LAB 04

WORKING WITH OPERATORS, INSTRUCTIONS & SYMBOLIC CONSTANTS

STUDENT NAME _____ ROLL NO _____ SEC ___

SIGNATURE & DATE _____

MARKS AWARDED: _____

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES (NUCES), KARACHI

Prepared by: Aashir Mahboo-Muhammad Nadeem

Version: 1.0
Date:

Lab Session 04 Working with Operators,Instructions & Symbolic Constants

Objectives:

- Assembly language Instructions: MOV , ADD , SUB , INC , DEC, MOVZX, MOVSX, XCHG
- Some useful Assembly Language Operators DUP, EQU

Operand Types:

As x86 instruction formats:

[label:] mnemonic [operands][; comment]

Because the number of operands may vary, we can further subdivide the formats to have zero, one, two, or three operands.
Here, we omit the label and comment fields for clarity:

mnemonic
mnemonic [destination]
mnemonic [destination],[source]
mnemonic [destination],[source-1],[source-2]

x86 assembly language uses different types of instruction operands. The following are the easiest to use:

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

Following table lists a simple notation for operands. We will use it from this point on to describe the syntax of individual instructions.

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:

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

Here is a list of the general variants of MOV, excluding segment registers:

Example:
MOV *bx, 2*
MOV *ax, cx*

Example:
'A' has ASCII code 65D (01000001B, 41H)

The following MOV instructions stores it in register BX:

MOV bx, 65d
MOV bx, 41h
MOV bx, 01000001b
MOV bx, 'A'
All of the above are equivalent.

Examples:
The following examples demonstrate compatibility between operands used with MOV instruction:

MOV ax, 2 ✓
MOV 2, ax ✗
MOV ax, *var* ✓
MOV *var*, ax ✓
MOV *var1*, *var2* ✗
MOV 5, *var* ✗

Overlapping Values
Example:

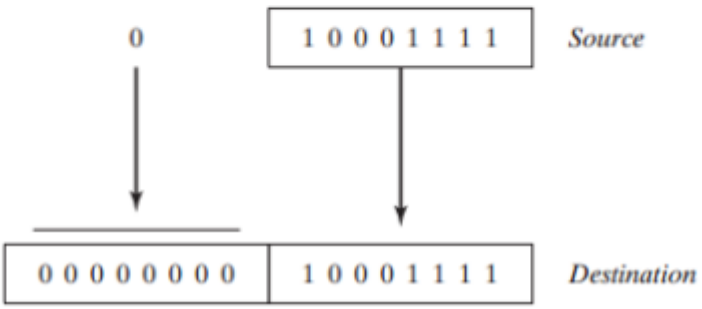
```
.data
oneByte BYTE 78h
oneWord WORD 1234h
oneDword DWORD 12345678h

.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 (MOV with zero-extend) instruction moves the contents and zero-extends the value to 16 or 32 bits. This instruction is only used with unsigned integers.

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



The following examples use registers for all operands, showing all the size variations:

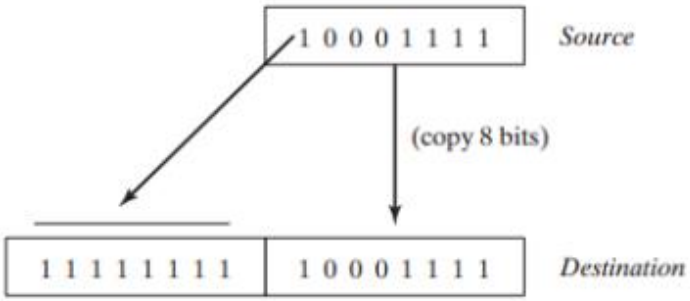
```
mov    bx,0A69Bh
movzx  eax,bx                  ; EAX = 0000A69Bh
movzx  edx,bl                  ; EDX = 0000009Bh
movzx  cx,bl                   ; CX  = 009Bh
```

The following examples use memory operands for the source and produce the same results:

