

# Chapter 3 :Elements of Assembly Language

---

**Chapter 3 explains how to write assembly language programs.**

- ✓ The first part describes the types and formats of statements that are accepted by MASM, the Microsoft Macro Assembler.
- ✓ Then follows an example of a complete assembly language program, with instructions on how to assemble, link, and execute this and other programs.
- ✓ The last portion of the chapter fills in details about constructs that have been illustrated in the example, laying the groundwork for programs in future chapters.

## Chapter 3 :Elements of Assembly Language

### 3.1 Assembly Language Statements

---

An assembly language source code file consists of a collection of **statements**. Most statements fit easily on an 80-character line. However, MASM 6.1 accepts statements up to 512 characters long; these can be extended over more than one physical line using backslash (\) characters at the end of each line except the last.

Because assembly language programs are far from self-documenting, it is important to use an adequate number of comments. Comments can be used with any statement.

**A semicolon (;) begins the comment,**

and the comment then extends until the end of the line. An entire line is a comment if the semicolon is in column 1 or.

# Chapter 3 :Elements of Assembly Language

## 3.1 Assembly Language Statements

---

There are three types of functional assembly language statements:

- ✓ **instructions**
- ✓ **Directives**
- ✓ **macros**

**An instruction** is translated by the assembler into one or more bytes of object code (machine code), which will be executed at run time. *Each instruction corresponds to one of the operations that can be executed by the 80x86 CPU.* The instruction

`add eax, 158`

was used before.

# Chapter 3 :Elements of Assembly Language

## 3.1 Assembly Language Statements

---

**A directive** tells the assembler to take some action. *Such an action does not result in machine instructions* and often has no effect on the object code. For example, the assembler can produce a listing file showing the original source code, the object code, and other information. The directive

### .NOLIST

anywhere in the source file tells the assembler to stop displaying source statements in the listing file. The object code produced is the same with or without the .NOLIST directive. (There is a .LIST directive to resume listing source statements.) These directives and many others start with a period, but others do not.

# Chapter 3 :Elements of Assembly Language

## 3.1 Assembly Language Statements

---

**A macro** is "shorthand" for a sequence of other statements, be they instructions, directives, or even other macros. The assembler expands a macro to the statements it represents and then assembles these new statements. Several macros will appear in the example program later in this chapter.

# Chapter 3 :Elements of Assembly Language

## 3.1 Assembly Language Statements

---

**A statement** that is more than just a comment almost always contains a mnemonic that identifies the purpose of the statement, and may have three other fields: name, operand, and comment. These components must be in the following order:

**name mnemonic operand(s) ;comment**

For example, a program might contain the statement

**ZeroCount: mov ecx,0 ;initialize count to zero**

The name field always ends with a colon (:) when used with an instruction. When used with a directive, the name field has no colon.

The mnemonic in a statement indicates a specific instruction, directive, or macro.

Some statements have no operand, others have one, others have more. If there is more than one operand, they are separated by commas; spaces can also be added. Sometimes a single operand has several components with spaces between them, making it look like more than one operand.

# Chapter 3 :Elements of Assembly Language

## 3.1 Assembly Language Statements

---

In the instruction

**add eax, 158**

the mnemonic is add and the operands are eax and 158.

The assembler recognizes add as a mnemonic for an instruction that will perform some sort of addition.

The operands provide the rest of the information that the assembler needs.

The first operand eax tells the assembler that the doubleword in the EAX register is to be one of the values added, and that the EAX register will be the destination of the sum.

Since the second operand is a number, the assembler knows that it is the actual value to be added to the doubleword in the EAX register. The resulting object code is 05 00 00 00 9E, where 05 stands for "add the doubleword immediately following this byte in memory to the doubleword already in EAX." The assembler takes care of converting the decimal number 158 to its doubleword length 2's complement representation 0000009E.

## Chapter 3 :Elements of Assembly Language

### 3.1 Assembly Language Statements

---

**One use for the name field is to label what will be symbolically,** following assembly and linking of the program, an address in memory for an instruction. Other instructions can then easily refer to the labeled instruction. If the above add instruction needs to be repeatedly executed in a program loop, then it could be coded

```
addLoop: add eax, 158
```

The instruction can then be the destination of a **jmp (jump) instruction**, the assembly language version of a *goto*:

```
jmp addLoop ; repeat addition
```

Notice that the colon does not appear at the end of the name addLoop in the jmp instruction.



## Chapter 3 :Elements of Assembly Language

### 3.1 Assembly Language Statements

---

It is sometimes useful to have a line of source code consisting of just a name, for example

**EndIfBlank:**

Such a label might be used as the last line of code implementing an if-then-else-endif structure. This name effectively becomes a label for whatever instruction follows it, but it is convenient to implement a structure without worrying about what comes afterwards.

It is considered good coding practice to make labels descriptive. The label **addLoop** might help to clarify the assembly language code, identifying the first instruction of a program loop that includes an addition. Other labels, like **EndIfBlank** above, may parallel key words in a **pseudocode design**.

# Chapter 3 :Elements of Assembly Language

## 3.1 Assembly Language Statements

---

- ✓ **Names and other identifiers used in assembly language are formed from letters, digits, and special characters.**
- ✓ The allowable special characters are underscore (\_), question mark (?), dollar sign (\$), and at sign (@).
- ✓ A name may not begin with a digit.
- ✓ An identifier may have up to 247 characters, so that it is easy to form meaningful names. The Microsoft Macro Assembler will not allow instruction mnemonics, directive mnemonics, register designations, and other words that have a special meaning to the assembler to be used as names. [Appendix C](#) contains a list of such reserved identifiers.

The assembler will accept code that is almost impossible for a human to read. However, since your programs must also be read by other people, you should make them as readable as possible. Two things that help are good program formatting and use of lowercase letters.

# Chapter 3 :Elements of Assembly Language

## 3.1 Assembly Language Statements

---

A well-formatted program has these fields aligned as you read down the program.

- ✓ **Always put names in column 1.**
- ✓ **Mnemonics might all start in column 12,**
- ✓ **operands might all start in column 18, and**
- ✓ **comments might all start in column 30**
- ✓ **Blank lines are allowed in an assembly language source file.**
- ✓ **the assembler does not distinguish between uppercase and lowercase.** (Mixed-case code is easier for people to read than code written all in uppercase or lowercase.)
- ✓ One convention is to use mostly lowercase source code except for uppercase directives.

# Chapter 3 :Elements of Assembly Language

## 3.2 A Complete Example

---

This section presents a complete example of an assembly language program. We start with a pseudocode design for the program. It is easy to get lost in the details of assembly language, so

*This program will prompt for two numbers and then find and display their sum.* The algorithm implemented by this program is

- ◆ `prompt for the first number;`
- ◆ `input ASCII characters representing the first number;`
- ◆ `convert the characters to a 2's complement doubleword;`
- ◆ `store the first number in memory;`
- ◆ `prompt for the second number;`
- ◆ `input ASCII characters representing the second number;`
- ◆ `convert the characters to a 2's complement doubleword;`
- ◆ `add the first number to the second number;`
- ◆ `convert the sum to a string of ASCII characters;`
- ◆ `display a label and the characters representing the sum;`

# Chapter 3 :Elements of Assembly Language

## 3.2 A Complete Example

---

Following program lists the complete program which implements this design.

```
; Example assembly language program -- adds two numbers
; Author: R. Detmer
; Date: revised 7/97
.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

INCLUDE io.h ; header file for input/output

cr EQU 0dh ; carriage return character
Lf EQU 0ah ; line feed

.STACK 4096 ; reserve 4096-byte stack

.DATA ; reserve storage for data
    Number1      DWORD ?
    Number2      DWORD ?
    Prompt1      BYTE "Enter first number: ", 0
    Prompt2      BYTE "Enter second number: ", 0
    String       BYTE 40 DUP (?)
    Labell       BYTE cr, Lf, "The sum is "
    Sum          BYTE 11 DUP (?)
               BYTE cr, Lf, 0
```

# Chapter 3 :Elements of Assembly Language

## 3.2 A Complete Example

---

```
.CODE ; start of main program code
_start:

    output    prompt1          ; prompt for first number
    input     string, 40        ; read ASCII characters
    atod      string           ; convert to integer
    mov       number1, eax      ; store in memory

    output    prompt2          ; repeat for second number
    input     string, 40
    atod      string
    mov       number2, eax

    mov       eax, number1      ; first number to EAX
    add       eax, number2      ; add second number
    dtoa      sum, eax          ; convert to ASCII characters
    output    labell            ; output label and sum

    INVOKE    ExitProcess, 0     ; exit with return code 0
    PUBLIC    _start             ; make entry point public

END                               ; end of source code
```

# Chapter 3 :Elements of Assembly Language

## 3.2 A Complete Example

---

The example program begins with comments identifying the purpose of the program, the author, and the date the program was written. This is minimal documentation for any program; most organizations require much more.

