Assembler Directives

This chapter describes assembler directives (also known as pseudo operations, or pseudo-ops), which allow control over the actions of the assembler. For organizational purposes, the directives are grouped here into the following functional categories:

- Directives for designating the current section
- Built-in directives for designating the current section
- Directives for moving the location counter
- Directives for generating data
- Directives for dealing with symbols
- Miscellaneous directives
- Processor-specific directives

Directives for Designating the Current Section

The assembler in NEXTSTEP Release 3.3 and later supports designation of arbitrary sections with the **.section** and **.zerofill** directives (descriptions apear below). Only those sections specified by a directive in the assembly file appear in the resulting object file (including implicit **.text** directives—see "Built-in Directives for Designating the Current Section"). Sections appear in the object file in the order their directives first appear in the assembly file. When object files are linked by the link editor, the output objects have their sections in the order the sections first appear in the object files that are linked. See the **ld**(1) UNIX man page for more details.

Associated with each section in each segment is an implicit location counter which begins at zero and is incremented by 1 for each byte assembled into the section. There is no way to explicitly reference a particular location counter, but the directives described here can be used to "activate" the location counter for a section, making it the *current* location counter. As a result, the assembler begins assembling into the section associated with that location counter.

Note: If the **-n** command line option isn't used, the (__TEXT,__text) section is used by default at the beginning of each file being assembled, just as if each file began with the **.text** directive.

.section

SYNOPSIS:

.section *segname* **,** *sectname* [[[**,** *type*] **,** *attribute*] **,** *sizeof_stub*]

The **.section** directive causes the assembler to begin assembling into the section given by *segname* and *sectname*. A section created with this directive contains initialized data or instructions and is referred to as a content section. *type* and *attribute* may be specified as described below under "Section Types and Attributes." If *type* is **symbol_stubs**, then the *sizeof_stub* field must be given as the size in bytes of the symbol stubs contained in the section.

.zerofill

SYNOPSIS:

.zerofill *segname* **,** *sectname* [**,** *symbolname* **,** *size* [**,** *align_expression*]]

The **.zerofill** directive causes *symbolname* to be created as uninitialized data in the section given by *segname* and *sectname*, with a size in bytes given by *size*. A power of 2 between 0 and 15 may be given for *align_expression* to indicate what alignment should be forced on *symbolname*, which will then

be placed on the next expression boundary having the given alignment. See the description of the **.align** built-in directive for more information.

Section Types and Attributes

A content section has a type, which informs the link editor about special processing needed for the items in that section. The most common form of special processing is for sections containing literals (strings, constants, and so on) where only one copy of the literal is needed in the output file and the same literal can be used by all references in the input files.

A section's attributes record supplemental information about the section that the link editor may use in processing that section. For example, the reloc_at_launch attribute indicates that a section should be relocated immediately when a program is launched.

A section's type and attribute are recorded in a Mach-O file as the flags field in the section header, using constants defined in the header file mach-o/loader.h. The following paragraphs describe the various types and attributes by the names used to identify them in a .section directive. The name of the related constant is also given in parentheses following the identifier.

Type Identifiers

regular (S_REGULAR)

> A **regular** section may contain any kind of data and gets no special processing from the link editor. This is the default section type. Examples of **regular** sections include program instructions or initialized data.

cstring_literals (S_CSTRING_LITERALS)

> A **cstring_literals** section contains null-terminated literal C language character strings. The link editor places only one copy of each literal into the output file's section and relocates references to different copies of the same literal to the one copy in the output file. There can be no relocation entries for a section of this type, and all references to literals in this section must be inside the address range for the specific literal being referenced. The last byte in a section of this type must be a null byte, and the strings can't contain null bytes in their bodies. An example of a **cstring_literals** section is one for the literal strings that appear in the body of an ANSI C function where the compiler chooses to make such strings read-only.

4byte_literals (S_4BYTE_LITERALS)

> A **4byte_literals** section contains 4-byte literal constants. The link editor places only one copy of each literal into the output file's section and relocates references to different copies of the same literal to the one copy in the output file. There can be no relocation entries for a section of this type, and all references to literals in this section must be inside the address range for the specific literal being referenced. An example of a **4byte_literals** section is one in which single-precision floating-point constants are stored for a RISC machine (these would normally be stored as immediates in CISC machine code).

8byte_literals (S_8BYTE_LITERALS)

> An **8byte_literals** section contains 8-byte literal constants. The link editor places only one copy of each literal into the output file's section and relocates references to different copies of the same literal to the one copy in the output file. There can be no relocation entries for a section of this type, and all

references to literals in this section must be inside the address range for the specific literal being referenced. An example of a **8byte_literals** section is one in which double-precision floating-point constants are stored for a RISC machine (these would normally be stored as immediates in CISC machine code).

literal_pointers (S_LITERAL_POINTERS)

A **literal_pointers** section contains 4-byte pointers to literals in a literal section. The link editor places only one copy of a pointer into the output file's section for each pointer to a literal with the same contents. The link editor also relocates references to each literal pointer to the one copy in the output file. There must be exactly one relocation entry for each literal pointer in this section, and all references to literals in this section must be inside the address range for the specific literal being referenced. The relocation entries can be external relocation entries referring to undefined symbols if those symbols identify literals in another object file. An example of a **literal_pointers** section is one containing selector references generated by the Objective C compiler.

symbol_stubs (S_SYMBOL_STUBS)

A **symbol_stubs** section contains symbol stubs, which are sequences of machine instructions (all the same size) used for lazily binding undefined function calls at run time. If a call to an undefined function is made, the compiler outputs a call to a symbol stub instead, and tags the stub with an indirect symbol that indicates what symbol the stub is for. On transfer to a symbol stub, a program executes instructions that eventually reach the code for the indirect symbol associated with that stub. Here's a sample of assemly code based on a function **func()** containing only a call to the undefined function **foo()**:

```
          .text
          .align 4, 0x90
_func:
          call        _foo_stub
          ret

          .symbol_stub
_foo_stub:
          .indirect_symbol _foo
          ljmp     _foo_lazy_ptr     # the symbol stub
_foo_stub_1:
          pushl   $_foo_lazy_ptr
          jmp     dyld_stub_binding_helper

          .lazy_symbol_pointer
_foo_lazy_ptr:                       # the symbol pointer
          .indirect_symbol _foo
          .long _foo_stub_1             # to be replaced by _foo's address
```

In the assembly code, **_func** calls **_foo_stub**, which is responsible for finding the definition of the function **foo()**. **_foo_stub** jumps to the contents of **_foo_lazy_ptr**, initially causing the code at **_foo_stub_1** to be executed. This value is initially the address for **_foo_stub1**, which calls the **dyld_stub_binding_helper()** function to overwrite the contents of **_foo_lazy_ptr** with the address of the real function, **_foo**. This way, jumps through **_foo_lazy_ptr** will immediately execute **foo()**'s code.

The indirect symbol entries for **_foo** provide information to the static and dynamic linkers for binding the symbol stub. Each symbol stub and lazy pointer entry must have exactly one such indirect

symbol, associated with the first address in the stub or pointer entry. See the description of the **.indirect_symbol** directive for more information.

The static link editor places only one copy of each stub into the output file's section for a particular indirect symbol, and relocates all references to the stubs with the same indirect symbol to the stub in the output file. Further, the static link editor eliminates a stub if a definition of the indirect symbol for that stub is present in the output file and that output file isn't a dynamically linked shared library file. The stub can refer only to itself, one lazy symbol pointer (referring to the same indirect symbol as the stub), and the **dyld_stub_binding_helper()** function. No global symbols can be defined in this type of section.

lazy_symbol_pointers (S_LAZY_SYMBOL_POINTERS)

A **lazy_symbol_pointers** section contains 4-byte symbol pointers that will eventually contain the value of the indirect symbol associated with the pointer. These pointers are used by symbol stubs to lazily bind undefined function calls at run time. A lazy symbol pointer initially contains an address in the symbol stub of instructions that cause the symbol pointer to be bound to the function definition (in the example above, the lazy pointer **_foo_lazy_ptr** initially contains the address for **_foo_stub_1** but gets overwritten with the address for **_foo**). The dynamic link editor binds the indirect symbol associated with the lazy symbol pointer by overwriting it with the value of the symbol.

The static link editor only places a copy of a lazy pointer in the output file if the corresponding symbol stub is in the output file. Only the corresponding symbol stub can make a reference to a lazy symbol pointer, and no global symbols can be defined in this type of section. There must be one indirect symbol assocated with each lazy symbol pointer. An example of a **lazy_symbol_pointers** section is one in which the compiler has generated calls to undefined functions, each of which can be bound lazily at the time of the first call to the function.

non_lazy_symbol_pointers (S_NON_LAZY_SYMBOL_POINTERS)

A **non_lazy_symbol_pointers** section contains 4-byte symbol pointers that will contain the value of the indirect symbol associated with a pointer that may be set at any time before any code makes a reference to it. These pointers are used by the code to reference undefined symbols. Initially these pointers have no interesting value, but will get overwritten by the dynamic link editor with the value of the symbol for the associated indirect symbol before any code can make a reference to it.

The static link editor places only one copy of each non-lazy pointer for its indirect symbol into the output file and relocates all references to the pointer with the same indirect symbol to the pointer in the output file. The static link editor futher can fill in the pointer with the value of the symbol if a definition of the indirect symbol for that pointer is present in the output file. No global symbols can be defined in this type of section. There must be one indirect symbol assocated with each non-lazy symbol pointer. An example of a **non_lazy_symbol_pointers** section is one in which the compiler has generated code to indirectly reference undefined symbols to be bound at run time—this preserves the sharing of the machine instructions by allowing the dynamic link editor to update references without writing on the instructions.

Here's an example of assembly code referencing an element in the undefined structure. The corresponding 'C' code would be:

```
struct s {
            int member1, member2;
};
extern struct s bar;
int func()
{
            return(bar.member2);
```

```
}
```

The i386 assembly code might look like this:

```
          .text
            .align 4, 0x90
.globl _func
_func:
          movl _bar_non_lazy_ptr,%eax
          movl 4(%eax),%eax
          ret

        .non_lazy_symbol_pointer
_bar_non_lazy_ptr:
        .indirect_symbol _bar
        .long 0
```

mod_init_funcs (S_MOD_INIT_FUNC_POINTERS)

A **mod_init_funcs** section contains 4-byte pointers to functions that are to be called just after the module containing the pointer is bound into the program by the dynamic link editor. The static link editor does no special processing for this section type except for disallowing section ordering. This is done to maintain the order the functions will be called (which is the order their pointers appear in the original module). There must be exactly one relocation entry for each pointer in this section. An example of a **mod_init_funcs** section is one in which the compiler has generated code to call C++ constructors for modules that get dynamicly bound at run time.

Attribute Identifiers

none (0)

No attributes for this section. This is the default section attribute.

pure_instructions (S_ATTR_PURE_INSTRUCTIONS)

The **pure_instructions** attribute means that this section contains nothing but machine instructions. This attribute would be used for the (__TEXT,__text) section of NeXT compilers and sections which have a section type of **symbol_stubs**.

reloc_at_launch (S_ATTR_RELOC_AT_LAUNCH)

The **reloc_at_launch** attribute means that this section is to be relocated by the dynamic linker when the program is first run or first loaded into memory, regardless of whether a module is needed to bind undefined symbols. Sections are normally relocated only when first referenced. This attribute would be used for the (__OBJC,__message_refs) section of NeXT Objective C compiler to allow the Objective C run-time system to initialize images being loaded into a program.

Built-in Directives for Designating the Current Section

The directives described here are simply built-in equivalents for **.section** directives with specific

arguments.

Designating Sections in the __TEXT Segment

The directives listed below cause the assembler to begin assembling into the indicated section of the __TEXT segment. Note that the underscore before __TEXT, __text, and the rest of the segment names is actually two underscore characters.

