack element 0 plus 1.0 and multiply the result by stack element 1 and return the result to stack element 1, then pop the stack.

```
fy12xpl
```

Constant Instructions

Load log$_2$E (fldl2e)

```
fldl2e
```

Example

Push $\log_2 e$ onto the FPU stack

```
fldl2e
```

Load $\log_2 10$ (fldl2t)

```
fldl2t
```

Example

Push $\log_2 10$ onto the FPU stack.

```
fldl2t
```

Load $\log_{10} 2$ (fldlg2)

```
fldlg2
```

Example

Push $\log_{10} 2$ onto the FPU stack.

```
fldlg2
```

Load $\log_e 2$ (fldln2)

```
fldln2
```

Example

Push $\log_2 e$ onto the FPU stack.

```
fldln2
```

Load pi (fldpi)

```
fldpi
```

Example

Push p onto the FPU stack.

```
fldpi
```

Load + 0 (fldz)

```
fldz
```

Example

Push +0.0 onto the FPU stack.

```
fldz
```

Processor Control Instructions

Initialize Processor (finit, fnint)

```
finit  fninit
```

Example

```
finit
```

No Operation (fnop)

```
fnop
```

Example

```
fnop
```

Save State (fsave, fnsave)

```
fsave
fnsave
```

Example

```
fsave 2(%ecx)
```

Store Control Word (fstcw, fnstcw)

```
fstcw
fnstcw
```

Example

```
fstcw 2(%ecx)
```

Store Environment (fstenv, fnstenv)

```
fstenv
  fnstenv
```

Example

```
fstenv 2(%ecx)
```

Store Status Word (fstsw, fnstsw)

```
fstsw
  fnstsw
```

Example

```
fstsw %ax
```

Restore State (frstor)

```
frstor
```

Example

```
frstor 2(%ecx)
```

CPU Wait (fwait, wait)

```
fwait
wait
```

Example

```
fwait
```

Clear Exceptions (fclex, fnclex)

```
fclex
  fnclex
```

Example

```
fclex
```

Decrement Stack Pointer (fdecstp)

```
fdecstp
```

Example

```
fdecstp
```

Free Registers (ffree)

```
ffree
```

Example

```
ffree %st(7)
```

Increment Stack Pointer (fincstp)

```
fincstp
```

Example

```
fincstp
```

CHAPTER 3

Assembler Output

This chapter is an overview of ELF *(Executable and Linking Format)* 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.

This chapter is organized as follows:

- “Introduction” on page 121
- “Object Files in Executable and Linking Format (ELF)” on page 122

Introduction

The main output produced by assembling an input assembly language source file is the translation of that file into an object file in (ELF). ELF files produced by the assembler are relocatable files that hold code and/or data. They are input files for the linker. The linker 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 can also produce ancillary output incidental to the translation process. For example, if the assembler is invoked with the `-V` option, it can 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, described in Chapter 2“, it is helpful to have some working knowledge of ELF, at least for relocatable files.

Object Files in Executable 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 a structure of fixed size and format. The fields, or members, of the structure describe the nature, organization and contents of the rest of the file. The ELF header has a field that specifies the location within the file where the section header table begins.

The section header table is an array of section headers that are structures of fixed size and format. The section headers are the elements of the array, or the entries in the table. The section header table has one entry for each section in the ELF file. However, the table can 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 can 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 ELF header, the section header table, and all the sections can omit some of the bytes. Bytes of a relocatable file that are not in the ELF header, or in the section header table, or in any of the sections constitute the 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 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 that can be used to decode and interpret the file's contents.

e_type
: Identifies the object file type. A value of 1, that has the name ET_REL, specifies a relocatable file. Table 3–1 describes all the object file types.

e_machine
: Specifies the required architecture for an individual file. A value of 3, that 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 where 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 3–1 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. A section header 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 3-3 describes the section types.