Below statements are bothe directives:

```
.386  
.MODEL FLAT
```

Without the directive `.386`, MASM [accepts only 8086/8088 instructions](#). The `.486` and `.586` directives enable use of even more instructions. There is also a `.386P` directive that allows the assembler to recognize privileged 80386 instructions.

The directive `.MODEL FLAT` tells the assembler to generate 32-bit code using a flat memory model. With MASM 6.1, this directive must follow the `.386` directive.

## Chapter 3 :Elements of Assembly Language

### 3.2 A Complete Example

---

The next statement

```
ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD
```

is another directive. [The PROTO directive is used to prototype a function.](#) In this instance, the name of the function is ExitProcess, a system function used to terminate a program. It has one parameter, a doubleword symbolically called dwExitCode.

The next statement

```
INCLUDE io.h
```

is yet another directive. [It instructs the assembler to copy the file IO.H into your program as the program is assembled.](#) The source file is not modified: It still contains just the INCLUDE directive, but for purposes of the assembly, the lines of IO.H are inserted at the point of the INCLUDE directive.



## Chapter 3 :Elements of Assembly Language

### 3.2 A Complete Example

---

The next two statements

```
cr      EQU      0dh      ;carriage return character
Lf      EQU      0ah      ;linefeed character
```

use the directive EQU to equate symbols to values.

Following an EQU directive, the symbol can be used as a synonym for the value in subsequent source code.

Using names rather than numbers can make clearer source code.

In this example, **cr** is being equated to the hexadecimal number **0D**, which is the ASCII code for a **carriage return character**; **Lf** is given the hex value **0A**, the ASCII code for a **linefeed character**.

An uppercase **L** has been used to avoid confusion with the number **1**.

In these EQU directives the assembler recognizes the values 0dh and 0ah as **hexadecimal** because each has a trailing **h**.

# Chapter 3 :Elements of Assembly Language

## 3.2 A Complete Example

---

The **.STACK** directive tells the assembler **how many bytes to reserve for a run-time stack**

4096 bytes is generous for the programs we will be writing. The stack is used primarily for procedure calls.

Each macro in IO.H generates a procedure call to an associated procedure that actually does the task, and some of these procedures in turn call other procedures.

## Chapter 3 :Elements of Assembly Language

### 3.2 A Complete Example

---

The directive **.DATA starts the data segment of the program**, the portion of the code where memory space for variables is reserved. In this program, the **BYTE** and **DWORD** directives are used to reserve **bytes** and **doublewords** of storage, respectively.

The directive

```
number1 DWORD ?
```

**reserves a single doubleword of storage**, associating the symbolic name `number1` with the address `00000000` since it is the first data item. The question mark (?) indicates that this doubleword will have no designated initial value, although actually MASM 6.1 will initialize it to zero.

# Chapter 3 :Elements of Assembly Language

## 3.2 A Complete Example

---

The statement

```
number2 DWORD ?
```

reserves another doubleword of storage, associating the symbolic name number2 with the next available address, 00000004, since it follows the doubleword already reserved. The run-time addresses for number1 and number2 will be different than 00000000 and 00000004, but these doublewords will be stored consecutively in memory.

# Chapter 3 :Elements of Assembly Language

## 3.2 A Complete Example

---

The directive

```
prompt1 BYTE "Enter first number: ", 0
```

**has two operands**, the string "Enter first number" and the number 0. It reserves one byte for each character inside the quotation marks and one byte for the number 0. For each character, the byte reserved is the ASCII code of the letter. For the number, it is simply its byte-length 2's complement representation. This directive thus reserves 22 bytes of memory containing 45 6E 74 65 72 20 66 69 72 73 74 20 6E 75 6D 62 65 72 3A 20 20 00. The name prompt1 is associated with the address 00000008 since eight previous bytes have been allocated.

## Chapter 3 :Elements of Assembly Language

### 3.2 A Complete Example

---

The next BYTE directive reserves 23 bytes of memory, with the name prompt2 associated with address 0000001E. Then the directive

```
string BYTE 40 DUP (?)
```

reserves 40 uninitialized bytes of memory that will have the symbolic address string. The DUP operator says to repeat the item(s) in parentheses. The directive

```
label1    BYTE cr, Lf, "The sum is "  
          BYTE      cr, Lf, 0
```

has three operands and reserves 13 bytes of storage. The first two bytes contain 0D and 0A since these are the values to which cr and Lf are equated. The next 11 bytes are the ASCII codes of the characters in quotation marks. The next-to-last BYTE directive reserves 11 uninitialized bytes with address associated with the name sum. Even though the last BYTE directive has no label, it reserves three initialized bytes of memory immediately following the 11 for sum.

## Chapter 3 :Elements of Assembly Language

### 3.2 A Complete Example

---

The next segment of the program contains executable statements. It begins with the directive

```
.CODE
```

The line of code with only the label

```
_start:
```

marks **the entry point of the program**, the address of the first statement to be executed. The name used is the programmer's choice, but we will consistently use `_start` for this purpose.

## Chapter 3 :Elements of Assembly Language

### 3.2 A Complete Example

---

Finally we come to the statements that really do something! Since this program performs mostly input and output, the bulk of its statements are macros to perform these functions. The macro

```
output prompt1
```

**displays characters stored at the address referenced by prompt1**, using a null (00) byte to terminate the display. In this program, the user will be prompted to enter the first number. Since there is no carriage return or line feed character at the end of the prompt, the cursor will remain on the line following the colon and the two blanks.



## Chapter 3 :Elements of Assembly Language

### 3.2 A Complete Example

---

The statement

```
input string,40      ;read ASCII characters
```

is a macro that functionally causes the computer to pause and wait for characters to be entered at the keyboard until the user presses the Enter key to terminate the input.

The first operand (string) identifies where the ASCII codes for these characters will be stored.

The second operand (40) identifies the maximum number of characters that can be entered. Notice that 40 uninitialized bytes were reserved at address string. More details about the input macro are in SECTION 3.6.

# Chapter 3 :Elements of Assembly Language

## 3.2 A Complete Example

---

The input macro inputs ASCII codes, but the CPU does arithmetic with numbers in 2's complement form. The `atod` (for "ASCII to double") macro scans memory at the address specified by its single operand and converts the ASCII codes there to the corresponding 2's complement doubleword; the result is stored in EAX.

In this program

```
atod string ; convert to integer
```

scans memory starting at string, skips leading blanks, notes any plus (+) or minus (-) sign, and builds a number from ASCII codes for digits. The scan stops when any non digit is encountered.

## Chapter 3 :Elements of Assembly Language

### 3.2 A Complete Example

---

The statement

```
mov    number1,eax    ;store in memory
```

is an instruction. The mnemonic `mov` stands for "move" but the instruction really performs a copy operation like an assignment statement in a high-level language. This particular instruction copies the value in the EAX register to the doubleword of memory at address number1.

## Chapter 3 :Elements of Assembly Language

### 3.2 A Complete Example

---

The next four statements

```
output      prompt2      ;repeat for second number
input       string,40
atod        string
mov         number2,eax
```

repeat the tasks just performed: prompt for the second number; input ASCII codes; convert the ASCII codes to a 2's complement doubleword; and copy the doubleword to memory. Note that the input area is reused.

