EE 213 Computer Organization and Assembly Language

Week # 5 Lecture # 13, 14

15th,17th Muharram ul Haram, 1440 A.H

26th, 28th September 2018

These slides contains materials taken from various sources. I fully acknowledge all copyrights.

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

- Specific coverage of topics of Chapter # 4 which are not covered before.
- Overall coverage of Chapter # 4

3.5.2 Calculating the Sizes of Arrays and Strings

When using an array, we usually like to know its size. The following example uses a constant named **ListSize** to declare the size of **list**:

```
list BYTE 10,20,30,40
ListSize = 4
```

Explicitly stating an array's size can lead to a programming error, particularly if you should later insert or remove array elements. A better way to declare an array size is to let the assembler calculate its value for you. The $ operator (*current location counter*) returns the offset associated with the current program statement. In the following example, **ListSize** is calculated by subtracting the offset of **list** from the current location counter ($):

```
list BYTE 10,20,30,40
ListSize = ($ - list)
```

ListSize must follow immediately after **list**. The following, for example, produces too large a value (24) for **ListSize** because the storage used by **var2** affects the distance between the current location counter and the offset of **list**:

```
list BYTE 10,20,30,40
var2 BYTE 20 DUP(?)
ListSize = ($ - list)
```

Rather than calculating the length of a string manually, let the assembler do it:

```
myString  BYTE "This is a long string, containing"
          BYTE "any number of characters"
myString_len = ($ – myString)
```

Arrays of Words and DoubleWords When calculating the number of elements in an array containing values other than bytes, you should always divide the total array size (in bytes) by the size of the individual array elements. The following code, for example, divides the address range by 2 because each word in the array occupies 2 bytes (16 bits):

```
list  WORD  1000h,2000h,3000h,4000h
ListSize = ($ - list) / 2
```

Similarly, each element of an array of doublewords is 4 bytes long, so its overall length must be divided by four to produce the number of array elements:

```
list  DWORD  10000000h,20000000h,30000000h,40000000h
ListSize = ($ -list) / 4
```

TABLE 4-1 Instruction Operand Notation, 32-Bit Mode.

Operand	Description
reg8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
reg16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
reg32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	Any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
reg/mem8	8-bit operand, which can be an 8-bit general register or memory byte
reg/mem16	16-bit operand, which can be a 16-bit general register or memory word
reg/mem32	32-bit operand, which can be a 32-bit general register or memory doubleword
mem	An 8-, 16-, or 32-bit memory operand

Alternative Notation. Some programmers prefer to use the following notation with direct operands because the brackets imply a dereference operation:

```
mov  al,[var1]
```

MASM permits this notation, so you can use it in your own programs if you want. Because so many programs (including those from Microsoft) are printed without the brackets, we will only use them in this book when an arithmetic expression is involved:

```
mov  al,[var1 + 5]
```

(This is called a direct-offset operand, a subject discussed at length in Section 4.1.8.)

In nearly all assembly language instructions, the left-hand operand is the destination and the right-hand operand is the source. MOV is very flexible in its use of operands, as long as the following rules are observed:

- Both operands must be the same size.
- Both operands cannot be memory operands.
- The instruction pointer register (IP, EIP, or RIP) cannot be a destination operand.

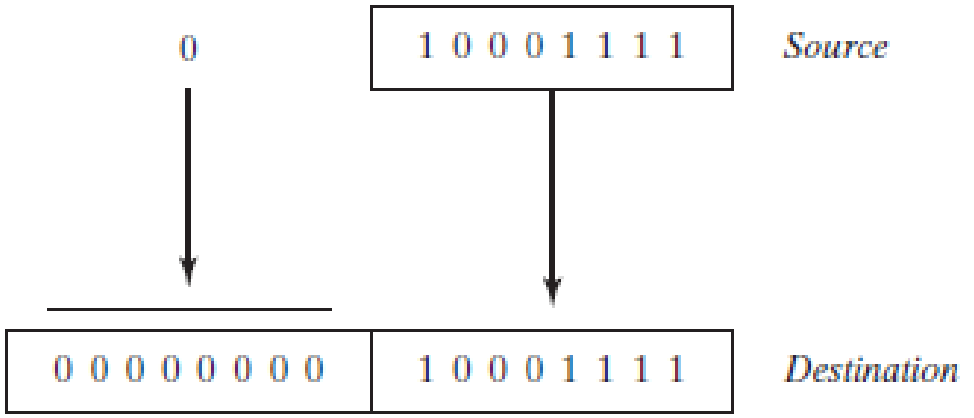
Here is a list of the standard MOV instruction formats:

```
MOV reg,reg
MOV mem,reg
MOV reg,mem
MOV mem,imm
MOV reg,imm
```