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

`sh_addr` Address where the first byte resides if the section appears in the memory image of a process; a value of 0 indicates the section does 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` can be nonzero; 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 3-3.

sh_info

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

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 has 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 that 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 3–2 Section Attribute Flags

Flag	Default Value	Description
SHF_WRITE	0x1	Contains data that is writable during process execution.
SHF_ALLOC	0x2	Occupies memory during process execution. This attribute is *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 3–3 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`
`SHT_SYMTAB`	2	Is a complete symbol table, usually for link editing. This table can also be used for dynamic linking; however, it can 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 can have multiple string table sections.	0	`SHN_UNDEF`
`SHT_RELA`	4	Contains relocation entries with explicit addends. A file can have multiple relocation sections.	The section header index of the section to where 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 3–3 Section Types *(continued)*

Name	Value	Description	Interpretation by	
			sh_info	**sh_link**
SHT_NOBITS	8	Contains information defined by the program, and in a format and with a meaning determined 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 can have multiple relocation sections.	The section header index of the section to where 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 can 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 does 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 the file information that 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 one section header entry in the section header table to describe the section. However, the section header table can have section header entries that correspond to no section in the file.
2. Each section occupies one contiguous sequence of bytes within a file. The section can be empty (even so, its section header entry in the section header table can have a nonzero 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 can have inactive space. Inactive space is the set of all bytes in the file that are not part of the ELF header, the section header table, 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 where there are entries in the file 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 nonuser 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 3–4 lists the names of the predefined user sections and briefly describes each.

TABLE 3–4 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 3–5 shows the predefined sections that are not user sections, because assembly language programmers cannot manipulate them by issuing section control directives in the source file.

TABLE 3–5 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.

TABLE 3–5 Predefined Non-user Sections *(continued)*

Section Name	Description
".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 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 can be an address, or it can 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 3–6 and Table 3–7 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 continue to point to the same location because the value of the symbol changes as well.

TABLE 3–6 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	loproc	Values reserved for processor-specific semantics.
15	hiproc	

TABLE 3–7 Binding Attributes

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

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

Attribute Expression

Attribute expressions are used to generate entries of relocation table. The form of the attribute expression is:

```
symbol-name@attribute
```

Attributes begin with the @ sign followed by a reserved identifier. Table 3–8 describes the identifier attributes used in the x86 assembler:.

TABLE 3–8 Identifier Attributes

Attribute	Description	Relocation Table Entry Generated
@GOT	Address of the Global Offset Table (GOT) entry for the identifier	R_386_GOT32
@GOTOFF	The difference between a symbol's value and the Global Offset Table (GOT).	R_386_GOTOFF
@PLT	Address of a function's Procedure Linkage Table ()LT) entry	R_386_PLT32

Note - To obtain correct position-independent code, conform to the Intel 386 Architecture and the System V ABI.

The special symbol _GLOBAL_OFFSET_TABLE_ can be used in the x86 Assembler for building of position-independent code.

APPENDIX **A**

Using the Assembler Command Line

This chapter describes how to invoke the assembler and use the command-line options.

This chapter is organized as follows:

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.)

Assembler Command Line Options

`-b`

Enable Sun SourceBrowser.

`-D`***name***

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.

`-K {pic.PIC}`

Generate position-independent code.

`-L`

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

`-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.x 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, that 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.

-S [[aAbBcClL]]

Produce a disassembly of the emitted code to the standard output. Adding the character to the S option:

a - disassembling with address

b - disassembling with ’.bof’

c - disassembling with comments (default)

l - disassembling with line numbers

Capital letters turn switch off for corresponding option

-T

This is a migration option for 4.1 assembly files to be assembled on 5.0 systems. With this option, the symbol names in 4.1 assembly files will be interpreted as 5.0 symbol names.

-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.

-xF

Generates additional information for performance analysis of the executable using WorkShop (SPARC-works) analyzer. If the input file does not contain any stabs (debugging directives), then the assembler will generate some default stabs which are needed by the SPARCworks analyzer.

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.