3

Assembly Language Fundamentals

3.1 Basic Elements of Assembly Language

Chapter 1 introduced number concepts and virtual machines. Chapter 2 introduced hardware basics. Now you're ready to begin programming. There is an element of truth in saying "*Assembly language is simple.*" It was designed to run in little memory and consists of mainly low-level,

simple operations. Then why does it have the reputation of being difficult to learn? After all, how hard can it be to move data between registers and do a calculation? Here's a proof of concept—a simple program in assembly language that adds two numbers and displays the result:

```
main PROC
     mov   eax,5                ; move 5 to the EAX register
     add   eax,6                ; add 6 to the EAX register
     call  WriteInt             ; display value in EAX
     exit                       ; quit
main ENDP
```

We simplified things a bit by calling a library subroutine named **WriteInt**, which itself contains a fair amount of code. But in general, assembly language is not hard to learn if you're happy writing short programs that do practically nothing.

Details, Details Becoming a skilled assembly language programmer requires a love of details. You will find it helpful to build a foundation of basic information and gradually fill in the details. If you were a cook, we would show you around the kitchen and explain how to use mixers, grinders, knives, stoves, and saucepans. Similarly, we will identify the ingredients of assembly language, mix them together, and cook up a few tasty programs.

Next, we describe basic elements of Microsoft Macro Assembler (MASM) syntax. Knowing these elements will help you to write your first programs in assembly language.

3.1.1 Integer Constants

An *integer constant* (or integer literal) is made up of an optional leading sign, one or more digits, and an optional suffix character (called a *radix*) indicating the number's base:

```
[{+|-}] digits [radix]
```

> Microsoft syntax notation is used throughout this chapter. Elements within square brackets [..] are optional and elements within braces {..} require a choice of one of the enclosed elements (separated by the | character). Elements in *italics* denote items that have known definitions or descriptions.

Radix may be one of the following (uppercase or lowercase):

h	Hexadecimal	r	Encoded real
q/o	Octal	t	Decimal *(alternate)*
d	Decimal	y	Binary *(alternate)*
b	Binary		

If no radix is given, the integer constant is assumed to be decimal. Here are some examples using different radixes:

26	Decimal	42o	Octal
26d	Decimal	1Ah	Hexadecimal
11010011b	Binary	0A3h	Hexadecimal
42q	Octal		

A hexadecimal constant beginning with a letter must have a leading zero to prevent the assembler from interpreting it as an identifier.

3.1.2 Integer Expressions

An *integer expression* is a mathematical expression involving integer values and arithmetic operators. The expression must evaluate to an integer, which can be stored in 32 bits (0 through FFFFFFFFh). The arithmetic operators are listed in Table 3-1 according to their precedence order, from highest (1) to lowest (4). The important thing to realize about integer expressions is that they can only be evaluated at assembly time. These are not runtime expressions.

TABLE 3-1 Arithmetic Operators.

Operator	Name	Precedence Level
( )	Parentheses	1
+, −	Unary plus, minus	2
*, /	Multiply, divide	3
MOD	Modulus	3
+, −	Add, subtract	4

Precedence refers to the implied order of operations when an expression contains two or more operators. The order of operations is shown for the following expressions:

```
4 + 5 * 2                    Multiply, add
12 - 1 MOD 5                 Modulus, subtract
-5 + 2                       Unary minus, add
(4 + 2) * 6                  Add, multiply
```

The following are examples of valid expressions and their values:

Expression	Value
16 / 5	3
−(3 + 4) * (6 − 1)	−35
−3 + 4 * 6 − 1	20
25 mod 3	1

Use parentheses in expressions to clarify the order of operations so you don't have to remember precedence rules.

