National Universi
of computer and emerging scien

Computer Organization and Assembly Language (EL-2003)

Semester: Fall 2025
Section: BCS-3D / BCS-H
Course Instructor: Muhammad Owais

LAB # 05

Working with Data related Operator and Directives, Addressing

Lab Objectives

By the end of this lab, students will be able to:

1. Analyze how arithmetic instructions (ADD, SUB) affect Flag Register flags (Zero, Carry, Sign, Auxiliary Carry, Parity, Overflow).
2. Use direct-offset operands to access array elements in memory.
3. Apply the OFFSET operator to get a variable's memory address.
4. Use the PTR operator to specify operand size (byte, word, doubleword).
5. Determine variable size in bytes using the TYPE operator.
6. Count array elements with the LENGTHOF operator.
7. Calculate total array size in bytes with the SIZEOF operator.
8. Access and modify data using indirect operands with registers.
9. Use indexed operands with scale factors to access array elements.

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

1) Z-Zero Flag:

The Zero flag is set when the result of an operation produces zero in the destination operand.
If the result is zero , ZF=1
Otherwise ,ZF=0

Example Source Code:

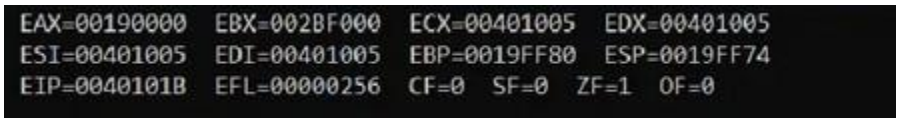
```
include irvine32.inc
.data
```

```
.code
Main Proc
mov AX,OFFFFH
inc AX
call dumpregs
exit
Main ENDP
END main
```

Explanation:

FFFFh+1h = 10000h. Since we have 0000h in AX after increment, ZF is set.

Output Snap:



2) C-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).

The `INC` and `DEC` (increment and decrement) instructions in assembly languages do not set the carry flag

1) Example Source Code:

```
include irvine32.inc
.data
.code
Main Proc
mov AL,OFH
add AL,OFIH
call dumpregs
exit
Main ENDP
END main
```

Explanation:

OFh+F1h = 100h. Since we have a carry, CF is set.

Output Snap:

```
EAX=0019FF00  EBX=002A6000  ECX=00401005  EDX=00401005
ESI=00401005  EDI=00401005  EBP=0019FF80  ESP=0019FF74
EIP=00401019  EFL=00000257  CF=1  SF=0  ZF=1  OF=0
```

2) Example Source Code:

```
include irvine32.inc
.data
.code
Main Proc
mov AX,12AEH
sub AX,12AFH
call dumpregs
exit
Main ENDP
END main
```

Explanation:

12AEh - 12AFh = -1h=FFFFh. Since it is negative (leading digits are 1), CF is set.

Output Snap:]]

```
EAX=0019FFFF  EBX=0021A000  ECX=00401005  EDX=00401005
ESI=00401005  EDI=00401005  EBP=0019FF80  ESP=0019FF74
EIP=0040101D  EFL=00000297  CF=1  SF=1  ZF=0  OF=0
```

3) S-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`
	`sub AL,97`
	sets the sign flag as the result is −82 (or 10101110 in binary)

4) AC-Auxilary Carry 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:

For example

```
mov al , OFh
Add al ,1                    ; AC=1
```

5) Parity flag :

Indicates even parity of the low 8 bits of the result, PF is set, if the lower 8 bits contain even number 1 bits. For 16- and 32-bit values, only the least significant 8 bits are considered for

computing parity value.

26 (11010)——————————— PF=0

102 (1100110)——————————— PF= 1

Examples

Example 1: 8-bit result

Result = 1010 1100 (0xAC)

Number of 1s = 4 → Even

PF = 1

Example 2: 16-bit result

Result = 1011 0011 0110 0101 (0xB365)

Least significant 8 bits = 0110 0101 (0x65)

Number of 1s = 4 → Even

PF = 1

Example 3: 32-bit result

Result = 1001 0100 1110 0110 0011 1100 1010 0001
Least significant 8 bits = 1010 0001 (0xA1)
Number of 1s = 3 → Odd
PF = 0

6) O-Over flow Flag:

It indicates out-of-range result on signed numbers. Signed number are counterpart of the carry flag. The following code sets the overflow flag but not the carry flag.

```
mov AL,72H                    ; 72H = 114D
add AL,OEH                    ; OEH = 14D
```

sum is +128, since it is out of range of 8 bit reg, so overflow flag become 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 ; AL = 10h
mov al,[arrayB+1] ; AL = 20h
mov ax,arrayW ; AX = 100h
 mov ax,[arrayW+2] ; AX = 200h
```

Data Related Operators and Directives:

1. OFFSET operator:

The OFFSET operator returns the number of bytes between the label and the beginning of its segment.

