EE 213 Computer Organization and Assembly Language

Week # 5 Lecture # 12

13th Muharram ul Haram, 1440 A.H

24th September 2018

Minds open...

... Laptops closed

This presentation helps in delivering the lecture.
Take notes, interact and read text book to learn and gain knowledge.

Today’s Topics

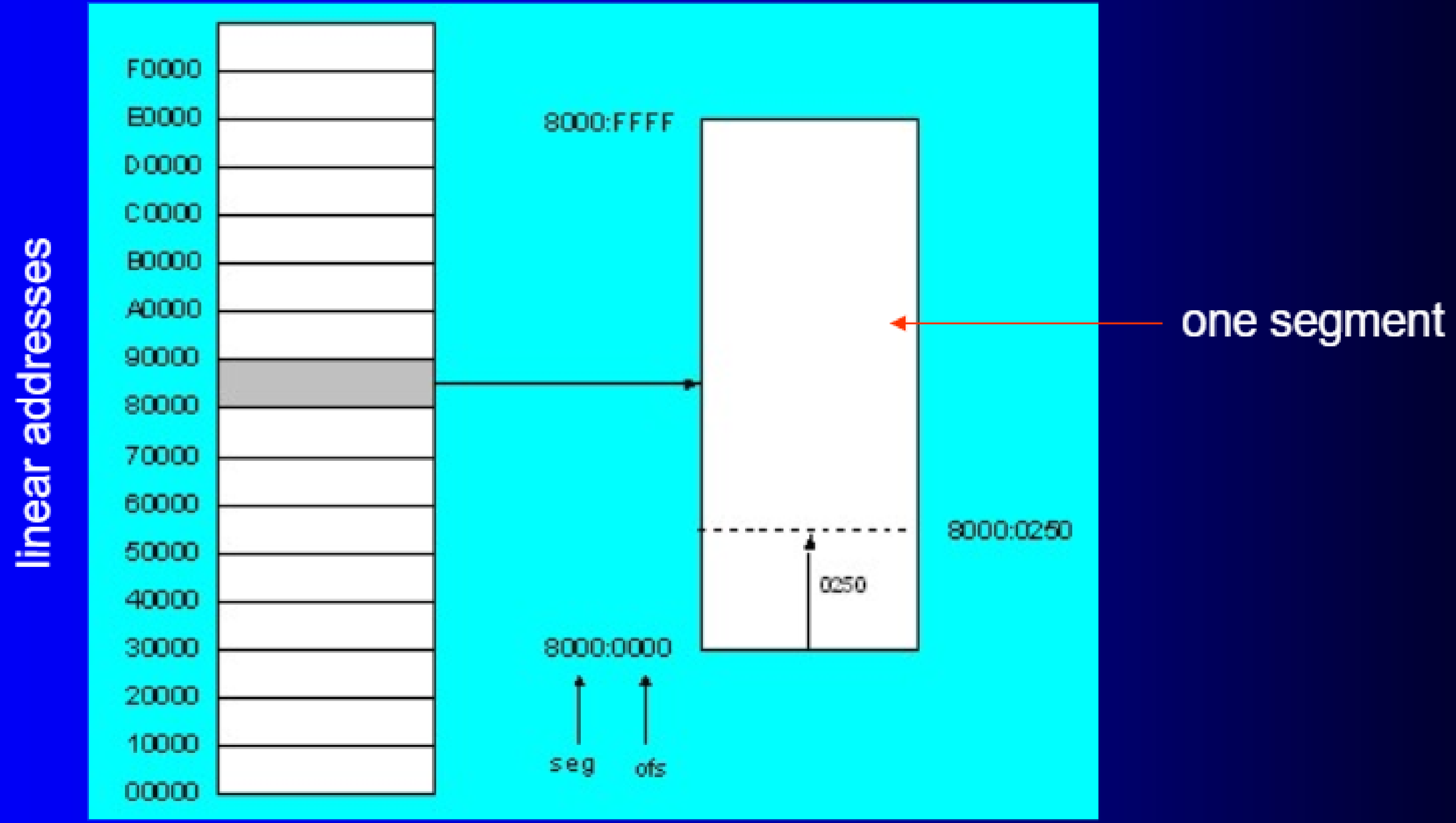
- Segmented Memory Model (Revision)