Directive	Section
.text	(__TEXT,__text)
.const	(__TEXT,__const)
.static_const	(__TEXT,__static_const)
.cstring	(__TEXT,__cstring)
.literal4	(__TEXT,__literal4)
.literal8	(__TEXT,__literal8)
.constructor	(__TEXT,__constructor)
.destructor	(__TEXT,__destructor)
.fvmlib_init0	(__TEXT,__fvmlib_init0)
.fvmlib_init1	(__TEXT,__fvmlib_init1)
.symbol_stub	(__TEXT,__symbol_stub)
.mod_init_func	(__TEXT,__mod_init_func)

The following paragraphs describe the sections in the __TEXT segment and the types of information that should be assembled into each of them:

(__TEXT,__text)
This is equivalent to .section __TEXT,__text,regular,pure_instructions

The compiler only places machine instructions in the (__TEXT,__text) section (no read-only data, jump tables or anything else). With this the entire (__TEXT,__text) section is pure instructions and tools that operate on object files can take advantage of this and can locate the instructions of the program and not get confused with data that could have been mixed in. To make this work all run-time support code linked into the program must also obey this rule (all NeXT library code follows this rule).

(__TEXT,__const)
This is equivalent to **.section __TEXT,__const**

The compiler places all data declared const in this section and all jump tables it generates for switch statements.

(__TEXT,__static_const)
This is equivalent to .**section __TEXT,__static_const**

This is not currently used by the compiler. It was added to the assembler so that the compiler may separate global and static const data into separate sections if it wished to.

(__TEXT,__cstring)
This is equivalent to **.section __TEXT,__cstring**, **cstring_literals**

This section is marked with the section type S_LITERAL_CSTRING, which the link editor recognizes. The link editor merges the like literal C strings in all the input object files to one unique C string in the output file. Therefore this section must only contain C strings (a C string in a sequence of bytes that

ends in a null byte, '\0', and does not contain any other null bytes except its terminator). The compiler places literal C strings found in the code that are not initializers and do not contain any imbedded nulls in this section.

(__TEXT,__literal4)
This is equivalent to **.section __TEXT,__literal4**,**4byte_literals**

This section is marked with the section type S_4BYTE_LITERALS, which the link editor recognizes. The link editor then can merge the like 4 byte literals in all the input object files to one unique 4 byte literal in the output file. Therefore this section must only contain 4 byte literals. This is typically intended for single precision floating-point constants and the compiler uses this section for that purpose. On some machines it is more efficient to place these constants in line as immediates as part of the instruction (this is what is done on NeXT 68k machines when the optimizer is turned on).

(__TEXT,__literal8)
This is equivalent to **.section __TEXT,__literal8**,**8byte_literals**

This section is marked with the section type S_8BYTE_LITERALS, which the link editor recognizes. The link editor then can merge the like 8 byte literals in all the input object files to one unique 8 byte literal in the output file. Therefore this section must only contain 8 byte literals. This is typically intended for double precision floating-point constants and the compiler uses this section for that purpose. On some machines it is more efficient to place these constants in line as immediates as part of the instruction (this is what is done on NeXT 68k machines when the optimizer is turned on).

(__TEXT,__constructor)
This is equivalent to **.section __TEXT,__constructor**
(__TEXT,__destructor)
This is equivalent to **.section __TEXT,__destructor**

These sections are used by the C++ run-time system, and are reserved exclusively for the C++ compiler.

(__TEXT,__fvmlib_init0)
This is equivalent to **.section __TEXT,__fvmlib_init0**
(__TEXT,__fvmlib_init1)
This is equivalent **to .section __TEXT,__fvmlib_init1**

These two sections are used by the fixed virtual memory shared library initialization. The compiler doesn't place anything in these sections, as they are reserved exclusively for the shared library mechanism.

(__TEXT,__symbol_stub)
This is equivalent to **.section __TEXT,__symbol_stub**, **symbol_stubs**, **pure_instructions**,**NBYTES**

This section is of type **symbol_stubs** and has the attribute **pure_instructions**. The compiler places symbol stubs in this section for undefined functions that are called in the module. This is the standard symbol stub section for non position-independent code. The value **NBYTES** is dependent on the target architecture. The standard symbol stub for the m68k is 20 bytes and has an alignment of 2 bytes (**.align 1** or **.even**) . For example, a stub for the symbol _foo would be (using a lazy symbol pointer Lfoo$stub_binder):

```
            .symbol_stub
Lfoo$stub:
            .indirect_symbol _foo
            movel       L_foo$lazy_ptr,a0
```

```
            jmp         a0@
Lfoo$stub_binder:
            movel        #L_foo$lazy_ptr,sp@-
            bra          dyld_stub_binding_helper

            .lazy_symbol_pointer
L_foo$lazy_ptr:
            .indirect_symbol _foo
            .long    Lfoo$stub_binder
```

The standard symbol stub for the i386 is 16 bytes and has an alignment of 1 byte (.align 0). For example a stub for the symbol _foo would be (using a lazy symbol pointer Lfoo$stub_binder):

```
            .symbol_stub
Lfoo$stub:
            .indirect_symbol _foo
            ljmp     L_foo$lazy_ptr
Lfoo$stub_binder:
            pushl   $L_foo$lazy_ptr
            jmp     dyld_stub_binding_helper

            .lazy_symbol_pointer
L_foo$lazy_ptr:
            .indirect_symbol _foo
            .long Lfoo$stub_binder
```

(__TEXT, __picsymbol_stub)
This is equivalent to **.section __TEXT**, **__picsymbol_stub**, **symbol_stubs**, **pure_instructions**, **NBYTES**

This section is of type **symbol_stubs** and has the attribute **pure_instructions**. The compiler places symbol stubs in this section for undefined functions that are called in the module. This is the standard symbol stub section for position-independent code. The value of **NBYTES** is dependent on the target architecture.

The standard position-independent symbol stub for the m68k is 24 bytes and has an alignment of 2 bytes (**.align 1** or **.even**). For example a stub for the symbol _foo would be (using a lazy symbol pointer Lfoo$stub_binder):

```
            .picsymbol_stub
Lfoo$stub:
            .indirect_symbol _foo
            movel       pc@(L_foo$lazy_ptr-.),a0
            jmp         a0@
Lfoo$stub_binder:
            pea         pc@(L_foo$lazy_ptr-.)
            bra         dyld_stub_binding_helper

            .lazy_symbol_pointer
L_foo$lazy_ptr:
            .indirect_symbol _foo
            .long Lfoo$stub_binder
```

The standard position-independent symbol stub for the i386 is 26 bytes and has an alignment of 1 byte (**.align 0**). For example a stub for the symbol _foo would be (using a lazy symbol pointer Lfoo$stub_binder):

```
              .picsymbol_stub
Lfoo$stub:
            .indirect_symbol _foo
            call        L1foo$stub
L1foo$stub:
            popl        %eax
            movl        L_foo$lazy_ptr-L1foo$stub(%eax),%ebx
            jmp         %ebx
Lfoo$stub_binder:
            lea         L_foo$lazy_ptr-L1foo$stub(%eax),%eax
            pushl       %eax
            jmp     dyld_stub_binding_helper

            .lazy_symbol_pointer
L_foo$lazy_ptr:
            .indirect_symbol _foo
            .long Lfoo$stub_binder
```

(__TEXT,__mod_init_func)
This is equivalent to **.section __TEXT**, **__mod_init_func**, **mod_init_funcs**

This section is of type **mod_init_funcs** and has no attributes. The C++ compiler places a pointer to a function in this section for each function it creates to call the constructors (if the module has them).

Designating Sections in the __DATA Segment

These directives cause the assembler to begin assembling into the indicated section of the __DATA segment:

Directive	**Section**
.data	(__DATA,__data)
.static_data	(__DATA,__static_data)
.non_lazy_symbol_pointer	(__DATA,__nl_symbol_pointer)
.lazy_symbol_pointer	(__DATA,__la_symbol_pointer)
.dyld	(__DATA,__dyld)

The following paragraphs describe the sections in the __DATA segment and the types of information that should be assembled into each of them:

(__DATA,__data)
This is equivalent to **.section __DATA, __data**

The compiler places all non-const initialized data (even initialized to zero) in this section.

(__DATA,__static_data)

This is equivalent to **.section __DATA**, **__static_data**

This is not currently used by the compiler. It was added to the assembler so that the compiler could separate global and static data symbol into separate sections if it wished to.

(__DATA,__nl_symbol_ptr)
This is equivalent to **.section __DATA**, **__nl_symbol_ptr**,**non_lazy_symbol_pointers**

This section is of type **non_lazy_symbol_pointers** and has no attributes. The compiler places a non-lazy symbol pointer in this section for each undefined symbol referenced by the module (except for function calls).

(__DATA,__la_symbol_ptr)
This is equivalent to **.section __DATA, __la_symbol_ptr,lazy_symbol_pointers**

This section is of type **lazy_symbol_pointers** and has no attributes. The compiler places a lazy symbol pointer in this section for each symbol stub it creates for undefined functions that are called in the module. (See __TEXT, __symbol_stub for examples.)

(__DATA,__dyld)
This is equivalent to **.section __DATA**, **__dyld**,**regular**

This section is of type **regular** and has no attributes. This section is used by the dynamic link editor. The compiler doesn't place anything in this section, as it is reserved exclusively for the dynamic link editor.

Designating Sections in the __OBJC Segment

These directives cause the assembler to begin assembling into the indicated section of the __OBJC segment:

Directive	**Section**
.objc_class	(__OBJC,__class)
.objc_meta_class	(__OBJC,__meta_class)
.objc_cat_cls_meth	(__OBJC,__cat_cls_meth)
.objc_cat_inst_meth	(__OBJC,__cat_inst_meth)
.objc_protocol	(__OBJC,__protocol)
.objc_string_object	(__OBJC,__string_object)
.objc_cls_meth	(__OBJC,__cls_meth)
.objc_inst_meth	(__OBJC,__inst_meth)
.objc_cls_refs	(__OBJC,__cls_refs)
.objc_message_refs	(__OBJC,__message_refs)
.objc_symbols	(__OBJC,__symbols)
.objc_category	(__OBJC,__category)
.objc_class_vars	(__OBJC,__class_vars)
.objc_instance_vars	(__OBJC,__instance_vars)
.objc_module_info	(__OBJC,__module_info)
.objc_class_names	(__OBJC,__class_names)
.objc_meth_var_names	(__OBJC,__meth_var_names)
.objc_meth_var_types	(__OBJC,__meth_var_types)
.objc_selector_strs	(__OBJC,__selector_strs)

All sections in the __OBJC segment, including old sections that are no longer used and future sections that may be added, are exclusively reserved for the Objective C compiler's use.

Directives for Moving the Location Counter

This section describes directives that advance the location counter to a location higher in memory. They have the additional effect of setting the intervening memory to some value.

.align

SYNOPSIS:

.align expression [**,** fill_expression]

The **.align** directive advances the location counter to the next expression boundary, if it isn't currently on such a boundary. *expression* is a power of 2 between 0 and 15 (not the result of the power of 2; for example, the argument of .**align 3** means 2 to the third). The fill expression, if specified, must be absolute. The space between the current value of the location counter and the desired value is filled with the low-order byte of the fill expression (or with zeros, if *fill_expression* isn't specified).

Note: The assembler enforces no alignment for any bytes created in the object file (data or machine instructions). You must supply the desired alignment before any directive or instruction.

EXAMPLE:

```
        .align 3
one:    .double 0r1.0
```

.org

SYNOPSIS:

.org expression [**,** fill_expression]

The **.org** directive sets the location counter to *expression*, which must be a currently known absolute expression. This directive can only move the location counter up in address. The fill expression, if specified, must be absolute. The space between the current value of the location counter and the desired value is filled with the low-order byte of the fill expression (or with zeros, if *fill_expression* isn't specified).

Note: If the output file is later link-edited, the **.org** directive isn't preserved.

EXAMPLE:

```
.org 0x100,0xff
```

Directives for Generating Data

The directives described in this section all generate data (unless specified otherwise, the data goes into the current section). In some respects they are similar to the directives in the previous section, "Directives for Moving the Location Counter"—they do have the effect of moving the location counter—but this isn't their primary purpose.