Two memory operands are not allowed.
DO NOT use these MOV instructions

- **MOV var1, var2**
- **MOV [EAX], [EBX]**

Figure 4–1 Using MOVZX to copy a byte into a 16-bit destination.



The following examples use registers for all operands, showing all the size variations:

```
mov    bx,0A69Bh
movzx  eax,bx                    ; EAX = 0000A69Bh
movzx  edx,bl                    ; EDX = 0000009Bh
movzx  cx,bl                     ; CX  = 009Bh
```

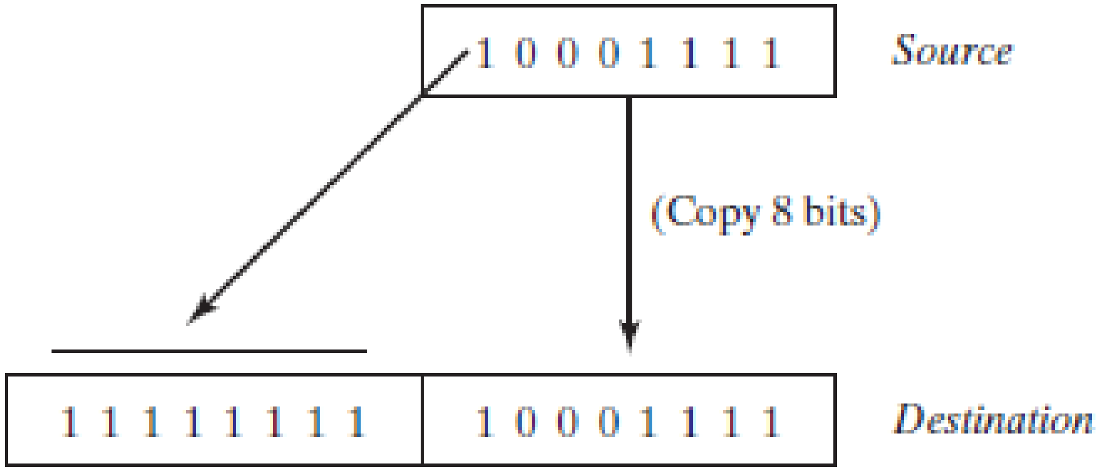
The following examples use memory operands for the source and produce the same results:

```
.data
byte1  BYTE 9Bh
word1  WORD 0A69Bh
.code
movzx  eax,word1                 ; EAX = 0000A69Bh
movzx  edx,byte1                 ; EDX = 0000009Bh
movzx  cx,byte1                  ; CX  = 009Bh
```

A hexadecimal constant has its highest bit set if its most significant hexadecimal digit is greater than 7. In the following example, the hexadecimal value moved to BX is A69B, so the leading "A" digit tells us that the highest bit is set. (The leading zero appearing before A69B is just a notational convenience so the assembler does not mistake the constant for the name of an identifier.)

```
mov    bx,0A69Bh
movsx  eax,bx                          ; EAX = FFFFA69Bh
movsx  edx,bl                          ; EDX = FFFFFF9Bh
movsx  cx,bl                           ; CX  = FF9Bh
```

Figure 4–2 Using MOVSX to copy a byte into a 16-bit destination.



