EE-2003
Computer Organization & Assembly Language

INSTRUCTOR

Engr. Aashir Mahboob
Lecturer, Department of Computer Science
FAST NUCES (Karachi)
aashir.mahboob@nu.edu.pk

Chapter No: 04

Data Transfers, Addressing, and Arithmetic

OUTLINE

- DATA TRANSFER INSTRUCTIONS
- ADDITION AND SUBTRACTION
- DATA RELATED OPERATORS AND DIRECTIVES
- INDIRECT ADDRESSING
- JMP AND LOOP INSTRUCTIONS

DATA TRANSFER INSTRUCTION

Operand Types

- •Instructions in assembly language can have zero, one, two, or three operands.
- •mnemonic
- •mnemonic [destination]
- •mnemonic [destination],[source]
- •mnemonic [destination],[source1],[source2]

DATA TRANSFER INSTRUCTION

- The three types of operands are:

1. **Immediate:** a numeric literal expression /a constant integer (8, 16, or 32 bits), value is encoded within the instruction.

2. **Register:** the name of a register, register name is converted to a number and encoded within the instruction.

3. **Memory:** references a location in memory, memory address is encoded within the instruction, or a register holds the address of a memory location.

```
mov al var1
 A0 00010400
```

DATA TRANSFER INSTRUCTION

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

DATA TRANSFER INSTRUCTION

- **Direct Memory Operands**
  - Variable names are references to offsets within the data segment.
  - A direct memory operand is a named reference to storage in memory.
  - The named reference (label) is automatically dereferenced by the assembler.

```
.data
var1 BYTE 10h
.code
mov al,var1                         ; AL = 10h
mov al,[var1]                       ; AL = 10h
```

alternate format

MOV INSTRUCTION

- The MOV instruction copies data from a source operand to a destination operand. Known as a data transfer instruction.

MOV destination,source

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

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

MOV INSTRUCTION

```
.data
    count BYTE 100
    wVal  WORD 2
```

```
.code
    mov bl,count
    mov ax,wVal
    mov count,al
```

ABOVE INSTRUCTIONS ARE CORRECT??

```
mov al,wVal
mov ax,count
mov eax,count
```

ABOVE INSTRUCTIONS ARE CORRECT??

Mistakes??

MOVZX INSTRUCTION

- **Zero Extension**

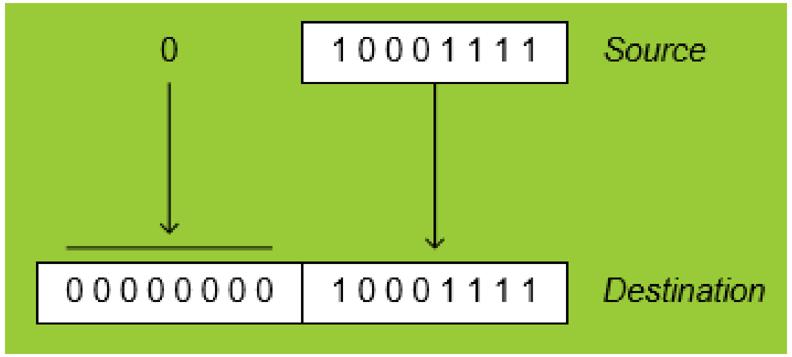
•MOV instruction cannot directly copy data from a smaller operand to a larger one.

mov bl,10001111b

mov ax,bl ; error

•**MOVZX (move with zero-extend) instruction** fills (extends) the upper half of the destination with zeros.

```
mov bl,10001111b
movzx ax,bl       ; zero-extension
```



EXERCISE

```
.data
        byte1  BYTE  9Bh
        word1  WORD  0A69Bh
```

```
.code
        movzx eax,word1
        movzx edx,byte1
        movzx cx,byte1
```

Write down values of registers

```
EAX  =  0000A69Bh
EDX  =  0000009Bh
CX  =  009Bh
```

