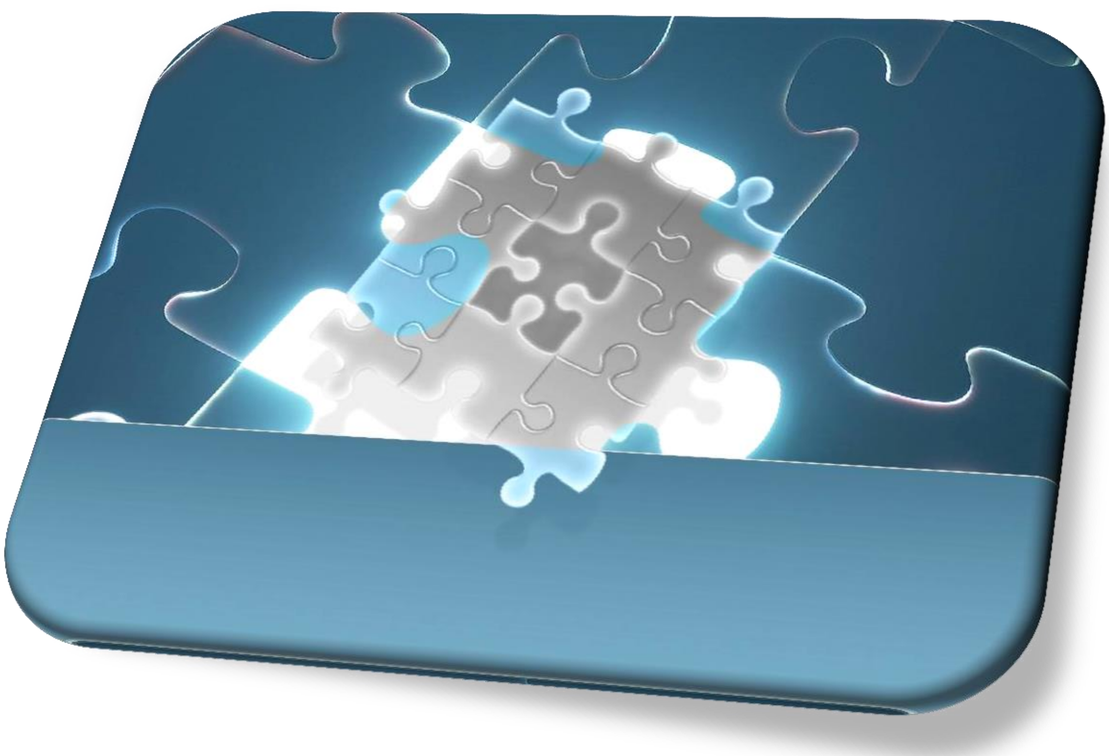
Islamic University – Gaza
Engineering Faculty
Department of Computer Engineering
ECOM 2025: Assembly Language Discussion

Chapter 4

Data Transfers, Addressing, and Arithmetic



Eng. Eman R. Habib

March, 2014

4.1 Data Transfer Instructions

❖ Operand Types:

- Immediate: uses a numeric literal expression
- Register: uses a named register in the CPU
- Memory: references a memory location

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

❖ MOV Instruction

The MOV instruction copies data from a source operand to a destination operand.

MOV *destination,source* → *destination = source*

The destination operand's contents change, but the source operand is unchanged.

MOV rules:

- Both operands must be the same size.
- Both operands cannot be memory operands.
- CS, EIP, and IP cannot be destination operands.
- An immediate value cannot be moved to a segment register.