The rules for operands in the XCHG instruction are the same as those for the MOV instruction (Section 4.1.4), except that XCHG does not accept immediate operands. In array sorting applications, XCHG provides a simple way to exchange two array elements. Here are a few examples using XCHG:

```
xchg  ax,bx                 ; exchange 16-bit regs
xchg  ah,al                 ; exchange 8-bit regs
xchg  var1,bx               ; exchange 16-bit mem op with BX
xchg  eax,ebx               ; exchange 32-bit regs
```

To exchange two memory operands, use a register as a temporary container and combine MOV with XCHG:

```
mov   ax,val1
xchg  ax,val2
mov   val1,ax
```

Word and Doubleword Arrays In an array of 16-bit words, the offset of each array element is 2 bytes beyond the previous one. That is why we add 2 to **ArrayW** in the next example to reach the second element:

```
.data
arrayW WORD 100h,200h,300h
.code
mov  ax,arrayW                        ; AX = 100h
mov  ax,[arrayW+2]                    ; AX = 200h
```

Similarly, the second element in a doubleword array is 4 bytes beyond the first one:

```
.data
arrayD DWORD 10000h,20000h
.code
mov  eax,arrayD                       ; EAX = 10000h
mov  eax,[arrayD+4]                   ; EAX = 20000h
```

4.2.1 INC and DEC Instructions

The INC (increment) and DEC (decrement) instructions, respectively, add 1 and subtract 1 from a register or memory operand. The syntax is

```
INC reg/mem
DEC reg/mem
```

Following are some examples:

```
.data
myWord WORD 1000h
.code
inc  myWord                          ; myWord = 1001h
mov  bx,myWord
dec  bx                              ; BX = 1000h
```

The Overflow, Sign, Zero, Auxiliary Carry, and Parity flags are changed according to the value of the destination operand. The INC and DEC instructions do not affect the Carry flag (which is something of a surprise).

4.2.2 ADD Instruction

The ADD instruction adds a source operand to a destination operand of the same size. The syntax is

```
ADD dest,source
```

Source is unchanged by the operation, and the sum is stored in the destination operand. The set of possible operands is the same as for the MOV instruction (Section 4.1.4). Here is a short code example that adds two 32-bit integers:

```
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code
mov  eax,var1                         ; EAX = 10000h
add  eax,var2                         ; EAX = 30000h
```

4.2.3 SUB Instruction

The SUB instruction subtracts a source operand from a destination operand. The set of possible operands is the same as for the ADD and MOV instructions. The syntax is

```
SUB dest,source
```

Here is a short code example that subtracts two 32-bit integers:

```
.data
var1 DWORD 30000h
var2 DWORD 10000h
.code
mov  eax,var1                         ; EAX = 30000h
sub  eax,var2                         ; EAX = 20000h
```

Flags The Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity flags are changed according to the value that is placed in the destination operand.

4.2.4 NEG Instruction

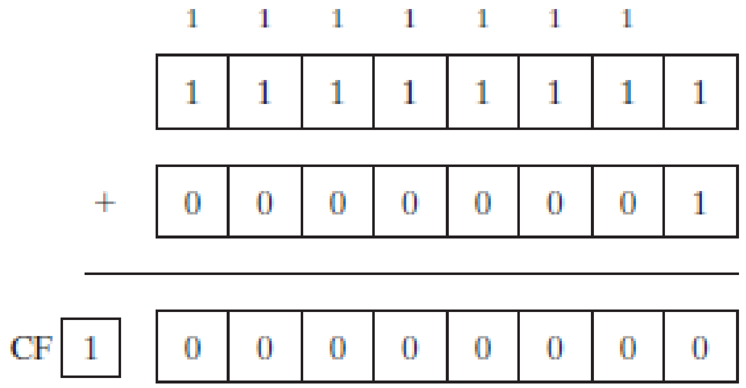
The NEG (negate) instruction reverses the sign of a number by converting the number to its two's complement. The following operands are permitted:

```
NEG reg
NEG mem
```

(Recall that the two's complement of a number can be found by reversing all the bits in the destination operand and adding 1.)

Flags The Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity flags are changed according to the value that is placed in the destination operand.

Figure 4–3 Adding 1 to 0FFh sets the Carry flag.