EXERCISE

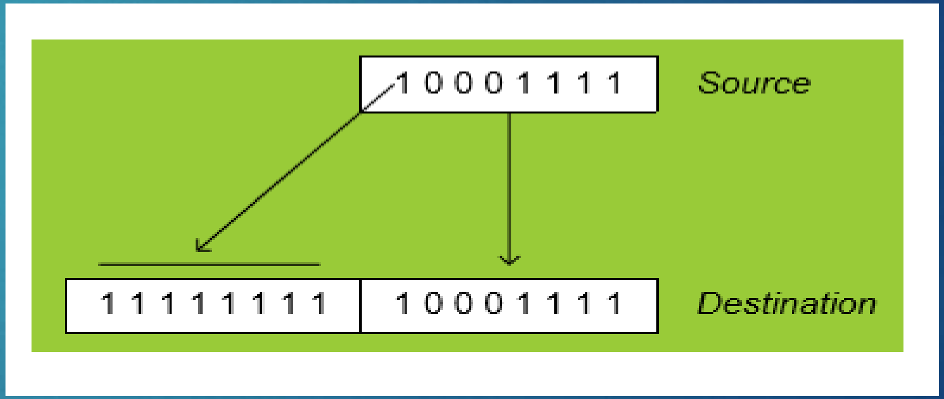
Write down values of registers

```
.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
```

MOVSX INSTRUCTION

- The **MOVSX instruction (move with sign-extend)** copies the contents of a source operand into a destination operand and fills the upper half of the destination with a copy of the source operand's sign bit.

```
mov bl,10001111b
movsx ax,bl      ; sign extension
```

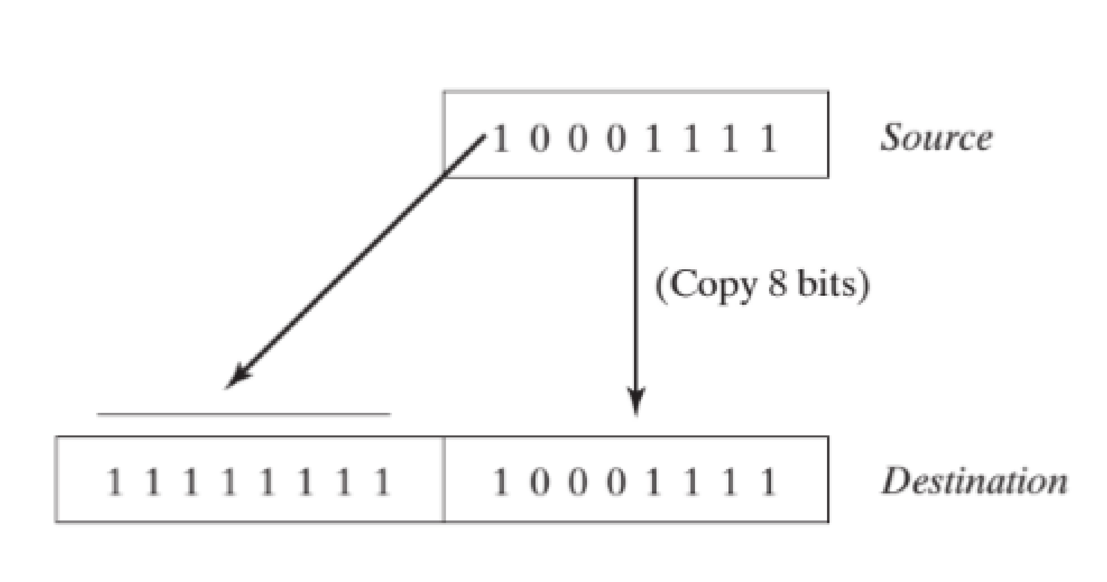


EXERCISE

```
.data
        byteVal BYTE 10001111b
```

AX = 1111111110001111b

```
.code
        movsx  ax,byteVal
```



Write down values ax.

XCHG INSTRUCTION

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

- There are three variants:

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

- You can exchange data between registers or between registers and memory, **but not from memory to memory**:

```
xchg    ax, bx          ; Put AX in BX and BX in AX
xchg    memory, ax      ; Put "memory" in AX and AX in "memory"
xchg    mem1, mem2      ; Illegal, can't exchange memory locations!
```

XCHG INSTRUCTION

- 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.

- In array sorting applications, XCHG provides a simple way to exchange two array elements.

```
xchg  ax,  bx   ; exchange 16-bit regs
xchg  ah,  al   ; exchange 8-bit regs
xchg  eax, ebx  ; exchange 32-bit regs
xchg  [response], cl   ; exchange 8-bit mem op with CL
xchg  [total],    edx  ; exchange 32-bit mem op with EDX
```

EXERCISE

```
.DATA
    val1  WORD 1000h
    val2  WORD 2000h
```

Write down contents of ax
And memory locations after
Execution of each instruction.

```
.CODE
    mov   ax,   [val1]
    xchg  ax,   [val2]
    mov   [val1], ax
```

Direct-Offset Operands

- lets you access memory locations that may not have explicit labels.