The variants of MOV:

```
MOV reg,reg
MOV mem,reg
MOV reg,mem
MOV mem,imm
MOV reg,imm
MOV reg/mem16,sreg
MOV sreg,reg/mem16
```

Example:

```
.data
oneByte BYTE 78h
oneWord WORD 1234h
oneDword DWORD 12345678h
.code
mov eax,0            ; EAX = 00000000h
mov al,oneByte       ; EAX = 00000078h
mov ax,oneWord       ; EAX = 00001234h
mov eax,oneDword     ; EAX = 12345678h
mov ax,0             ; EAX = 12340000h
```

❖ MOVZX Instruction

The MOVZX instruction (move with zero-extend) copies the contents of a source operand into a destination operand and zero-extends the value to 16 or 32 bits. (Fills the upper part of the destination with zeros)

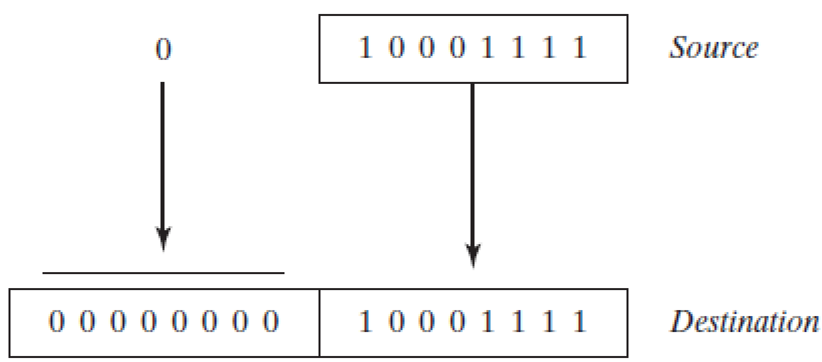
MOVZX destination, source

The variants of MOVZX:

```
MOVZX reg32,reg/mem8
MOVZX reg32,reg/mem16
MOVZX reg16,reg/mem8
```

Examples:

```
.data
byteVal BYTE 10001111b
.code
movzx ax,byteVal     ; AX = 0000000010001111b
```




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

❖ MOVSX Instruction

The MOVSX instruction (move with sign-extend) copies the contents of a source operand into a destination operand and sign-extends the value to 16 or 32 bits. (Fills the upper part of the destination with a copy of the source operand's highest bit)

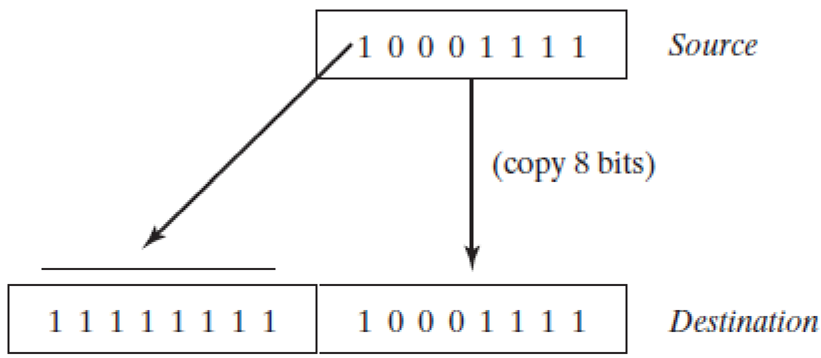
MOVSX destination, source

The variants of MOVSX:

```
MOVSX reg32,reg/mem8
MOVSX reg32,reg/mem16
MOVSX reg16,reg/mem8
```

Examples:

```
.data
byteVal BYTE 10001111b
.code
movsx ax,byteVal ; AX = 1111111110001111b
```




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

❖ XCHG Instruction

The XCHG (exchange data) instruction exchanges the contents of two operands.

The variants of XCHG:

```
XCHG reg,reg
XCHG reg,mem
XCHG mem,reg
```

Examples:

```
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
```

The rules for operands in the XCHG instruction are the same as those for the MOV instruction except that XCHG does not accept immediate operands.