Logical and Physical Addresses

- Addresses specify the location of instructions and data
- Addresses that specify an absolute location in main memory are physical addresses (called linear addresses)
  - They appear on the address bus
- Addresses that specify a location relative to a point in the program are logical (or virtual) addresses
  - They are addresses used in the code and are independent of the structure of main memory
- Each logical address for the x86 consist of 2 parts:
  - A segment number used to specify a (logical) part of the program [The physical address of the segment]
  - A offset number used to specify a location relative to the beginning of the segment

Segmented Memory

Segmented memory addressing: absolute (linear) address is a combination of a 16-bit segment value added to a 16-bit offset



How is a 20 bit physical memory address calculated in the 8086 microprocessor?

8086 has a concept of Memory Segmentation. It is a method where the whole memory is segmented (divided) into smaller parts called segments. These segments are

- Code Segment (CS)
- Stack Segment (SS)
- Data Segment (DS)
- Extra Segment (ES)

Each Segment has a corresponding 16-bit Segment Register which holds the Base Address (starting Address) of the Segment. At any given time, 8086 can address 16-bit x 64KB = 256 KB of memory chunk out of 1MB.

8086 has 20bit address line. So the maximum value of address that can be addressed by 8086 is 2^20 = 1MB. So 8086 can address the locations ranging between 00000 H to FFFFF H. This 1MB memory is divided into 16 logical segments, each with a memory of 64KB.

How is a 20 bit physical memory address calculated in the 8086 microprocessor?

To locate any adress in the memory bank, it needs the Physical address of that memory location. It cannot get the 20-bit Physical adress using the 8086 Address Line or 16-bit Segment Registers alone.

In order to access memory location, you cannot pass 20-bit address directly to the processor. You need to tell the 16-bit address with respect to the segment. This 16-bit address with respect to the part (segment of 64KB) of the memory bank is called the offset.

How is a 20 bit physical memory address calculated in the 8086 microprocessor?

So, Physical Address = Base Address + Offset.

Suppose the Data Segment holds the Base Aaddress as 1000h and the data you need is present in the 0020h memory location (Offset) of the Data Segment. The calculation of the actual address is done as follows.

1. Left shift the 16-bit address present in the segment register by 4-bits

 0001 0000 0000 0000 (0000)

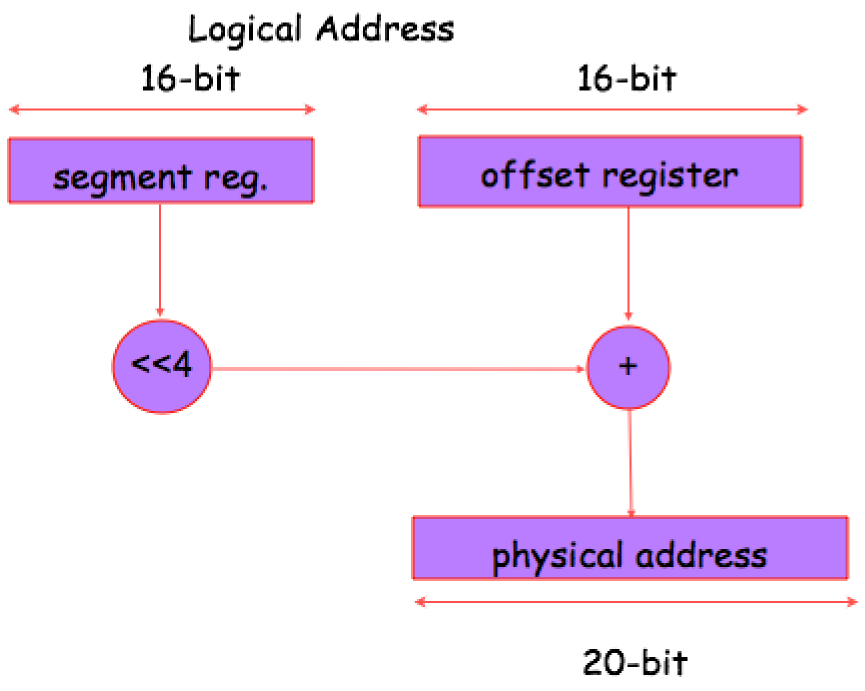
2. Add the 16-bit offset address to this shifted base address

