Quick reminder

Dimensions	8 bits	8 bits	16 bits	32 bits	64 bits
Data Type	**byte**	**byte**	**word**	**doubleword**	**quadword**
Number of Hexadecimal digits	2	2	4	8	16
Registers	AH	AL	AX	EAX	EDX:EAX
	BH	BL	BX	EBX	
	CH	CL	CX	ECX	ECX:EBX
	DH	DL	DX	EDX	

Base 10	Base 16	Base 2
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Multiplication Instruction
(for unsigned representation)

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

Syntax: **MUL op**

op is called explicit operand

The MUL is realized different accordind to the explicit operand:

op is *reg/mem8* => **MUL *reg/mem8*** => *AL * reg/mem8 = AX*

op is *reg/mem16* => **MUL *reg/mem16*** => *AX * reg/mem16 = DX:AX*

op is *reg/mem32* => **MUL *reg/mem32*** => *EAX * reg/mem32 = EDX:EAX*

Division Instruction
(for unsigned representation)

Dividend	Divisor	Quotient	Remainder
AX	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX

Syntax: **DIV op**

op is called explicit operand

The DIV is realized different according to the explicit operand:

op is reg/mem8 => **DIV reg/mem8** => AX / reg/mem8 = AL – quotient
and AH – remainder

op is reg/mem16 => **DIV reg/mem16** => DX:AX / reg/mem16 = AX – quotient
and DX – remainder

op is reg/mem32 => **DIV reg/mem32** => EDX:EAX / reg/mem32 = EAX – quotient
and EDX – remainder

segment data
a db -2
b dw -2
c dd -2

$(-2)_{10} = (1111.1110)_2$ - byte

$|-2| = 2 = (0000.0010)_2$

$$\begin{array}{r} (1111.1101)_2 + \\ 0000.0001 \\ \hline (1111.1110)_2 \end{array}$$

$\Rightarrow (-2)$ - word: $(1111.1111.1111.1110)_2$

(-2) - double word: $(\underbrace{11\ldots1}_{24}.1111.1110)_2$

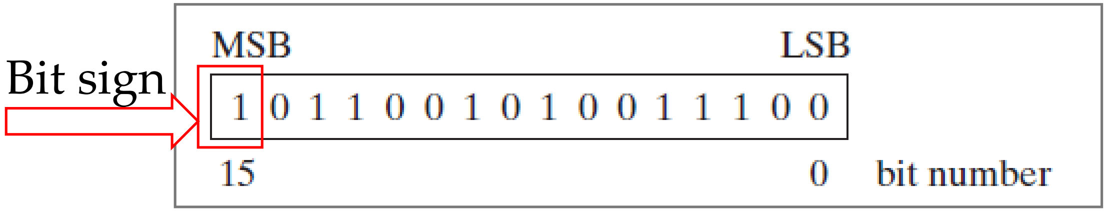
b16: (-2) - byte = FE
- word = FF.FE
- doubleword = FF.FF.FF.FE

$\Rightarrow$ in memory: FE FE FF FE FF FF FF - values

a+0	b+0	b+1	c+0	c+1	c+2	c+3	-	addresses

Signed conversions

- **Extension from a smaller data type to a larger data type based on a sign bit**
- **In sign representation, the most significat bit is sign bit**

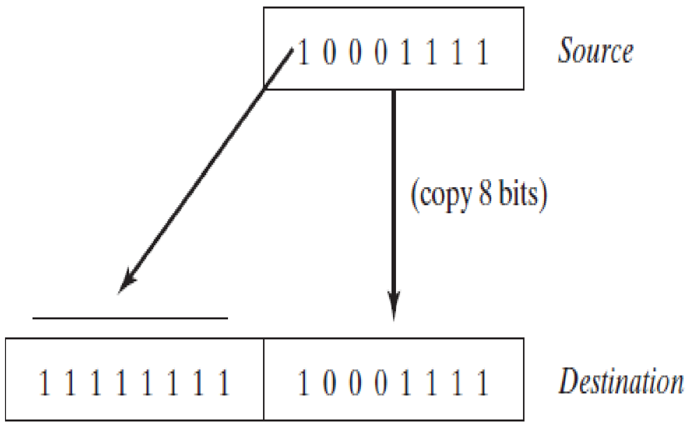


MOVSX instruction (move with sign-extend)

copies the contents of a source operand into a destination operand and sign-extends the value to 16 or 32 bits

- **MOVSX *reg16, reg/mem8***
- **MOVSX *reg32, reg/mem8***
- **MOVSX *reg32, reg/mem16***

Using MOVSX to copy a byte into a 16-bit destination.



