Lecture No.5

Lecture Outlines

3.1 Basic Language Elements

3.1.1 First Assembly Language Program

Assembly language programming might have a reputation for being obscure and tricky, but we like to think of it another way—it is a language that gives you nearly total information. You get to see everything that is going on, even in the CPU's registers and flags! With this powerful ability, however, you have the responsibility to manage data representation details and instruction formats. You work at a very detailed level. To see how this works, let's look at a simple assembly language program that adds two numbers and saves the result in a register. We will call it the *AddTwo* program:

```
1: main PROC
2:    mov eax,5                ; move 5 to the eax register
3:    add eax,6                ; add 6 to the eax register
4:
5:    INVOKE ExitProcess,0     ; end the program
6: main ENDP
```

Although line numbers have been inserted in the beginning of each line to aid our discussion, you never actually type line numbers when you create assembly programs. Also, don't try to type in and run this program just yet—it's missing some important declarations that we will include later on in this chapter.

Let's go through the program one line at a time: Line 1 starts the **main** procedure, the entry point for the program. Line 2 places the integer 5 in the **eax** register. Line 3 adds 6 to the value in EAX, giving it a new value of 11. Line 5 calls a Windows service (also known as a function) named **ExitProcess** that halts the program and returns control to the operating system. Line 6 is the ending marker of the main procedure.

You probably noticed that we included comments, which always begin with a semicolon character. We've left out a few declarations at the top of the program that we can show later, but essentially this is a working program. It does not display anything on the screen, but we could run it with a utility program called a *debugger* that would let us step through the program one line at a time and look at the register values. Later in this chapter, we will show how to do that.

Adding a Variable

Let's make our program a little more interesting by saving the results of our addition in a variable named sum. To do this, we will add a couple of markers, or declarations, that identify the code and data areas of the program:

```
 1: .data                    ; this is the data area
 2: sum DWORD 0              ; create a variable named sum
 3:
 4: .code                    ; this is the code area
 5: main PROC
 6:    mov eax,5             ; move 5 to the eax register
 7:    add eax,6             ; add 6 to the eax register
 8:    mov sum,eax
 9:
10:    INVOKE ExitProcess,0  ; end the program
11: main ENDP
```

The **sum** variable is declared on Line 2, where we give it a size of 32 bits, using the DWORD keyword. There are a number of these size keywords, which work more or less like data types. But they are not as specific as types you might be familiar with, such as int, double, float, and so on. They only specify a size, but there's no checking into what actually gets put inside the variable. Remember, you are in total control.

By the way, those code and data areas we mentioned, which were marked by the .code and .data directives, are called *segments*. So you have the code segment and the data segment. Later on, we will see a third segment named **stack**.

Next, let's dive deeper into some of the language details, showing how to declare literals (also known as constants), identifiers, directives, and instructions. You will probably have to read this chapter a couple of times to retain it all, but it's definitely worth the time. By the way, throughout

this chapter, when we refer to syntax rules imposed by the assembler, we really mean rules imposed by the Microsoft MASM assembler. Other assemblers are out there with different syntax rules, but we will ignore them. We will probably save at least one tree (somewhere in the world) by not reprinting the word MASM every time we refer to the assembler.

3.1.2 Integer Literals

An *integer literal* (also known as an *integer constant*) is made up of an optional leading sign, one or more digits, and an optional radix character that indicates the number's base:

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

> We will use Microsoft syntax notation throughout the book. 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* identify items that have known definitions or descriptions.

So, for example, 26 is a valid integer literal. It doesn't have a radix, so we assume it's in decimal format. If we wanted it to be 26 hexadecimal, we would have to write it as 26h. Similarly, the number 1101 would be considered a decimal value until we added a "b" at the end to make it 1101b (binary). Here are the possible radix values:

h	hexadecimal	r	encoded real
q/o	octal	t	decimal (alternate)
d	decimal	y	binary (alternate)
b	binary		

And here are some integer literals declared with various radixes. Each line contains a comment:

```
26                      ; decimal
26d                     ; decimal
11010011b               ; binary
42q                     ; octal
42o                     ; octal
1Ah                     ; hexadecimal
0A3h                    ; hexadecimal
```