.ascii and .asciz

SYNOPSIS:
.ascii ["*string*"] [, "*string*"] ...
.asciz ["*string*"] [, "*string*"] ...

These two directives translate character strings into their ASCII equivalents for use in the source program. Each directive takes zero or more comma-separated, quoted strings. Each string can contain any character or escape sequence that can appear in a character string; the newline character cannot appear, but it can be represented by the escape sequence **\012** or **\n**.

- The .ascii directive generates a sequence of ASCII characters.

- The .asciz directive is similar, except that it automatically terminates the sequence of ASCII characters with the null character, \0 (necessary when generating strings usable by C programs).

If no strings are specified, the directive is ignored.

EXAMPLE:
```
.ascii "Can't open the DSP.\0"
.asciz "%s has changes.\tSave them?"
```

.byte, .short, and .long

SYNOPSIS:
.byte [*expression*] [**,** *expression*] ...
.short [*expression*] [**,** *expression*] ...
.long [*expression*] [**,** *expression*] ...

These directives reserve storage locations in the current section and initialize them with specified values. Each directive takes zero or more comma-separated absolute expressions and generates a sequence of bytes for each expression. The expressions are truncated to the size generated by the directive:

- .byte generates one byte per expression
- .short generates two bytes per expression
- .long generates four bytes per expression

EXAMPLE:
```
.byte  74,0112,0x4A,0x4a,'J           | all the same byte
.short 64206,0175316,0xface           | all the same short
.long  -1234,037777775456,0xfffffb2e  | all the same long
```

.single and .double

SYNOSIS:
.single [*number*] [**,** *number*] ...
.double [*number*] [**,** *number*] ...

These two directives reserve storage locations in the current section and initialize them with specified values. Each directive takes zero or more comma-separated decimal floating-point numbers:

- .single takes IEEE single-precision floating point numbers; it reserves four bytes for each number, and initializes them to the value of the corresponding number

- .double takes IEEE double-precision floating point numbers; it reserves eight bytes for each number, and initializes them to the value of the corresponding number

EXAMPLE:

```
.single 3.3333333333333310000e-01
.double 0.0000000000000000000e+00
.single +Infinity
.double -Infinity
.single NaN
```

.fill

SYNOPSIS:

.fill *repeat_expression* **,** *fill_size* **,** *fill_expression*

The **.fill** directive advances the location counter by *repeat_expression* times *fill_size* bytes.

- ***fill_size*** is in bytes, and must have the value 1, 2, or 4
- ***repeat_expression*** must be an absolute expression greater than zero
- ***fill_expression*** may be any absolute expression (it gets truncated to the fill size)

EXAMPLE:

```
.fill 69,4,0xfeadface   | put out 69 0xfeadface's
```

.space

SYNOPSIS:

.space num_bytes [**,** fill_expression]

The **.space** directive advances the location counter by *num_bytes*, where *num_bytes* is an absolute expression greater than zero. The fill expression, if specified, must be absolute. The space between the current value of the location counter and the desired value is filled with the low-order byte of the fill expression (or with zeros, if *fill_expression* isn't specified).

EXAMPLE:

```
ten_ones:
          .space 10,1
```

.comm

SYNOPSIS:

.**comm** *name*, *size*

The .comm directive creates a common symbol named name of size bytes. If the symbol isn't defined elsewhere, its type is "common."

The link editor allocates storage for common symbols that aren't otherwise defined. Enough space is left after the symbol to hold the maximum size (in bytes) seen for each symbol in the (__DATA,__common) section.

The link editor will align each such symbol (based on its size aligned to the next greater power of two) to the maximum alignment of the (__DATA,__common) section. For information about how to change the maximum alignment, see the description of -sectalign in the ld(1) UNIX manual page.

EXAMPLE:

```
.comm _global_uninitialized,4
```

.lcomm

SYNOPSIS:

.lcomm *name*, *size* [, *align*]

The **.lcomm** directive creates a symbol named *name* of *size* bytes in the (__DATA,__bss) section. It will contain zeros at execution. The name isn't declared as global, and hence will be unknown outside the object module.

The optional *align* expression, if specified, causes the location counter to be rounded up to an *align* power-of-two boundary before assigning the location counter to the value of *name*.

EXAMPLE:

```
.lcomm abyte,1      | or:  .lcomm abyte,1,0
.lcomm padding,7
.lcomm adouble,8    | or:  .lcomm adouble,8,3
```

These are the same as:

```
.zerofill __DATA,__bss,abyte,1
.lcomm __DATA,__bss,padding,7
.lcomm __DATA,__bss,adouble,8
```

Directives for Dealing with Symbols

This section describes directives that have an effect on symbols and the symbol table.

.globl

SYNOPSIS:

.globl symbol_name

The **.globl** directive makes *symbol_name* external. If *symbol_name* is otherwise defined (by **.set** or by appearance as a label), it acts within the assembly exactly as if the **.globl** statement were not given; however, the link editor may be used to combine this object module with other modules referring to this symbol.

EXAMPLE:

```
        .globl abs
        .set abs,1

        .globl var
var:    .long 2
```

.indirect_symbol

SYNOPSIS:

.indirect_symbol *symbol_name*

The **.indirect_symbol** directive creates an indirect symbol with *symbol_name* and associates the current location with the indirect symbol. An indirect symbol must be defined immediately before each item in a **symbol_stub**, **lazy_symbol_pointers**, and **non_lazy_symbol_pointers** section. The static and dynamic linkers use *symbol_name* to identify the symbol associated with the following item.

.reference

SYNOPSIS:

.reference *symbol_name*

The **.reference** directive causes *symbol_name* to be an undefined symbol that will be present in the output's symbol table. This is useful in referencing a symbol without generating any bytes to do it (used, for example, by the Objective C run-time system to reference superclass objects).

EXAMPLE:

```
.reference  .objc_class_name_Object
```

.private_extern

SYNOPSIS:

.private_extern *symbol_name*

The **.private_extern** directive makes *symbol_name* a private external symbol. When the link editor combines this module with other modules (and the **-keep_private_externs** command-line option is not specified) the symbol turns it from global to static.

.lazy_reference

SYNOPSIS:

.lazy_reference *symbol_name*

The **.reference** directive causes *symbol_name* to be a lazy undefined symbol that will be present in the output's symbol table. This is useful in referencing a symbol without generating any bytes to do it (used, for example, by the Objective C run-time system with the dynamic linker to reference superclass objects

but to allow the runtime to bind them on first use).

```
EXAMPLE:
    .lazy_reference  .objc_class_name_Object
```

.stabs, .stabn, and .stabd

SYNOPSIS:
.stabs *n_name* , *n_type* , *n_other* , *n_desc* , *n_value*
.stabn *n_type* , *n_other* , *n_desc* , *n_value*
.stabd *n_type* , *n_other* , *n_desc*

These three directives are used to place symbols in the symbol table for the symbolic debugger (a “stab” is a *s*ymbol *tab*le entry).

- **.stabs** specifies all the fields in a symbol table entry. The *n_name* is the name of a symbol; if the symbol name is null, the .stabn directive may be used instead.
- **.stabn** is like **.stabs**, except that it uses a NULL (“”) name.
- **.stabd** is like **.stabn**, except that it uses the value of the location counter (.) as the *n_value* field.

In each case, the *n_type* field is assumed to contain a 4.3BSD-like value for the N_TYPE bits. For **.stabs** and **.stabn** the **n_sect** field of the Mach-O file’s **nlist** is set to the section number of the symbol for the specified *n_value* parameter. For **.stabd** the **n_sect** field is set to the current section number for the location counter. The **nlist** structure is defined in **mach-o/nlist.h**.

Note: The *n_other* field of a stab directive is ignored.

EXAMPLE:
```
    .stabs  “hello.c”,100,0,0,Ltext
    .stabn  192,0,0,LBB2
    .stabd  68,0,15
```

.desc

SYNOPSIS:
.desc *symbol_name* , *absolute_expression*

The **.desc** directive sets the **n_desc** field of the specified symbol to *absolute_expression*.

EXAMPLE:
```
    .desc  _main,0xface
```

.set

SYNOPSIS:
.set *symbol_name* , *absolute_expression*

The **.set** directive creates the symbol *symbol_name* and sets its value to *absolute_expression*. This is the same as using *symbol_name* **=** *absolute_expression*.

EXAMPLE:
```
.set one,1
two = 2
```

.lsym

SYNOPSIS:
.lsym *symbol_name* **,** *expression*

A unique and otherwise unreferenceable symbol of the (*symbol_name*, *expression*) pair is created in the symbol table. Some Fortran 77 compilers use this mechanism to communicate with the debugger.

Miscellaneous Directives

This section describes additional directives that don't fit into any of the previous sections.

.abort

SYNOPSIS:
.abort [**"***abort_string***"**]

The .abort directive causes the assembler to ignore all further input and quit processing. No files are created. The directive would be used, for example, in a pipe interconnected version of a compiler—the first major syntax error would cause the compiler to issue this directive, saving unnecessary work in assembling code that would have to be discarded anyway.

The optional "abort_string" is printed as part of the error message when the **.abort** directive is encountered.

EXAMPLE:
```
#ifndef VAR
    .abort "You must define VAR to assemble this file."
#endif
```

.file and .line

SYNOPSIS:
.file *file_name*
.line *line_number*

The .file directive causes the assembler to report error messages as if it were processing the file file_name.

The .line directive causes the assembler to report error messages as if it were processing the line line_number. The next line after the .line directive is assumed to be line_number.

The assembler turns C preprocessor comments of the form

```
# line_number file_name level
```

into

```
.line line_number; .file file_name
```

EXAMPLE:

```
.line 6
nop       | this is line 6
```

.if, .elseif, .else, and .endif

SYNOPSIS:

.if *expression*
.elseif *expression*
.else
.endif

These directives are used to delimit blocks of code that are to be assembled conditionally, depending on the value of an expression. A block of conditional code may be nested within another block of conditional code. Expression must be an absolute expression.

For each .if directive,

- there must be a matching **.endif**
- there may be as many intervening **.elseif**'s as desired
- there may be no more than one intervening **.else** before the tailing **.endif**

Labels or multiple statements must not be placed on the same line as any of these directives; otherwise, statements including these directives won't be recognized and will produce errors or incorrect conditional assembly.

EXAMPLE:

```
.if a==1
.long 1
.elseif a==2
.long 2
.else
.long 3
.endif
```

.include

SYNOPSIS:

.include *filename*

The **.include** directive causes the named file to be included at the current point in the assembly. The **-I***dir* option to the assembler specifies alternative paths to be used in searching for the file if it isn't found

in the current directory (the default path, **/usr/include**, is always searched last).

EXAMPLE:

```
.include  macros.h
```

.macro, .endmacro, .macros_on, and .macros_off

SYNOPSIS:

.macro
.endmacro
.macros_on
.macros_off

These directives allow your to define simple macros (once a macro is defined, however, you can't redefine it). For example:

```
.macro  var
instruction_1  $0,$1
instruction_2  $2
 . . .
instruction_N
.long $n
.endmacro
```

$*d* (where *d* is a single decimal digit, 0 through 9) represents each argument—there can be at most 10 arguments. *$n* is replaced by the actual number of arguments the macro was invoked with.

When you use a macro, arguments are separated by a comma (except inside matching parentheses—for example, `xxx(1,3,4),yyy` contains only two arguments). You could use the macro defined above as follows:

```
var  #0,@sp,4
```

This would be expanded to:

```
instruction_1 #0,@sp
instruction_2 4
 . . .
instruction_N
.long 3
```

The directives **.macros_on** and **.macros_off** allow macros to be written that override an instruction or directive while still using the instruction or directive. For example:

```
.macro .long
.macros_off
.long $0,$0
.macros_on
.endmacro
```

If you don't specify an argument, the macro will substitute nothing (also see the **.abs** directive below).

