

LAB 04

Computer Organization & Assembly Language (COAL) | EL 2003



Student Name: _____

Roll No.: _____

Section: _____

Marks Awarded _____

Prepared by: Talha Shahid

Objectives:

- Assembly Language Instruction: [MOV](#), [ADD](#), [SUB](#), [INC](#), [DEC](#), [MOVZX](#), [MOVSX](#), [XCHG](#)
- Symbolic Constants
- Example Program

Section 01

Assembly Language Instruction: [MOV](#), [ADD](#), [SUB](#), [INC](#), [DEC](#), [MOVZX](#), [MOVSX](#), [XCHG](#)

x86 Instruction Format:

```
[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
<i>reg8</i>	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
<i>reg16</i>	16-bit general-purpose register: AX, BX, CX, DX, SI, DL SP, BP
<i>reg32</i>	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
<i>reg</i>	Any general-purpose register
<i>sreg</i>	16-bit segment register: CS, DS, SS, ES, FS, GS
<i>imm</i>	8-, 16-, or 32-bit immediate value
<i>imm8</i>	8-bit immediate byte value
<i>imm16</i>	16-bit immediate word value
<i>imm32</i>	32-bit immediate double word value
<i>reg/mem8</i>	8-bit operand, which can be an 8-bit general register or memory byte
<i>reg/mem16</i>	16-bit operand, which can be a 16-bit general register or memory word
<i>reg/mem32</i>	32-bit operand, which can be a 32-bit general register or memory double word
<i>mem</i>	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:

Examples

```
MOV bx, 2  
MOV ax, cx
```

Examples

'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

Examples

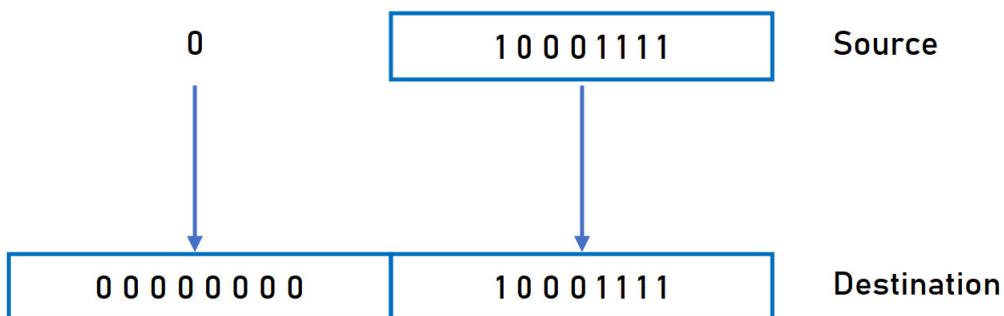
```
.data  
    oneByte BYTE 78h  
    oneWord WORD 1234h  
    oneDword DWORD 12345678h  
.code  
    mov eax, 0          ; EAX = 00000000h  
    mov al, oneByte     ; EAX = 00000078h  
    mov ax, oneWord     ; EAX = 0001234h  
    mov eax, oneDword   ; EAX = 12345678h  
    mov ax, 0          ; EAX = 12340000h
```

MOVZX Instruction

The **MOVZX** (MOV with zero-ex tend) 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:

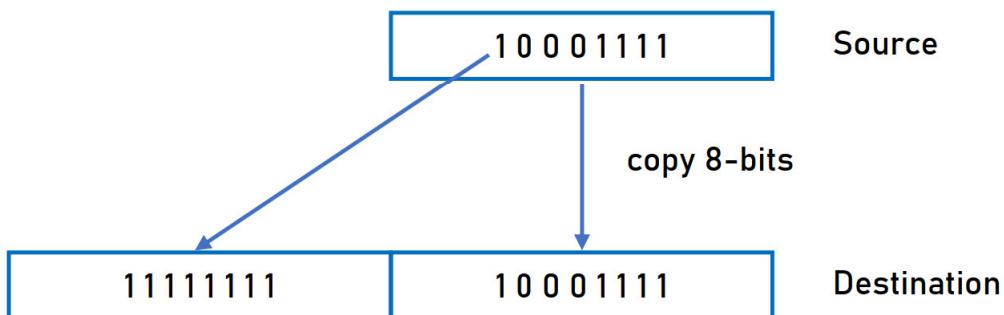
```
mov bx, 0A69Bh  
mov bl, 09Bh  
movzx eax, bx ; EAX = 0000A69Bh  
movzx edx, bl ; EDX = 0000009Bh  
movzx cx, bl ; CX = 009Bh
```

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 ex, byte1 ; CX = 009Bh
```

MOVSX Instruction

The [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 = FFFFFFF9Bh
mov bl, 7Bh    ; BL = 7Bh
movsx cx, bl ; CX = 007Bh
```

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
```

XCHG Instruction

The **XCHG** (exchange data) instruction exchanges the contents of two operands. **XCHG** does not accept immediate operands. There are three variants:

```
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.

Syntax:

```
ADD dest, source
```

SUB Instruction

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

Syntax:

```
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
```

Section 02 | Symbolic Constant

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 equal-sign (=) directive associates a symbol name with an integer expression

Syntax:

```
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.

Syntax:

```
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 with 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 [M1](#) and [M2](#). The integer expression in [matrix1](#) is evaluated and assigned to [M1](#). On the other hand, the text in [matrix2](#) is copied directly into the data definition for [M2](#).

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

Section 03 | Example Program

The following program implements various arithmetic expressions using the [ADD](#), [SUB](#), and [NEG](#) instructions.

```
INCLUDE Irvine32.inc

; Rval = -Xval + (Yval - Zval)
; The following signed 32-bit variables will be used:

.data
    Rval SDWORD ?
    XVAL SDWORD 26
    Yval SDWORD 30
    Zval SDWORD 40

.code
main PROC

    ; 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       ; Rval = -36
    mov EAX, Rval

    call WriteInt
    exit
main ENDP
END main
```