## Chapter 3 :Elements of Assembly Language

### 3.2 A Complete Example

---

The next two instructions add the numbers. Addition must be done in a register, so the first number is copied to the EAX register with the instruction

```
mov     eax,number1      ;first number to AX
```

and then the second is added to the first with the instruction

```
add     eax,number2      ;add second number
```

(Do you see a more efficient way to get the sum in EAX?)

## Chapter 3 :Elements of Assembly Language

### 3.2 A Complete Example

---

The sum is now in the EAX register in 2's complement form. For display purposes we need a sequence of ASCII characters that represent this value.

The dtoa ("double to ASCII") macro takes the doubleword specified by its second operand and converts it to a string exactly 11 bytes long at the destination specified by the first operand.

In this program, the macro

```
dtoc    sum,eax        ;convert to ASCII characters
```

uses the contents of EAX as the source, and fills in the 11 bytes at sum with ASCII codes corresponding to this value. For a typical small number, leading space characters are used fill a total of 11 bytes.

# Chapter 3 :Elements of Assembly Language

## 3.2 A Complete Example

---

The macro

```
output label1          ;output label and sum
```

will display bytes of memory, starting at label1 and continuing until a null byte (00) is encountered.

Since the undefined bytes at sum have been replaced by ASCII codes, the first null byte in memory will be the one following the carriage return and line feed codes in the unlabeled BYTE directive. A total of 26 characters will be displayed.

# Chapter 3 :Elements of Assembly Language

## 3.2 A Complete Example

---

The statement

```
INVOKE ExitProcess,0      ;exit with return code 0
```

is a directive that generates code to call the procedure `ExitProcess` with the value 0 for the parameter symbolically called `dwExitCode` in the prototype.

Functionally this terminates the program, with exit code value 0 telling the operating system that the program terminated normally. (Nonzero values can be used to indicate error conditions.)



## Chapter 3 :Elements of Assembly Language

### 3.2 A Complete Example

---

Normally the names used inside a file are visible only inside the file. The directive

```
PUBLIC _start           ;make entry point public
```

makes the entry point address visible outside this file, so that the linker can identify the first instruction to be executed as it constructs an executable program file. We will later use this directive to make names of separately assembled procedures visible.

The final statement in an assembly language source file is the directive END. This marks the physical end of the program. There should be no statement following END.

## Chapter 3 :Elements of Assembly Language

### 3.3 How to Assemble, Link, and Run a Program

---

The source code for a program is entered using any standard text editor such as [Notepad](#) or [Edit](#). Assembly language source code is normally stored in a file with a .ASM type. For this section, we will assume that the source program from last example is stored in the file EXAMPLE.ASM.

We will use the ML assembler from MASM 6.1 to assemble programs. To assemble EXAMPLE.ASM, you enter

```
ml /c /coff example.asm
```

at a DOS prompt in a MS-DOS window. Assuming there is no error in your program, you will see a display like

```
Microsoft (R) Macro Assembler Version 6.11
```

```
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.
```

```
Assembling: example.asm
```

## Chapter 3 :Elements of Assembly Language

### 3.3 How to Assemble, Link, and Run a Program

---

The file EXAMPLE.OBJ will be added to your directory. If your program contains errors, error messages are displayed and no .OBJ file is produced.

There are two switches, `/c` and `/coff`, in this invocation of the assembler. The ML product is capable of both assembly and linking, and the switch `/c` says to assemble only. The `/coff` switch says to generate common object file format. *ML switches are case-sensitive: They must be entered exactly as shown-these are in lowercase.*

The linker we will use is named LINK. For this example, invoke it at the DOS prompt with

```
link /subsystem:console /entry:start /out:example.exe  
example.obj io.obj kernel32.lib
```

This is entered as a single command, although it may wrap to a new line as you type.

## Chapter 3 :Elements of Assembly Language

### 3.3 How to Assemble, Link, and Run a Program

---

Again, assuming no errors, you will see

```
Microsoft (R) 32-Bit Incremental Linker Version 5.10.7303  
Copyright (C) Microsoft Corp 1992-1997. All rights reserved.
```

This LINK command links together EXAMPLE.OBJ, IO.OBJ, and KERNEL32.LIB to produce the output file EXAMPLE.EXE.

The switch /subsystem:console tells LINK to produce a console application, one that runs in a DOS window. The switch /entry:start identifies the label of the program entry point; *notice that you do **not** use an underscore here even though `_start` was the actual label for the entry point.*

## Chapter 3 :Elements of Assembly Language

### 3.3 How to Assemble, Link, and Run a Program

---

A program is executed by simply typing its name at the DOS prompt. This shows a sample run of EXAMPLE.EXE, with user input underlined. Once the executable file has been produced, you can run the program as many times as you wish without assembling or linking it again.

---

```
C:\AsmFiles>example
```

```
Enter first number: 98
```

```
Enter second number: -35
```

```
The sum is 63
```

```
C:\AsmFiles>
```

---

## Chapter 3 :Elements of Assembly Language

### 3.3 How to Assemble, Link, and Run a Program

---

The book's software package includes [Microsoft's Windbg](#), a symbolic debugger that can be used to trace execution of an assembly language program. This is a useful tool for finding errors and for seeing how the computer works at the machine level.

To use Windbg, you must add the **/Zi** switch (uppercase Z, lowercase i) to the assembly with ML. This causes the assembler to add debug information to its output. The assembly command now looks like

```
ml /c /coff /Zi example.asm
```

Linking is changed to add one new switch, **/debug**, giving

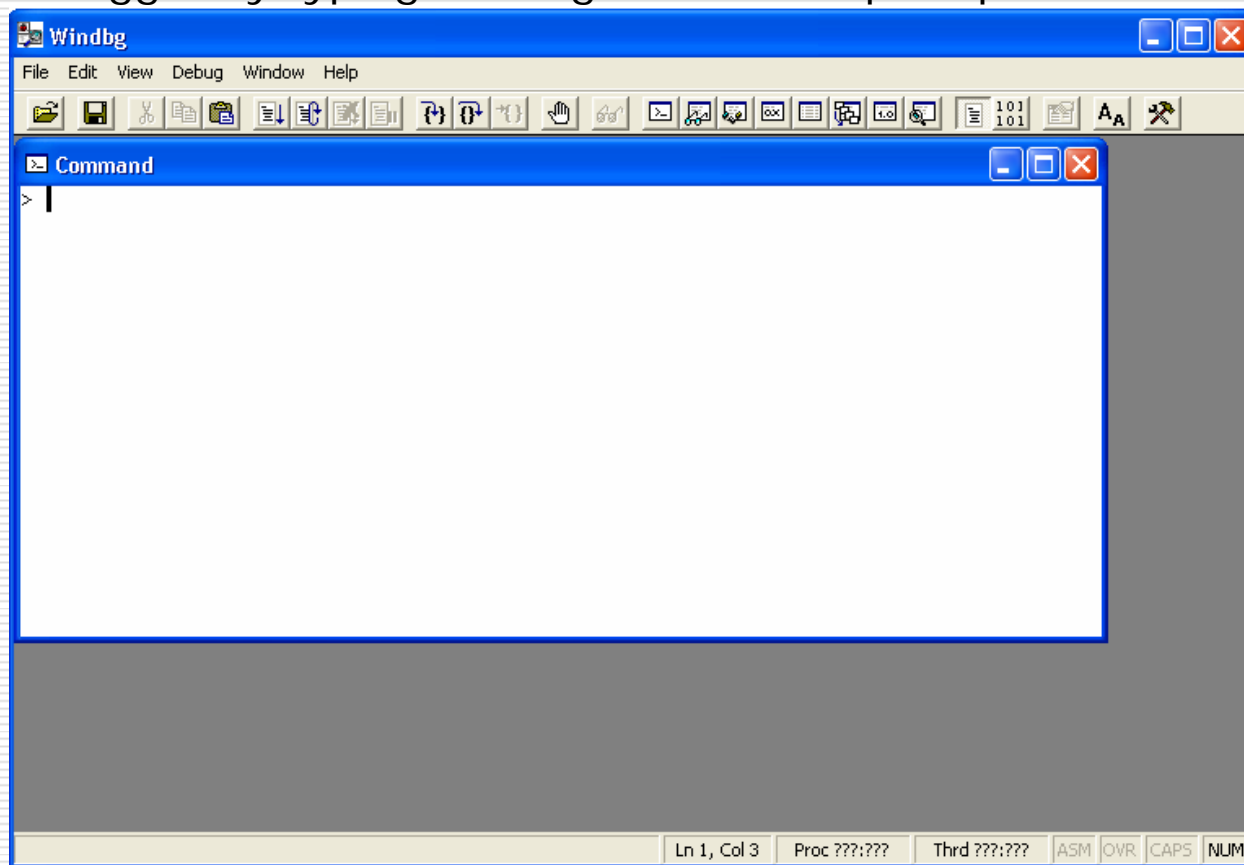
```
link /debug /subsystem:console /entry:start /out:example.exe  
example.obj io.obj kernel32.lib
```

