**EL-213: Comp. Organization & Assembly Language Lab**

**Lab#03:** *Operators, Instructions & Flags*

# OBJECTIVES:

* Assembly language Instructions : MOV , ADD , SUB , INC , DEC, MOVZX, MOVSX
* Some useful Assembly Language Operators DUP, EQU
* Observing affect of Arithmetic Instructions on Flag Register

## MOV Instruction:

It is used to move data from source operand to destination operand

• 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.

## Syntax:

MOV DESTINATION, SOURCE

## Example:

MOV BX, 2 MOV CX, BX

## Store the ASCII character in register:

‘A’ has ASCII code 65D (01000001B, 41H)

The following mov instruction carries out the task: MOV BX, 65d ;

MOV BX, 41h;

MOV BX, 01000001b; MOV BX, ‘A’;

All of the above are equivalent.

## INC Instruction:

The INC instruction takes one operand and adds 1 to it.

## Example:

MOV AX, 08d

INC AX ; add 1 to ax ; ax now contains 9

## DEC Instuction:

The dec instruction like inc takes one operand and subtracts 1 from it.

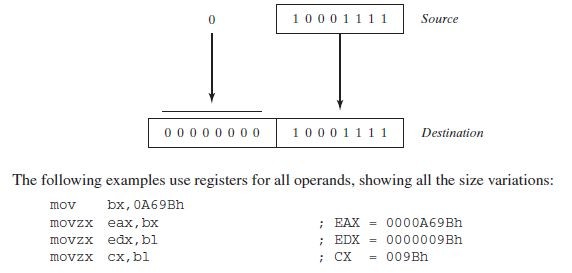
## Example:

DEC AX ; subtract 1 from ax

## MOVZX Instruction (MOV with zero-extend):

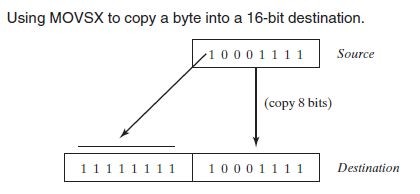
The MOVZX instruction moves the contents and zero­extends the value to 16 or 32 bits. This instruction is only used with unsigned integers. There are three variants:

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



## MOVSX Instruction (MOV with sign-extend):

The MOVSX instruction moves the contents and sign­extends the value to 16 or 32 bits. This instruction is only used with signed integers.



* **OPERATORS:**

## DUP Operator

The DUP operator allocates storage for multiple data items, using a constant expression as a counter. It is particularly useful when allocating space for a string or array, and can be used with initialized or uninitialized data:

BYTE 20 DUP(0) ; 20 bytes, all equal to zero

BYTE 20 DUP(?) ; 20 bytes, uninitialized

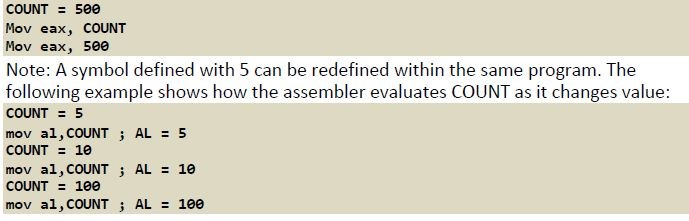
BYTE 4 DUP("STACK") ;20 bytes: "STACKSTACKSTACKSTACK"

* **SYMBOLIC CONSTANTS:**

A symbolic constant (or symbol definition) is created by associating an identifier (a symbol) with an integer expression or some text. Symbols do not reserve storage. They are used only by the assembler when scanning a program, and they cannot change at runtime.

## 1. Equal­Sign (=) Directive

The equal­sign (=) directive associates a symbol name with an integer expression



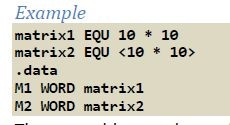
## 2. EQU Directive

The EQU directive associates a symbolic name with an integer expression or some arbitrary text. Ordinarily, expression is a 32­bit integer value. When a program is assembled, all occurrences of name are replaced by expression during the assembler’s preprocessor step.

**Syntax:**

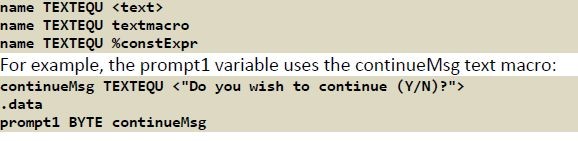
*name EQU expression*

### *name EQU symbol name EQU <text>*



## 3. TEXTEQU Directive

The TEXTEQU directive, similar to EQU, creates what is known as a text macro. There are three different formats: the first assigns text, the second assigns the contents of an existing text macro, and the third assigns a constant integer expression.