On the other hand, if 1 is added to 00FFh in AX, the sum easily fits into 16 bits and the Carry flag is clear:

```
mov  ax,00FFh
add  ax,1                           ; AX = 0100h, CF = 0
```

But adding 1 to FFFFh in the AX register generates a Carry out of the high bit position of AX:

```
mov  ax,0FFFFh
add  ax,1                           ; AX = 0000, CF = 1
```

Subtraction and the Carry Flag A subtract operation sets the Carry flag when a larger unsigned integer is subtracted from a smaller one. Figure 4-4 shows what happens when we subtract 2 from 1, using 8-bit operands. Here is the corresponding assembly code:

```
mov  al,1
sub  al,2                           ; AL = FFh, CF = 1
```

4.2.8 Section Review

Use the following data for Questions 1-5:

```
.data
val1 BYTE  10h
val2 WORD  8000h
val3 DWORD 0FFFFh
val4 WORD  7FFFh
```

1. Write an instruction that increments **val2**.
2. Write an instruction that subtracts **val3** from EAX.
3. Write instructions that subtract **val4** from **val2**.
4. If **val2** is incremented by 1 using the ADD instruction, what will be the values of the Carry and Sign flags?
5. If **val4** is incremented by 1 using the ADD instruction, what will be the values of the Overflow and Sign flags?
6. Where indicated, write down the values of the Carry, Sign, Zero, and Overflow flags after each instruction has executed:

```
mov  ax,7FF0h
add  al,10h          ; a. CF =    SF =    ZF =    OF =
add  ah,1            ; b. CF =    SF =    ZF =    OF =
add  ax,2            ; c. CF =    SF =    ZF =    OF =
```

4.3 Data-Related Operators and Directives

Operators and directives are not executable instructions; instead, they are interpreted by the assembler. You can use a number of assembly language directives to get information about the addresses and size characteristics of data:

- The OFFSET operator returns the distance of a variable from the beginning of its enclosing segment.
- The PTR operator lets you override an operand's default size.
- The TYPE operator returns the size (in bytes) of an operand or of each element in an array.
- The LENGTHOF operator returns the number of elements in an array.
- The SIZEOF operator returns the number of bytes used by an array initializer.

OFFSET Examples

In the next example, we declare three different types of variables:

```
.data
bVal  BYTE  ?
wVal  WORD  ?
dVal  DWORD ?
dVal2 DWORD ?
```

If **bVal** were located at offset 00404000 (hexadecimal), the OFFSET operator would return the following values:

```
mov  esi,OFFSET bVal                 ; ESI = 00404000h
mov  esi,OFFSET wVal                 ; ESI = 00404001h
mov  esi,OFFSET dVal                 ; ESI = 00404003h
mov  esi,OFFSET dVal2                ; ESI = 00404007h
```

OFFSET can also be applied to a direct-offset operand. Suppose **myArray** contains five 16-bit words. The following MOV instruction obtains the offset of **myArray**, adds 4, and moves the resulting address to ESI. We can say that ESI points to the third integer in the array:

```
.data
myArray WORD 1,2,3,4,5
.code
mov  esi,OFFSET myArray + 4
```

4.3.3 PTR Operator

You can use the PTR operator to override the declared size of an operand. This is only necessary when you're trying to access the operand using a size attribute that is different from the one assumed by the assembler.

Suppose, for example, that you would like to move the lower 16 bits of a doubleword variable named **myDouble** into AX. The assembler will not permit the following move because the operand sizes do not match:

```
.data
myDouble  DWORD  12345678h
.code
mov  ax,myDouble                    ; error
```

But the WORD PTR operator makes it possible to move the low-order word (5678h) to AX:

```
mov  ax,WORD PTR myDouble
```

Similarly, we could use the BYTE PTR operator to move a single byte from **myDouble** to BL:

```
mov   bl,BYTE PTR myDouble              ; 78h
```

Note that PTR must be used in combination with one of the standard assembler data types, BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD, or TBYTE.

