3

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

M68000 (including MC68882)

- Many M68000 machine instructions can operate on byte, word, or long word data. The desired size is indicated as part of the instruction mnemonic by adding a trailing **b**, **w**, or **l**:

  b byte (8-bit) data

  w word (16-bit) data

  l long word (32-bit) data

  For instance, a **movb** instruction moves a byte of data, but a **movw** instruction moves a 16-bit word of data. In general, the default size for data manipulation instructions on the 68030 and 68040 processors is 16-bit word.

- Many 68882 instructions (as well as built-in floating-point instructions on the 68040) can operate on other types of data besides byte, word, or long word integer data. Again, the size required is specified as part of the instruction mnemonic by a trailing letter:

  s single-precision (32-bit) floating-point data

  d double-precision (64-bit) floating-point data

  x extended-precision (96-bit) floating-point data

  p packed decimal (96-bit) floating-point data (note that the assembler currently doesn't support packed immediate formats)

Intel i386 Architecture

- As with the Motorola 68000 family, i386 instructions can operate on byte, word, or long word data (the last is called ªdouble wordº 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 ª.varº 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 redefinedÐits 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.