# Chapter 3 :Elements of Assembly Language

## 3.3 How to Assemble, Link, and Run a Program

---

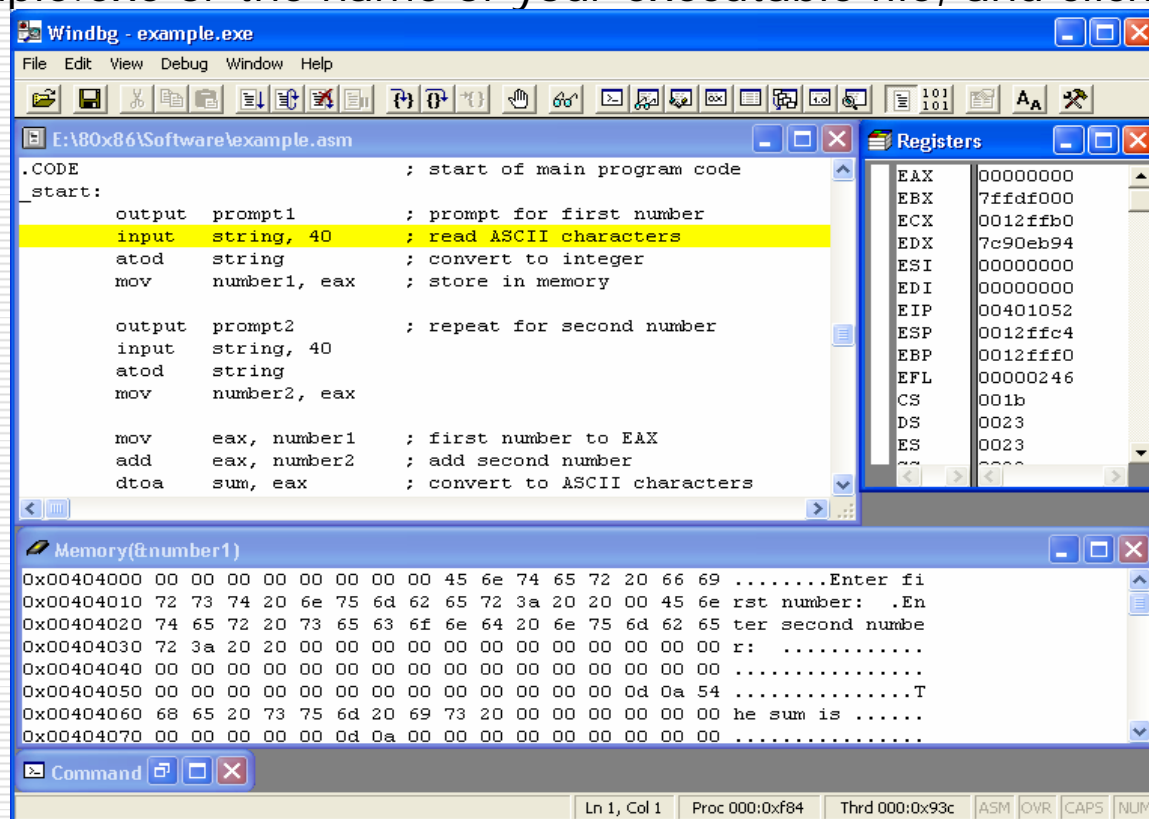
Start the debugger by typing Windbg at the DOS prompt.



# Chapter 3 :Elements of Assembly Language

## 3.3 How to Assemble, Link, and Run a Program

From the menu bar choose File, then Open Executable... Select example.exe or the name of your executable file, and click OK





## Chapter 3 :Elements of Assembly Language

### 3.3 How to Assemble, Link, and Run a Program

---

Now press the step into button



Click OK in the information window and then press the step into button again. Your source code now appears in a Windbg child window behind the Command window.

Minimize the Command window. Next select View and then Registers to open a window that shows contents of the 80×86 registers.

Then select View and Memory... to open a window that shows contents of memory. For this window you must enter the starting memory address; for the example program, use &number1 as the starting address-the C/C++ address-of operator (&) is used to indicate the address of number1, the first item in the data section.

Finally size and rearrange windows until your screen looks something like the screen shown in previous slide.

## Chapter 3 :Elements of Assembly Language

### 3.3 How to Assemble, Link, and Run a Program

---

The first statement of the example program is highlighted. Clicking the step into button causes this statement to be executed.

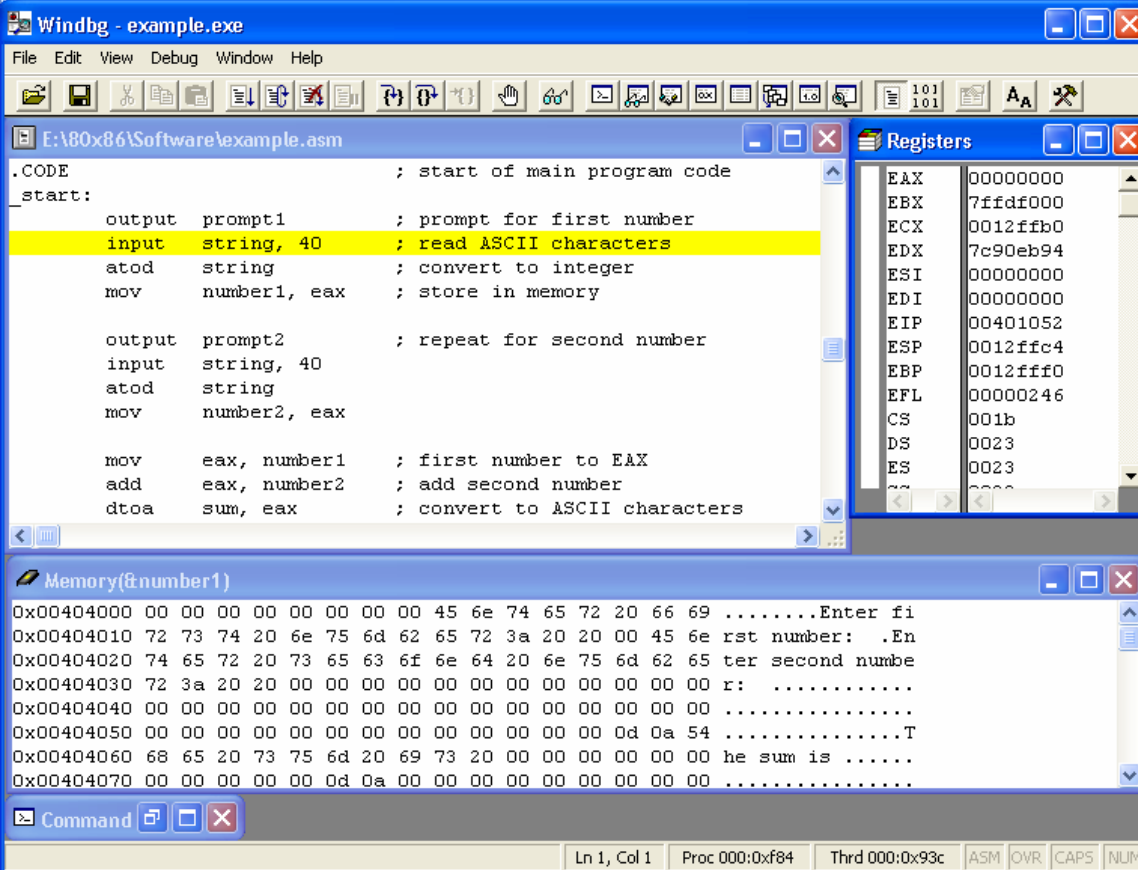
Although this statement is a macro, it is executed as a single instruction, and **Enter first number:** appears on the output screen. (You can click on the edge of the output screen to put it on top.)