0001 0000 0000 0000 0000

+ 0000 0000 0010 0000

0001 0000 0000 0010 0000

So the actual address turns out to be 10020h.



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]
```

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

Example	Operand Type
96	*Integer literal*
2 + 4	Integer expression
eax	Register
count	Memory

Figure 3–7 Assemble-Link-Execute cycle.

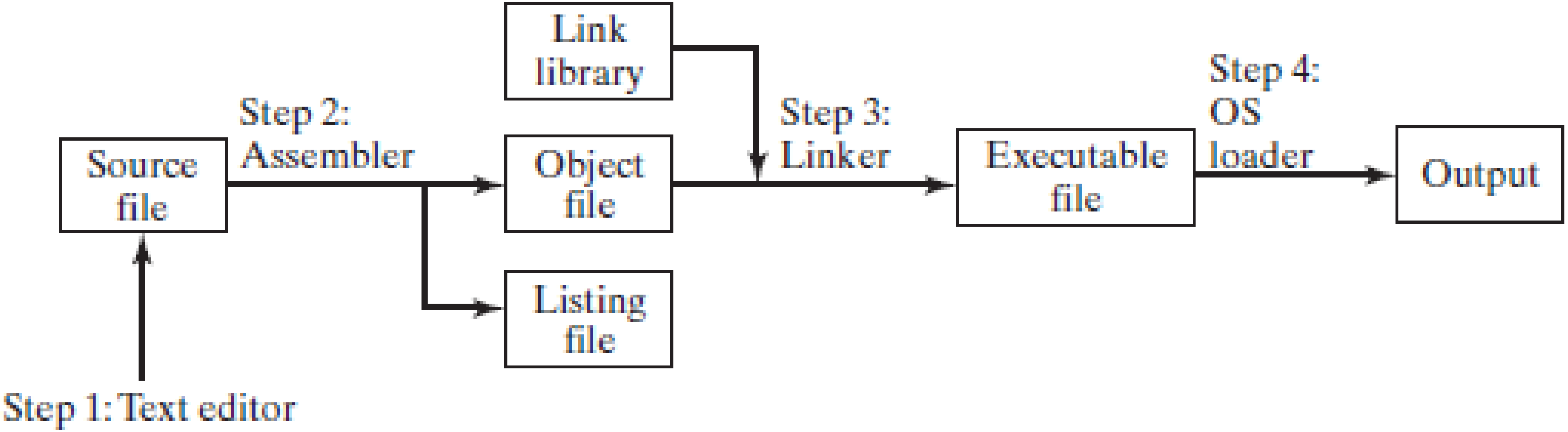


FIGURE 3–8 Excerpt from the AddTwo source listing file.