3.1.3 Real Number Constants

Real number constants are represented as decimal reals or encoded (hexadecimal) reals. A *decimal real* contains an optional sign followed by an integer, a decimal point, an optional integer that expresses a fraction, and an optional exponent:

```
[sign]integer.[integer][exponent]
```

Following are the syntax for the sign and exponent:

```
sign       {+,-}
exponent   E[{+,-}]integer
```

Following are examples of valid real number constants:

```
2.
+3.0
-44.2E+05
26.E5
```

At least one digit and a decimal point are required.

Encoded Reals An encoded real represents a real number in hexadecimal, using the IEEE floating-point format for short reals (see Chapter 12). The binary representation of decimal +1.0, for example, is

```
0011 1111 1000 0000 0000 0000 0000 0000
```

The same value would be encoded as a short real in assembly language as

```
3F800000r
```

3.1.4 Character Constants

A *character constant* is a single character enclosed in single or double quotes. MASM stores the value in memory as the character's binary ASCII code. Examples are

```
'A'
"d"
```

A complete list of ASCII codes is printed on the inside back cover of this book.

3.1.5 String Constants

A *string constant* is a sequence of characters (including spaces) enclosed in single or double quotes:

```
'ABC'
'X'
"Good night, Gracie"
'4096'
```

Embedded quotes are permitted when used in the manner shown by the following examples:

```
"This isn't a test"
'Say "Good night," Gracie'
```

3.1.6 Reserved Words

Reserved words have special meaning in MASM and can only be used in their correct context. There are different types of reserved words:

- Instruction mnemonics, such as MOV, ADD, and MUL
- Register names
- Directives, which tell MASM how to assemble programs
- Attributes, which provide size and usage information for variables and operands. Examples are BYTE and WORD
- Operators, used in constant expressions
- Predefined symbols, such as @data, which return constant integer values at assembly time

A common list of MASM reserved words can be found in Appendix A.

3.1.7 Identifiers

An *identifier* is a programmer-chosen name. It might identify a variable, a constant, a procedure, or a code label. Keep the following in mind when creating identifiers:

- They may contain between 1 and 247 characters.
- They are not case sensitive.
- The first character must be a letter (A..Z, a..z), underscore (_), @ , ?, or $. Subsequent characters may also be digits.
- An identifier cannot be the same as an assembler reserved word.

> You can make all keywords and identifiers case sensitive by adding the –Cp command line switch when running the assembler.

The @ symbol is used extensively by the assembler as a prefix for predefined symbols, so avoid it in your own identifiers. Make identifier names descriptive and easy to understand. Here are some valid identifiers:

```
var1          Count          $first
_main         MAX            open_file
myFile        xVal           _12345
```

3.1.8 Directives

A *directive* is a command embedded in the source code that is recognized and acted upon by the assembler. Directives do not execute at runtime. Directives can define variables, macros, and procedures. They can assign names to memory segments and perform many other housekeeping tasks related to the assembler. In MASM, directives are case insensitive. For example, it recognizes **.data**, **.DATA**, and **.Data** as equivalent.

The following example helps to show the difference between directives and instructions. The DWORD directive tells the assembler to reserve space in the program for a doubleword variable. The MOV instruction, on the other hand, executes at runtime, copying the contents of **myVar** to the EAX register:

```
myVar  DWORD 26                    ; DWORD directive
mov    eax,myVar                   ; MOV instruction
```

Although all assemblers for Intel processors share the same instruction set, they have completely different sets of directives. The Microsoft assembler's REPT directive, for example, is not recognized by some other assemblers.

Defining Segments One important function of assembler directives is to define program sections, or *segments*. The .DATA directive identifies the area of a program containing variables:

```
.data
```

The .CODE directive identifies the area of a program containing executable instructions:

```
.code
```

The .STACK directive identifies the area of a program holding the runtime stack, setting its size:

```
.stack 100h
```

Appendix A contains a useful reference for MASM directives and operators.

3.1.9 Instructions

An *instruction* is a statement that becomes executable when a program is assembled. Instructions are translated by the assembler into machine language bytes, which are loaded and executed by the CPU at runtime. An instruction contains four basic parts:

- Label (optional)
- Instruction mnemonic (required)
- Operand(s) (usually required)
- Comment (optional)

This is the basic syntax:

```
[label:] mnemonic [operands] [;comment]
```

Let's explore each part separately, beginning with the *label* field.