```
.data
 bVal BYTE ?
```

wVal WORD ?
dVal DWORD ?
dVal2 DWORD ?
If bVal is located at offset 00404000h, we would get:

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

2. PTR Operator:

Assembly instructions require operands to be the same size. However, it may be required at some point to operate on data in a size other than that originally declared. This can be done with the PTR operator (override a variable's default size).

For example, the PTR operator can be used to access the high-order word of a DWORD-size variable.

The syntax for the PTR operator is: **type PTR expression**

Must be used in combo with a data type: BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD, TBYTE.

```
.data
myDouble DWORD 12345678h
```

Suppose want to move 5678 to AX?

o Can NOT do mov ax,myDouble ; error since mismatched sizes

Remember little-endian storage:

Doubleword	Word	Byte	Offset	
12345678	5678	78	0000	myDouble
		56	0001	myDouble + 1
	1234	34	0002	myDouble + 2
		12	0003	myDouble + 3

Example 1: The PTR Operator

```
.data
myDouble DWORD 12345678h
.code
mov ax,WORD PTR myDouble        ; 5678
mov ax,WORD PTR [myDouble+2]    ; 1234
mov bl,BYTE PTR myDouble        ; 78
```

Can use it to move smaller to larger too…

The DWORD PTR operator makes this possible:

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

3. TYPE Operator:

Returns the size, in bytes, of a single element of a variable.
Syntax: TYPE var_name
BYTE: 1
WORD: 2
Etc.

```
.data
v1 BYTE ?
v2 WORD ?
v3 DWORD ?
v4 QWORD ?
TYPE v1        ;returns 1
TYPE v2        ;returns 2
TYPE v3        ;returns 4
 TYPE v4       ;returns 8
```

Example

```
.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
```

4. LENGTHOF Operator:

Returns number of elements in an array, base on values appearing on 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
```

```
LENGTHOF byte1          returns 3
LENGTHOF array1         returns 30+2
LENGTHOF array2         returns 5*3
LENGTHOF array3         returns 4
LENGTHOF digitStr       returns 9
```

Example: LENGTHOF Operator

```
.data
        ;declaration of arrays
        array_1   WORD      40 DUP (5)
        num       DWORD     4, 5, 6, 7, 8, 9, 10, 11
        warray    WORD      40 DUP (40 DUP (5))
        larray EQU    LENGTHOF array      ; 40 elements
        lnum   EQU    LENGTHOF num        ; 8 elements
        len    EQU    LENGTHOF warray     ; 1600 elements
```

Be careful:

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

LENGTHOF myArray is 5

as opposed to

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

LENGTHOF myArray is 10

5. SIZEOF Operator:

The number of bytes taken up by a structure, Basically:

LENGTHOF * TYPE

Example 1: SIZEOF Operator :

```
.data
intArray WORD 32 DUP(0)
.code
mov eax,SIZEOF intArray ; returns 64 = 32 * 2
```

Indirect Operands:

In 32-bit protected mode, an indirect operand can be any 32-bit general-purpose register (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP) surrounded by brackets []. The register contains the address of some data, and the value is accessed using that address.