```
; AddTwo.asm - adds two 32-bit integers.
; Chapter 3 example

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO,dwExitCode:DWORD

 00000000                          .code
 00000000                          main PROC
 00000000  B8 00000005                mov   eax,5
 00000005  83 C0 06                   add   eax,6

                                      invoke ExitProcess,0
 00000008  6A 00                      push   +00000000h
 0000000A  E8 00000000 E              call   ExitProcess
 0000000F                          main ENDP
                                   END main
```

Table 3-2 Intrinsic Data Types.

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer. D stands for double
SDWORD	32-bit signed integer. SD stands for signed double
FWORD	48-bit integer (Far pointer in protected mode)
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit (10-byte) integer. T stands for Ten-byte
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

Table 3-3 Legacy Data Directives.

Directive	Usage
DB	8-bit integer
DW	16-bit integer
DD	32-bit integer or real
DQ	64-bit integer or real
DT	define 80-bit (10-byte) integer

3.4.4 Defining BYTE and SBYTE Data

The BYTE (define byte) and SBYTE (define signed byte) directives allocate storage for one or more unsigned or signed values. Each initializer must fit into 8 bits of storage. For example,

```
value1 BYTE  'A'                ; character literal
value2 BYTE   0                 ; smallest unsigned byte
value3 BYTE  255                ; largest unsigned byte
value4 SBYTE -128               ; smallest signed byte
value5 SBYTE +127               ; largest signed byte
```

Multiple Initializers

If multiple initializers are used in the same data definition, its label refers only to the offset of the first initializer. In the following example, assume list is located at offset 0000. If so, the value 10 is at offset 0000, 20 is at offset 0001, 30 is at offset 0002, and 40 is at offset 0003:

```
list BYTE 10,20,30,40

list BYTE 10,20,30,40
     BYTE 50,60,70,80
     BYTE 81,82,83,84
```

```
list1 BYTE 10, 32, 41h, 00100010b
list2 BYTE 0Ah, 20h, 'A', 22h

greeting1 BYTE "Good afternoon",0
greeting2 BYTE 'Good night',0
```

```
greeting1 BYTE "Welcome to the Encryption Demo program "
  BYTE "created by Kip Irvine.",0dh,0ah
  BYTE "If you wish to modify this program, please "
  BYTE "send me a copy.",0dh,0ah,0
```

DUP Operator

The *DUP operator* allocates storage for multiple data items, using a integer expression as a counter. It is particularly useful when allocating space for a string or array, and can be used with initialized or uninitialized data:

```
BYTE 20 DUP(0)              ; 20 bytes, all equal to zero
BYTE 20 DUP(?)              ; 20 bytes, uninitialized
BYTE  4 DUP("STACK")        ; 20 bytes: "STACKSTACKSTACKSTACK"
```

3.4.5 Defining WORD and SWORD Data

The WORD (define word) and SWORD (define signed word) directives create storage for one or more 16-bit integers:

```
word1  WORD   65535          ; largest unsigned value
word2  SWORD  -32768         ; smallest signed value
word3  WORD   ?              ; uninitialized, unsigned
```

The legacy DW directive can also be used:

```
val1  DW 65535               ; unsigned
val2  DW -32768              ; signed
```

Array of 16-Bit Words Create an array of words by listing the elements or using the DUP operator. The following array contains a list of values:

```
myList  WORD 1,2,3,4,5
```

Little endian vs Big endian order

Figure 3–14 Little-endian representation of 12345678h.

0000:	78
0001:	56
0002:	34
0003:	12

Figure 3–15 Big-endian representation of 12345678h.

0000:	12
0001:	34
0002:	56
0003:	78

3.4.12 Declaring Uninitialized Data

The .DATA? directive declares uninitialized data. When defining a large block of uninitialized data, the .DATA? directive reduces the size of a compiled program. For example, the following code is declared efficiently:

```
.data
smallArray DWORD 10 DUP(0)       ; 40 bytes
.data?
bigArray DWORD 5000 DUP(?)       ; 20,000 bytes, not initialized
```

The following code, on the other hand, produces a compiled program 20,000 bytes larger:

```
.data
smallArray DWORD 10 DUP(0)       ; 40 bytes
bigArray DWORD 5000 DUP(?)       ; 20,000 bytes
```

3.5.1 Equal-Sign Directive

The *equal-sign directive* associates a symbol name with an integer expression (see Section 3.1.3). The syntax is