- A constant is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location.

```
.data
   arrayB BYTE 10h,20h,30h,40h
```

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

Direct-Offset Operands

- **WORD Arrays**; In an array of 16-bit words, the offset of each array element is 2 bytes beyond the previous one.
- That's why 2 is added as offset in Array (for each next element of an array)

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

- **DWORD Arrays**; In an array of 32-bit words, the offset of each array element is 4 bytes beyond the previous one.
- That's why 4 is added as offset in Array (for each next element of an array)

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

EXAMPLE PROGRAM (MOVES)

```
.data
val1 WORD 1000h
val2 WORD 2000h
arrayB BYTE 10h,20h,30h,40h,50h
arrayW WORD 100h,200h,300h
arrayD DWORD 10000h,20000h
```

```
.code
main PROC

    mov bx,0A69Bh
    movzx eax,bx
    movzx edx,bl
    movzx cx,bl

    mov bx,0A69Bh
    movsx eax,bx
    movsx edx,bl
    movsx bl,7Bh
    movsx cx,bl

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

```
mov al,arrayB
mov al,[arrayB+1]
mov al,[arrayB+2]

mov ax,arrayW
mov ax,[arrayW+2]

mov eax,arrayD
mov eax,[arrayD+4]
mov eax,[arrayD+4]
```

ADDITION AND SUBTRACTION

- **INC and DEC Instructions**

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

```
.data
    myWord   WORD 1000h
    myDword DWORD 10000000h
```

```
.code
    inc myWord        ; 1001h
    dec myWord        ; 1000h
    inc myDword       ; 10000001h

    mov ax,00FFh
    inc ax            ; AX = 0100h
```

ADDITION AND SUBTRACTION

ADD Instruction

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

```
ADD dest,source
```

SUB Instruction

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

```
SUB dest,source
```

- The set of possible operands is the same as for the MOV instruction

ADDITION AND SUBTRACTION

```
.data
    var1 DWORD 10000h
    var2 DWORD 20000h
```

```
.code
    mov eax,var1
    add eax,var2
    add ax,0FFFFh
    add eax,1
    sub ax,1
```

```
; ---EAX---
; 00010000h
; 00030000h
; 0003FFFFh
; 00040000h
; 0004FFFFh
```

NEG Instruction

- The neg (negate) instruction takes the two's complement of a byte or word.

- It takes a single (destination) operation and negates it. The syntax for this instruction is:

```
NEG reg
NEG mem
```

- Neg always updates the A, S, P, and Z flags as though you were using the sub instruction.

Implementing Arithmetic Expressions

- HLL compilers translate mathematical expressions into assembly language. You can do it also. For example:

```
Rval = -Xval + (Yval - Zval)
```

```
.data
 Rval SDWORD ?
 Xval SDWORD 26
 Yval SDWORD 30
 Zval SDWORD 40
```

```
.code
 mov eax, Xval
 neg eax

 mov ebx,Yval
 sub ebx,Zval

 add eax,ebx
 mov Rval,eax
```

```
;EAX=-26

;EBX = -10

; -36
```

Flags Affected by Addition and Subtraction

- The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations.
  - based on the contents of the destination operand.

- We use the values of CPU status flags to check the outcome of arithmetic operations and to activate conditional branching instructions.
- **Essential flags:**
- Zero flag – set when destination equals zero
- Sign flag – set when destination is negative; if the MSB of the destination operand is set,
- Carry flag – set when unsigned value is out of range.
- Overflow flag – set when signed value is out of range .

Flags Affected by Addition and Subtraction

```
mov cx,1
sub cx,1            ; CX = 0, ZF = 1

mov ax,0FFFFh
inc ax              ; AX = 0, ZF = 1
inc ax              ; AX = 1, ZF = 0

mov cx,0
sub cx,1            ; CX = -1, SF = 1
add cx,2            ; CX = 1, SF = 0
```

ADD & SUB Instructions

- Example:

```
mov al,0FFh
add al,1                        ; AL = 00, CF = 1
```

		1	1	1	1	1	1	1	
		1	1	1	1	1	1	1	1
	+	0	0	0	0	0	0	0	1
CF	1	0	0	0	0	0	0	0	0

ADD & SUB Instructions

- Example:

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

		0	0	0	0	0	0	0	1	(1)
+		1	1	1	1	1	1	1	0	(−2)
CF	1	1	1	1	1	1	1	1	1	(FFh)

ADD & SUB Instructions

Sign and Overflow Flags:

- The Sign flag is set when the result of a signed arithmetic operation is negative.