Example:

```
.data
byteVal BYTE 10h
.code
mov esi, OFFSET byteVal          ; ESI = address of 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.

  Example:

```
mov [esi], bl ; Store BL into memory at address ESI
```

- When incrementing memory values, the assembler must know the operand size.

  Otherwise, it shows an error:

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

  Solution: Use the PTR operator to specify size:

```
inc BYTE PTR [esi]       ; Increment 1 byte
inc WORD PTR [esi]       ; Increment 2 bytes
inc DWORD PTR [esi]      ; Increment 4 bytes
```

Arrays:

Indirect addressing is useful when working with arrays.

1) Array of Bytes

```
.data
arrayB BYTE 10h, 20h, 30h
.code
mov esi, OFFSET arrayB     ; ESI = address of first element
mov al, [esi]              ; AL = 10h
inc esi                    ; Move to next element
mov al, [esi]              ; AL = 20h
```

2) Array of 16-bit Integers (WORD)

```
.data
 arrayW WORD 1000h, 2000h, 3000h
.code
mov esi, OFFSET arrayW     ; ESI = address of first element
mov ax, [esi]              ; AX = 1000h
add esi, 2                 ; Move to next element
mov ax, [esi]              ; AX = 2000h
```

3) Array of 32-bit Integers (DWORD)

```
.data
arrayD DWORD 11111111h, 22222222h, 33333333h
.code
mov esi, OFFSET arrayD      ; ESI = address of first element
mov eax, [esi]               ; EAX = 11111111h
add esi, 4                   ; Move to next element
mov eax, [esi]               ; EAX = 22222222h
```

IndexedOperands

1. Indexed Operands Overview

An **indexed operand** adds a **constant** to a **register** to generate an **effective address**.
Any 32-bit general-purpose register (`EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP`) can be used as an **index register**.

Syntax:

constant [reg32]
[constant + reg32]

Example 1: Indexed Operand with Byte Array

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

2. Adding Displacements

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.

Example 2: Using Displacement with WORD Array

```
INCLUDE Irvine32.inc
.data
arrayW WORD 1000h, 2000h, 3000h
.code
main PROC
mov esi, OFFSET arrayW
mov ax, [esi]          ; AX = 1000h
mov bx, [esi+2]        ; BX = 2000h
mov cx, [esi+4]        ; CX = 3000h
```

3. Scale Factors in Indexed Operands

When working with **arrays of words** or **doublewords**, the offset must consider the **size** of each element.
We use the **TYPE** operator to automatically calculate the correct offset.

Syntax:

constant [reg32 * TYPE constant]

Example 3: Using TYPE Operator for WORD Array

```
INCLUDE Irvine32.inc
.data
arrayW WORD 1000h, 2000h, 3000h, 4000h
.code
main PROC
mov esi, 1
mov ax, arrayW[esi * TYPE arrayW]          ; AX = 2000h
mov esi, 2
mov bx, arrayW[esi * TYPE arrayW]          ; BX = 3000h
mov esi, 3
mov cx, arrayW[esi * TYPE arrayW]          ; CX = 4000h
```

LAB TASKS

Task # 01:

Write an assembly program with the following instructions:

```
mov AL, 7Fh
add AL, 1
```

a) Determine the values of the Zero (ZF), Sign (SF), Carry (CF), and Overflow (OF) flags after execution.
b) Replace 'add AL, 1' with 'sub AL, 80h'. Again, state the values of ZF, SF, CF, and OF.

Task # 02:

```
.data
myByte   BYTE  12h
myWord   WORD  1234h
myDword  DWORD 12345678h
```

a) Use the OFFSET operator to load the address of each variable into the ESI register.
b) Use the PTR operator to move the high-order word of myDword into AX.
c) Use the TYPE operator to load the size (in bytes) of each variable into BX.

Task # 03:

Declare following arrays.

```
.data
arr1 BYTE  10,20,30,40
arr2 WORD  100h,200h,300h
arr3 DWORD 5 DUP(0)
```

a) Use LENGTHOF to calculate the number of elements in each array.
b) Use SIZEOF to calculate the total storage (in bytes) of each array.
c) Store the results in registers AX, BX, and CX respectively.

Task # 04:

```
.data
arrayB BYTE  11h,22h,33h
arrayW WORD  4444h,5555h,6666h
```

a) Use ESI with indirect addressing to read all elements of arrayB.
b) Use ESI with indirect addressing to read all elements of arrayW.
c) Explain why the increment of ESI is different in both cases.

Task # 05:

Consider the following arrays.

```
.data
arrayD DWORD 1000h,2000h,3000h,4000h
```

a) Load the 2nd element of arrayD into EAX using an indexed operand with a scale factor and the TYPE operator.
b) Load the last element of arrayD into EBX using the same method.
c) Explain the role of the TYPE operator in this example.