Lab 3: *Registers, Operators & Instructions*	**Session:** Fall 2019
Instructor(s): Faheem Ahmad, Miss Sumaiyah	

Introduction to Registers

To speed up the processor operations, the processor includes some internal memory storage locations, called Registers. The registers store data elements for processing without having to access the memory.

There are ten 32-bit and six 16-bit processor registers in IA-32 architecture. The registers are grouped into three categories:

- General-Purpose registers,
- Control registers, and
- Segment registers

Furthermore, the general registers are further divided into the following groups:

- Data registers,
- Pointer registers &
- Index registers

Data Registers

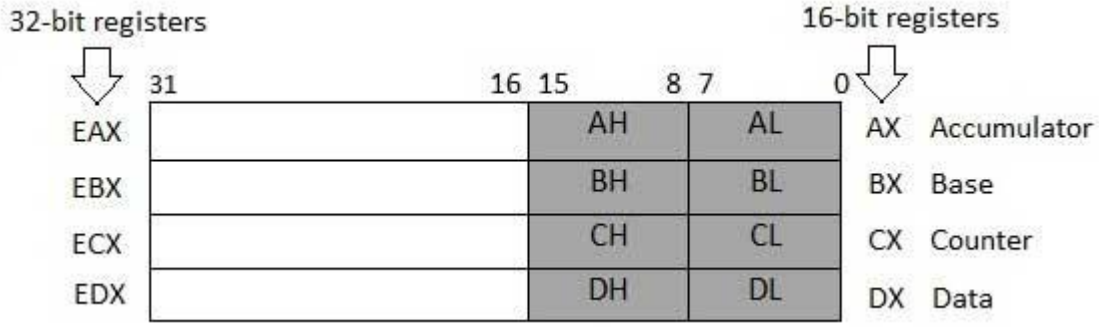
EAX (Accumulator register)
It is used in input/output and most arithmetic instructions. For example, in multiplication operation, one operand is stored in EAX or AX or AL register according to the size of the operand.

EBX (Base register)
It could be used in indexed addressing.

ECX (Counter register)
The ECX, CX registers store the loop count in iterative operations.

EDX (Data register)
It is also used in input/output operations. It is also used with AX register along with DX for multiply and division operations involving large values.

These four 32-bit registers are used for arithmetic, logical, and other operations.



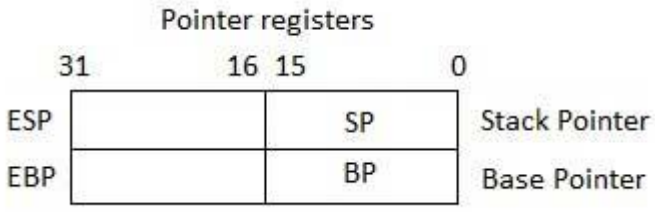
Pointer Registers

The pointer registers are 32-bit EIP, ESP, and EBP registers and their corresponding 16-bit portions IP, SP, and BP.

Extended Instruction Pointer (EIP)
The EIP register stores the offset address of the next instruction to be executed.

Extended Stack Pointer (ESP)
The ESP register provides the offset value within the program stack.

Extended Base Pointer (EBP)
The EBP register mainly helps in referencing the parameter variables passed to a subroutine.

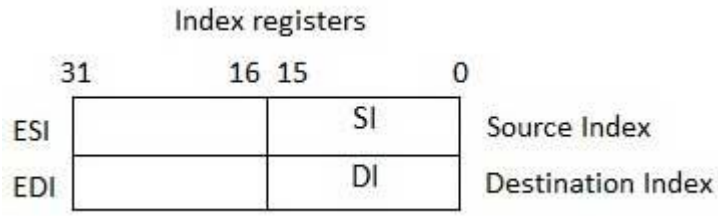


Index Registers

The 32-bit index registers, ESI and EDI, and their 16-bit rightmost portions. SI and DI, are used for indexed addressing and sometimes used in addition and subtraction.

Extended Source Index (ESI)
It is used as source index for string operations.