```
mov eax,4
sub eax,5          ; EAX = -1, SF = 1
```

- The Overflow flag is set when the result of a signed arithmetic operation overflows or underflows the destination operand.

```
mov al,127
add al,1           ; OF = 1
```

```
mov al,-128
sub al,1           ; OF = 1
```

LAHF/SAHF (load/store status flag from/to AH)

- **LAHF instruction loads lower byte of the EFLAGS register into AH register.**
- **The lowest 8 bits of the flags are transferred:**
  - Sign
  - Zero
  - Auxiliary Carry
  - Parity
  - Carry

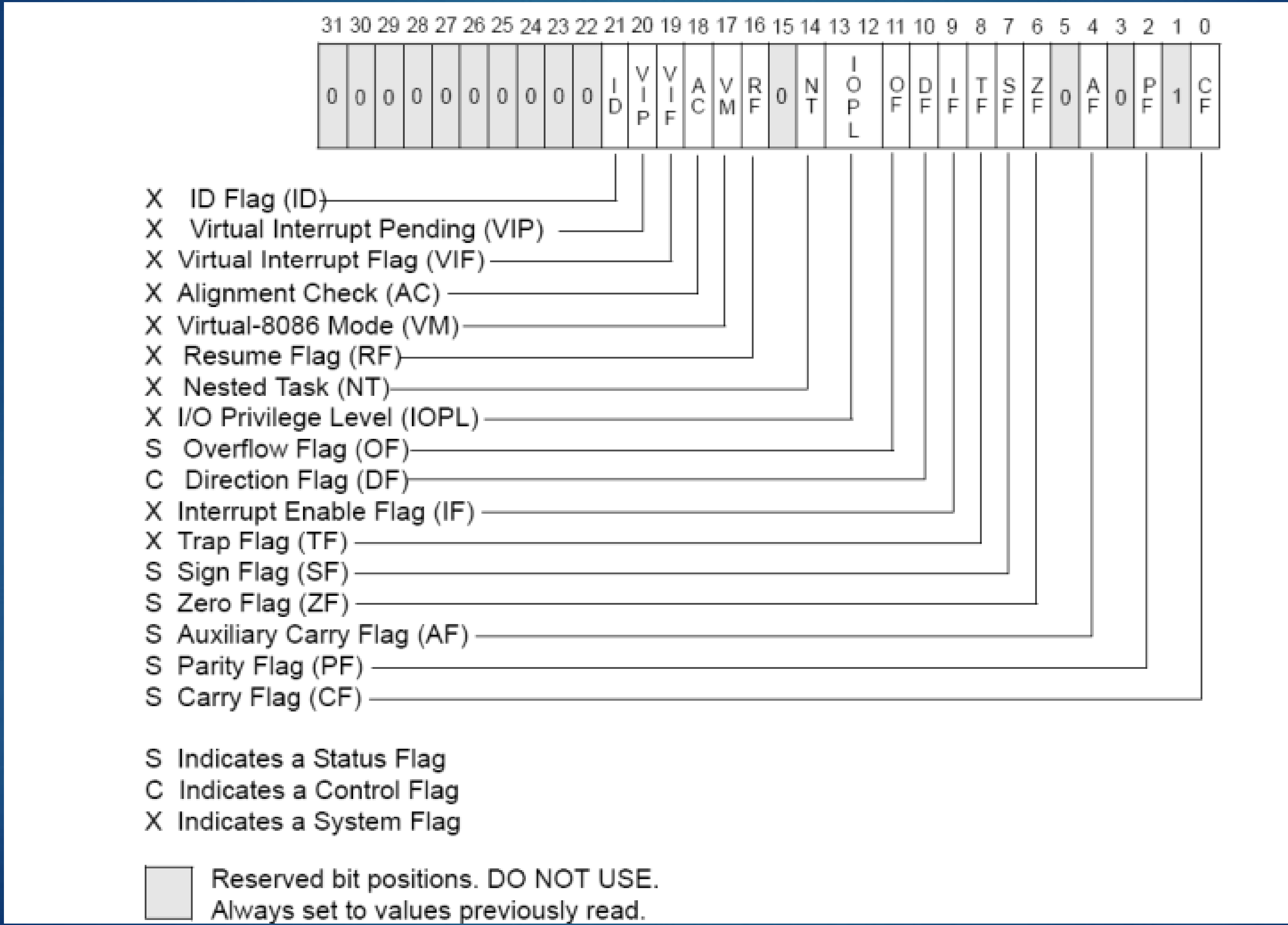
```
.data
saveflags BYTE ?
.code
lahf                          ; load flags into AH
mov  saveflags,ah             ; save them in a variable
```

LAHF/SAHF (load/store status flag from/to AH)

- SAHF restores the value of lower byte flags.
- This instruction copies, AH into low byte of EFLAGS Register.

