2

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 programÐthese are characters, symbols, labels, and constants.

Characters

The following characters are used in assembly language programs

- alphanumeric charactersÐ`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 ªExpression Syntaxº 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, ªLabelsº)
- 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 significantÐfor 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 bitsÐhaving 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 ª.º
[*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}
M88000	{rRdDfF}	{eE}
M98000	{dDfF}	{eE}
i386	{fFdDxX}	{eE}
i860	{rRsSfFdDxXpP}	{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 ªdot,º 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 expressionsÐfor example, to differentiate between **x * y + z** and **x * (y + z)** or to apply a unary operator to an entire expressionÐfor 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 sectionÐthat 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.