Extended Destination Index (EDI)
It is used as destination index for string operations.



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 *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* ✗

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.

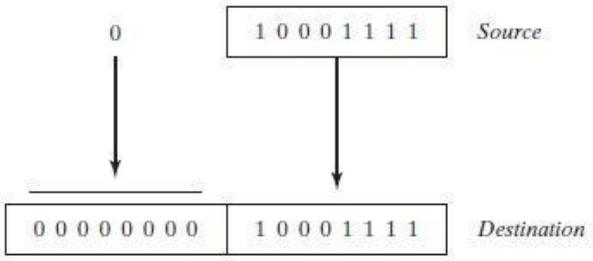
Example:
MOV *ax*, *5*
DEC *ax* *; ax now contains 4*

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*

Example:



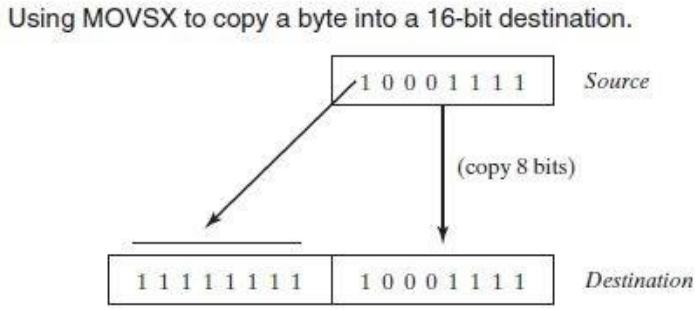
The following examples use registers for all operands, showing all the size variations:

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

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.

Example:



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.

Examples:

```
v1      BYTE  20    DUP(0)          ; 20 bytes, all equal to zero
v2      BYTE  20    DUP(?)          ; 20 bytes, uninitialized
v3      BYTE  4     DUP("STACK")    ;20 bytes, "STACKSTACKSTACKSTACK"
```

FLAGS Register

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 (2^8 - 1)
- 16 bits 0 to 65,535 (2^{16} - 1)
- 32 bits 0 to 4,294,967,295 (2^{32}-1)

S-Sign Flag:
This flag indicates the sign of the result of an operation. A 0 for positive number and 1 for a negative number.

AC-Auxilary 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 32-bit signed numbers:

- 8 bits (- 128 to +127)
- 16 bits (- 32,768 to +32,767 2^{15})
- 32 bits (-2,147,483,648 to +2,147,483,647 2^{31})

Sample Code:

```
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
```

The MUL/IMUL Instruction

There are two instructions for multiplying binary data. The MUL (Multiply) instruction handles unsigned data and the IMUL (Integer Multiply) handles signed data. Both instructions affect the Carry and Overflow flag.

Syntax

The syntax for the MUL/IMUL instructions is as follows –

```
MUL/IMUL multiplier
```

Multiplicand in both cases will be in an accumulator, depending upon the size of the multiplicand and the multiplier and the generated product is also stored in two registers depending upon the size of the operands. Following section explains MUL instructions with three different cases –

Sr.No.	Scenarios
1	**When two bytes are multiplied –** The multiplicand is in the AL register, and the multiplier is a byte in the memory or in another register. The product is in AX. High-order 8 bits of the product is stored in AH and the low-order 8 bits are stored in AL. 
2	**When two one-word values are multiplied –** The multiplicand should be in the AX register, and the multiplier is a word in memory or another register. For example, for an instruction like MUL DX, you must store the multiplier in DX and the multiplicand in AX. The resultant product is a doubleword, which will need two registers. The high-order (leftmost) portion gets stored in DX and the lower-order (rightmost) portion gets stored in AX. 
3	**When two doubleword values are multiplied –** When two doubleword values are multiplied, the multiplicand should be in EAX and the multiplier is a doubleword value stored in memory or in another register. The product generated is stored in the EDX:EAX registers, i.e., the high order 32 bits gets stored in the EDX register and the low order 32-bits are stored in the EAX register. 