❖ Direct-Offset Operands

You can add a displacement to the name of a variable, creating a direct-offset operand. This lets you access memory locations that may not have explicit labels.

```
arrayB BYTE 10h,20h,30h,40h,50h
```

If we use MOV with **arrayB** as the source operand, we automatically move the first byte in the array:

```
mov al,arrayB ; AL = 10h
```

We can access the second byte in the array by adding 1 to the offset of **arrayB**:

```
mov al,[arrayB+1] ; AL = 20h
```

The third byte is accessed by adding 2:

```
mov al,[arrayB+2] ; AL = 30h
```

The brackets are not required by MASM, so the following statements are equivalent:

```
mov al,[arrayB+1]
mov al,arrayB+1
```

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
```

Note:
The MOV instruction never affects the flags.

❖ Section 4.1 Review

1. **What are the three basic types of operands?**
   Register, immediate, and memory
2. ***(True/False):*** **The destination operand of a MOV instruction cannot be a segment register.**

3. ***(True/False):*** **In a MOV instruction, the second operand is known as the *destination* operand.**

4. ***(True/False):*** **The EIP register cannot be the destination operand of a MOV instruction.**

5. **In the operand notation used by Intel, what does *reg/mem32* indicate?**
   A 32-bit register or memory operand

6. **In the operand notation used by Intel, what does *imm16* indicate?**
   A 16-bit immediate (constant) operand

Use the following variable definitions for the remaining questions in this section:

```
.data
var1 SBYTE -4,-2,3,1
var2 WORD 1000h,2000h,3000h,4000h
var3 SWORD -16,-42
var4 DWORD 1,2,3,4,5
```

7. **For each of the following statements, state whether or not the instruction is valid:**

a. `mov ax,var1` not valid, both operands must be the same size
b. `mov ax,var2` valid
c. `mov eax,var3` not valid, both operands must be the same size
d. `mov var2,var3` not valid, both operands cannot be memory operands.
e. `movzx ax,var2` not valid, destination must be larger than source
f. `movzx var2,al` not valid, destination must be register
g. `mov ds,ax` valid
h. `mov ds,1000h` not valid, an immediate value cannot be moved to a segment register.

8. **What will be the hexadecimal value of the destination operand after each of the following instructions execute in sequence?**

```
mov al,var1 ;          a. FCh
mov ah,[var1+3] ;      b. 01h
```

9. What will be the value of the destination operand after each of the following instructions execute in sequence?

```
mov ax,var2 ;          a. 1000h
mov ax,[var2+4] ;      b. 3000h
mov ax,var3 ;          c.FFF0h
mov ax,[var3-2] ;      d.4000h
```

10. What will be the value of the destination operand after each of the following instructions execute in sequence?

```
mov edx,var4 ;         a. 00000001h
movzx edx,var2 ;       b. 00001000h
mov edx,[var4+4] ;     c. 00000002h
movsx edx,var1 ;       d. FFFFFFFCh
```

4.2 Addition and Subtraction

❖ INC and DEC Instructions

The INC (increment) adds 1 to a single operand.

```
INC reg/mem
```

The DEC (decrement) subtracts 1 from a single operand.

```
DEC reg/mem
```

Examples:

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

The INC and DEC instructions affect Overflow, Sign, Zero, Auxiliary Carry, and Parity flags. Do not affect the Carry flag.

❖ ADD Instruction

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

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.

Examples:

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

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

❖ SUB Instruction

The SUB instruction subtracts a source operand from a destination operand.

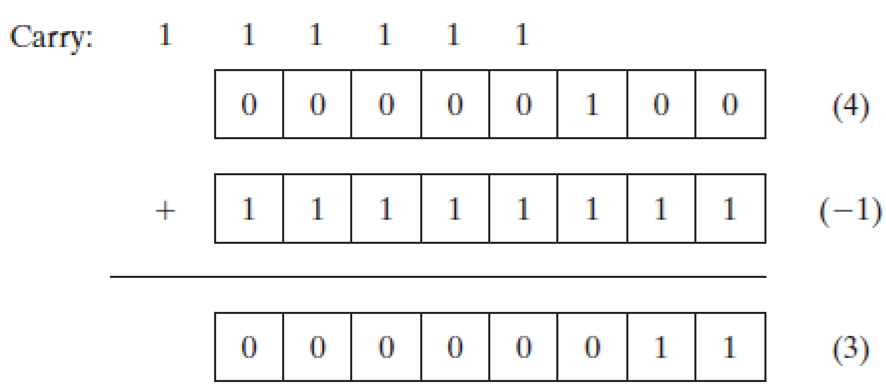
SUB *dest, source*

Examples:

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

Internally, the CPU can implement subtraction as a combination of negation and addition. Two's-complement notation is used for negative numbers.

4 - 1 can be rewritten as 4 + (-1).
-1 is represented by 11111111.



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

❖ NEG Instruction

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