```
name = expression
```

Ordinarily, expression is a 32-bit integer value. When a program is assembled, all occurrences of *name* are replaced by *expression* during the assembler's preprocessor step. Suppose the following statement occurs near the beginning of a source code file:

```
COUNT = 500
```

Further, suppose the following statement should be found in the file 10 lines later:

```
mov eax, COUNT
```

When the file is assembled, MASM will scan the source file and produce the corresponding code lines:

```
mov eax, 500
```

Using the DUP Operator Section 3.4.4 showed how to use the DUP operator to create storage for arrays and strings. The counter used by DUP should be a symbolic constant, to simplify program maintenance. In the next example, if COUNT has been defined, it can be used in the following data definition:

```
array dword COUNT DUP(0)
```

Redefinitions A symbol defined with = can be redefined within the same program. The following example shows how the assembler evaluates COUNT as it changes value:

```
COUNT = 5
mov al,COUNT                        ; AL = 5
COUNT = 10
mov al,COUNT                        ; AL = 10
COUNT = 100
mov al,COUNT                        ; AL = 100
```

The changing value of a symbol such as COUNT has nothing to do with the runtime execution order of statements. Instead, the symbol changes value according to the assembler's sequential processing of the source code during the assembler's preprocessing stage.

3.5.3 EQU Directive

The *EQU directive* associates a symbolic name with an integer expression or some arbitrary text. There are three formats:

```
name EQU expression
name EQU symbol
name EQU <text>
```

In the first format, *expression* must be a valid integer expression (see Section 3.1.3). In the second format, *symbol* is an existing symbol name, already defined with = or EQU. In the third format, any text may appear within the brackets <. . .>. When the assembler encounters *name* later in the program, it substitutes the integer value or text for the symbol.

EQU can be useful when defining a value that does not evaluate to an integer. A real number constant, for example, can be defined using EQU:

```
PI EQU <3.1416>
```

Example Suppose we would like to define a symbol that counts the number of cells in a 10-by-10 integer matrix. We will define symbols two different ways, first as an integer expression and second as a text expression. The two symbols are then used in data definitions:

```
matrix1  EQU   10 * 10
matrix2  EQU  <10 * 10>
.data
M1 WORD matrix1
M2 WORD matrix2
```

The assembler produces different data definitions for **M1** and **M2**. The integer expression in **matrix1** is evaluated and assigned to **M1**. On the other hand, the text in **matrix2** is copied directly into the data definition for **M2**:

```
M1 WORD  100
M2 WORD  10 * 10
```

No Redefinition Unlike the = directive, a symbol defined with EQU cannot be redefined in the same source code file. This restriction prevents an existing symbol from being inadvertently assigned a new value.

3.5.4 TEXTEQU Directive

The *TEXTEQU directive*, similar to EQU, creates what is known as a *text macro*. There are three different formats: the first assigns text, the second assigns the contents of an existing text macro, and the third assigns a constant integer expression:

```
name TEXTEQU <text>
name TEXTEQU textmacro
name TEXTEQU %constExpr
```

For example, the **prompt1** variable uses the **continueMsg** text macro:

```
continueMsg TEXTEQU <"Do you wish to continue (Y/N)?">
.data
prompt1 BYTE continueMsg
```

Text macros can build on each other. In the next example, **count** is set to the value of an integer expression involving **rowSize**. Then the symbol **move** is defined as **mov**. Finally, **setupAL** is built from **move** and **count**:

```
rowSize = 5
count   TEXTEQU  %(rowSize * 2)
move    TEXTEQU  <mov>
setupAL TEXTEQU  <move al,count>
```

Therefore, the statement

```
setupAL
```

would be assembled as

```
mov al,10
```

A symbol defined by TEXTEQU can be redefined at any time.