.abs

SYNOPSIS:
.abs *symbol_name* **,** *expression*

This directive sets the value of *symbol_name* to 1 if *expression* is an absolute expression; otherwise, it sets the value to 0.

EXAMPLE:
```
.macro var
.abs is_abs,$0
.if is_abs==1
.abort "must be absolute"
.endif
.endmacro
```

.dump and .load

SYNOPSIS:
.dump *filename*
.load *filename*

These directives let you dump and load the absolute symbols and macro definitions, for faster loading and faster assembly.

These work like this:

```
.include "big_file_1"
.include "big_file_2"
.include "big_file_3"
. . .
.include "big_file_N"
.dump    "symbols.dump"
```

The **.dump** directive writes out all the N_ABS symbols and macros. You can later use the **.load** directive to load all the N_ABS symbols and macros faster than you could with **.include**:

```
.load "symbols.dump"
```

One useful side effect of loading symbols this way is that they aren't written out to the object file.

Additional Processor-Specific Directives

The following processor-specific directives are synonyms for other standard directives described earlier in this chapter; although they are listed here for completeness, their use isn't recommended; wherever possible, you should use the standard directive instead.

The following are i386-specific directives:

i386 Directive	Standard Directive

.ffloat	.single
.dfloat	.double
.tfloat	[expression] ← 80-bit IEEE extended precision floating-point
.word	.short
.value	.short
.ident	(ignored)
.def	(ignored)
.optim	(ignored)
.version	(ignored)
.ln	(ignored)

Assembly Language Statements

This chapter describes the assembly language statements that make up an assembly language program.

The general format of an assembly language statement is shown below. Each of the fields shown here is described in detail in one of the following sections.

```
[ label_field ] [ opcode_field [ operand_field ] ] [ comment_field ]
```

A line may contain multiple statements separated by semicolons, or by at (@) signs for the hppa, which may then be followed by a single comment:

```
[ statement [ ; statement ...] ] [ comment_field ]
[ statement [ @ statement ...] ] [ comment_field ]
```

The following rules apply to the use of whitespace within a statement:

- Spaces or tabs are used to separate fields.
- At least one space or tab must occur between the opcode field and the operand field.
- Spaces may appear within the operand field.
- Spaces and tabs are significant when they appear in a character string.

Label Field

Labels are identifiers that you use to tag the locations of program and data objects. Each label is composed of an identifier and a terminating colon. The format of the label field is:

```
identifier: [ identifier: ] ...
```

The optional label field can only occur first in a statement. The following example shows a label field containing two labels, followed by a (M68000-style) comment:

```
var: VAR:  | two labels defined here
```

As shown here, letters in identifiers are case-sensitive, and both uppercase and lowercase letters may be used.

Operation Code Field

The operation code field of an assembly language statement identifies the statement as a machine instruction, an assembler directive, or a macro defined by the programmer:

- A machine instruction is indicated by an instruction mnemonic. An assembly language statement that contains an instruction mnemonic is intended to produce a single executable machine instruction. The operation and use of each instruction is described in the manufacturer's user manual.

- An assembler directive (or pseudo-op) performs some function during the assembly process. It doesn't produce any executable code, but it may assign space for data in the program.

- Macros are defined with the .macro directive (see Chapter 4 for more information).

One or more spaces or tabs must separate the operation code field from the following operand field in a statement. Spaces or tabs are optional between the label and operation code fields, but they help to improve the readability of the program.

Architecture- and Processor-Specific Caveats

Intel i386 Architecture

- As with the Motorola 68000 family, i386 instructions can operate on byte, word, or long word data (the last is called “double word” by Intel). The size can be indicated in the same way as it is for the MC68000. If no size is specified, the assembler attempts to determine the size from the operands. For example, if the 16-bit names for registers are used as operands, a 16-bit operation will be performed. When both a size specifier and a size-specific register name are given, the size specifier is used. Thus, the following are all correct and result in the same operation:

```
movw    %bx,%cx
mov     %bx,%cx
movw    %ebx,%ecx
```

- An i386 operation code can also contain optional prefixes, which are separated from the operation code by a slash (‘/’) character. The prefix mnemonics are:

  data16 operation uses 16-bit data

  addr16 operation uses 16-bit addresses

  lock exclusive memory lock

  wait wait for pending numeric exceptions

  cs, **ds**, **es**, **fs**, **gs**, **ss** segment register override

  rep, **repe**, **repne** repeat prefixes for string instructions

More than one prefix may be specified for some operation codes. For example:

```
lock/fs/xchgl    %ebx,4(%ebp)
```

Segment register overrides and the 16-bit data specifications are usually given as part of the operation code itself or of its operands. For example, the following two lines of assembly generate the same instructions:

```
movw            %bx,%fs:4(%ebp)
data16/fs/movl  %bx,4(%ebp)
```

Not all prefixes are allowed with all instructions. The assembler does check that the repeat prefixes for strings instructions are used correctly, but doesn’t otherwise check for correct usage.

Operand Field

The operand field of an assembly language statement supplies the arguments to the machine instruction,

assembler directive, or macro.

The operand field may contain one or more operands, depending on the requirements of the preceding machine instruction or assembler directive. Some machine instructions and assembler directives don't take any operand, and some take two or more. If the operand field contains more than one operand, the operands are generally separated by commas, as shown here:

```
[ operand [ , operand ] ... ]
```

The following types of objects can be operands:

- register operands
- register pairs
- address operands
- string constants
- floating-point constants
- register lists
- expressions

Register operands in a machine instruction refer to the machine registers of the processor or coprocessor. Register names may appear in mixed case.

Architecture-and Processor-Specific Caveats

Intel 386 Architecture

- The NeXT assembler orders operand fields for i386 instructions in the reverse order from Intel's conventions. Intel's convention is destination first, source second; NeXT's is source first, destination second. Where Intel documentation would describe the Compare and Exchange instruction for 32-bit operands as follows:

  ```
  CMPXCHG  r/m32,r32    # Intel processor manual convention
  ```

  The NeXT assembler syntax for this same instruction is:

  ```
  cmpxchg  r32,r/m32    # NeXT assembler syntax
  ```

  So an example of actual assembly code for the NeXT would be:

  ```
  cmpxchg  %ebx,(%eax)  # NeXT assembly code
  ```

Comment Field

The assembler recognizes two types of comments in source code:

- A line whose first non-whitespace character is the hash character (#) is a comment. This style of comment is useful for passing C preprocessor output through the assembler. Note that comments of the form

  ```
  # line_number file_name level
  ```

  get turned into

```
.line line_number; .file file_name
```

This can cause problems when comments of this form which aren't intended to specify line numbers precede assembly errors, since the error will be reported as occurring on a line relative to that specified in the comment. Suppose a program contains these two lines of assembly source:

```
# 500
  .var
```

If ".var" hasn't been defined, this fragment will result in the following error message:

```
var.s:500:Unknown pseudo-op: .var
```

- A comment field, appearing on a line after one or more statements. The comment field consists of the appropriate comment character and all the characters that follow it on the line:

  | comment character for MC68000 processors

  ; comment character for hppa processors

  # comment character for i386 architecture processors

An assembly language source line can consist of just the comment field; in this case, it's equivalent to using the hash character comment style:

```
# This is a comment.
| This is a comment.
```

Note the warning given above for hash character comments beginning with a number.

Direct Assignment Statements

This section describes direct assignment statements, which don't conform to the normal statement syntax described throughout this chapter. A direct assignment statement can be used to assign the value of an expression to an identifier. The format of a direct assignment statement is:

```
identifier = expression
```

If expression in a direct assignment is absolute, identifier is also absolute, and it may be treated as a constant in subsequent expressions. If expression is relocatable, identifier is also relocatable, and it is considered to be declared in the same program section as the expression.

The use of an assignment statement is analogous to using the .set directive (described in the following chapter), except that the .set directive requires that expression be absolute.

Once an identifier has been defined by a direct assignment statement, it may be redefined—its value is then the result of the last assignment statement. There are a few restrictions, however, concerning the redefinition of identifiers:

- Register identifiers may not be redefined.
- An identifier that has already been used as a label should not be redefined, since this would

amount to redefining the address of a place in the program. Moreover, an identifier that has been defined in a direct assignment statement cannot later be used as a label. Only the second situation produces an assembler error message.

Assembly Language Syntax

This chapter first describes the basic lexical elements of assembly language programming, and then describes how those elements combine to form complete assembly language expressions. The following chapter goes on to explain how sequences of expressions are put together to form the statements that make up an assembly language program.

Elements of Assembly Language

This section describes the basic building blocks of an assembly language program—these are characters, symbols, labels, and constants.

Characters

The following characters are used in assembly language programs

- alphanumeric characters—'A' through 'Z', 'a' through 'z', and '0' through '9'
- other printable ASCII characters (such as #, $, :, ., +, -, *, /, !, and |)
- non-printing ASCII characters (such as space, tab, return, and newline)

Some of these characters have special meanings, which are described in the section "Expression Syntax" and in the following chapter.

Identifiers

An *identifier* (also known as a *symbol*) can be used for several purposes:

- as the *label* for an assembler statement (see the following section, "Labels")
- as a location tag for data
- as the symbolic name of a constant

Each identifier consists of a sequence of alphanumeric characters (which may include other printable ASCII characters such as ., _, and $). The first character must not be numeric. Identifiers may be of any length, and all characters are significant. Case of letters is significant—for example, the identifier var is different from the identifier Var.

It is also possible to define a new identifier by enclosing multiple identifiers within a pair of double quotes. For example:

```
"Object +new:":
.long "Object +new:"
```

Labels

A label is written as an identifier immediately followed by a colon (**:**). The label represents the current value of the current location counter; it can be used in assembler instructions as an operand.

Note: You may not use a single identifier to represent two different locations.

Numeric Labels

Local numeric labels allow compilers and programmers to use names temporarily. A numeric label consists of a digit (between 0 and 9) followed by a colon. These ten local symbol names can be reused any number of times throughout the program. As with alphanumeric labels, a numeric label assigns the current value of the location counter to the symbol.

Although multiple numeric labels with the same digit may be used within the same program, only the next definition and the most recent previous definition of a label can be referenced:

- To refer to the most recent previous definition of a local numeric label, write *digit***b**, (using the same digit as when you defined the label).
- To refer to the next definition of a numeric label, write *digit***f**.

The Scope of a Label

The scope of a label is the distance over which it is visible to (and referenceable by) other parts of the program. Normally, a label that tags a location or data is visible only within the current assembly unit.

The .globl directive (described in Chapter 4) may be used to make a label external. In this case, the symbol is visible to other assembly units at link time.

Constants

Four types of constants are available: *numeric constants*, *character constants*, *string constants*, and *floating point constants*. All constants are interpreted as absolute quantities when they appear in an expression.

Numeric Constants

A numeric constant is a token that starts with a digit. Numeric constants can be decimal, hexadecimal, or octal. The following restrictions apply:

- Decimal constants contain only digits between 0 and 9, and normally aren't longer than 32 bits—having a value between -2,147,483,648 and 2,147,483,647 (values that don't fit in 32 bits are bignums, which are legal but which should fit within the designated format). Decimal constants cannot contain leading zeros or commas.
- Hexadecimal constants start with 0x (or 0X), followed by between one and eight decimal or hexadecimal digits (0 through 9, 'a' through 'f', and 'A' through 'F'). Values that don't fit in 32 bits are bignums.
- Octal constants start with 0, followed by from one to eleven octal digits (0 through 7). Values that don't fit in 32 bits are bignums.

Character Constants

A single-character constant consists of a single quote (') followed by any ASCII character. The constant's value is the code for the given character.

String Constants

A string constant is a sequence of 0 or more ASCII characters surrounded by quotation marks ("*characters*").