```
.data
byte1  BYTE 9Bh
word1  WORD 0A69Bh
.code
movzx  eax,word1               ; EAX = 0000A69Bh
movzx  edx,byte1               ; EDX = 0000009Bh
movzx  cx,byte1                ; CX  = 009Bh
```

MOVSX Instruction

The MOVSX (MOV with sign extend) instruction moves the contents and sign extends the value to 16 or 32 bits. This instruction is only used with signed integers.



The following examples use registers for all operands, showing all the size variations:

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

INC Instruction

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

Example:
MOV *ax, 8*
INC *ax* ; *ax now contains 9*

DEC Instruction

The DEC instruction takes an operand and subtracts 1 from it.

Example:
MOV *ax, 5*
DEC *ax* *; ax now contains 4*

Lab Exercise:

1. What errors are present in the following?
   - MOV AX 3d
   - MOV 23, AX
   - MOV CX, CH
   - MOVE AX, 1h
   - ADD 2, CX
   - ADD 3, 6
   - INC AX, 2

2. Store the ASCII codes for starting three letters of your name in a register.

3. Use following array declarations:

varB BYTE +10
varW WORD -150
varD DWORD 600

Now move every element to EAX,EBX and ECX.

XCHG Instruction

The XCHG (exchange data) instruction exchanges the contents of two operands. There are three variants:

XCHG does not accept immediate operands.

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

To exchange two memory operands, use a register as a temporary container and combine MOV with XCHG:

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

ADD Instruction

The ADD instruction adds a source operand to a destination operand of the same size. *Source* is unchanged by the operation, and the sum is stored in the destination operand.

The syntax is

ADD *dest,source*

SUB Instruction

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

The syntax is

SUB *dest,source*

NEG Instruction

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

NEG *reg*

NEG *mem*

Lab Exercise:

4. Implement the following high-level mathematical equations into assembly language ***using x86 general purpose registers.***

1. EAX = 89 + 74Fh - 47o - 28 +11011b
2. EAX = Val1 + Val2 - 654h + Val3

Val1 DWORD 25h
Val2 BYTE 36o
Val3 WORD 20d

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. Equal-Sign (=) Directive

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

The syntax is

name = expression

Example:

```
COUNT = 500
mov eax, COUNT
mov eax, 500
```

Redefinitions: A symbol defined with can be redefined within the same program. The following example shows how the assembler evaluates COUNT as it changes value:

```
COUNT = 5
mov al,COUNT                    ; AL = 5
COUNT = 10
mov al,COUNT                    ; AL = 10
```

2. EQU Directive:

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

```
name EQU expression
name EQU symbol
name EQU <text>
```

- Expression must be a valid integer expression
- Symbol is an existing symbol name, already defined with = or EQU.
- any text may appear within the brackets <. . .>.

Example:

```
matrix1  EQU   10 * 10
matrix2  EQU  <10 * 10>
.data
M1 WORD matrix1
M2 WORD matrix2
```

The assembler produces different data definitions for M1and M2. The integer expression in matrix1 is evaluated and assigned to M1. On the other hand, the text in matrix2is copied directly into the data definition for M2:

```
M1 WORD 100
M2 WORD 10 * 10
```

Example:

The following program implements various arithmetic expressions using the ADD, SUB, INC, DEC, and NEG instructions.

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

The following signed 32-bit variables will be used:

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

When translating an expression, evaluate each term separately and combine the terms at the end. First, we negate a copy of **Xval**:

```
; first term: -Xval
mov  eax,Xval
neg  eax                        ; EAX = -26
```

Then **Yval** is copied to a register and **Zval** is subtracted:

```
; second term: (Yval - Zval)
mov  ebx,Yval
sub  ebx,Zval                   ; EBX = -10
```

Finally, the two terms (in EAX and EBX) are added:

```
; add the terms and store:
add  eax,ebx
mov  Rval,eax                   ; -36
```

Lab Exercise:

6. 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.
7. Let A = 0FF10 h and B = 0E10B h, you need to write an assembly language code to swap the contents.
8. Use this data 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.

1. ***Submit screenshots of each task containing register values i.e. Debugging window. (in single word file).***
2. ***The codes of all task in a text file. Use Notepad.***
3. ***Submissions should be made on Google Classroom***.