Moving Smaller Values into Larger Destinations We might want to move two smaller values from memory to a larger destination operand. In the next example, the first word is copied to the lower half of EAX and the second word is copied to the upper half. The DWORD PTR operator makes this possible:

```
.data
wordList WORD 5678h,1234h
.code
mov  eax,DWORD PTR wordList             ; EAX = 12345678h
```

Using PTR with Indirect Operands The size of an operand may not be evident from the context of an instruction. The following instruction causes the assembler to generate an "operand must have size" error message:

```
inc [esi]                          ; error: operand must have size
```

The assembler does not know whether ESI points to a byte, word, doubleword, or some other size. The PTR operator confirms the operand size:

```
inc BYTE PTR [esi]
```

4.3.4 TYPE Operator

The TYPE operator returns the size, in bytes, of a single element of a variable. For example, the TYPE of a byte equals 1, the TYPE of a word equals 2, the TYPE of a doubleword is 4, and the TYPE of a quadword is 8. Here are examples of each:

```
.data
var1 BYTE  ?
var2 WORD  ?
var3 DWORD ?
var4 QWORD ?
```

The following table shows the value of each TYPE expression.

Expression	Value
TYPE var1	1
TYPE var2	2
TYPE var3	4
TYPE var4	8

4.3.5 LENGTHOF Operator

The LENGTHOF operator counts the number of elements in an array, defined by the values appearing on the same line as its label. We will use the following data as an example:

```
.data
byte1    BYTE  10,20,30
array1   WORD  30 DUP(?),0,0
array2   WORD  5 DUP(3 DUP(?))
array3   DWORD 1,2,3,4
digitStr BYTE  "12345678",0
```

When nested DUP operators are used in an array definition, LENGTHOF returns the product of the two counters. The following table lists the values returned by each LENGTHOF expression:

Expression	Value
LENGTHOF byte1	3
LENGTHOF array1	30 + 2
LENGTHOF array2	5 * 3
LENGTHOF array3	4
LENGTHOF digitStr	9

4.3.6 SIZEOF Operator

The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE. In the following example, **intArray** has TYPE = 2 and LENGTHOF = 32. Therefore, SIZEOF **intArray** equals 64:

```
.data
intArray WORD 32 DUP(0)
.code
mov  eax,SIZEOF intArray          ; EAX = 64
```

4.4.2 Arrays

Indirect operands are ideal tools for stepping through arrays. In the next example, **arrayB** contains 3 bytes. As ESI is incremented, it points to each byte, in order:

```
.data
arrayB  BYTE 10h,20h,30h
.code
mov  esi,OFFSET arrayB
mov  al,[esi]                        ; AL = 10h
inc  esi
mov  al,[esi]                        ; AL = 20h
inc  esi
mov  al,[esi]                        ; AL = 30h
```

If we use an array of 16-bit integers, we add 2 to ESI to address each subsequent array element:

```
.data
arrayW  WORD 1000h,2000h,3000h
.code
mov  esi,OFFSET arrayW
mov  ax,[esi]                        ; AX = 1000h
add  esi,2
mov  ax,[esi]                        ; AX = 2000h
add  esi,2
mov  ax,[esi]                        ; AX = 3000h
```

4.4.3 Indexed Operands

An *indexed operand* adds a constant to a register to generate an effective address. Any of the 32-bit general-purpose registers may be used as index registers. There are different notational forms permitted by MASM (the brackets are part of the notation):

```
constant[reg]
[constant + reg]
```

The first notational form combines the name of a variable with a register. The variable name is translated by the assembler into a constant that represents the variable's offset. Here are examples that show both notational forms:

arrayB[esi]	[arrayB + esi]
arrayD[ebx]	[arrayD + ebx]

Indexed operands are ideally suited to array processing. The index register should be initialized to zero before accessing the first array element:

```
.data
arrayB BYTE 10h,20h,30h
.code
mov  esi,0
mov  al,arrayB[esi]              ; AL = 10h
```

In the following example, we add 1 to AX each time the loop repeats. When the loop ends, AX = 5 and ECX = 0:

```
        mov   ax,0
        mov   ecx,5
L1:
        inc   ax
        loop L1
```

A common programming error is to inadvertently initialize ECX to zero before beginning a loop. If this happens, the LOOP instruction decrements ECX to FFFFFFFFh, and the loop repeats 4,294,967,296 times! If CX is the loop counter (in real-address mode), it repeats 65,536 times.

