Lecture : 4

WORKING WITH DATA RELATED OPERATORS, DIRECTIVES & ADDRESSING

Objectives:

- Observing Effect of Arithmetic Instructions on Flags
- Register Direct-Offset Operands
- `OFFSET` Operator
- `PTR` Operator
- `TYPE` Operator
- `LENGTH` Operator
- `SIZEOF` Operator
- `Indirect` Operands
- `Indexed` Operands

Effect of Arithmetic Instructions on Flag Registers

Status flags are updated to indicate certain properties of the result Once a flag is set, it remains in that state until another instruction that affects the flags is executed.

ZF-Zero Flag

The Zero flag is set when the result of an operation produces `zero` in the destination operand.

```
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
```

Remember…

- A flag is set when it equals to 1.
- A flag is clear when it equals to 0.

CF-Carry Flag

This flag is set, when there is a carry out of `MSB` in case of addition and borrow in case of subtraction.

The Carry flag is set when the result of an operation generates an unsigned value that is out of range (too big or too small for the destination operand).

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

; Try to go below zero;

mov al, 0
sub al, 1             ; CF = 1, AL = FF
```

SF-Sign Flag

This flag indicates the sign of the result of an operation. A 0 for positive number and 1 for a negative number.

```
mov AL, 15
add AL, 97

; clears the sign flag as the result
is 112 (or 0111000 in binary)
```

```
mov AL, 15
add AL, 97

; sets the sign flag as the result is
-82 (or 1010110 in binary)
```

AC-Auxiliary Flag

This flag is set, if there is a carry from the lowest nibble, i.e., bit three during addition, or borrow for the lowest nibble, i.e., bit three, during subtraction.

Suppose we add 1 to 0Fh. The sum (10h) contains a 1 in bit position 4 that was carried out of bit position 3.

```
mov al, 0Fh
add al, 1             ; AC = 1
```

PF-Parity Flag

The Parity flag (PF) is set when the least significant byte of the destination has an even number of 1 bit. The following ADD and SUB instructions alter the parity of AL.

```
mov al, 10001100b
add al, 00000010b               ; AL = 10001110, PF = 1
sub al, 10000000b               ; AL = 00001110, PF = 0
```

OF-Over Flow Flag

The Overflow flag is set when the result of a signed arithmetic operation over-flows or underflows the destination operand. For example, the largest possible integer signed value is +127; adding 1 to it causes overflow:

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

Similarly, the smallest possible negative integer byte value is 128. Subtracting 1 from it causes underflow. The destination operand value does not hold a valid arithmetic result, and the Over-flow flag is set.

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

Direct-Offset Operands

You can add a displacement to the name of a variable, creating a direct-offset operand.

Example

```
.data
     arrayB BYTE 10h, 20h, 30h, 40h
     arrayW WORD 100h, 200h, 300h

.code
     mov AL, arrayB
     mov AL, [arrayB+1]
     mov AX, arrayW
     mov AX, [arrayW+2]

;    similarly, the second element in a doubleword array is 4 bytes beyond
     the first one.
```

Data-Related Operators and Directives

OFFSET Operator

The OFFSET operator returns the offset of a data label.

Syntax

```
mov reg32, OFFSET mem      ; reg32 points to count
```

Example

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

; if bVal is located at offset 00404000h, we would get:

.code
     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

We can use the PTR operator to override the declared size of an operand. Note, PTR must be used in combination with one of the standard assembler data types.

For example, that we 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     ; AX = 5678h
```

and higher WORD (1234h) to AX.

```
     mov DX, WORD PTR myDouble + 2  ; DX = 1234h
```

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
```

TYPE Operator

The TYPE operator returns the size, in bytes, of a single element of a variable.

Syntax

```
mov reg16, TYPE mem
```

Example 01

```
.data
     var1 BYTE ?                    ; TYPE var1 = 1
     var2 WORD ?                    ; TYPE var2 = 2
     var3 DWORD ?                   ; TYPE var3 = 4
     var4 QWORD ?                   ; TYPE var4 = 8
```

Example 02

```
.data
     var1 BYTE 20h
     var2 WORD 1000h
     var3 DWORD ?
     var4 BYTE 10, 20, 30, 40, 50
     msg BYTE 'File not found', 0

.code
     mov AX, TYPE var1              ; AX = 0001
     mov AX, TYPE var2              ; AX = 0002
     mov AX, TYPE var3              ; AX = 0004
     mov AX, TYPE var4              ; AX = 0001
     mov AX, TYPE msg               ; AX = 0001
```

LENGTHOF Operator

The LENGTHOF operator counts the number of individual elements in a variable that has been defined using DUP.

Syntax

```
mov reg16, LENGTHOF mem
```

Example

```
.data
     val1 WORD 1000h
     val2 SWORD 10, 20, 30
     array WORD 10 DUP(?), 0
     array2 WORD 5 DUP(3 DUP(0))
     msg BYTE 'File not found', 0

.code
     mov AX, LENGTHOF val1          ; AX = 1
```

```
mov AX, LENGTHOF val2          ; AX = 3
mov AX, LENGTHOF array         ; AX = 11
mov AX, LENGTHOF array2        ; AX = 15
mov AX, LENGTHOF msg           ; AX = 15
```

SIZEOF Operator

The SIZEOF operator returns the number of bytes an array takes up. It is similar in effect to multiplying LENGTHOF with TYPE.

Syntax

```
mov reg16/32, SIZEOF mem
```

Example

```
data
     intArray WORD 32 DUP(0)
.code
     mov EAX, SIZEOF intArray
```

Indirect Operands

In protected mode, an indirect operand can be any 32-bit general-purpose register (EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP) surrounded by brackets. 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
```

If the destination operand uses indirect addressing, a new value is placed in memory at the location pointed to by the register.

```
mov [esi],bl
```

Using PTR with Indirect Operands

```
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

Indirect operands are ideal tools for stepping through arrays.

Example

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

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
```

If we use an array of 32-bit integers, we `add 4` to `ESI` to address each subsequent array element.

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.

Syntax

```
constant [reg32]
[constant + reg32]
```

Example

```
.data
     arrayB BYTE 20, 40, 60, 80
.code
     mov esi, 1
     mov al, arrayB[esi]
     inc esi
     mov al, arrayB[esi]
     mov esi, 3
     mov al, [arrayB + esi]
```

Adding Displacements: The second type of indexed addressing combines a register with a constant offset. The index register holds the base address of an array.

Scale Factors in Indexed Operands

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

Syntax

```
constant [reg32* TYPE constant]
```

Example

```
.data
     arrayW WORD 1000h, 2000h, 3000h, 4000h
.code
     mov eax,0
     mov ebx,0
     mov ecx,0
     mov esi, 1
     mov ax, arrayW[esi * TYPE arrayW]
     mov esi, 2
     mov bx, arrayW[esi * TYPE arrayW]
     mov esi, 3
     mov cx, arrayW[esi * TYPE arrayW]
```

```
.data
     arrayW WORD 1000h,2000h,3000h
.code
     mov eax,0
     mov ebx,0
     mov ecx,0
     mov esi,OFFSET arrayW
     mov ax,[esi]                ; AX = 1000h
     mov bx,[esi+2]              ; AX = 2000h
     mov cx,[esi+4]              ; AX = 3000h
```