Floating Point Constants

The general lexical form of a floating point number is:

```
0flt_char[{+-}]dec...[.][dec...][exp_char[{+-}][dec...]]
```

where:

flt_char	a required type specification character (see the following table)
[{+-}]	the optional occurrence of either + or -, but not both
dec...	a required sequence of 1 or more decimal digits
[.]	a single optional "."
[*dec*...]	an optional sequence of 1 or more decimal digits
[*exp_char*]	an optional exponent delimiter character (see the following table)

The type specification character, *flt_char*, specifies the type and representation of the constructed number; the set of legal type specification characters with the processor architecture, as shown here:

Architecture

flt_char	*exp_char*
M68000 {rRsSfFdDxXeEpP}	{eE}
i386	{fFdDxX} {eE}
hppa	{dDfF} {eE}

On the M68000 architecture, **0b** can be used to specify an immediate hexadecimal bit pattern. For example:

```
fmoves #0b7f80001,fp0
```

moves the signaling Nan into the register **fp0** and

```
fmoves #0x7f80001,fp0
```

moves the decimal number 2,139,095,041 (0x7f80001 in hexadecimal) into the register **fp0**.

When floating-point constants are used as arguments to the .single and .double directives, the type specification character isn't actually used in determining the type of the number. For convenience, r or R can be used consistently to specify all types of floating-point numbers.

Collectively, all floating point numbers, together with quad and octal scalars, are called Bignums. When

as requires a Bignum, a 32-bit scalar quantity may also be used.

Floating point constants are internally represented as flonums, in a machine-independent, precision-independent floating point format (for accurate cross-assembly).

Assembly Location Counter

A single period (.), usually referred to as “dot,” is used to represent the current location counter. There is no way to explicitly reference any other location counters besides the current location counter.

Even if it occurs in the operand field of a statement, dot refers to the address of the first byte of that statement; the value of dot isn’t updated until the next machine instruction or assembler directive.

Expression Syntax

Expressions are combinations of operand terms (which can be numeric constants or symbolic identifiers) and operators. This section lists the available operators, and describes the rules for combining these operators with operands in order to produce legal expressions.

Operators

Identifiers and numeric constants can be combined, through the use of operators, to form expressions. Each operator operates on 32-bit values. If the value of a term occupies 8 or 16 bits, it is sign extended to a 32-bit value.

The assembler provides both unary and binary operators. A unary operator precedes its operand; a binary operator follows its first operand, and precedes its second operand. For example:

```
!var    | unary expression
var+5   | binary expression
```

The assembler recognizes the following unary operators:

- **-** *Unary minus*: the result is the two’s complement of the operand
- **~** *One’s complement*: the result is the one’s complement of the operand
- **!** *Logical negation*: the result is 0 if the operand is non-zero, and 1 if the operand is 0

The assembler recognizes the following binary operators:

- **+** *Addition*: the result is the arithmetic addition of the two operands
- **-** *Subtraction*: the result is the arithmetic subtraction of the two operands
- ***** *Multiplication*: the result is the arithmetic multiplication of the two operands
- */* *Division*: the result is the arithmetic division of the two operands; this is integer division, which truncates towards zero

% *Modulus*: the result is the remainder that's produced when the first operand is divided by the second (this operator applies only to integral operands)

>> *Right shift*: the result is the value of the first operand shifted to the right, where the second operand specifies the number of bit positions by which the first operand is to be shifted (this operator applies only to integral operands). This is always an arithmetic shift since all operators operate on signed operands.

<< *Left shift*: the result is the value of the first operand shifted to the left, where the second operand specifies the number of bit positions by which the first operand is to be shifted (this operator applies only to integral operands)

& *Bitwise AND*: the result is the bitwise AND function of the two operands (this operator applies only to integral operands)

^ *Bitwise exclusive OR*: the result is the bitwise exclusive OR function of the two operands (this operator applies only to integral operands)

| *Bitwise inclusive OR*: the result is the bitwise inclusive OR function of the two operands (this operator applies only to integral operands); this operator can't be used on the M68000 microprocessor family, because the '|' character is used there to mark the start of a comment

< *Less than*: the result is 1 if the first operand is less than the second operand, and 0 otherwise

> *Greater than*: the result is 1 if the first operand is greater than the second operand, and 0 otherwise

<= *Less than or equal*: the result is 1 if the first operand is less than or equal to the second operand, and 0 otherwise

>= *Greater than or equal*: the result is 1 if the first operand is greater than or equal to the second operand, and 0 otherwise

== *Equal*: the result is 1 if the two operands are equal, and 0 otherwise

!= *Not equal* (same as **<>**): the result is 0 if the two operands are equal, and 1 otherwise

Terms

A term is a part of an expression; it may be:

- An identifier.
- A numeric constant (its 32-bit value is used). The assembly location counter (.), for example, is a valid numeric constant.
- An expression or term enclosed in parentheses. Any quantity enclosed in parentheses is evaluated before the rest of the expression. This can be used to alter the normal evaluation of expressions—for example, to differentiate between x * y + z and x * (y + z) or to apply a unary operator to an entire expression—for example, -(x * y + z).
- A term preceded by a unary operator (for example, ~var). Multiple unary operators may be used in a term (for example, !~var).

Expressions

Expressions are combinations of terms joined together by binary operators. An expression is always evaluated to a 32-bit value, but in some situations a different value will be used:

- If the operand requires a one-byte value (a .byte directive, for example), the low-order eight bits of the expression are used.
- If the operand requires a 16-bit value (a .short directive or a movem instruction, for example), the low-order 16 bits of the expression are used.

All expressions are evaluated using the same operator precedence rules that are used by the C programming language.

When an expression is evaluated its value is absolute, relocatable, or external, as described below.

Absolute Expressions

An expression is absolute if its value is fixed. The following, for example, are absolute:

- An expression whose terms are constants
- An identifier whose value is a constant via a direct assignment statement
- A relocatable expression minus a relocatable term, if both items belong to the same program section.

Relocatable Expressions

An expression (or term) is relocatable if its value is fixed relative to a base address, but will have an offset value when it is linked or loaded into memory. For example, all labels of a program defined in relocatable sections are relocatable.

Expressions that contain relocatable terms must only add or subtract constants to their value. For example, if the identifiers var and dat were defined in a relocatable section of the program, then the following examples demonstrate the use of relocatable expressions:

var is a simple relocatable term. Its value is an offset from the base address of the current control section.

var+5 is a simple relocatable expression. Since the value of var is an offset from the base address of the current control section, adding a constant to it doesn't change its relocatable status.

var*2 is not relocatable. Multiplying a relocatable term by a constant invalidates the relocatable status of the expression.

2-var is not relocatable. The expression can't be linked by adding var's offset to it.

var+dat+5 is a relocatable expression if both var and dat are both defined in some section—that is, if neither is undefined. This form of relocatable expression is used for position-independent

code and is supported in NEXTSTEP Release 3.3 and later.

External Expressions

An expression is *external* (or *global*) if it contains an external identifier not defined in the current program. In general, the same restrictions on expressions containing relocatable identifiers apply to expressions containing external identifiers. An exception is that the expression **var-dat** is incorrect when both **var** and **dat** are external identifiers (that is, you cannot subtract two external relocatable expressions). Also, you cannot multiply or divide any relocatable expression.

Contents

Using the Assembler

This chapter describes how to run the **as** assembler, which produces an object file from one or more files of assembly language source code.

Note: Although **a.out** is the default file name that **as** gives to the object file that's created (as is conventional with most UNIX-style compiler systems), the format of the object file is not standard UNIX 4.3BSD **a.out** format. Object files produced by the assembler are in Mach-O (Mach object) file format. For more information about the Mach-O file format, see the *NEXTSTEP Development Tools and Techniques* manual.

Command Syntax

To run the assembler, type the following command in a shell window:

```
as [ option ] ... [ file ] ...
```

You can specify one or more command-line options. These assembler options are described in the following section.

You can specify one or more files containing assembly language source code. If no files are specified, **as** uses the standard input (stdin) for the assembly source input.

Note: By convention, files containing assembly language source code should have a **.s** extension.

Assembler Options

The following command-line options are recognized by the assembler:

-o *name* The *name* argument after **-o** is used as the name of the **as** output file, instead of **a.out**.

-- Use the standard input (stdin) for the assembly source input.

-f Fast; no need to run **app** (the assembler preprocessor). This option is intended for use by compilers that produce assembly code in a strict "clean" format that specifies exactly where whitespace can go. The **app** preprocessor needs to be run on handwritten assembly files and on file that have been preprocessed by **cpp** (the C preprocessor). This typically is needed when assembler files are assembled through the use of the **cc**(1) command, which automatically runs the C preprocessor on assembly source files. The assembler preprocessor strips out excess spaces, turns each single-quoted character into a decimal constant, and turns occurrences of

```
# number filename level
```

into:

`.line` *`number`*`;.file` *`filename`*

The assembler preprocessor can also be turned off by starting the assembly file with **#NO_APP\n**. When the assembler preprocessor has been turned off in this way, it can be turned on and off with pairs of **#APP\n** and **#NO_APP\n** at the beginning of lines. This is used by the compiler to wrap assembly statements produced from **asm**() statements.

-g Produce debugging information for the symbolic debugger gdb(1) so the the assembly source can be debugged symbolically. For include files (included by the C preprocessor's #include or by the assembler directive .include) that produce instructions in the (__TEXT,__text) section, the include file must be included while a .text directive is in effect (that is, there must be a .text directive before the include) and end with the a .text directive in effect (at the end of the include file). Otherwise the debugger will have trouble dealing with that assembly file.

-v Print the version of the assembler (both the NeXT version and the GNU version that it is based on).

-n Don't assume that the assembly file starts with a .text directive.

-**I***dir* Add dir to the list of directories to search for files included with the .include directive. The default places to search are the current directory, and then /usr/include.

-**L** Save defined labels beginning with an 'L' (the compiler generates these temporary labels). Temporary labels are normally discarded to save space in the resulting symbol table.

-**W** Suppress warnings.

Architecture Options

The program **/bin/as** is a driver that executes assemblers for specific target architectures. If no target architecture is specified, it defaults to the architecture of the host it is running on.

-arch *arch_type*
Specifies to the target architecture, *arch_type*, the assembler to be executed. The target assemblers for each architecture are in **/lib/***arch_type***as**.

-**arch_multiple**
This is used by the **cc**(1) driver program when it is run with multiple **-arch** *arch_type* flags and instructs programs like **as**(1) that if it prints any messages to precede the messages with one line stating the program name—in this case **a**

—and the architecture (from the **-arch** *arch_type* flag) to distinguish which architecture the error messages refer to. This flag is accepted only by the actual assemblers (in **/lib** *arch_type***/as**) and not by the assembler driver, **/bin/as**.

i386 Addressing Modes and Assembler Instructions

This chapter contains information specific to the Intel i386 processor architecture, which includes the i386, i486, and Pentium processors. The first section, “i386 Registers and Addressing Modes,” lists the registers available and describes the addressing modes used by assembler instructions. The second section, “i386 Assembler Instructions,” lists each assembler instruction with NeXT assembler syntax.

Note: Don’t confuse the i386 *architecture* with the i386 *processor*. NEXTSTEP makes use of instructions specific to the i486 processor, and will not run on an i386 processor.

i386 Registers and Addressing Modes

This section describes the conventions used to specify addressing modes and instruction mnemonics for the Intel i386 processor architecture. The instructions themselves are detailed in the next section, “i386 Assembler Instructions.”

Instruction Mnemonics

The instruction mnemonics that the assembler uses are based on the mnemonics described in the relevant Intel processor manuals.

Note: Branch instructions are always long (32 bits) for non-local labels on the NeXT i386 architecture machines. This allows the link editor to do procedure ordering (see the description of the **-sectorder** option in the **ld**(1) man page, and the “Link Optimization” paper in the directory **/NextLibrary/Documentation/NextDev/Concepts/Performance**).

Registers

Many instructions accept registers as operands. The available registers are listed in this section. The NeXT assembler for Intel i386 processors always uses names beginning with a percent sign (‘%’) for registers, so naming conflicts with identifiers aren’t possible; further, all register names are in lowercase letters.

