Computer Org. & Assembly Language Lab

Lab#04

Agenda

- Instructions of Variation of MOV Introductions
  - MOV
  - MOVZX
  - MOVSX
- XCHG Instruction for Swapping
- Direct Offset Operand
- Addition and Subtraction
  - INC and DEC Instructions
  - NEG Instruction
- Flags Affected by Arithmetic
  - Zero and Sign Flags
  - Carry Flag
  - Overflow Flag
  - Demonstration of Arithmetic & Flags

Instructions of Variation of MOV Introductions

MOV

Copies data from a source operand to a destination operand of the same size, it is equivalent to assignment operator as in C/C++. General syntax of the MOV instruction is as follows:

```
MOV [destination], [source]
e.g. MOV eax, ebx
```

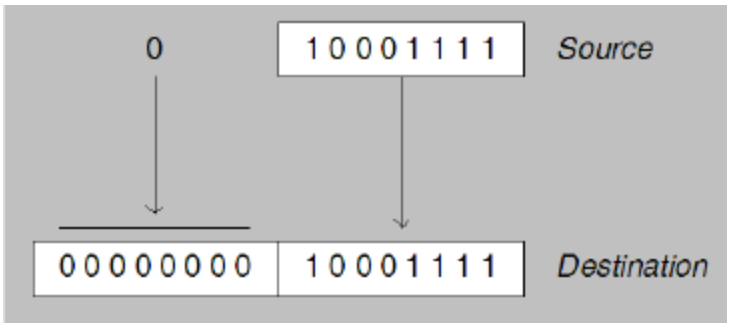
MOVZX

When you copy a smaller value into a larger destination, the MOVZX instruction fills (extends) the upper half of the destination with zeros.

Only register is allowed as a destination operand. General possible instruction formats are:

```
MOVZX reg32, reg16        MOVZX reg32, mem16
MOVZX reg16, reg8         MOVZX reg16, mem8
MOVZX reg32, reg8         MOVZX reg32, mem8
```

Following figure shows an 8 – bit source operand zero – extended into a 16 – bit destination




```
mov bx,0A69Bh
movzx eax,bx        ;EAX = 0000A69B
movzx edx,bl        ;EDX = 0000009B
movzx cx,bl         ;CX = 009B
```

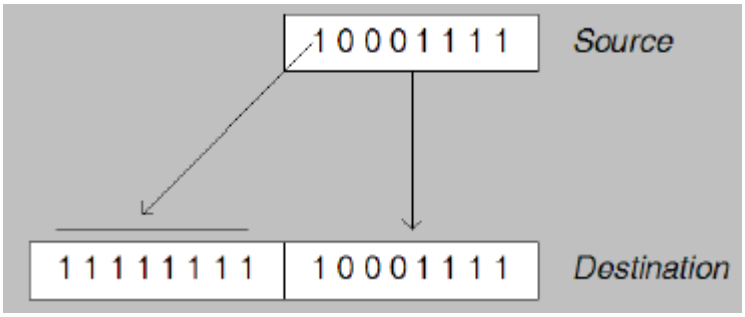
MOVSX

Copies data from a source operand to destination register and sign extends into the upper half of the destination. This instruction is used to copy and 8 – bit or 16 – bit operand into a larger destination.

Only register is allowed as destination operand. General possible instruction formats are given below

MOVSX reg32, reg16	*MOVSX reg32, mem16*
MOVSX reg16, reg8	*MOVSX reg16, mem8*
MOVSX reg32, reg8	*MOVSX reg32, mem8*

If an 8 – Bit value of 1000111b is moved to a 16 - bit destination, the lowest 8 bits are copied as it. The highest bit of the source is copied into each of the high 8 bit positions of the destination




```
.data
        count SBYTE -16d
.code
main PROC
        XOR EBX,EBX                 ;clears the register
        XOR EAX,EAX                 ;clears the register
        mov al,count                ;ax=000000F0
        movsx bx,count ;bx=FFF0
                                    ; it fills the sign bit(1) to whole next 8-bits
        call DumpRegs;
```

XCHG Instruction for Swapping

XCHG exchanges the values of two operands. At least one operand must be a register. No immediate operands are allowed. This instruction is equivalent to swap () in C++.