Clicking step into again causes the input macro to be executed. When you enter a number and press return, Windbg returns to the debugger screen with the third statement highlighted.

Two more clicks of the step into button causes the ASCII to double the macro to be executed, and the first mov instruction to be executed. The Windbg window now looks like the one shown in next slide.

# Chapter 3 :Elements of Assembly Language

## 3.3 How to Assemble, Link, and Run a Program



Windbg - example.exe

File Edit View Debug Window Help

E:\80x86\Software\example.asm

```
.CODE ; start of main program code
_start:
    output prompt1 ; prompt for first number
    input string, 40 ; read ASCII characters
    atod string ; convert to integer
    mov number1, eax ; store in memory

    output prompt2 ; repeat for second number
    input string, 40
    atod string
    mov number2, eax

    mov eax, number1 ; first number to EAX
    add eax, number2 ; add second number
    dtoa sum, eax ; convert to ASCII characters
```

Registers

EAX	00000000
EBX	7ffdf000
ECX	0012ffb0
EDX	7c90eb94
ESI	00000000
EDI	00000000
EIP	00401052
ESP	0012ffc4
EBP	0012fff0
EFL	00000246
CS	001b
DS	0023
ES	0023
SS	0023

Memory(fix number1)

```
0x00404000 00 00 00 00 00 00 00 00 45 6e 74 65 72 20 66 69 .....Enter fi
0x00404010 72 73 74 20 6e 75 6d 62 65 72 3a 20 20 00 45 6e rst number: .En
0x00404020 74 65 72 20 73 65 63 6f 6e 64 20 6e 75 6d 62 65 ter second numbe
0x00404030 72 3a 20 20 00 00 00 00 00 00 00 00 00 00 00 00 r: .....
0x00404040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x00404050 00 00 00 00 00 00 00 00 00 00 00 00 0d 0a 54 .....T
0x00404060 68 65 20 73 75 6d 20 69 73 20 00 00 00 00 00 00 he sum is .....
0x00404070 00 00 00 00 00 0d 0a 00 00 00 00 00 00 00 00 .....

```

Command

Ln 1, Col 1 Proc 000:0xf84 Thrd 000:0x93c ASM OVR CAPS NUM

## Chapter 3 :Elements of Assembly Language

### 3.3 How to Assemble, Link, and Run a Program

---

At this point, the Registers window shows that EAX contains 00000062, the 2's complement doubleword version of 98. The number 98 was entered in response to the prompt. You can see its ASCII codes stored in memory on the fourth line of the Memory window.

Each line of the Memory window has three parts: its starting address, hex values for the bytes stored at that address, and printable characters that correspond to those bytes, if any.

The first five characters of the fourth line are the end of prompt2, ASCII codes for r and colon, two spaces, and a null byte. The 40 bytes reserved for string come next in memory, and the first four have been replaced by 39, 38, 00, and 0A, ASCII codes for 98, a null byte, and a line feed.

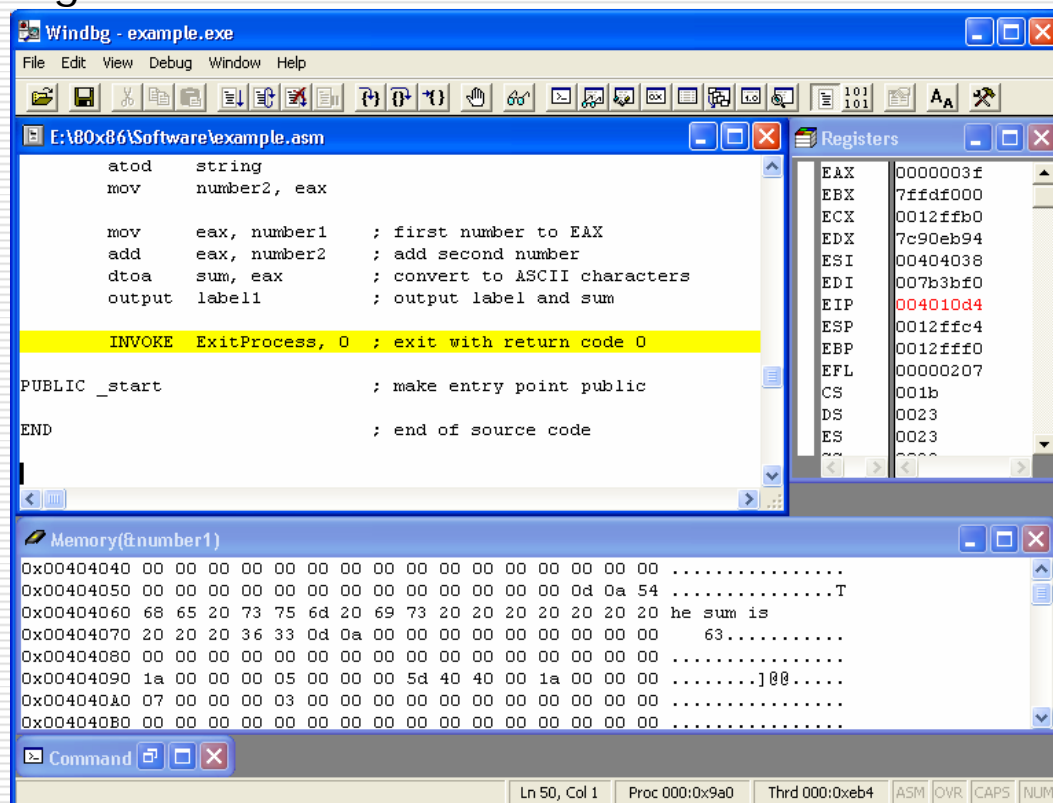
When 98 and Enter were pressed, the operating system stored 39 and 38 plus a carriage return character and a line feed character.

The input macro replaced the carriage return by a null byte, but you can still see the line feed in memory. The atod macro scanned these ASCII codes to produce the value in EAX. The Memory window also shows a value of 62 00 00 00 for number1, the bytes of the number stored backwards, copied there by the mov instruction.

# Chapter 3 :Elements of Assembly Language

## 3.3 How to Assemble, Link, and Run a Program

The rest of the program is traced similarly. This figure shows Windbg just before program termination.



## Chapter 3 :Elements of Assembly Language

### 3.4 The Assembler Listing File

---

The ML assembler can produce a listing file as part of the assembly process. This .LST file shows the source code, the object code to which it is translated, and additional information.

When your source file contains errors, this .LST file displays error messages at the points of the errors, helping to locate the offending statements.

Suppose that we modify the example program EXAMPLE.ASM from 3.1 part, changing

```
    atod    string          ;convert to integer
    mov     number1,eax     ;store in memory
```

to

```
    atod    eax,string      ;convert to integer
    mov     number1,ax      ;store in memory
```

This introduces two errors: The atod macro only allows one operand, and the source and destination operands for the mov instruction are different sizes.

## Chapter 3 :Elements of Assembly Language

### 3.4 The Assembler Listing File

---

Suppose that this revised file is stored in EXAMPLE1.ASM. An additional switch, /Fl (uppercase F, lowercase letter l), is needed to generate a listing file during assembly

```
ml /c /coff /Fl example1.asm
```

When this command is entered at a DOS prompt, the console shows

```
Assembling: example1.asm
example1.asm(32): error A2022: instruction operands must be the same size
example1.asm(31): error A2052: forced error : extra operand(s) in ATOD
atod(7): Macro Called From
example1.asm(31): Main Line Code
```

These error messages are fairly helpful-they indicate errors on lines 32 and 31 of the source file and describe the errors.

