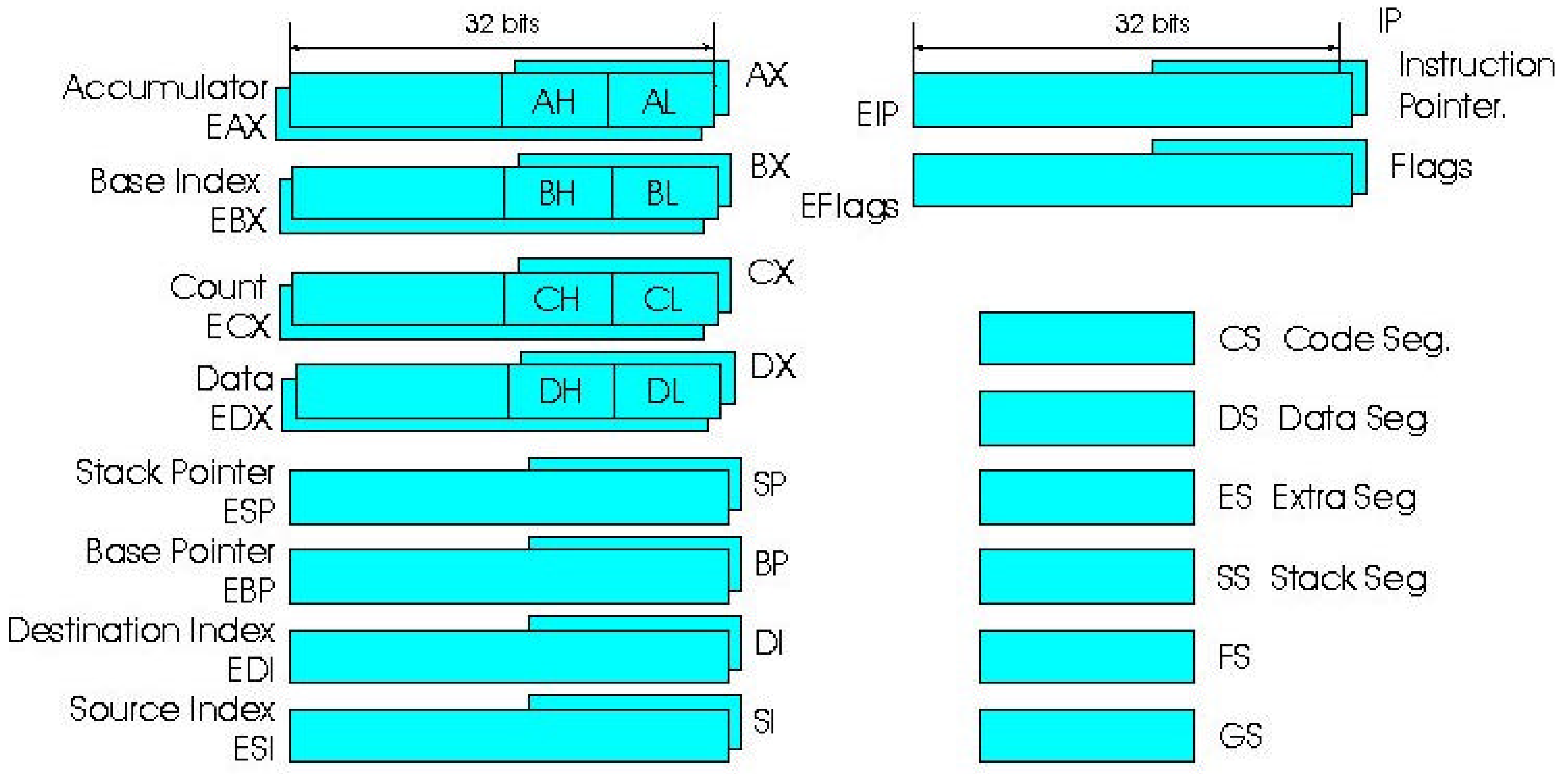
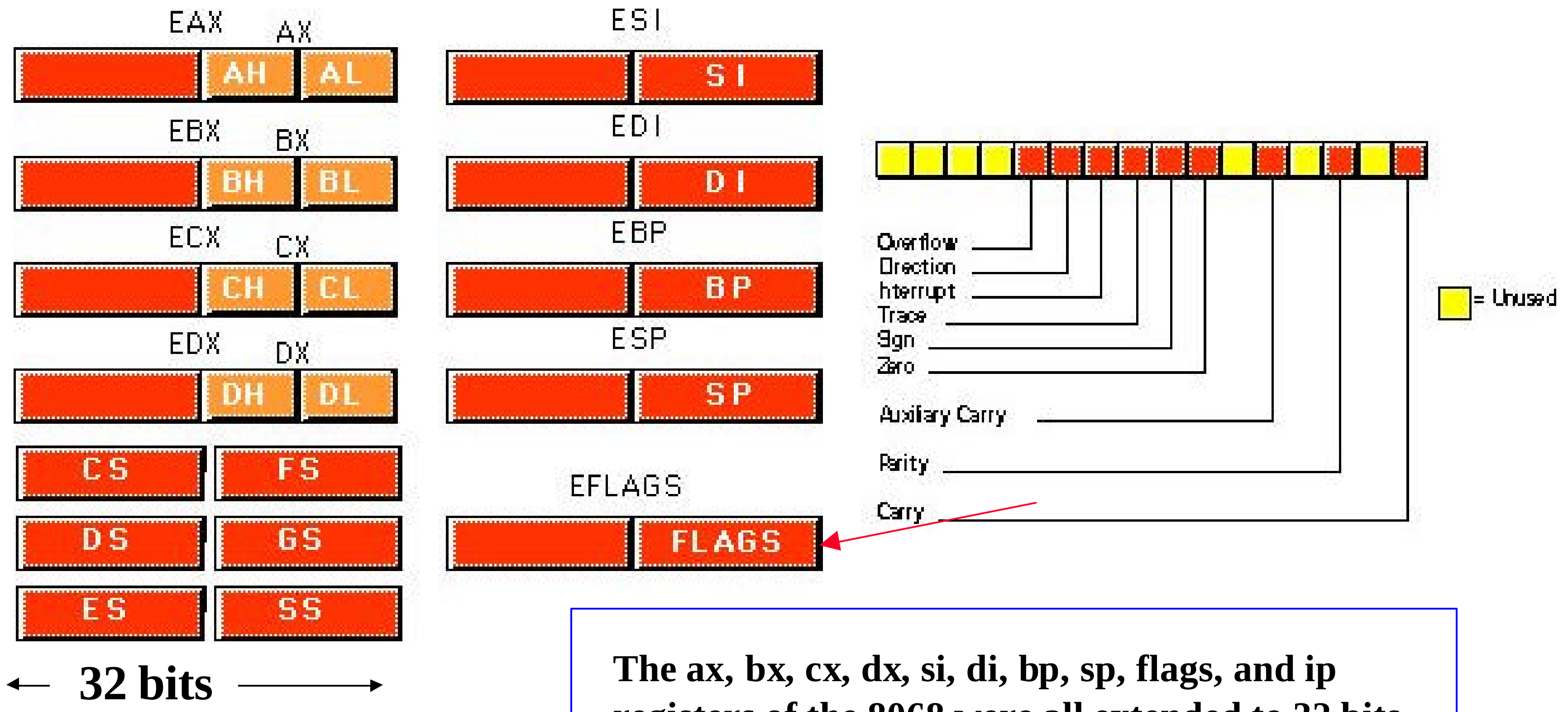
Intel 80x86 Register Organization