```
NEG reg
NEG mem
```

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

❖ Flags Affected by Addition and Subtraction

- The Carry flag indicates unsigned integer overflow. For example, if an instruction has an 8-bit destination operand but the instruction generates a result larger than 11111111 binary, the Carry flag is set.
  - In ADD CF = (carry out of the MSB)
  - In SUB CF = INVERT (carry out of the MSB)
- The Overflow flag indicates signed integer overflow. For example, if an instruction has an 8-bit destination operand but it generates a negative result smaller than 10000000 binary, the Overflow flag is set. The Overflow flag is only set when:
  - Two positive operands are added and their sum is negative.
  - Two negative operands are added and their sum is positive.
  - OF = (carry out of the MSB) XOR (carry into the MSB).
- The Zero flag indicates that an operation produced zero. For example, if an operand is subtracted from another of equal value, the Zero flag is set.
- The Sign flag indicates that an operation produced a negative result. If the most significant bit (MSB) of the destination operand is set, the Sign flag is set.
- The Parity flag indicates whether or not an even number of 1 bits occurs in the least significant byte of the destination operand, immediately after an arithmetic or boolean instruction has executed.
- The Auxiliary Carry flag is set when a 1 bit carries out of position 3 in the least significant byte of the destination operand.

❖ Section 4.2 Review

Use the following data for the next several questions:

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

1. Write an instruction that increments **val2**.
   inc val2

2. Write an instruction that subtracts **val3** from **EAX**.
   sub eax,val3

3. Write instructions that subtract **val4** from **val2**.
   mov ax,val4
   sub val2,ax

4. If **val2** is incremented by 1 using the ADD instruction, what will be the values of the Carry and Sign flags?
   CF = 0, SF = 1

5. If **val4** is incremented by 1 using the ADD instruction, what will be the values of the Overflow and Sign flags?
   OF = 1, SF = 1

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 = 1 SF = 0 ZF = 1 OF = 0
add ah,1 ;      b. CF = 0 SF = 1 ZF = 0 OF = 1
add ax,2 ;      c. CF = 0 SF = 0 ZF = 0 OF = 0
```

7. Implement the following expression in assembly language: AX = (- val2 + BX) - val4.
   mov ax,val2
   neg ax
   add ax,bx
   sub ax,val4

8. *(Yes/**No**):* Is it possible to set the Overflow flag if you add a positive integer to a negative integer?

9. *(**Yes**/No):* Will the Overflow flag be set if you add a negative integer to a negative integer and produce a positive result?

10. *(**Yes**/No):* Is it possible for the NEG instruction to set the Overflow flag?

11. *(Yes/**No**):* Is it possible for both the Sign and Zero flags to be set at the same time?

12. Write a sequence of two instructions that set both the Carry and Overflow flags at the same time.
    mov al,80h
    add al,80h

4.3 Data-Related Operators and Directives

Operators and directives are not executable instructions; instead, they are interpreted by the assembler.

• 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.

❖ Section 4.3 Review

1. *(True/**False**):* The OFFSET operator always returns a 16-bit value.
2. *(True/**False**):* The PTR operator returns the 32-bit address of a variable.
3. *(**True**/False):* The TYPE operator returns a value of 4 for doubleword operands.
4. *(True/**False**):* The LENGTHOF operator returns the number of bytes in an operand.
5. *(**True**/False):* The SIZEOF operator returns the number of bytes in an operand.