- *Examples:*
  - byteVal db 10001111b
  - movsx ax, byte[byteVal] ; AX = 1111111110001111b
  - mov bx, 0**F6**FBh
  - movsx eax, bx ; EAX = FFFFF6FBh
  - movsx edx, bl ; EDX = FFFFFFFBh
  - movsx cx, bl ; CX = FFFBh

CBW

- The instruction does not have any explicitly specified operands because it is always converting AL → AX
- Converts the byte AL to the word AX in the signed interpretation (saves in AH the bit sign)
- The conversion refers to the extension of the representation from 8 bits to 16 bits, by filling AH with the sign bit of AL

Eg1:

mov AL, 01110111b

cbw ; AX ← 00000000 01110111b

Eg2:

mov AL, 11110111b

cbw ;   AX ← 11111111 11110111b

Eg3:

Mov bl, -1

Mov al, bl

Cbw   ; ax = -1

CWD

- The instruction does not have any explicitly specified operands because it is always converting AX → DX:AX
- Converts the word AX to the doubleword DX:AX in the signed interpretation
- The conversion refers to the extension of the representation from 16 bits to 32 bits, by filling DX with the sign bit of AX

Eg1.

mov ax, **0**0110011 11001100b

cwd ; DX:AX ← 00000000 00000000 00110011 11001100b

Eg2.

mov ax, 10110011 11001100b

cwd ; DX:AX ← 11111111 11111111 10110011 11001100b

CWDE

- The instruction does not have any explicitly specified operands because it is always converting AX → EAX
- Converts the word AX to the doubleword EAX in the signed interpretation
- The conversion refers to the extension of the representation from 16 bits to 32 bits, by filling the high word of EAX with the sign bit of AX

Eg1:

mov ax, 00110011 11001100b

cwde ; EAX ← 00000000 00000000 00110011 11001100b

Eg2:

mov ax, 10110011 11001100b

cwde ; EAX ← 11111111 11111111 10110011 11001100b

CDQ

- The instruction does not have any explicitly specified operands because it is always converting EAX → EDX:EAX
- Converts the doubleword EAX to the qword EDX:EAX in the signed interpretation
- The conversion refers to the extension of the representation from 32 bits to 64 bits, by filling EDX (the high doubleword) with the sign bit of EAX.

Eg1:

mov eax, **0**0110011 11001100 00110011 11001100b

cdq ; EDX:EAX ← 00000000 00000000 00000000 00000000 00110011 11001100 00110011 11001100b

Eg2:

mov eax, 10110011 11001100 10110011 11001100b

cdq   ;EDX:EAX ← 11111111 11111111 11111111 11111111 10110011 11001100 10110011 11001100b

Multiplication Instruction
(for signed representation)

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

Syntax: **IMUL op**

op is called explicit operand

The MUL is realized different accordind to the explicit operand:

op is *reg/mem8* => **IMUL *reg/mem8*** => *AL * reg/mem8 = AX*

op is *reg/mem16* => **IMUL *reg/mem16*** => *AX * reg/mem16 = DX:AX*

op is *reg/mem32* => **IMUL *reg/mem32*** => *EAX * reg/mem32 = EDX:EAX*

Division Instruction
(for signed representation)

Dividend	Divisor	Quotient	Remainder
AX	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX

Syntax: **IDIV op**

op is called explicit operand

The DIV is realized different according to the explicit operand:

op is reg/mem8 => **IDIV reg/mem8** => AX / reg/mem8 = AL – quotient
and AH – remainder

op is reg/mem16 => **IDIV reg/mem16** => DX:AX / reg/mem16 = AX – quotient
and DX – remainder

op is reg/mem32 => **IDIV reg/mem32** => EDX:EAX / reg/mem32 = EAX – quotient
and EDX – remainder

Comparisons Unsigned vs. Signed (1)

```
;unsigned
segment code use32 class=code
    start:
     ;ex1 unsigned
        mov al, 1
        mov bl, -1    ; bl=255 in unsigned
        mul bl        ; ax = 255 in unsigned
        ;----------------
     ;ex2 unsigned
        mov ax, 6
        mov cl, -2    ; cl = 254 in unsigned
        div cl     ; al = 0      , ah = 6
```

```
;signed
segment code use32 class=code
    start:
     ;ex1 signed
        mov al, 1
        mov bl, -1     ; bl = -1 in signed
        imul bl        ; ax = -1 in signed
        ;----------------
      ;ex2 signed
         mov ax, 6
         mov cl, -2    ; cl = -2 in signed
         idiv cl    ; al = -3      , ah = 0
```

Comparisons: Unsigned vs. Signed (2)

```
;unsigned
segment data use32 class=data
a db 5
b db 2
c dw 3
d dw 2
; our code starts here
segment code use32 class=code
    start:
        ; [(a+b - c)*3]/d
        mov al, [a]
        add al, [b]
        mov ah,0
        sub ax, [c]
        mov bx, 3
        mul bx   ; dx:ax = rez, dx=0000h=0, ax = 000ch = 12

        div word[d]  ; dx:ax/d = ax - quotient and dx-remainder
            ;ax = 0006h = 6
            ;dx = 0
```

```
;signed
segment data use32 class=data
a db 0FEh
b db 0FDh
c dw 0FFFBh
d dw 0FFFFFFFEh
; our code starts here
segment code use32 class=code
    start:
         ; [(a+b - c)*3]/d
        mov al, [a]
        add al, [b]
        cbw  ; movsx ax, al
        sub ax, [c]
        mov bx, 3
        imul bx   ; dx:ax rez deci dx = FFFFh = -1, ax = FFF4h = -12

        idiv word[d] ; dx:ax/d = ax - quotient and dx-remainder
            ; ax = 0006h = 6
            ; dx =0
```