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

3.1.3 Constant Integer Expressions

A *constant integer expression* is a mathematical expression involving integer literals and arithmetic operators. Each 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 constant integer expressions is that they can only be evaluated at assembly time. From now on, we will just call them *integer 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

Operator 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

Expression	Value
−3 + 4 * 6 − 1	20
25 mod 3	1

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

3.1.4 Real Number Literals

Real number literals (also known as *floating-point literals*) are represented as either 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]
```

These are the formats for the sign and exponent:

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

Following are examples of valid decimal reals:

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

At least one digit and a decimal point are required.

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
```

We will not be using real-number constants for a while, because most of the x86 instruction set is geared toward integer processing. However, Chapter 12 will show how to do arithmetic with real numbers, also known as floating-point numbers. It's very interesting, and very technical.

3.1.5 Character Literals

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

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

Recall that Chapter 1 showed that character literals are stored internally as integers, using the ASCII encoding sequence. So, when you write the character constant "A," it's stored in memory as the number 65 (or 41 hex). We have a complete table of ASCII codes on the inside back cover of this book, so be sure to look over them from time to time.

3.1.6 String Literals

A *string literal* 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'
```

Just as character constants are stored as integers, we can say that string literals are stored in memory as sequences of integer byte values. So, for example, the string literal "ABCD" contains the four bytes 41h, 42h, 43h, and 44h.

3.1.7 Reserved Words

Reserved words have special meaning and can only be used in their correct context. Reserved works, by default, are not case-sensitive. For example, MOV is the same as mov and Mov. There are different types of reserved words:

- Instruction mnemonics, such as MOV, ADD, and MUL
- Register names
- Directives, which tell the assembler 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 reserved words can be found in Appendix A.

3.1.8 Identifiers

An *identifier* is a programmer-chosen name. It might identify a variable, a constant, a procedure, or a code label. There are a few rules on how they can be formed:

- 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.

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

In general, it's a good idea to use descriptive names for identifiers, as you do in high-level programming language code. Although assembly language instructions are short and cryptic, there's no reason to make your identifiers hard to understand also! Here are some examples of well-formed names:

```
lineCount     firstValue    index     line_count
myFile        xCoord        main      x_Coord
```

The following names are legal, but not as desirable:

```
_lineCount    $first        @myFile
```

Generally, you should avoid the @ symbol and underscore as leading characters, since they are used both by the assembler and by high-level language compilers.

3.1.9 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, but they let you define variables, macros, and procedures. They can assign names to memory segments and perform many other housekeeping tasks related to the assembler. Directives are not, by default, case sensitive. For example, **.data**, **.DATA**, and **.Data** are 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
mov    eax,myVar
```

Although all assemblers for Intel processors share the same instruction set, they usually have 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*. Segments are sections of a program that have different purposes. For example, one segment can be used to define variables, and is identified by the .DATA directive:

```
.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 directives and operators.

3.1.10 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 how the different parts are arranged:

```
[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. There are two types of labels: Data labels and Code 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.

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 follow the same rules we described for identifiers in Section 3.1.8. You can use the same code label more than once in a program as long as each label is unique within its enclosing procedure. We will show how to create procedures in Chapter 5.

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:

Mnemonic	Description
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

An operand is a value that is used for input or output for an instruction. Assembly language instructions can have between zero and three operands, each of which can be a register, memory operand, integer expression, or input–output port. We discussed register names in Chapter 2, and we discussed integer expressions in Section 3.1.2. There are different ways to create memory operands—using variable names, registers surrounded by brackets, for example. We will go into more details about that later. 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	*Integer literal*
2 + 4	Integer expression
eax	Register
count	Memory

Let's look at 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
```

There is a natural ordering of operands. When instructions have multiple operands, the first one is typically called the destination operand. The second operand is usually called the *source operand*. 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 three operands, in which the first operand is the destination, and the following two operands are source operands, which are multiplied together:

```
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. Here is an example:

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

We can also use any other symbol, as long as it does not appear within the comment lines:

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

Of course, you should provide comments throughout a program, particularly where the intent of your code is not obvious.

The NOP (No Operation) Instruction

The safest (and the most useless) instruction is 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 efficient 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.