## Chapter 3 :Elements of Assembly Language

### 3.4 The Assembler Listing File

---

if you look at the corresponding part of EXAMPLE1.LST, you see

```
00000000          _start:
                                output prompt1      ;prompt for first number
                                input string,40     ;read ASCII characters
                                atod  eax,string ;convert to integer
                                1                   .ERR <extra operand(s) in ATOD>
example1.asm(31): error A2052: forced error : extra operand(s) in ATOD
atod(7): Macro Called From
example1.asm(31): Main Line Code
                                mov number1,ax     ;store in memory
example1.asm(32): error A2022: instruction operands must be the same size

                                output prompt2     ;repeat for second number
```

with the error messages under the statements with the errors. Viewing the listing file frequently makes it easier to find errors in source code.

---



## Chapter 3 :Elements of Assembly Language

### **3.4 The Assembler Listing File**

---

Following figure shows a listing file for EXAMPLE.ASM, the original example program without errors. Parts of this file will be examined to better understand the assembly process.

Look the file EXAMPLE.LST

## Chapter 3 :Elements of Assembly Language

### 3.4 The Assembler Listing File

---

The listing begins by echoing comments and directives at the beginning of the source code file. Following the INCLUDE directive, several lines from IO.H are shown. These lines are marked with the letter C to show they come from an included file. In particular, you see the .NOLIST directive that suppressed listing of most of IO.H, and the .LIST directive that resumed listing of the rest of the source file. For each EQU directive the assembler shows the value to which the symbol is equated as eight hex digits.

**This listing shows 0000000D for cr and 0000000A for Lf.**

```
00000000      00000000      number1  DWORD ?
```

## Chapter 3 :Elements of Assembly Language

### 3.4 The Assembler Listing File

---

00000000	00000000	number1	DWORD ?
00000004	00000000	number2	DWORD ?
00000008	45 6E 74 65 72	prompt1	BYTE "Enter first"
	20 66 69 72 73		
	74 20 6E 75 6D		
	62 65 72 3A 20		
	20 00		
0000001E	45 6E 74 65 72	prompt2	BYTE "Enter second"
	20 73 65 63 6F		
	6E 64 20 6E 75		
	6D 62 65 72 3A		
	20 20 00		

## Chapter 3 :Elements of Assembly Language

### 3.5 Constant Operands

---

This section discusses formats of constant operands used in BYTE, DWORD, and WORD directives.

Numeric operands can be expressed in decimal, hexadecimal, binary, or octal notations. The assembler assumes that a number is decimal unless the number has a suffix indicating another base. The suffixes that may be used are

<i><b>Suffix</b></i>	<i><b>Base</b></i>	<i><b>Number System</b></i>
H	16	hexadecimal
B	2	binary
O or Q	8	octal
none	10	decimal

## Chapter 3 :Elements of Assembly Language

### 3.5 Constant Operands

---

The directive

**mask          BYTE    01111101b**

reserves one byte of memory and initializes it to 7D. This is equivalent to any of the following directives

**mask          BYTE    7dh**  
**mask          BYTE    125**  
**mask          BYTE    175q**

since  $1111101_2 = 7D_{16} = 125_{10} = 175_8$ . The choice of number systems often depends on the use planned for the constant. A binary value is appropriate when you need to think of the value as a sequence of eight separate bits, for instance in a logical operation.

## Chapter 3 :Elements of Assembly Language

### 3.5 Constant Operands

---

A BYTE directive reserves storage for one or more bytes of data.

Byte1	BYTE	255 ; value is FF
Byte2	BYTE	127 ; value is 7F
byte3	BYTE	91 ; value is 5B
byte4	BYTE	0 ; value is 00
byte5	BYTE	-1 ; value is FF
Byte6	BYTE	-91 ; value is A5
byte7	BYTE	-128 ; value is 80

If a data value is numeric, it can be thought of as signed or unsigned. The decimal range of unsigned values that can be stored in a single byte is 0 to 255. The decimal range of signed values that can be stored in a single byte is -128 to 127. Although the assembler will allow larger or smaller values, normally you restrict numeric operands for BYTE directives to 128 to 255.

## Chapter 3 :Elements of Assembly Language

### 3.5 Constant Operands

---

**A DWORD directive reserves a doubleword of storage;** since four bytes can store a signed number in the range -2,147,483,648 to 2,147,483,647 or an unsigned number from 0 to 4,294,967,295, it makes sense to restrict operand values to the range -2,147,483,648 to 4,294,967,295.

The examples below give the initial values reserved for a few doublewords:

<b>double1</b>	<b>DWORD</b>	<b>4294967295</b>	<b>; value is FFFFFFFF</b>
<b>double2</b>	<b>DWORD</b>	<b>4294966296</b>	<b>; value is FFFFFFFC18</b>
<b>double3</b>	<b>DWORD</b>	<b>0</b>	<b>; value is 00000000</b>
<b>double4</b>	<b>DWORD</b>	<b>-1</b>	<b>; value is FFFFFFFF</b>
<b>double5</b>	<b>DWORD</b>	<b>-1000</b>	<b>; value is FFFFFFFC18</b>
<b>double6</b>	<b>DWORD</b>	<b>-2147483648</b>	<b>; value is 80000000</b>

## Chapter 3 :Elements of Assembly Language

### 3.5 Constant Operands

---

Similarly, operands for a WORD directive should be restricted to the range 32,768 to 65,535. The examples below give the initial values reserved for a few doublewords and words.

The examples below give the initial values reserved for a few doublewords:

<b>word1</b>	<b>WORD</b>	<b>65535</b>	<b>; value is FFFF</b>
<b>word2</b>	<b>WORD</b>	<b>32767</b>	<b>; value is 7FFF</b>
<b>word3</b>	<b>WORD</b>	<b>1000</b>	<b>; value is 03E8</b>
<b>word4</b>	<b>WORD</b>	<b>0</b>	<b>; value is 0000</b>
<b>word5</b>	<b>WORD</b>	<b>-1</b>	<b>; value is FFFF</b>
<b>word6</b>	<b>WORD</b>	<b>-1000</b>	<b>; value is FC18</b>
<b>word7</b>	<b>WORD</b>	<b>-32768</b>	<b>; value is 8000</b>



## Chapter 3 :Elements of Assembly Language

### 3.5 Constant Operands

---

**The bytes of a word or doubleword  
are actually stored **backwards****

**for example, the initial value of  
**word6** previous is actually **18FC**.**

## Chapter 3 :Elements of Assembly Language

### 3.5 Constant Operands

---

The **BYTE directive allows character operands with a single character or string operands with many characters.** Either apostrophes (') or quotation marks (") can be used to designate characters or delimit strings. They must be in pairs; you can not put an apostrophe on the left and a quotation mark on the right.

Each of the following BYTE directives is allowable.

<b>char1</b>	<b>BYTE 'm'</b>	<b>; value is 6D</b>
<b>char2</b>	<b>BYTE 6dh</b>	<b>; value is 6D</b>
<b>string1</b>	<b>BYTE "Joe"</b>	<b>; value is 4A 6F 65</b>
<b>string2</b>	<b>BYTE "Joe's"</b>	<b>; value is 4A 6F 65 27 73</b>

If you are trying to store the letter m rather than the number  $6D_{16}$ , then there is no reason to look up the ASCII code and enter it as in char2- the assembler has a built-in ASCII chart!

## Chapter 3 :Elements of Assembly Language

### 3.5 Constant Operands

---

DWORD and WORD directives also allows multiple operands. The directive

**words      WORD      10, 20, 30, 40**

reserves four words of storage with initial values 000A, 0014, 001E, and 0028. The DUP operator can be used to generate multiple bytes or words with known values as well as uninitialized values. The directive

**DbIArray    DWORD    100 DUP(999)**

reserves 100 doublewords of storage, each initialized to 000003E7. This is an effective way to initialize elements of an array.