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

```
XOR EAX,EAX         ; to clear the register
XOR EBX,EBX         ;to clear the register
mov bx,0FFFFh
call DumpRegs       ;display contents before exchange
xchg ax,bx          ; exchange 16-bit regs and ax = bx & bx = ax
```

```
call DumpRegs          ;display contents after exchange
mov al,0EEh
call DumpRegs          ;display contents before exchange
xchg ah,al             ;exchange 8-bit regs
call DumpRegs          ;display contents after exchange
```

Direct Offset Operand

A displacement can be added to a variable by creating a direct offset operand. It provides direct access to memory location having no explicit labels like variables. A constant offset is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location. As shown in below.

```
;example of BYTE array
arrayB BYTE 10h,20h,30h,40h     ;declare an array in .data section
mov al,arrayB                          ; AL = 10h
mov ah,[arrayB+1]                      ; AH = 20h
mov ah,[arrayB+2]                      ; AL = 30h
; As BYTE variable reserves 1 – Byte for storage, so to access next element we add 1 to pervious element address (EA)
```

In the same way we can access the other members of the BYTE Array keeping in mind the range of array. MASM does not require square brackets necessarily, to access 4th value from the array.

```
mov al,[arrayB+3]              ; AL = 40h
                ;OR
mov al,arrayB+3                ; AL = 40h ; alternative way to access
```

Note

MASM has no built-in range checking of array (EA), as the given below instruction will be executed and MASM will retrieve a BYTE form memory. Execute the following instructions and find what data is retrieved

```
Mov al, [arrayB + 20]          ; AL = ??
Mov al, [arrayB -1]            ; AL = ??

; As you the array size is 4, and this element should not be accessible as per logic but MASM retrieves the data from memory.
```

So this creates a hard to find logic bug, so you should be extra careful about this when checking array references.

```
;example of WORD array
arrayW WORD 1000h,2000h,3000h,4000h ; declare an array in .data section
mov ax , arrayW ; AX = 1000h
mov ax , [arrayW + 2] ; AX = 2000h
mov ax , [arrayW + 4] ; AX = 3000h
mov ax , [arrayW + 6] ; AX = 4000h
; As WROD variable reserves 2 – Byte for storage, so to access next element we add 2 to pervious element address (EA)
```

```
INCLUDE Irvine32.inc
.data
        val1 WORD 1000h
        val2 WORD 2000h
        arrayB BYTE 10h,20h,30h,40h,50h
        arrayW WORD 1000h,2000h,3000h
        arrayD DWORD 10000000h,2000000h
.code
main PROC
        ; MOVZX
        mov bx,0A69Bh
        movzx eax,bx                    ; EAX = 0000A69Bh
        movzx edx,bl                    ; EDX = 0000009Bh
        movzx cx,bl                     ; CX = 009Bh

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

        ; Memory-to-memory exchange:
        mov ax,val1                     ; AX = 10000h
        xchg ax,val2                    ; AX = 20000h, val2 = 10000h
        mov val1,ax                     ; val1 = 20000h

        ; Direct-Offset Addressing (byte array):
        mov al,arrayB                   ; AL = 10h
        mov al,[arrayB+1]               ; AL = 20h
        mov al,[arrayB+2]               ; AL = 30h
```

```
        ; Direct-Offset Addressing (word array):
        mov ax,arrayW                   ; AX = 100h
        mov ax,[arrayW+2]               ; AX = 200h

        ; Direct-Offset Addressing (doubleword array):
        mov eax,arrayD                  ; EAX = 10000000h
        mov eax,[arrayD+4]              ; EAX = 20000000h
exit
main ENDP
END main
```

Addition and Subtraction

Operations like addition and subtraction are very common on numerical data. These operations are very commonly performed by CPU. Increment, decrement and negation are also the applications of them. In assembly following instructions are fore theses operations INC, DEC, ADD, SUB and NEG

INC and DEC Instructions

INC (increment) instruction adds one (1) to the operand result is also stored though same operand. DEC (decrement) instruction subtracts one (1) from operand result is also stored in same operand.
The sample format/syntax is given below

```
INC reg/mem
DEC reg/mem
```

```
mov eax , 512d          ;EAX = 512
mov ebx , 512d          ;EBX = 512
INC eax                 ;EAX = 513 (EAX = EAX+1)
DEC ebx                 ;EBX = 511 (EBX = EBX-1)
```

So, in conclusion we can say, INC is replaceable with ADD eax, 1 and DEC is SUB ebx, 1.