```
mov  ah,saveflags          ; load saved flags into AH
sahf                       ; copy into Flags register
```

EFLAGS


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE.
Always set to values previously read.

ALIGN DIRECTIVE

- The ALIGN directive aligns a variable on a byte, word, doubleword, or paragraph boundary.
- The syntax is :

ALIGN bound

- Bound can be 1, 2, 4, 8, or 16. A value of 1 aligns the next variable on a 1- byte boundary (the default).
- If bound is 2, the next variable is aligned on an even-numbered address. If bound is 4, the next address is a multiple of 4. If bound is 16, the next address is a multiple of 16, a paragraph boundary.

ALIGN DIRECTIVE

- The assembler can insert one or more empty bytes before the variable to fix the alignment.
- Why bother aligning data?
- Because the CPU can process data stored at even-numbered addresses more quickly than those at odd-numbered addresses.

ALIGN DIRECTIVE

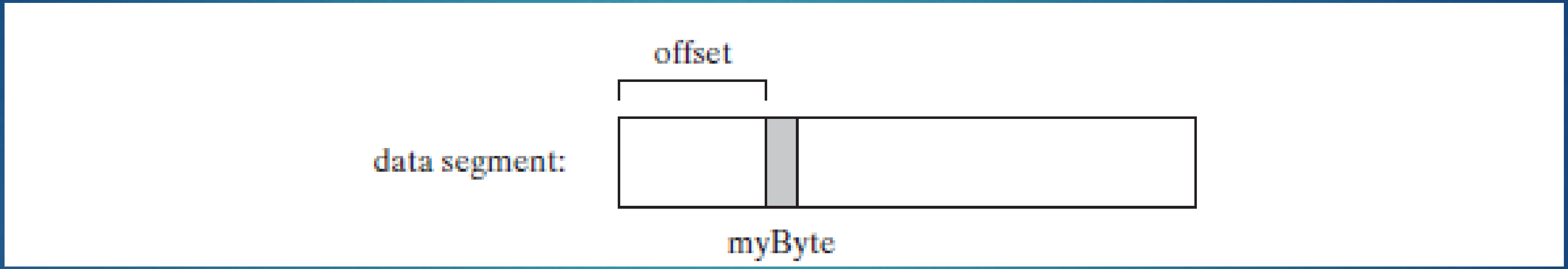
- In the following example, bVal is arbitrarily located at offset 00404000. Inserting the ALIGN 2 directive before wVal causes it to be assigned an even-numbered offset:

```
bVal   BYTE   ?                    ; 00404000h
ALIGN 2
wVal   WORD   ?                    ; 00404002h
bVal2  BYTE   ?                    ; 00404004h
ALIGN 4
dVal   DWORD ?                     ; 00404008h
dVal2  DWORD ?                     ; 0040400Ch
```

- Note that dVal would have been at offset 00404005, but the ALIGN 4 directive bumped it up to offset 00404008.

OFFSET Operator

- The OFFSET operator return the offset of a data label
- The offset represents the distance, in bytes, of the label from beginning of the data segment
- Figure shows a variable named myByte inside the data segment



OFFSET Operator

- Example:
- Declaration of different types:

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

- If bVal were located at offset 00404000 (hexa-decimal), the OFFSET operator would return the following values:

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

PTR Operator

- You can use the PTR operator to override the declared size of an operand.