## Chapter 3 :Elements of Assembly Language

### 3.5 Constant Operands

---

If one needs a string of 50 asterisks, then

```
stars          BYTE    50 DUP('*')
```

will do the job. If one wants 25 asterisks, separated by spaces,

```
starsAndSpaces  BYTE    24 DUP("* "), '*'
```

reserves these 49 bytes and assigns the desired initial values.

## Chapter 3 :Elements of Assembly Language

### 3.5 Constant Operands

---

An operand of a BYTE, DWORD, WORD, or other statement can be an expression involving arithmetic or other operators. These expressions are evaluated by the assembler at assembly time, not at run time, with the resulting value used for assembly. The following directives are equivalent, each reserving a word with an initial hex value of 0090.

```
gross    WORD    144
gross    WORD    12*12
gross    WORD    10*15 - 7 + 1
```

## Chapter 3 :Elements of Assembly Language

### 3.5 Constant Operands

---

Each symbol defined by a BYTE, DWORD, or WORD directive is associated with a length. The assembler notes this length and checks to be sure that symbols are used appropriately in instructions. For example, the assembler will generate an error message if

```
char    BYTE    'x'
```

is used in the data segment and

```
mov     ax,char
```

appears in the code segment-the AX register is a word long, but char is associated with a single byte of storage.

## Chapter 3 :Elements of Assembly Language

### 3.5 Constant Operands

---

The Microsoft assembler recognizes several additional directives for reserving storage. These include

- ✓ **QWORD** for reserving a quadword,
- ✓ **TBYTE** for a 10-byte integer,
- ✓ **REAL4**, for reserving a 4-byte floating point number,
- ✓ **REAL8** for 8-byte floating point, and
- ✓ **REAL10** for 10-byte floating point.

It also has directives to distinguish signed bytes, words, and doublewords from unsigned.

# Chapter 3 :Elements of Assembly Language

## 3.6 Instruction Operands

---

There are **three basic types of instruction** operands;

- ✓ **some are constants,**
- ✓ **some designate CPU registers,**
- ✓ **some reference memory locations.**

There are several ways of referencing memory; two simpler ways will be discussed in this section, and more complex methods will be introduced as needed.



## Chapter 3 :Elements of Assembly Language

### 3.6 Instruction Operands

---

Many instructions have two operands. In general, **the first operand gives the destination of the operation**, and **the second operand gives the source**. For example, when

```
mov    al, '/'
```

is executed, the byte 2F will be loaded into the AL register, replacing the previous byte. The second operand '/' specifies the constant source. When

```
add     eax, number1
```

is executed, EAX gets the sum of the doubleword designated by number1 and the old contents of EAX. **The first operand EAX specifies the source for one doubleword as well as the destination for the sum; the second operand number1 specifies the source for the other of the two doublewords that are added together.**

## Chapter 3 :Elements of Assembly Language

### 3.6 Instruction Operands

---

Following table lists the addressing modes used by the Intel 80×86 microprocessor, giving the location of the data for each mode.

mode	Location of data
immediate	In the instruction itself
register	In a register
memory	at some address in memory

For an **immediate mode** operand, the data to be used is built into the instruction before it is executed. For a **register mode** operand, the data to be used is in a register. To indicate a register mode operand, the programmer simply codes the name of the register.

# Chapter 3 :Elements of Assembly Language

## 3.6 Instruction Operands

---

Memory addresses can be calculated several ways; Following table lists the two most common.

Memory mode	Location of data
Direct	at a memory location whose address (offset) is built into the instruction
Register indirect	at a memory location whose address is in a register

## Chapter 3 :Elements of Assembly Language

### 3.6 Instruction Operands

---

In each of the following examples, the first operand is register mode and the second operand is immediate mode.

```
mov    al, '/'    ; B0 2F
```

the ASCII code 2F for the slash is the second byte of the instruction, and is placed there by the assembler.

For the instruction

```
add     eax, 135    ; 05 00000087
```

the doubleword length 2's complement version of 135 is assembled into the last four bytes of the instruction.

## Chapter 3 :Elements of Assembly Language

### 3.6 Instruction Operands

---

Any memory mode operand specifies using data in memory or specifies a destination in memory. A direct mode operand has the 32-bit address built into the instruction. Usually the programmer will code a symbol that is associated with a BYTE, DWORD, or WORD directive in the data segment or with an instruction in the code segment. The location corresponding to such a symbol will be **relocatable** so that the assembler listing shows an assembly-time address that may be adjusted later. In the statement

```
add    eax, number2    ;05 00000004
```

the first operand is register mode and the second operand is direct mode. The memory operand has been encoded as the 32-bit address 00000004, the offset of number2 in the data segment.

## Chapter 3 :Elements of Assembly Language

### 3.6 Instruction Operands

---

The first operand of the instruction

```
add    eax,[edx]    ;03 02
```

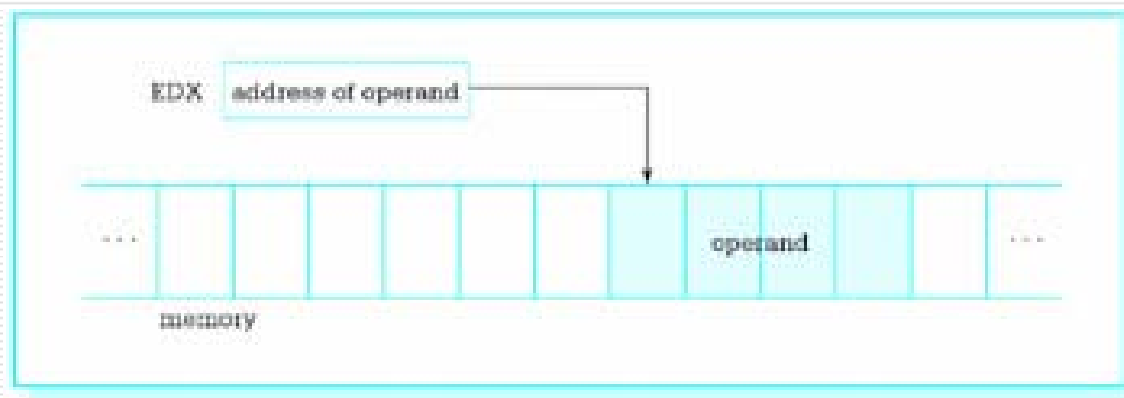
is register mode, and the second operand is **register indirect mode**. notice that it is not long enough to contain a 32-bit memory address. Instead, it contains bits that say to use the *address* in the EDX register to locate a doubleword in memory to add to the doubleword already in EAX. In other words, the second number is not in EDX, but its address is. The square bracket notation ([]) indicates indirect addressing with MASM 6.11. Following figure illustrates how register indirect addressing works in this example.

## Chapter 3 :Elements of Assembly Language

### 3.6 Instruction Operands

---

Following figure illustrates how register indirect addressing works in this example.



Any of the general registers EAX, EBX, ECX, and EDX or the index registers ESI and EDI can be used for register indirect addressing. The base pointer EBP can also be used, but for an address in the stack rather than for an address in the data segment.

## Chapter 3 :Elements of Assembly Language

### 3.6 Instruction Operands

---

With register indirect mode, the register serves like a pointer variable in a high-level language. The register contains the location of the data to be used in the instruction, not the data itself.

When the size of the memory operand is ambiguous, the PTR operator must be used to give the size to the assembler. For example, the assembler will give an error message for

```
mov    [ebx],0
```

since it cannot tell whether the destination is a byte, word, or doubleword. If it is a byte, you can use

```
mov    BYTE PTR [ebx],0
```

For a word or doubleword destination, use **WORD PTR** or **DWORD PTR**, respectively.



# Chapter 3 :Elements of Assembly Language

## 3.6 Instruction Operands

---

In an instruction like

**add      eax,[edx]**

it is not necessary to use DWORD PTR [edx] since the assembler assumes that the source will be a doubleword, the size of the destination EAX.