General Registers

Each of the 32-bit general registers of the i386 architecture are accessible by different names, which specify parts of that register to be used. For example, the AX register can be accessed as a single byte (**%ah** or **%al**), a 16-bit value (**%ax**), or a 32-bit value (**%eax**). Figure 6-1 shows the names of these registers and their relation to the full 32-bit storage for each register:

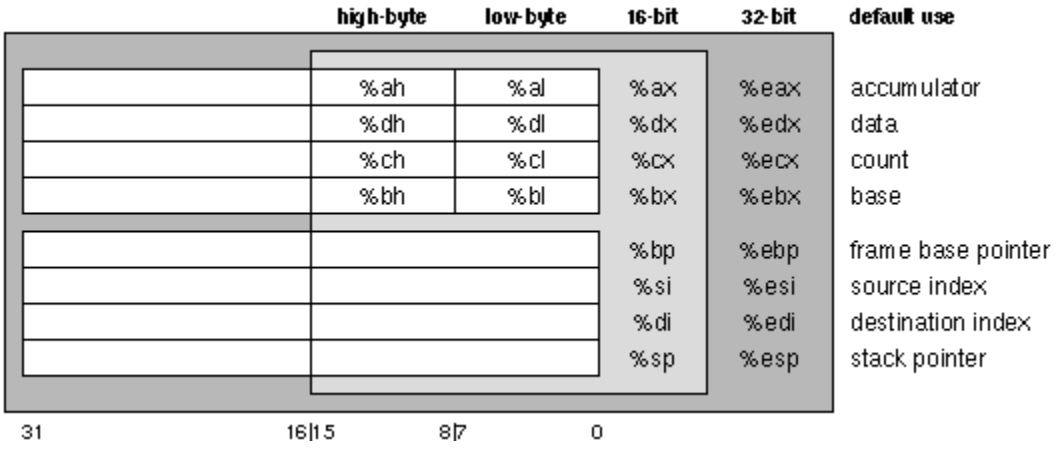

Figure 6-1

Floating-Point Registers

%st

%st(0)-%st(7)

Segment Registers

%cs	code segment register
%ss	stack segment register
%ds	data segment register
%es	data segment register (string operation destination segment)
%fs	data segment register
%gs	data segment register

Other Registers

%cr0-%cr3	control registers
%db0-%db7	debug registers
%tr3-%tr7	test registers

Operands and Addressing Modes

The i386 architecture uses four kinds of instruction operands:

- Register

- Immediate
- Direct Memory
- Indirect Memory

Each type of operand corresponds to an addressing mode. Register operands specify that the value stored in the named register is to be used by the operator. Immediate operands are constant values specified in assembler code. Direct memory operands are the memory location of labels, or the value of a named register treated as an address. Indirect memory operands are calculated at run time from the contents of registers and optional constant values.

Register Operands

A register operand is given simply as the name of a register. It can be any of the identifiers beginning with '%' listed above; for example, **%eax**. When an operator calls for a register operand of a particular size, the operand is listed as *r8*, *r16*, or *r32*.

Immediate Operands

Immediate operands are specified as numeric values preceded by a dollar sign ('$'). They are decimal by default, but can be marked as hexadecimal by beginning the number itself with '0x'. Simple calculations are allowed if grouped in parentheses. Finally, an immediate operand can be given as a label, in which case its value is the address of that label. Here are some examples:

```
$100
$0x5fec4
$(10*6)     # calculated by the assembler
$begloop
```

A reference to an undefined label is allowed, but that reference must be resolved at link time.

Direct Memory Operands

Direct memory operands are references to labels in assembler source. They act as static references to a single location in memory relative to a specific segment, and are resolved at link time. Here's an example:

```
     .data
var: .byte 0       # declare a byte-size variable labelled "var"
     .text
     .
     .
     .
     movb %al,var  # move the low byte of the AX register into the
                   #  memory location specified by "var"
```

By default, direct memory operands use the **%ds** segment register. This can be overridden by prefixing the operands with the segment register desired and a colon:

```
     movb %es:%al,var  # move the low byte of the AX register into the
                       #  memory location in the segment given by %es
                       # and "var"
```

Note that the segment override applies only to the memory operands in an instruction; “var” is affected, but not **%al**. The string instructions, which take two memory operands, use the segment override for both. A less common way of indicating a segment is to prefix the operator itself:

```
es/movb %al,%var   # same as above
```

Indirect Memory Operands

Indirect memory operands are calculated from the contents of registers at run time. An indirect memory operand can contain a base register, and index register, a scale, and a displacement. The most general form is:

displacement(*base_register*,*index_register*,*scale*)

displacement is an immediate value. The base and index registers may be any 32-bit general register names, except that **%esp** can’t be used as an index register. *scale* must be 1, 2, 4, or 8; no other values are allowed. The displacement and scale can be omitted, but at least one register must be specified. Also, if items from the end are omitted, the preceding commas can also be omitted, but the comma following an omitted item must remain:

```
10(%eax,%edx)
(%eax)
12(,%ecx,2)
12(,%ecx)
```

The value of an indirect memory operand is the memory location given by the contents of the register, relative to a segment’s base address. The segment register used is **%ss** when the base register is **%ebp** or **%esp**, and **%ds** for all other base registers. For example:

```
movl (%eax),%edx    # default segment register here is %ds
```

The above assembler instruction moves 32 bits from the address given by **%eax** into the **%edx** register. The address **%eax** is relative to the **%ds** segment register. A different segment register from the default can be specified by prefixing the operand with the segment register name and a colon (‘:’):

```
movl %es:(%eax),%edx
```

A segment override can also be specified as an operator prefix:

```
es/movl (%eax),%edx
```

i386 Assembler Instructions

Note the following points about the information contained in this section:

- **Name** is the name that appears in the upper left corner of a page in the Intel manuals.
- **Operation Name** is the name that appears after the operator name in the Intel manuals. Processor-specific instructions are marked as they occur.
- The form of operands is that used in Intel’s *i486 Microprocessor Programmer’s Reference Manual*.

- The order of operands is **source** → **destination**, the opposite of the order in Intel's manuals.

Instructions

Name	Operator	Operand	Operation Name
aaa	aaa		ASCII Adjust after Addition
aad	aad		ASCII Adjust AX before Division
aam	aam		ASCII Adjust AX after Division
aas	aas		ASCII Adjust AL after Subtraction
adc	adc	$*imm8,r/m8*	Add with Carry
	adc	$*imm16,r/m*16	
	adc	$*imm32,r/m32*	
	adc	$*imm8,r/m16*	
	adc	$*imm8,r/m32*	
	adc	*r8,r/m8*	
	adc	*r16,r/m16*	
	adc	*r32,r/m32*	
	adc	*r/m8,r8*	
	adc	*r/m16,r16*	
	adc	*r/m32,r32*	

add	add	$*imm8,r/m8*	Add
	add	$*imm16,r/m*16	
	add	$*imm32,r/m32*	
	add	$*imm8,r/m16*	
	add	$*imm8,r/m32*	
	add	*r8,r/m8*	
	add	*r16,r/m16*	
	add	*r32,r/m32*	
	add	*r/m8,r8*	
	add	*r/m16,r16*	
	add	*r/m32,r32*	
and	and	$*imm8,r/m8*	Logical AND
	and	$*imm16,r/m*16	
	and	$*imm32,r/m32*	
	and	$*imm8,r/m16*	
	and	$*imm8,r/m32*	
	and	*r8,r/m8*	
	and	*r16,r/m16*	
	and	*r32,r/m32*	
	and	*r/m8,r8*	
	and	*r/m16,r16*	
	and	*r/m32,r32*	
arpl	arpl	*r16,r/m16*	Adjust RPL Field of Selector
bound	bound	*m16&16,r16*	Check Array Index Against
	bound	*m32&32,r32*	Bounds

bsf	bsf	*r/m16,r16*	Bit Scan Forward
	bsf	*r/m32,r16*	
bsr	bsr	*r/m16,r16*	Bit Scan Reverse
	bsr	*r/m32,r16*	
bswap	bswap	*r32*	Byte Swap (i486-specific)
bt	bt	*r16,r/m16*	Bit Test
	bt	*r32,r/m32*	
	bt	$*imm8,r/m16*	
	bt	$*imm8,r/m32*	
btc	btc	*r16,r/m16*	Bit Test and Complement
	btc	*r32,r/m32*	
	btc	$*imm8,r/m16*	
	btc	$*imm8,r/m32*	
btr	btr	*r16,r/m16*	Bit Test and Reset
	btr	*r32,r/m32*	
	btr	$*imm8,r/m16*	
	btr	$*imm8,r/m32*	
bts	bts	*r16,r/m16*	Bit Test and Set
	bts	*r32,r/m32*	
	bts	$*imm8,r/m16*	
	bts	$*imm8,r/m32*	

call	call	*rel16*	Call Procedure
	call	*r/m16*	
	call	*ptr16:16*	
	call	*m16:16*	
	call	*rel32*	
	call	*r/m32*	
	lcall	$*imm16,*$*imm32*	
	lcall	*m16*	
	lcall	*m32*	
cbw/cwde	cbw		Convert Byte to Word/
	cwde		Convert Word to Doubleword
clc	clc		Clear Carry Flag
cld	cld		Clear Direction Flag
cli	cli		Clear Interrupt Flag
clts	clts		Clear Task-Switched Flag in CR0
cmc	cmc		Complement Carry Flag
cmp	cmp	$*imm8,r/m8*	Compare Two Operands
	cmp	$*imm16,r/m*16	
	cmp	$*imm32,r/m32*	

	cmp	$*imm8,r/m16*
	cmp	$*imm8,r/m32*
	cmp	*r8,r/m8*
	cmp	*r16,r/m16*
	cmp	*r32,r/m32*
	cmp	*r/m8,r8*
	cmp	*r/m16,r16*
	cmp	*r/m32,r32*

cmps/cmpsb/cmpsw/cmpsd — Compare String Operands

	cmps	*m8,m8*
	cmps	*m16,m16*
	cmps	*m32,m32*
	cmpsb	
	cmpsw	
	cmpsd	
	(optional forms with segment override)	
	cmpsb	*%seg*:0(%esi),%es:0(%edi)
	cmpsw	*%seg*:0(%esi),%es:0(%edi)
	cmpsd	*%seg*:0(%esi),%es:0(%edi)

cmpxchg	cmpxchg	*r8,r/m8*	Compare and Exchange
	cmpxchg	*r16,r/m16*	(i486-specific)
	cmpxchg	*r32,r/m32*	

cmpxchg8b	cmpxchg8b	*m32*	Compare and Exchange 8 Bytes
			(Pentium-specific)

cpuid	cpuid		CPU Identification (Pentium-specific)
cwd/cdq	cwd cdq		Convert Word to Doubleword/ Convert Doubleword to Quadword
daa	daa		Decimal Adjust AL after Addition
das	das		Decimal Adjust AL after Subtraction
dec	dec dec dec dec dec	*r/m8* *r/m16* *r/m32* *r16* *r32*	Decrement by 1
div	div div div	*r/m8,***%al** *r/m16,***%ax** *r/m32,***%eax**	Unsigned Divide
enter	enter	$*imm16,*$*imm8*	Make Stack Frame for Procedure Parameters
f2xm1	f2xm1		Computer 2x-1

fabs	fabs		Absolute Value
fadd/faddp/fiadd			Add
	fadd	*m32real*	
	fadd	*m64real*	
	fadd	ST(i),ST	
	fadd	ST,ST(i)	
	faddp	ST,ST(i)	
	fadd		
	fiadd	*m32int*	
	fiadd	*m16int*	
fbld	fbld	*m80dec*	Load Binary Coded Decimal
fbstp	fbstp	*m80dec*	Store Binary Coded Decimal and Pop
fchs	fchs		Change Sign
fclex/fnclex			Clear Exceptions
	fclex		
	fnclex		
fcom/fcomp/fcompp			Compare Real
	fcom	*m32real*	
	fcom	*m64real*	
	fcom	ST(i)	