The DIV/IDIV Instructions

The division operation generates two elements - a **quotient** and a **remainder**. In case of multiplication, overflow does not occur because double-length registers are used to keep the product. However, in case of division, overflow may occur. The processor generates an interrupt if overflow occurs.

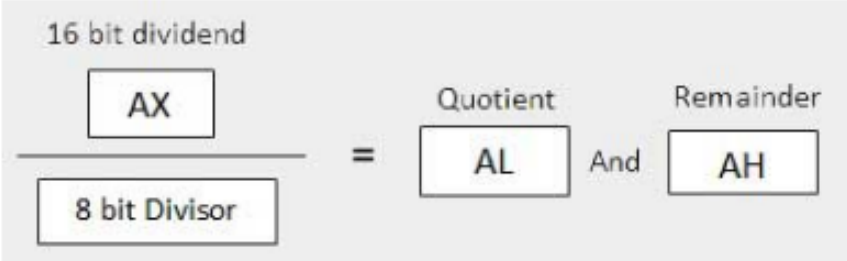
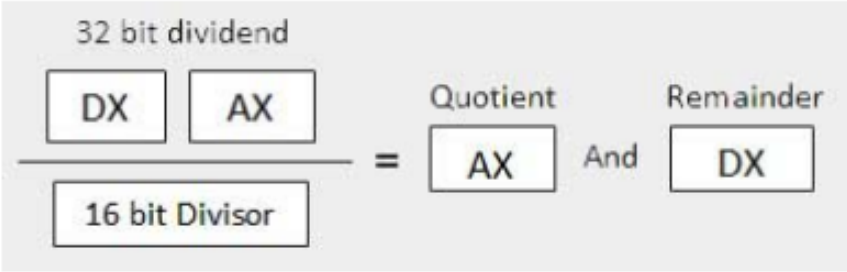
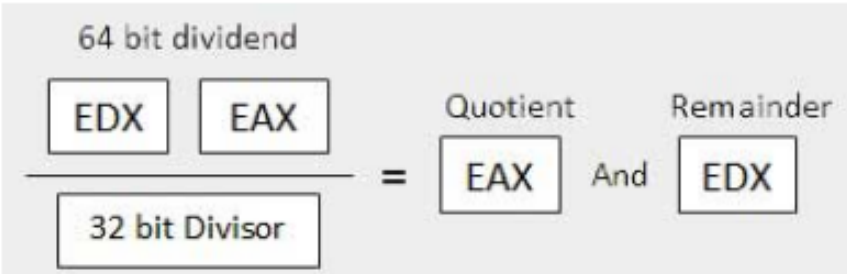
The DIV (Divide) instruction is used for unsigned data and the IDIV (Integer Divide) is used for signed data.

Syntax

The format for the DIV/IDIV instruction –

```
DIV/IDIV        divisor
```

The dividend is in an accumulator. Both the instructions can work with 8-bit, 16-bit or 32-bit operands. The operation affects all six status flags. Following section explains three cases of division with different operand size –

Sr.No.	Scenarios
1	**When the divisor is 1 byte –** The dividend is assumed to be in the AX register (16 bits). After division, the quotient goes to the AL register and the remainder goes to the AH register. 
2	**When the divisor is 1 word –** The dividend is assumed to be 32 bits long and in the DX:AX registers. The high-order 16 bits are in DX and the low-order 16 bits are in AX. After division, the 16-bit quotient goes to the AX register and the 16-bit remainder goes to the DX register. 
3	**When the divisor is doubleword –** The dividend is assumed to be 64 bits long and in the EDX:EAX registers. The high-order 32 bits are in EDX and the low-order 32 bits are in EAX. After division, the 32-bit quotient goes to the EAX register and the 32-bit remainder goes to the EDX register. 

Exercises:

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

2. 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

3. Write a program to find area of a square. Declare necessary variable side for the program (assign any arbitrary value to the variable).

4. Write a program to find area of a rectangle. Declare necessary variables length & width for the program (assign arbitrary values to the variables).

5. Write a program to find area of a triangle. Declare all necessary variables for the program (give arbitrary values to the variables).

6. 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.