- ✓ A few instructions have no operands.
- ✓ Many have a single operand.
- ✓ Sometimes an instruction with no operands requires no data to operate on or an instruction with one operand needs only one value. Other times the location of one or more operands is implied by the instruction and is not coded.
- ✓ For example, one 80×86 instruction for multiplication is **mul**; it might be coded

**mul      bh**

Only one operand is given for this instruction; the other value to be multiplied is always in the AL register.

---

## Chapter 3 :Elements of Assembly Language

### 3.7 Input/Output Using Macros Defined in IO.H

---

In order to write useful programs, you need to be able to input and output data. High-level languages usually provide for input or output of numeric data in addition to character or string data.

#### **A numeric input routine in a high-level language**

- ✓ accepts a string of character codes representing a number,
- ✓ converts the characters to a 2's complement or floating point form, and
- ✓ stores the value in a memory location associated with some variable name.

#### **Conversely, output routines of high-level languages start with**

- ✓ a 2's complement or floating point number in some memory location,
- ✓ convert it to a string of characters that represent the number, and then
- ✓ output the string.

Operating systems usually do not provide these services, so the assembly language programmer must code them.

## Chapter 3 :Elements of Assembly Language

### 3.7 Input/Output Using Macros Defined in IO.H

The file IO.H provides a set of macro definitions that make it possible to do input, output, and numeric conversion fairly easily.

The source code for these external procedures is in the file IO.ASM; the assembled version of this file is IO.OBJ.

Name	Parameter(s)	Action	Flags affected
dtoa	<i>destination, source</i>	Converts the doubleword at source (register or memory) to an eleven-byte-long ASCII string at <i>destination</i> .	None
Atod	<i>source</i>	Scans the string starting at source for + or - followed by digits, interpreting these characters as an integer. The corresponding 2's complement number is put in EAX. The offset of the terminating nondigit character is put in ESI. For input error, 0 is put in EAX. Input error occurs if the number has no digits or is out of the range -2,147,483,648 to 2,147,483,647.	OF = 1 for input error; OF = 0 otherwise. Other flag values correspond to the result in EAX.

## Chapter 3 :Elements of Assembly Language

### 3.7 Input/Output Using Macros Defined in IO.H

---

#### Macros in IO.H

Name	Parameter(s)	Action	Flags affected
itoa	<i>destination, source</i>	Converts the word at source (register or memory) to a six-byte-long ASCII string at destination.	None
atoi	<i>source</i>	Similar to atod, except that the resulting number is placed in AX. The range accepted is -32,768 to 32,767.	similar to atod
output	<i>source</i>	Displays the string starting at source. The string must be null-terminated.	None
input	<i>destination, length</i>	Inputs a string up to length characters long and stores it at destination.	None

## Chapter 3 :Elements of Assembly Language

### 3.7 Input/Output Using Macros Defined in IO.H

---

**The output macro** is used to output a string of characters to the monitor. Its *source* operand references a location in the data segment, usually the name on a BYTE directive. **Characters starting at this address are displayed until a null character is reached**; the null character terminates the output. It is important that source string contains ASCII codes for characters that can be displayed.

**The input macro** is used to input a string of characters from the keyboard. **It has two parameters, *destination* and *length***. The destination operand references a string of bytes in the data segment and the length operand references the number of bytes reserved in that string. The destination string should be at least two bytes longer than the actual number of characters to be entered; this allows for the operating system to add carriage return and linefeed characters when you press Enter. The input macro replaces the carriage return character by a null byte.

## Chapter 3 :Elements of Assembly Language

### 3.7 Input/Output Using Macros Defined in IO.H

---

**The name `dtoa`** (double to ASCII) describes the function of this macro. It takes a doubleword length source containing a 2's complement integer, and produces a string of exactly 11 ASCII characters representing the same integer in the decimal number system.

The source operand is normally a register or memory operand. The destination will always be a 11-byte area of storage in the data segment reserved with a `BYTE` directive. The string of characters produced will have leading blanks if the number is shorter than 11 characters.

If the number is negative, a minus sign will immediately precede the digits. Since the decimal range for a word-length 2's complement number is -2147483648 to 2147483647, there is no danger of generating too many characters to fit in a 11-byte-long field.

A positive number will always have at least one leading blank.

**The `dtoa` macro** alters only the 11-byte area of memory that is the destination for the ASCII codes. No registers are changed, including the flag register.

## Chapter 3 :Elements of Assembly Language

### 3.7 Input/Output Using Macros Defined in IO.H

---

**The `atod` (ASCII to double) macro** is in many ways the inverse of the `dtoa` macro. It has only a single operand, the address of a string of ASCII character codes in storage, and it scans this area of memory for characters that represent a decimal number.

If it finds characters for a decimal number in the range 2,147,483,648 to 2,147,483,647, then the doubleword-length 2's complement form of the number is placed in the EAX register.

The *source* string may contain any number of leading blanks. These are skipped by `atod`. There may then be the ASCII code for `-` (minus) or the ASCII code for `+` (plus). A If `atod` is able to successfully convert a string of ASCII characters, then the overflow flag OF is set to 0. In all cases, the SF, ZF, and PF flags are set according to the value returned in EAX as follows:

- ✓ SF is 1 if the number is negative, and 0 otherwise
- ✓ ZF is 1 if the number is 0, and 0 if the number is nonzero
- ✓ PF reflects the parity of the number returned in EAX

## Chapter 3 :Elements of Assembly Language

### 3.7 Input/Output Using Macros Defined in IO.H

---

**The `atod` macro** will typically be used immediately after the input macro.

The input macro produces a string of ASCII codes, including a trailing null character. When `atod` is applied to this string, the null character serves as a terminating character for the scan. If `atod` is applied to a string that comes from some source other than input, the programmer must ensure that it has some trailing nondigit character to prevent `atod` from scanning too far

**The `atoi` (ASCII to integer) and `itoa` (integer to ASCII) macros** are the word-length versions of `atod` and `dtoa`. The `atoi` macro scans a string of characters and produces the corresponding word-length 2's complement value in AX. The `itoa` macro takes the 2's complement value stored in a word-length source and produces a string of exactly six characters representing this value in decimal. These macros are useful if you are dealing with values in the range -32,768 to 32,767.



# Chapter 3 :Elements of Assembly Language

## Chapter Summary

---

Chapter 3 introduced 80×86 assembly language as translated by the Microsoft MASM assembler.

An assembly language comment always starts with a semicolon. Other statements have the format

**name      mnemonic      operand(s)      ;comment**

where some of these fields may be optional.

The three types of assembly language statements are:

- ✓ **instructions-each corresponds to a CPU instruction**
- ✓ **directives-tell the assembler what to do**
- ✓ **macros-expand into additional statements**

# Chapter 3 :Elements of Assembly Language

## Chapter Summary

---

An assembly language program consists mainly of a data segment in which variables are defined and a code segment that contains statements to be executed at run time. To get an executable program, one must translate the program to object code using an assembler and then link the program using a linker. An executable program can be traced with a debugger like Windbg.

BYTE, DWORD, or WORD directives reserve bytes, doublewords, or words of storage and optionally assign initial values.

Instruction operands have three modes:

- ✓ **immediate-data built into the instruction**
- ✓ **register-data in a register**
- ✓ **memory-data in storage**

# Chapter 3 :Elements of Assembly Language

## Chapter Summary

---

Memory mode operands come in several formats, two of which are

- ✓ **direct-at an address in the instruction**
- ✓ **register indirect-data at an address in a register**

Several macros for input and output are defined in the file IO.H. They call procedures whose assembled versions are in the file IO.OBJ. The macros are:

- ✓ **output-to display a string on the monitor**
- ✓ **input-to input a string from the keyboard**
- ✓ **atod-to convert a string to a doubleword-length 2's complement number**
- ✓ **dtoa-to convert a doubleword-length 2's complement number to a string**
- ✓ **atoi-to convert a string to a word-length 2's complement number**
- ✓ **itoa-to convert a word-length 2's complement number to a string**