	fcom		
	fcomp	*m32real*	
	fcomp	*m64real*	
	fcomp	ST(i)	
	fcomp		
	fcompp		
fcos	fcos		Cosine
fdecstp	fdecstp		Decrement Stack-Top Pointer
fdiv/fdivp/fidiv			Divide
	fdiv	*m32real*	
	fdiv	*m64real*	
	fdiv	ST(i),ST	
	fdiv	ST,ST(i)	
	fdivp	ST,ST(i)	
	fdiv		
	fidiv	*m32int*	
	fidiv	*m16int*	
fdivr/fdivpr/fidivr			Reverse Divide
	fdivr	*m32real*	
	fdivr	*m64real*	
	fdivr	ST(i),ST	
	fdivr	ST,ST(i)	
	fdivrp	ST,ST(i)	
	fdivr		

	fidivr	*m32int*	
	fidivr	*m16int*	
ffree	ffree	ST(i)	Free Floating-Point Register
ficom/ficomp			Compare Integer
	ficom	*m16real*	
	ficom	*m32real*	
	ficomp	*m16int*	
	ficomp	*m32int*	
fild	filds	*m16int*	Load Integer
	fildl	*m32int*	
	fildq	*m64int*	
fincstp	fincstp		Increment Stack-Top Pointer
finit/fninit	finit		Initialize Floating-Point Unit
	fninit		
fist/fistp	fists	*m16int*	Store Integer
	fistl	*m32int*	
	fistps	*m16int*	
	fistpl	*m32int*	
	fistpq	*m64int*	
fld	flds	*m32real*	Load Real
	fldl	*m64real*	

	fldt	*m80real*	
	fld	ST(i)	
fld1/fldl2t/fldl2e/fldpi/fldlg2/gldln2/fldz			Load Constant
	fld1		
	fld2t		
	fld2e		
	fldpi		
	fldlg2		
	fldln2		
	fldz		
fldcw	fldcw	*m2byte*	Load Control Word
fldenv	fldenv	*m14/28byte*	Load FPU Environment
fmul/fmulp/fimul			Multiply
	fmul	*m32real*	
	fmul	*m64real*	
	fmul	ST(i),ST	
	fmul	ST(i),ST	
	fmulp	ST,ST(i)	
	fmul		
	fimul	*m32int*	
	fimul	*m16int*	
fnop	fnop		No Operation

fpatan	fpatan		Partial Arctangent
fprem	fprem		Partial Remainder
fprem1	fprem1		Partial Remainder
fptan	fptan		Partial Tangent
frndint	frndint		Round to Integer
frstor	frstor	*m94/108byte*	Restore FPU State
fsave/fnsave			Store FPU State
	fsave	*m94/108byte*	
	fnsave	*m94/108byte*	
fscale	fscale		Scale
fsin	fsin		Sine
fsincos	fsincos		Sine and Cosine
fsqrt	fsqrt		Square Root
fst/fstp	fst	*m32real*	Store Real
	fst	*m64real*	
	fst	ST(i)	
	fstp	*m32real*	

fstp *m64real*

fstp *m80real*

fstp ST(i)

fstcw/fnstcw — Store Control Word

fstcw *m2byte*

fnstcw *m2byte*

fstenv/fnstenv — Store FPU Environment

fstenv *m14/28byte*

fnstenv *m14/28byte*

fstsw/fnstsw — Store Status Word

fstsw *m2byte*

fstsw **%ax**

fnstsw *m2byte*

fnstsw **%ax**

fsub/fsubp/fisub — Subtract

fsub *m32real*

fsub *m64real*

fsub ST(i),ST

fsub ST,ST(i)

fsubp ST,ST(i)

fsub

fisub *m32int*

fisub *m16int*

fsubr/fsubpr/fisubr			Reverse Subtract
	fsubr	*m32real*	
	fsubr	*m64real*	
	fsubr	ST(i),ST	
	fsubr	ST,ST(i)	
	fsubpr	ST,ST(i)	
	fsubr		
	fisubr	*m32int*	
	fisubr	*m16int*	
ftst	ftst		Test
fucom/fucomp/fucompp			Unordered Compare Real
	fucom	ST(i)	
	fucom		
	fucomp	ST(i)	
	fucomp		
	fucompp		
fwait	fwait		Wait
fxam	fxam		Examine
fxch	fxch	ST(i)	Exchange Register Contents
	fxch		
fxtract	fxtract		Extract Exponent and Significand

fyl2x	fyl2x		Compute y $\times$ log2
fyl2xp1	fyl2xp1		Compute y $\times$ log2(x+1)
hlt	hlt		Halt
idiv	idiv	*r/m8*	Signed Divide
	idiv	*r/m16,***%ax**	
	idiv	*r/m32,***%eax**	
imul	imul	*r/m8*	Signed Multiply
	imul	*r/m16*	
	imul	*r/m32*	
	imul	*r/m16,r16*	
	imul	*r/m32,r32*	
	imul	$*imm8,r/m16,r16*	
	imul	$*imm8,r/m32,r32*	
	imul	$*imm8,r16*	
	imul	$*imm8,r32*	
	imul	$*imm16,r/m16,r16*	
	imul	$*imm32,r/m32,r32*	
	imul	$*imm16,r16*	
	imul	$*imm32,r32*	
in	in	$*imm8,***%al**	Input from Port
	in	$*imm8,***%ax**	
	in	$*imm8,***%eax**	

	in	**%dx**,**%al**	
	in	**%dx**,**%ax**	
	in	**%dx**,**%eax**	
inc	inc	*r/m8*	Increment by 1
	inc	*r/m16*	
	inc	*r/m32*	
	inc	*r16*	
	inc	*r32*	
ins/insb/insw/insd			Input from Port to String
	ins		
	insb		
	insw		
	insd		
int/into	int	3	Call to Interrupt Procedure
	int	$*imm8*	
	into		
invd	invd		Invalidate Cache (i486-specific)
invlpg	invlpg	m	Invalidate TLB Entry (i486-specific)
iret/iretd	iret		Interrupt Return
	iretd		

j*cc*			Jump if Condition is Met
	ja	*rel8*	short if above
	jae	*rel8*	short if above or equal
	jb	*rel8*	short if below
	jbe	*rel8*	short if below or equal
	jc	*rel8*	short if carry
	jcxz	*rel8*	short if **%cx** register is 0
	jecxz	*rel8*	short if **%ecx** register is 0
	je	*rel8*	short if equal
	jz	*rel8*	short if 0
	jg	*rel8*	short if greater
	jge	*rel8*	short if greater or equal
	jl	*rel8*	short if less
	jle	*rel8*	short if less or equal
	jna	*rel8*	short if not above
	jnae	*rel8*	short if not above or equal
	jnb	*rel8*	short if not below
	jnbe	*rel8*	short if not below or equal
	jnc	*rel8*	short if not carry
	jne	*rel8*	short if not equal
	jng	*rel8*	short if not greater
	jnge	*rel8*	short if not greater or equal
	jnl	*rel8*	short if not less
	jnle	*rel8*	short if not less or equal
	jno	*rel8*	short if not overflow
	jnp	*rel8*	short if not parity
	jns	*rel8*	short if not sign
	jnz	*rel8*	short if not 0

jo	*rel8*	short if overflow
jp	*rel8*	short if parity
jpe	*rel8*	short if parity even
jpo	*rel8*	short if parity odd
js	*rel8*	short if sign
jz	*rel8*	short if zero
ja	*rel16/32*	near if above
jae	*rel16/32*	near if above or equal
jb	*rel16/32*	near if below
jbe	*rel16/32*	near if below or equal
jc	*rel16/32*	near if carry
je	*rel16/32*	near if equal
jz	*rel16/32*	near if 0
jg	*rel16/32*	near if greater
jge	*rel16/32*	near if greater or equal
jl	*rel16/32*	near if less
jle	*rel16/32*	near if less or equal
jna	*rel16/32*	near if not above
jnae	*rel16/32*	near if not above or equal
jnb	*rel16/32*	near if not below
jnbe	*rel16/32*	near if not below or equal
jnc	*rel16/32*	near if not carry
jne	*rel16/32*	near if not equal
jng	*rel16/32*	near if not greater
jnge	*rel16/32*	near if not greater or less
jnl	*rel16/32*	near if not less
jnle	*rel16/32*	near if not less or equal
jno	*rel16/32*	near if not overflow

	jnp	*rel16/32*	near if not parity
	jns	*rel16/32*	near if not sign
	jnz	*rel16/32*	near if not 0
	jo	*rel16/32*	near if overflow
	jp	*rel16/32*	near if parity
	jpe	*rel16/32*	near if parity even
	jpo	*rel16/32*	near if parity odd
	js	*rel16/32*	near if sign
	jz	*rel16/32*	near if 0
jmp	jmp	*rel8*	Jump
	jmp	*rel16*	
	jmp	*r/m16*	
	jmp	*rel32*	
	jmp	*r/m32*	
	ljmp	$*imm16*,$*imm32*	
	ljmp	*m16*	
	ljmp	*m32*	
lahf	lahf		Load Flags into AH Register
lar	lar	*r/m16,r16*	Load Access Rights Byte
	lar	*r/m32,r32*	
lea	lea	*m,r16*	Load Effective Address
	lea	*m,r32*	
leave	leave		High Level Procedure Exit

lgdt/lidt	lgdt	*m16&32*	Load Global/Interrupt
	lidt	*m16&32*	Descriptor Table Register
lgs/lss/lds/les/lfs			Load Full Pointer
	lgs	*m16:16,r16*	
	lgs	*m16:32,r32*	
	lss	*m16:16,r16*	
	lss	*m16:32,r32*	
	lds	*m16:16,r16*	
	lds	*m16:32,r32*	
	les	*m16:16,r16*	
	les	*m16:32,r32*	
	lfs	*m16:16,r16*	
	lfs	*m16:32,r32*	
lldt	lldt	*r/m16*	Load Local Descriptor Table Register
lmsw	lmsw	*r/m16*	Load Machine Status Word
lock	lock		Assert LOCK# Signal Prefix
lods/lodsb/lodsw/lodsd			Load String Operand
	lods	*m8*	
	lods	*m16*	
	lods	*m32*	
	lodsb		

	lodsw		
	lodsd		
	(optional forms with segment override)		
	lodsb	*%seg*:**0(%esi),%al**	
	lodsw	*%seg*:**0(%esi),%al**	
	lodsd	*%seg*:**0(%esi),%al**	
loop/loop*cond*			Loop Control with CX Counter
	loop	*rel8*	
	loope	*rel8*	
	loopz	*rel8*	
	loopne	*rel8*	
	loopnz	*rel8*	
lsl	lsl	*r/m16,r16*	Load Segment Limit
	lsl	*r/m32,r32*	
ltr	ltr	*r/m16*	Load Task Register
mov	mov	*r8,r/m8*	Move Data
	mov	*r16,r/m16*	
	mov	*r32,r/m32*	
	mov	*r/m8,r8*	
	mov	*r/m16,r16*	
	mov	*r/m16,r16*	
	mov	*Sreg,r/m16*	
	mov	*r/m16,Sreg*	

	mov	*moffs8,***%al**	
	mov	*moffs8,***%ax**	
	mov	*moffs8,***%eax**	
	mov	**%al**,*moffs8*	
	mov	**%ax**,*moffs16*	
	mov	**%eax**,*moffs32*	
	mov	$*imm8,reg8*	
	mov	$*imm16,reg*16	
	mov	$*imm32,reg32*	
	mov	$*imm8,r/m8*	
	mov	$*imm16,r/m*16	
	mov	$*imm32,r/m32*	
mov	mov	*r32,***%cr0**	Move to/from Special Registers
	mov	**%cr0**/**%cr2**/**%cr3**,*r32*	
	mov	**%cr2**/**%cr3**,*r32*	
	mov	**%dr0-3**,*r32*	
	mov	**%dr6**/**%dr7**,*r32*	
	mov	*r32,***%dr0-3**	
	mov	*r32,***%dr6**/**%dr7**	
	mov	**%tr4**/**%tr5**/**%tr6**/**%tr7**,*r32*	
	mov	*r32***,%tr4**/**%tr5**/**%tr6**/**%tr7**	
	mov	**%tr3**,*r32*	
	mov	*r32***,%tr3**	
movs/movsb/movsw/movsd			Move Data from String to String
	movs	*m8,m8*	
	movs	*m16,m16*	