If you need to modify ECX inside a loop, you can save it in a variable at the beginning of the loop and restore it just before the LOOP instruction:

```
.data
count DWORD ?
.code
    mov   ecx,100              ; set loop count
top:
    mov   count,ecx            ; save the count
    .
    mov   ecx,20               ; modify ECX
    .
    mov   ecx,count            ; restore loop count
    loop  top
```

Nested Loops When creating a loop inside another loop, special consideration must be given to the outer loop counter in ECX. You can save it in a variable:

```
.data
count DWORD ?
.code
    mov   ecx,100              ; set outer loop count
L1:
    mov   count,ecx            ; save outer loop count
    mov   ecx,20               ; set inner loop count
L2:
    .
    .
    loop  L2                   ; repeat the inner loop
    mov   ecx,count            ; restore outer loop count
    loop  L1                   ; repeat the outer loop
```

```
; Summing an Array                          (SumArray.asm)
.386
.model flat,stdcall
.stack 4096
ExitProcess proto,dwExitCode:dword
.data
intarray DWORD 10000h,20000h,30000h,40000h

.code
main PROC

     mov  edi,OFFSET intarray       ; 1: EDI = address of intarray
     mov  ecx,LENGTHOF intarray     ; 2: initialize loop counter
     mov  eax,0                     ; 3: sum = 0
L1:                                 ; 4: mark beginning of loop
     add  eax,[edi]                 ; 5: add an integer
     add  edi,TYPE intarray         ; 6: point to next element
     loop L1                        ; 7: repeat until ECX = 0

     invoke ExitProcess,0
main ENDP
END main
```

```
; Copying a String                              (CopyStr.asm)

.386
.model flat,stdcall
.stack 4096
ExitProcess proto,dwExitCode:dword
.data
source  BYTE  "This is the source string",0
target  BYTE  SIZEOF source DUP(0)

.code
main PROC
     mov  esi,0                        ; index register
     mov  ecx,SIZEOF source            ; loop counter
L1:
     mov  al,source[esi]               ; get a character from source
     mov  target[esi],al               ; store it in the target
     inc  esi                          ; move to next character
     loop L1                           ; repeat for entire string

     invoke ExitProcess,0
main ENDP
END main
```

★★★ 6. Reverse an Array

Use a loop with indirect or indexed addressing to reverse the elements of an integer array in place. Do not copy the elements to any other array. Use the SIZEOF, TYPE, and LENGTHOF operators to make the program as flexible as possible if the array size and type should be changed in the future.

★★★ 7. Copy a String in Reverse Order

Write a program with a loop and indirect addressing that copies a string from **source** to **target**, reversing the character order in the process. Use the following variables:

```
source BYTE "This is the source string",0
target BYTE SIZEOF source DUP('#')
```

★★★ 8. Shifting the Elements in an Array

Using a loop and indexed addressing, write code that rotates the members of a 32-bit integer array forward one position. The value at the end of the array must wrap around to the first position. For example, the array [10,20,30,40] would be transformed into [40,10,20,30].

Protected Mode Memory Access

- 32-bit **Protected Mode** supports *much larger data structures* than **Real mode**.
- Because code, data, and stack reside in the *same segment*, each segment register can hold the same value that *never needs to change*.
- Rather than using a formula (such as CS:IP) to determine the physical address, protected mode processors use a *look up table*.
- Segment registers simply point to OS data structures that contain the information needed to access a location.
- **Protected mode** uses *privilege levels* to maintain system integrity and security.
- Programs cannot access data or code that is in a higher privilege level.
- Application programs cannot make use of protected mode by themselves.
- The operating system must set up and manage a protected mode.
- Capability provided by Linux and Windows NT/2000/XP/Vista systems.
- Each address is a 32-bit quantity.
- All of the general-purpose registers are 32 bits in size.

Protected Mode Memory Access