Use the following data definitions for the next seven exercises:

```
.data
myBytes BYTE 10h,20h,30h,40h
myWords WORD 3 DUP(?),2000h
myString BYTE "ABCDE"
```

7. What will be the value of EAX after each of the following instructions execute?

```
mov eax,TYPE myBytes ;         a. 1
mov eax,LENGTHOF myBytes ;     b. 4
mov eax,SIZEOF myBytes ;       c. 4
mov eax,TYPE myWords ;         d. 2
mov eax,LENGTHOF myWords ;     e. 4
mov eax,SIZEOF myWords ;       f. 8
mov eax,SIZEOF myString ;      g. 5
```

8. Write a single instruction that moves the first two bytes in **myBytes** to the DX register. The resulting value will be 2010h.
mov dx, WORD PTR myBytes

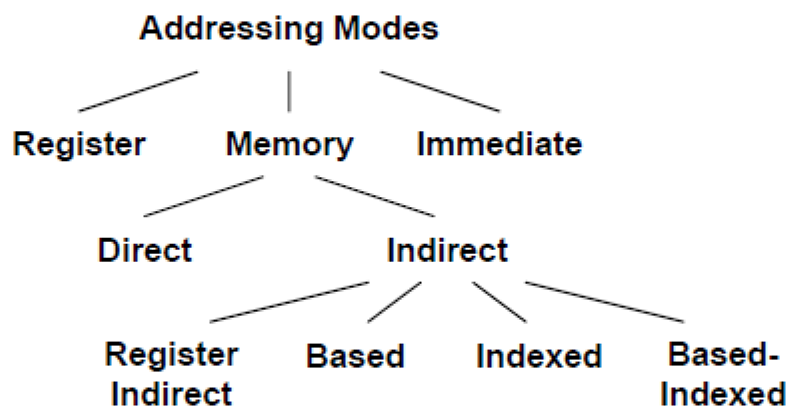
9. Write an instruction that moves the second byte in **myWords** to the AL register.
mov al, BYTE PTR myWords+1

10. Write an instruction that moves all four bytes in **myBytes** to the EAX register.
mov eax,DWORD PTR myBytes

4.4 Indirect Addressing

❖ Addressing Modes

The assembly language instructions require the specification of the location of data for source and destination operands. The specification of the location of data is called the **data addressing mode**. It can be classified as shown in the following diagram:



Register addressing is when a register is used to specify the source or destination of an operand. This is the most efficient addressing mode because registers are implemented inside the processor and their access is very fast.

Immediate addressing is when an immediate value (a constant) is used for a source operand. It cannot be used to specify a destination operand. The immediate constant is part of the instruction itself.

Memory addressing is used to specify the address of the source and destination operands located in memory. It can be divided into **direct** and **indirect** memory addressing.

Direct memory addressing is when the address of a memory operand is specified directly by name.

For example:

```
mov sum, eax          ; sum is a variable in memory
```

Direct memory addressing is useful for accessing simple variables in memory, but it is useless for addressing arrays or data structures. To address the elements of an array, we need to use a register as a pointer to the array elements. This is called **indirect memory addressing**.

❖ Register Indirect

Register Indirect can be any 32-bit general-purpose register (EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP) surrounded by brackets. In real-address mode, a 16-bit register holds the offset of a variable. If the register is used as an indirect operand, it may only be SI, DI, BX, or BP. Avoid BP unless you are using it to index into the stack. The register is assumed to contain the address of some data.

Example:

```
.data
byteVal BYTE 10h
.code
mov esi,OFFSET byteVal
mov al,[esi]              ; AL = 10h
```

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]
```

Arrays:

```
.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
```

Example: Adding 32-Bit Integers

The following code example adds three doublewords. A displacement of 4 must be added to ESI as it points to each subsequent array value because doublewords are 4 bytes long:

```
.data
arrayD DWORD 10000h,20000h,30000h
.code
mov esi,OFFSET arrayD
mov eax,[esi] ; first number
add esi,4
add eax,[esi] ; second number
add esi,4
add eax,[esi] ; third number
```

❖ Indexed Addressing

Indexed Addressing adds a constant to a register to generate an effective address.

```
constant[indexReg]
[constant + indexReg]
```

Example:

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

❖ Index Scaling

The scale factor is the size of the array component (word = 2, doubleword = 4, quadword = 8).