	movs	*m32,m32*	
	movsb		
	movsw		
	movsd		
	(optional forms with segment override)		
	movsb	***%seg*:0(%esi),%es:0(%edi)**	
	movsw	***%seg*:0(%esi),%es:0(%edi)**	
	movsd	***%seg*:0(%esi),%es:0(%edi)**	
movsx	movsx	*r/m8,r16*	Move with Sign-Extend
	movsx	*r/m8,r32*	
	movsx	*r/m16,r32*	
movzx	movzx	*r/m8,r16*	Move with Zero-Extend
	movzx	*r/m8,r32*	
	movzx	*r/m16,r32*	
mul	mul	*r/m8,***%al**	Unsigned Multiplication of AL
	mul	*r/m16,***%ax**	or AX
	mul	*r/m32,***%eax**	
neg	neg	*r/m8*	Two's Complement Negation
	neg	*r/m16*	
	neg	*r/m32*	
nop	nop		No Operation

not	not	*r/m8*	One’s Complement Negation
	not	*r/m16*	
	not	*r/m32*	
or	or	$*imm8,r/m8*	Logical Inclusive OR
	or	$*imm16,r/m*16	
	or	$*imm32,r/m32*	
	or	$*imm8,r/m16*	
	or	$*imm8,r/m32*	
	or	*r8,r/m8*	
	or	*r16,r/m16*	
	or	*r32,r/m32*	
	or	*r/m8,r8*	
	or	*r/m16,r16*	
	or	*r/m32,r32*	
out	out	**%al**,$*imm8*	Output to Port
	out	**%ax**,$*imm8*	
	out	**%eax**,$*imm8*	
	out	**%al**,**%dx**	
	out	**%ax**,**%dx**	
	out	**%eax**,**%dx**	
outs/outsb/outsw/outsd			Output String to Port
	outs	*r/m8,***%dx**	
	outs	*r/m16,***%dx**	
	outs	*r/m32,***%dx**	
	outsb		

	outsw		
	outsd		
pop	pop	*m16*	Pop a Word from the Stack
	pop	*m32*	
	pop	*r16*	
	pop	*r32*	
	pop	**%ds**	
	pop	**%es**	
	pop	**%ss**	
	pop	**%fs**	
	pop	**%gs**	
popa/popad			Pop all General Registers
	popa		
	popad		
popf/popfd	popf		Pop Stack into FLAGS or
	popfd		EFLAGS Register
push	push	*m16*	Push Operand onto the Stack
	push	*m32*	
	push	*r16*	
	push	*r32*	
	push	$*imm8*	
	push	$*imm16*	
	push	$*imm32*	
	push	*Sreg*	

pusha/pushad — Push all General Registers

pusha

pushad

pushf/pushfd — Push Flags Register onto the Stack

pushf

pushfd

rcl/rcr/rol/ror — Rotate

rcl 1,*r/m8*

rcl **%cl**,*r/m8*

rcl $*imm8,r/m8*

rcl 1,*r/m16*

rcl **%cl**,*r/m16*

rcl $*imm8,r/m16*

rcl 1,*r/m32*

rcl **%cl**,*r/m32*

rcl $*imm8,r/m32*

rcr 1,*r/m8*

rcr **%cl**,*r/m8*

rcr $*imm8,r/m8*

rcr 1,*r/m16*

rcr **%cl**,*r/m16*

rcr $*imm8,r/m16*

rcr 1,*r/m32*

rcr **%cl**,*r/m32*

rcr $*imm8,r/m32*

	rol	1,*r/m8*	
	rol	**%cl**,*r/m8*	
	rol	$*imm8,r/m8*	
	rol	1,*r/m16*	
	rol	**%cl**,*r/m16*	
	rol	$*imm8,r/m16*	
	rol	1,*r/m32*	
	rol	**%cl**,*r/m32*	
	rol	$*imm8,r/m32*	
	ror	1,*r/m8*	
	ror	**%cl**,*r/m8*	
	ror	$*imm8,r/m8*	
	ror	1,*r/m16*	
	ror	**%cl**,*r/m16*	
	ror	$*imm8,r/m16*	
	ror	1,*r/m32*	
	ror	**%cl**,*r/m32*	
	ror	$*imm8,r/m32*	
rdmsr	rdmsr		Read from Model-Specific Register (Pentium-specific)
rdstc	rdstc		Read from Time Stamp Counter (Pentium-specific)
rep/repe/repz/repne/repnz			Repeat Following String
	rep ins	**%dx**,*rm8*	Operation
	rep ins	**%dx**,*rm16*	

	rep ins	**%dx**,*rm32*	
	rep movs	*m8,m8*	
	rep movs	*m16,m16*	
	rep movs	*m32,m32*	
	rep outs	*rm8,***%dx**	
	rep outs	*rm16,***%dx**	
	rep outs	*rm32,***%dx**	
	rep lods	*m8*	
	rep lods	*m16*	
	rep lods	*m32*	
	rep stos	*m8*	
	rep stos	*m16*	
	rep stos	*m32*	
	repe cmps	*m8,m8*	
	repe cmps	*m16,m16*	
	repe cmps	*m32,m32*	
	repe scas	*m8*	
	repe scas	*m16*	
	repe scas	*m32*	
	repne cmps	*m8,m8*	
	repne cmps	*m16,m16*	
	repne cmps	*m32,m32*	
	repne scas	*m8*	
	repne scas	*m16*	
	repne scas	*m32*	
ret	ret		Return from Procedure
	ret	$*imm16*	

rsm	rsm		Resume from System-Management Mode (Pentium-specific)
sahf	sahf		Store AH into Flags
sal/sar/shl/shr			Shift Instructions
	sal	1,*r/m8*	
	sal	**%cl**,*r/m8*	
	sal	$*imm8,r/m8*	
	sal	1,*r/m16*	
	sal	**%cl**,*r/m16*	
	sal	$*imm8,r/m16*	
	sal	1,*r/m32*	
	sal	**%cl**,*r/m32*	
	sal	$*imm8,r/m32*	
	sar	1,*r/m8*	
	sar	**%cl**,*r/m8*	
	sar	$*imm8,r/m8*	
	sar	1,*r/m16*	
	sar	**%cl**,*r/m16*	
	sar	$*imm8,r/m16*	
	sar	1,*r/m32*	
	sar	**%cl**,*r/m32*	
	sar	$*imm8,r/m32*	
	shl	1,*r/m8*	
	shl	**%cl**,*r/m8*	

	shl	$*imm8,r/m8*	
	shl	1,*r/m16*	
	shl	**%cl**,*r/m16*	
	shl	$*imm8,r/m16*	
	shl	1,r/m32	
	shl	%cl,r/m32	
	shl	$imm8,r/m32	
	shr	1,r/m8	
	shr	%cl,r/m8	
	shr	$imm8,r/m8	
	shr	1,r/m16	
	shr	%cl,r/m16	
	shr	$imm8,r/m16	
	shr	1,r/m32	
	shr	%cl,r/m32	
	shr	$imm8,r/m32	
sbb	sbb	$imm8,r/m8	Integer Subtraction with Borrow
sbb	$imm16,r/m16		
	sbb	$imm32,r/m32	
	sbb	$imm8,r/m16	
	sbb	$imm8,r/m32	
	sbb	r8,r/m8	
	sbb	r16,r/m16	
	sbb	r32,r/m32	
	sbb	r/m8,r8	
	sbb	r/m16,r16	
	sbb	r/m32,r32	

scas/scasb/scasw/scasd Compare String Data

scas	m8	
scas	m16	
scas	m32	
scasb		
scasw		
scasd		

(optional forms with segment override)

scasb	%al,%seg:0(%edi)	
scasw	%ax,%seg:0(%edi)	
scasd	%eax,%seg:0(%edi)	

setcc Byte Set on Condition

seta	r/m8	above
setae	r/m8	above or equal
setb	r/m8	below
setbe	r/m8	below or equal
setc	r/m8	carry
sete	r/m8	equal
setg	r/m8	greater
setge	r/m8	greater or equal
setl	r/m8	less
setle	r/m8	less or equal
setna	r/m8	not above
setnae	r/m8	not abover or equal
setnb	r/m8	not below

	setnbe	r/m8	not below or equal
	setnc	r/m8	not carry
	setne	r/m8	not equal
	setng	r/m8	not greater
	setnge	r/m8	not greater or equal
	setnl	r/m8	not less
	setnle	r/m8	not less or equal
	setno	r/m8	not overflow
	setnp	r/m8	not parity
	setns	r/m8	not sign
	setnz	r/m8	not zero
	seto	r/m8	overflow
	setp	r/m8	parity
	setpe	r/m8	parity even
	setpo	r/m8	parity odd
	sets	r/m8	sign
	setz	r/m8	zero
sgdt/sidt	sgdt	m	Store Global/Interrupt
	sidt	m	Descriptor Table Register
shld	shld	$imm8,r16,r/m16	Double Precision Shift Left
	shld	$imm8,r32,r/m32	
	shld	%cl,r16,r/m16	
	shld	%cl,r32,r/m32	
shrd	shrd	$imm8,r16,r/m16	Double Precision Shift Right
	shrd	$imm8,r32,r/m32	

	shrd	%cl,r16,r/m16	
	shrd	%cl,r32,r/m32	
sldt	sldt	r/m16	Store Local Descriptor Table
Register			
smsw	smsw	r/m16	Store Machine Status Word
stc	stc		Set Carry Flag
std	std		Set Direction Flag
sti	sti		Set Interrupt Flag
stos/stosb/stosw/stosd			Store String Data
	stos	m8	
	stos	m16	
	stos	m32	
	stosb		
	stosw		
	stosd		
(optional forms with segment override)			
	stosb	%al,%seg:0(%edi)	
	stosw	%ax,%seg:0(%edi)	
	stosd	%eax,%seg:0(%edi)	
str	str	r/m16	Store Task Register

sub	sub	$imm8,r/m8	Integer Subtraction
	sub	$imm16,r/m16	
	sub	$imm32,r/m32	
	sub	$imm8,r/m16	
	sub	$imm8,r/m32	
	sub	r8,r/m8	
	sub	r16,r/m16	
	sub	r32,r/m32	
	sub	r/m8,r8	
	sub	r/m16,r16	
	sub	r/m32,r32	
test	test	$imm8,r/m8	Logical Compare
	test	$imm16,r/m16	
	test	$imm32,r/m32	
	test	r8,r/m8	
	test	r16,r/m16	
	test	r32,r/m32	
verr, verw	verr	r/m16	Verify a Segment for Reading or
	verw	r/m16	Writing
wait	wait		Wait
wbinvd	wbinvd		Write-Back and Invalidate
			Cache (i486-specific)
wrmsr	wrmsr		Write to Model-Specific

Register (Pentium-specific)

xadd	xadd	r8,r/m8	Exchange and Add
	xadd	r16,r/m16	(i486-specific)
	xadd	r32,r/m32	
xchg	xchg	r16,%ax	Exchange Register/Memory
	xchg	%ax,r16	with Register
	xchg	%eax,r32	
	xchg	r32,%eax	
	xchg	r8,r/m8	
	xchg	r/m8,r8	
	xchg	r16,r/m16	
	xchg	r/m16,r16	
	xchg	r32,r/m32	
	xchg	r/m32,r32	
xlat/xlatb	xlat	m8	Table Look-up Translation
	xlatb		
xor	xor	$imm8,r/m8	Logical Exclusive OR
	xor	$imm16,r/m16	
	xor	$imm32,r/m32	
	xor	$imm8,r/m16	
	xor	$imm8,r/m32	
	xor	r8,r/m8	
	xor	r16,r/m16	

```
xor     r32,r/m32
xor     r/m8,r8
xor     r/m16,r16
xor     r/m32,r32
```