NEG Instruction

The NEG(negate) instruction reverses the sign of an operand. Operand can be a register or memory operand. It converts a number to its two's complement. General format for this instruction is given below:

```
        NEG reg
        NEG mem
```

For example

```
mov eax, 512d     ; EAX = 512
NEG eax           ; EAX = -512
call DumpRegs
```

Flags Affected by Arithmetic

The ALU has a number of status flags that reflect the outcome of arithmetic operations based on the contents of the destination operand. Some essential flags are given below

- ◊ Zero flag – set when destination equals zero
- ◊ Sign flag – set when destination is negative
- ◊ Carry flag – set when unsigned value is out of range
- ◊ Overflow flag – set when signed value is out of range

NOTE

A flag is set => Flag value = 1

A flag is clear =>Flag value = 0

Zero and Sign Flags

The Zero flag is set (=1) when the destination operand of an operand of an arithmetic instruction is assigned a value. For example

```
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
```

The Sign flag is set when the destination operand is negative. The flag is clear when the destination is positive.
Carry Flag

```
mov cx,0
sub cx,1          ; CX = -1, SF = 1
add cx,2          ; CX = 1, SF = 0
```

Carry Flag

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
```

Overflow Flag

Overflow flag relevant only for signed arithmetic. The Overflow flag is set when the signed result of an operation is invalid or out of range.

For example

```
        ; Example 1
        mov al,+127
        add al,1
```

```
        ; Example 2
        mov al,-128         ; OF = 1
        sub al,1
```

Demonstration of Arithmetic & Flags

```
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 ; 10001h
        dec ax ; 10000h

        ; 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

        ; 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
```

NOTE

Students often ask how the CPU knows whether a number is signed or unsigned. One can only give what seems to be a dumb answer: The CPU doesn' t know-only the programmer knows. The CPU sets all the status flags after an operation, not knowing which of the flags will be important to the programmer. The programmer chooses which flags to interpret, and which flags to ignore.

The Carry Flag (CF)

- CF = 1 if there is a carry out from the msb (most significant bit) on addition, or there is a borrow into the msb on subtraction
- CF = 0 otherwise
- CF is also affected by shift and rotate instructions

The Parity Flag (PF)

- PF = 1 if the low byte of a result has an even number of one bits (even parity)
- PF = 0 otherwise (odd parity)

The Auxiliary Carry Flag (AF)

- AF = 1 if there is a carry out from bit 3 on addition, or there is a borrow into the bit 3 on subtraction
- AF = 0 otherwise
- AF is used in binary-coded decimal (BCD) operations

The Zero Flag (ZF)

- ZF = 1 for a zero result
- ZF = 0 for a non-zero result

The Sign Flag (SF)

- SF = 1 if the msb of a result is 1; it means the result is negative if you are giving a signed interpretation
- SF = 0 if the msb is 0

The Overflow Flag (OF)

- OF = 1 if signed overflow occurred
- OF = 0 otherwise

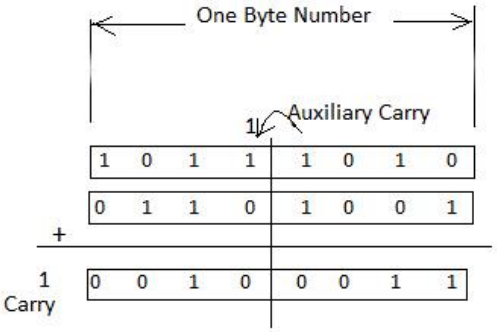


Figure-(b) : Example of Auxiliary Carry

Practice Session

Execute the following code and clear your mind about flags.

```
mov ax,00FFh
add ax,1        ; AX=? SF=? ZF=? CF=?
sub ax,1        ; AX=? SF=? ZF=? CF=?
add al,1        ; AL=? SF=? ZF=? CF=?
mov bh,6Ch
add bh,95h      ; BH= ? SF=? ZF=? CF=?
mov al,2
sub al,3        ; AL=? SF=? ZF=? CF=?
```