Label

A *label* is an identifier that acts as a place marker for instructions and data. A label placed just before an instruction implies the instruction's address. Similarly, a label placed just before a variable implies the variable's address.

Data Labels A data label identifies the location of a variable, providing a convenient way to reference the variable in code. The following, for example, defines a variable named count:

```
count  DWORD 100
```

The assembler assigns a numeric address to each label. It is possible to define multiple data items following a label. In the following example, array defines the location of the first number (1024). The other numbers following in memory immediately afterward:

```
array  DWORD 1024, 2048
       DWORD 4096, 8192
```

Variables will be explained in Section 3.4.2, and the MOV instruction will be explained in Section 4.1.4.

Code Labels A label in the code area of a program (where instructions are located) must end with a colon (:) character. Code labels are used as targets of jumping and looping instructions. For example, the following JMP (jump) instruction transfers control to the location marked by the label named **target**, creating a loop:

```
target:
     mov   ax,bx
     ...
     jmp   target
```

A code label can share the same line with an instruction, or it can be on a line by itself:

```
L1:  mov   ax,bx
L2:
```

Label names are created using the rules for identifiers discussed in Section 3.1.7. You can use the same code label more than once in a program as long as each label is unique within its enclosing procedure. (A procedure is like a function.)

Instruction Mnemonic

An *instruction mnemonic* is a short word that identifies an instruction. In English, a *mnemonic* is a device that assists memory. Similarly, assembly language instruction mnemonics such as mov, add, and sub provide hints about the type of operation they perform. Following are examples of instruction mnemonics:

```
mov     Move (assign) one value to another
add     Add two values
sub     Subtract one value from another
mul     Multiply two values
jmp     Jump to a new location
call    Call a procedure
```

Operands Assembly language instructions can have between zero and three operands, each of which can be a register, memory operand, constant expression, or input-output port. We discussed register names in Chapter 2, and we discussed constant expressions in Section 3.1.2. A *memory operand* is specified by the name of a variable or by one or more registers containing the address of a variable. A variable name implies the address of the variable and instructs the computer to reference the contents of memory at the given address. The following table contains several sample operands:

Example	Operand Type
96	Constant (*immediate value*)
2 + 4	Constant expression
eax	Register
count	Memory

Following are examples of assembly language instructions having varying numbers of operands. The STC instruction, for example, has no operands:

```
stc                          ; set Carry flag
```

The INC instruction has one operand:

```
inc  eax                    ; add 1 to EAX
```

The MOV instruction has two operands:

```
mov count,ebx               ; move EBX to count
```

In a two-operand instruction, the first operand is called the *destination*. The second operand is the *source*. In general, the contents of the destination operand are modified by the instruction. In a MOV instruction, for example, data is copied from the source to the destination.

The IMUL instruction has 3 operands, in which the first operand is the destination, and the following 2 operands are source operands:

```
imul eax,ebx,5
```

In this case, EBX is multiplied by 5, and the product is stored in the EAX register.

Comments

Comments are an important way for the writer of a program to communicate information about the program's design to a person reading the source code. The following information is typically included at the top of a program listing:

- Description of the program's purpose
- Names of persons who created and/or revised the program
- Program creation and revision dates
- Technical notes about the program's implementation

Comments can be specified in two ways:

- Single-line comments, beginning with a semicolon character (;). All characters following the semicolon on the same line are ignored by the assembler.
- Block comments, beginning with the COMMENT directive and a user-specified symbol. All subsequent lines of text are ignored by the assembler until the same user-specified symbol appears. For example,

```
COMMENT  !
     This line is a comment.
     This line is also a comment.
!
```

We can also use any other symbol:

```
COMMENT &
    This line is a comment.
    This line is also a comment.
&
```

Of course, it is important to provide comments throughout your program, particularly where the intent of your code is not obvious.