```
constant[indexReg * scale]
[constant + indexReg * scale]
```

Example:

```
.data
arrayD DWORD 1,2,3,4
.code
mov esi,3
mov eax,arrayD[esi*4] ; EAX = 4
```

The TYPE operator can make the indexing more flexible:

```
mov esi,3
mov eax,arrayD[esi*TYPE arrayD] ; EAX = 4
```

❖ Based Addressing

The based addressing combines a register with a constant offset. The base register holds the base address of an array or structure, and the constant identifies offsets of various array elements.

[BaseReg + Offset]

Example:

```
.data
arrayW WORD 1000h,2000h,3000h
.code
mov esi,OFFSET arrayW
mov ax,[esi] ; AX = 1000h
mov ax,[esi+2] ; AX = 2000h
mov ax,[esi+4] ; AX = 3000h
```

❖ Based-Indexed Addressing

*[BaseReg + (IndexReg * Scale) + Offset]*

Useful in accessing two-dimensional arrays:

- Offset: array address → we can refer to the array by name
- Base register: holds row address → relative to start of array
- Index register: selects an element of the row → column index
- Scaling factor: when array element size is 2, 4, or 8 bytes

Example:

```
.data
     matrix  DWORD  0, 1, 2, 3, 4  ; 4 rows, 5 cols
             DWORD 10,11,12,13,14
             DWORD 20,21,22,23,24
             DWORD 30,31,32,33,34
     ROWSIZE EQU   SIZEOF matrix   ; 20 bytes per row
.code
     mov ebx, 2*ROWSIZE            ; row index = 2
     mov esi, 3                    ; col index = 3
     mov eax, matrix[ebx+esi*4]    ; EAX = matrix[2][3]

     mov ebx, 3*ROWSIZE            ; row index = 3
     mov esi, 1                    ; col index = 1
     mov eax, matrix[ebx+esi*4]    ; EAX = matrix[3][1]
```

❖ LEA Instruction

LEA = Load Effective Address

LEA r32, mem

Example:

```
.data
     array WORD 1000 DUP(?)
.code                        ; Equivalent to . . .
lea eax, array               ; mov eax, OFFSET array
lea eax, array[esi]          ; mov eax, esi
                             ; add eax, OFFSET array

lea eax, array[esi*2]        ; mov eax, esi
                             ; add eax, eax
                             ; add eax, OFFSET array

lea eax, [ebx+esi*2]         ; mov eax, esi
                             ; add eax, eax
                             ; add eax, ebx
```

❖ Section 4.4 Review

1. *(True/**False**):* Any 16-bit general-purpose register can be used as an indirect operand.
2. *(**True**/False):* Any 32-bit general-purpose register can be used as an indirect operand.
3. *(True/**False**):* The BX register is usually reserved for addressing the stack.
5. *(**True**/False):* The following instruction is invalid: inc [esi]
6. *(**True**/False):* The following is an indexed operand: array[esi]

Use the following data definitions for the remaining questions in this section:

```
myBytes BYTE 10h,20h,30h,40h
myWords WORD 8Ah,3Bh,72h,44h,66h
myDoubles DWORD 1,2,3,4,5
myPointer DWORD myDoubles
```

7. Fill in the requested register values on the right side of the following instruction sequence:

```
mov esi,OFFSET myBytes
mov al,[esi]                ; a. AL = 10h
mov al,[esi+3]              ; b. AL = 40h
mov esi,OFFSET myWords + 2
mov ax,[esi]                ; c. AX = 003Bh
mov edi,8
mov edx,[myDoubles + edi]   ; d. EDX = 3
mov edx,myDoubles[edi]      ; e. EDX = 3
mov ebx,myPointer
mov eax,[ebx+4]             ; f. EAX = 2
```

8. Fill in the requested register values on the right side of the following instruction sequence:

```
mov esi,OFFSET myBytes
mov ax,[esi]                ; a. AX = 2010h
mov eax,DWORD PTR myWords   ; b. EAX = 003B008Ah
mov esi,myPointer
```

```
mov ax,[esi+2]                ; c. AX = 0000
mov ax,[esi+6]                ; d. AX = 0000
mov ax,[esi-4]                ; e. AX = 0044h
```

4.5 JMP and LOOP Instructions

There are two basic types of transfers:

• **Unconditional Transfer:** Control is transferred to a new location in all cases; a new address is loaded into the instruction pointer, causing execution to continue at the new address. The JMP instruction does this.

• **Conditional Transfer:** The program branches if a certain condition is true. A wide variety of conditional transfer instructions can be combined to create conditional logic structures. The CPU interprets true/false conditions based on the contents of the ECX and Flags registers.

❖ JMP Instruction

The JMP instruction causes an unconditional transfer to a destination, identified by a code label.