```
.data
        myDouble DWORD 12345678h
.code
        mov ax,myDouble                   ;error - why?
        mov ax,WORD PTR myDouble          ; loads 5678h
```

- Why wasn't 1234h moved into AX?
- x86 processors use the little endian storage format in which the low-order byte is stored at the variable's starting address.

TYPE Operator

- The TYPE operator returns the size, in bytes, of a single element of a data declaration.

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

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

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.

```
.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
```

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

- If you declare an array that spans multiple program lines, LENGTHOF only regards the data from the first line as part of the array (here LENGTHOF myArray returns 5).

```
myArray BYTE 10,20,30,40,50
        BYTE 60,70,80,90,100
```

SIZEOF Operator

- The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE.

```
.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
```

SIZEOF

```
; 3
; 64
; 30
; 16
; 9
```

```
.code
    mov ecx, SIZEOF array1
```

```
; 64
```

LABEL Directive

- **The LABEL directive assigns an alternate label name and type to an existing storage location. LABEL does not allocate any storage of its own.**

  - A common use of LABEL is to provide an alternative name and size attribute for the variable declared next in the data segment.

```
.data
      val16 LABEL WORD
      val32 DWORD 12345678h
.code
      mov ax,val16                ; AX = 5678h
      mov dx,[val16+2]            ; DX = 1234h
```

```
.data
      LongValue LABEL DWORD
      val1 WORD 5678h
      val2 WORD 1234h
.code
      mov eax,LongValue ; EAX = 12345678h
```

INDIRECT ADDRESSING

- Direct addressing is rarely used for array processing because it is impractical to use constant offsets to address more than a few array elements.
- An indirect operand holds the address of a variable, usually an array or string. It can be dereferenced (just like a pointer).

```
.data
    val1 BYTE 10h,20h,30h
.code
    mov esi,OFFSET val1
    mov al,[esi]                ; dereference ESI (AL = 10h)

    inc esi
    mov al,[esi]                ; AL = 20h

    inc esi
    mov al,[esi]                ; AL = 30h
```

INDIRECT ADDRESSING

- The size of an operand may not be evident from the context of an instruction.

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

- Because 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

- Indirect operands are ideal tools for stepping through 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
```

Indexed Operands

- **An indexed operand adds a constant to a register to generate an effective address. There are two notational forms:**
  - [label + reg]		label[reg]

```
.data
    arrayW WORD 1000h,2000h,3000h

.code
    mov esi,0
    mov ax,[arrayW + esi]         ; AX = 1000h

    mov ax, arrayW[esi]           ; alternate format

    add esi,2
    add ax,[arrayW + esi]
```

Scale Factors in Indexed Operands

- Indexed operands must take into account the size of each array element when calculating offsets.

- Using an array of doublewords, as in the following example, we multiply the subscript (3) by 4 (the size of a doubleword) to generate the offset of the array element containing 400h:

```
.data
arrayD DWORD 100h, 200h, 300h, 400h
.code
mov esi,3 * TYPE arrayD            ; offset of arrayD[3]
mov eax,arrayD[esi]                ; EAX = 400h
```

Scale Factors in Indexed Operands

- The x86 instruction set provides a way for offsets to be calculated, using a scale factor .
- The scale factor is the size of the array component (WORD=2, DWORD=4 or QWORD=8).

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

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

EXERCISE

- **What will be the value of EAX after each of the following instructions execute?**

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

```
mov   eax,TYPE myBytes                    ; a.
mov   eax,LENGTHOF myBytes                ; b.
mov   eax,SIZEOF myBytes                  ; c.
mov   eax,TYPE myWords                    ; d.
mov   eax,LENGTHOF myWords                ; e.
mov   eax,SIZEOF myWords                  ; f.
mov   eax,SIZEOF myString                 ; g.
```

EXERCISE

Write down values of destination registers

```
.data
arrayB  BYTE    20, 40, 60, 80
arrayW  WORD    100, 150, 250, 300
.code
mov si, 1
mov al, arrayB[si]
mov al, [arrayB + 3]
mov si, 2
mov cx, arrayW[si]
mov cx, [arrayW + 4]
```

```
;       SI = 0001
;       AL = 40
;       AL = 80
;       SI = 2
;       CX = 150
;       CX = 250
```