3.1.10 The NOP (No Operation) Instruction

The safest (and the most useless) instruction you can write is called NOP (no operation). It takes up 1 byte of program storage and doesn't do any work. It is sometimes used by compilers and

assemblers to align code to even-address boundaries. In the following example, the first MOV instruction generates three machine code bytes. The NOP instruction aligns the address of the third instruction to a doubleword boundary (even multiple of 4):

```
00000000  66 8B C3    mov ax,bx
00000003  90          nop               ; align next instruction
00000004  8B D1       mov edx,ecx
```

x86 processors are designed to load code and data more quickly from even doubleword addresses.

3.1.11 Section Review

1. Identify valid suffix characters used in integer constants.
2. *(Yes/No):* Is A5h a valid hexadecimal constant?
3. *(Yes/No):* Does the multiplication operator (*) have a higher precedence than the division operator (/) in integer expressions?
4. Write a constant expression that divides 10 by 3 and returns the integer remainder.
5. Show an example of a valid real number constant with an exponent.
6. *(Yes/No):* Must string constants be enclosed in single quotes?
7. Reserved words can be instruction mnemonics, attributes, operators, predefined symbols, and _____.
8. What is the maximum length of an identifier?
9. *(True/False):* An identifier cannot begin with a numeric digit.
10. *(True/False):* Assembly language identifiers are (by default) case insensitive.
11. *(True/False):* Assembler directives execute at runtime.
12. *(True/False):* Assembler directives can be written in any combination of uppercase and lowercase letters.
13. Name the four basic parts of an assembly language instruction.
14. *(True/False):* MOV is an example of an instruction mnemonic.
15. *(True/False):* A code label is followed by a colon (:), but a data label does not have a colon.
16. Show an example of a block comment.
17. Why would it not be a good idea to use numeric addresses when writing instructions that access variables?

3.2 Example: Adding and Subtracting Integers

We now introduce a short assembly language program that adds and subtracts integers. Registers are used to hold the intermediate data, and we call a library subroutine to display the contents of the registers on the screen. Here is the program source code:

```
TITLE Add and Subtract            (AddSub.asm)

; This program adds and subtracts 32-bit integers.

INCLUDE Irvine32.inc
```

```
.code
main PROC

    mov   eax,10000h          ; EAX = 10000h
    add   eax,40000h          ; EAX = 50000h
    sub   eax,20000h          ; EAX = 30000h
    call  DumpRegs            ; display registers

    exit
main ENDP
END main
```

Let's go through the program line by line. Each line of program code will appear before its explanation.

```
TITLE Add and Subtract            (AddSub.asm)
```

The TITLE directive marks the entire line as a comment. You can put anything you want on this line.

```
; This program adds and subtracts 32-bit integers.
```

All text to the right of a semicolon is ignored by the assembler, so we use it for comments.

```
INCLUDE Irvine32.inc
```

The INCLUDE directive copies necessary definitions and setup information from a text file named *Irvine32.inc*, located in the assembler's INCLUDE directory. (The file is described in Chapter 5.)

```
.code
```

The **.code** directive marks the beginning of the *code segment*, where all executable statements in a program are located.

```
main PROC
```

The PROC directive identifies the beginning of a procedure. The name chosen for the only procedure in our program is **main**.

```
mov   eax,10000h              ; EAX = 10000h
```

The MOV instruction moves (copies) the integer 10000h to the EAX register. The first operand (EAX) is called the *destination operand,* and the second operand is called the *source operand.* The comment on the right side shows the expected new value in the EAX register.

```
add   eax,40000h              ; EAX = 50000h
```

The ADD instruction adds 40000h to the EAX register. The comment shows the expected new value in EAX.

```
sub   eax,20000h              ; EAX = 30000h
```

The SUB instruction subtracts 20000h from the EAX register.

```
call DumpRegs                 ; display registers
```