```
JMP destination
```

The JMP instruction provides an easy way to create a loop by jumping to a label at the top of the loop:

```
top:
.
.
jmp top ; repeat the endless loop
```

JMP is unconditional, so a loop like this will continue endlessly unless another way is found to exit the loop.

❖ LOOP Instruction

The LOOP instruction, formally known as Loop According to ECX Counter, repeats a block of statements a specific number of times. ECX is automatically used as a counter and is decremented each time the loop repeats.

```
LOOP destination
```

The loop destination must be within -128 to +127 bytes of the current location counter.
The execution of the LOOP instruction involves two steps:

ECX ← ECX – 1
If ECX != 0, jump to *destination* label

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
```

When initialize ECX to zero before beginning a loop, the LOOP instruction decrements ECX to FFFFFFFFh, and the loop repeats 4,294,967,296 times!

❖ Nested Loops

If you need to code a loop within a loop, you must save the outer loop counter's ECX value.

```
.DATA
     count DWORD ?
.CODE
     mov ecx, 100           ; set outer loop count to 100
L1:
     mov count, ecx         ; save outer loop count
     mov ecx, 20            ; set inner loop count to 20

L2:  .
     .
     loop L2                ; repeat the inner loop
     mov ecx, count         ; restore outer loop count
     loop L1                ; repeat the outer loop
```

❖ Summing an Integer Array

```
TITLE Summing an Array (SumArray.asm)
INCLUDE Irvine32.inc
.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
exit
main ENDP
END main
```

❖ Copying a String

```
TITLE Copying a String (CopyStr.asm)
INCLUDE Irvine32.inc
.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 char from source
     mov  target[esi],al       ; store it in the target
     inc  esi                  ; increment index
     loop L1                   ; loop for entire string
     exit
main ENDP
END main
```

❖ Section 4.5 Review

1. *(**True**/False):* A JMP instruction can only jump to a label inside the current procedure.

2. *(True/**False**):* JMP is a conditional transfer instruction.

3. If ECX is initialized to zero before beginning a loop, how many times will the LOOP instruction repeat? (Assume ECX is not modified by any other instructions inside the loop.)
   4,294,967,296 times

4. *(True/**False**):* The LOOP instruction first checks to see whether ECX is not equal to zero; then LOOP decrements ECX and jumps to the destination label.

5. *(**True**/False):* The LOOP instruction does the following: It decrements ECX; then, if ECX is not equal to zero, LOOP jumps to the destination label.

6. In real-address mode, which register is used as the counter by the LOOP instruction?
   CX

8. *(True/**False**):* The target of a LOOP instruction must be within 256 bytes of the current location.

9. *(Challenge):* What will be the final value of EAX in this example?

```
mov eax,0
mov ecx,10 ; outer loop counter
L1:
mov eax,3
mov ecx,5 ; inner loop counter
L2:
add eax,5
loop L2 ; repeat inner loop
loop L1 ; repeat outer loop
```

The program does not stop, because the first LOOP instruction decrements ECX to zero. The second LOOP instruction decrements ECX to FFFFFFFFh, causing the outer loop to repeat.

10. Revise the code from the preceding question so the outer loop counter is not erased when the inner loop starts.

```
.DATA
     count DWORD ?
.CODE
mov eax,0
mov ecx,10 ; outer loop counter
L1:
mov count, ecx
mov eax,3
mov ecx,5 ; inner loop counter
L2:
add eax,5
loop L2 ; repeat inner loop
mov ecx, count
loop L1 ; repeat outer loop
```

☺ *Best Wishes* ☺