# 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
* Not all instructions affect all status flags:
* ADD and SUB affect all six flags
* INC and DEC affect all but the carry flag
* MOV, PUSH, and POP do not affect any flags

## Z­ Zero Flag:

This flag is set, if the result of the computation or comparison performed by the previous instruction is zero.



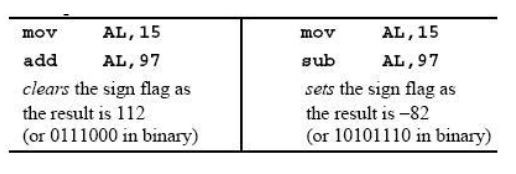
## C­ Carry Flag:

This flag is set, when there is a carry out of MSB in case of addition and borrow in case of subtraction. Ranges of 8, 16, and 32 bit unsigned numbers are:

* 8 bits 0 to 255 (28 ­ 1)
* 16 bits 0 to 65,535 (216 ­ 1)
* 32 bits 0 to 4,294,967,295 (232­1)

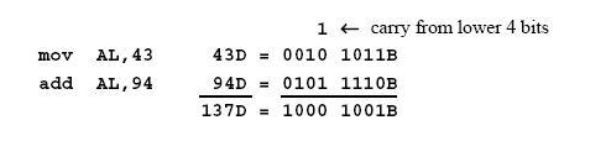
## S­Sign Flag:

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



## AC­Auxilary 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.



## P­ Parity Flag:

This flag is set to 1, if the lower byte of the result contains even number of 1’s

## O­ Over flow Flag:

This flag is set, if an overflow occurs, i.e., if the result of a signed operation is too large to fit into a destination register. Range of 8­, 16­, and 32­bit signed numbers:

* 8 bits (­ 128 to +127)
* 16 bits (­ 32,768 to +32,767 215)
* 32 bits (­2,147,483,648 to +2,147,483,647 231)

## Example:

The following program implements various arithmetic expressions using the ADD, SUB, INC, DEC, and NEG instructions, and shows how certain status flags are affected:

INCLUDE Irvine32.inc

.data

**Rval SDWORD ? Xval SDWORD 26**

**Yval SDWORD 30**

**Zval SDWORD 40**

.code

main PROC

; INC and DEC

mov ax,1000h inc ax ; 1001h dec ax ; 1000h

; Expression: Rval = ­Xval + (Yval ­ Zval)

mov eax,Xval

neg eax ; ­26 mov ebx,Yval

sub ebx,Zval ; ­10 add eax,ebx

mov Rval,eax ; ­36

; Zero flag example:

mov cx,1

sub cx,1 ; ZF = 1 mov ax,0FFFFh inc ax ; ZF = 1

; Sign flag example:

mov cx,0

sub cx,1 ; SF = 1 mov ax,7FFFh add ax,2 ; SF = 1

; Carry flag example: mov al,0FFh

add al,1 ; CF = 1, AL = 00

; Overflow flag example: mov al,+127

add al,1 ; OF = 1 mov al,­128

sub al,1 ; OF = 1 exit

main ENDP END main

**EXERCISES:**

### Convert the following high-level instruction into Assembly Language: x = (x+1) – (y­1) + y

1. Write a program in assembly language that implements following expression:

*eax = ­val2 + 7 – val3 +val1*

Use these data definitions:

*val1 word 8*

*val2 word 15*

*val3 word 20*

1. Write a program which declares a symbolic constant named ***SecondsInDay*** using the equal-sign directive and assign it an arithmetic expression that calculates the number of seconds in a 24-hour period.
2. Write a program to find area of a square. Declare necessary variable *side* for the program (assign any arbitrary value to the variable).
3. Write a program to find area of a rectangle. Declare necessary variables *length* & *width* for the program (assign arbitrary values to the variables).
4. Write a program to find area of a triangle. Declare all necessary variables for the program (give arbitrary values to the variables).
5. Use this code for the following questions:

**.data**

**val1 BYTE 10h val2 WORD 8000h**

**val3 DWORD 0FFFFh**

**val4 WORD 7FFFh**

i. Write an instruction that increments val2.

ii. Write an instruction that subtracts val3 from EAX.

iii. Write instructions that subtract val4 from val2.

iv. If val2 is incremented by 1 using the ADD instruction, note down the values of Carry and Sign flags?

v. If val4 is incremented by 1 using the ADD instruction, note down the values of Overflow and Sign flag.