32-bit registers not present in 8086, 8088, or 80286

80386/80486/Pentium Registers



The ax, bx, cx, dx, si, di, bp, sp, flags, and ip registers of the 8068 were all extended to 32 bits.

The 80386 calls these new 32 bit versions EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EFLAGS, and EIP
Memory segments can be as large as 4GB

80x86 Flag Settings

ZF	Zero Flag	1:ZR:Zero 0:NZ:Non-zero	1 indicates that the result was zero
CF	Carry Flag	1:CY 0:NC	Unsigned math and shifting Needed a carry or borrow.
OF	Overflow Flag	1:OV 0:NV	
SF	Sign Flag	1:NG:- 0:PL:+	MSB of result

EECC250 - Shaaban

80x86 Register Usage

General Registers

AX (AH,AL): Accumulator, Math operations (General Purpose)

BX (BH,BL): Address/Pointer

CX(CH,CL): Counting & Looping

DX(DH,DL): Data, Most Significant bytes of a 16-bit MUL/DIV

Index Registers

SP: Stack pointer

BP: Base pointer Stack operations: parameters of a subroutine

SI: Source Index (arrays & strings)

DI: Destination Index (arrays & strings)

Segment Registers

CS: Code Segment: Used with Instruction Pointer (IP) to fetch instructions

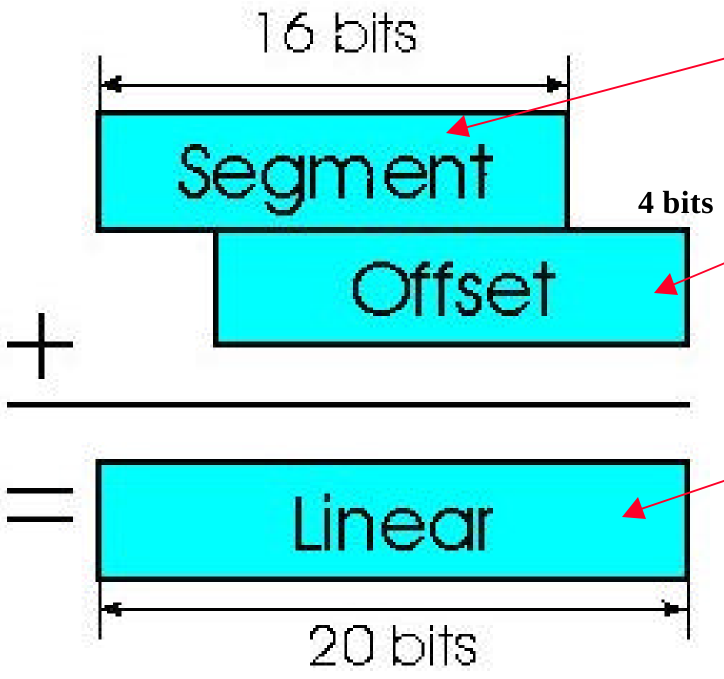
DS: Default (Data) Segment: Variables

SS: Stack Segment: Subroutines & local variables in procedures

ES: Extra Segment: String operations

80x86 Real Mode Memory Addressing

- Real Mode 20-bit memory addressing is used by default
- Memory addresses have the form: **Segment:Offset**



Both the segment and offset are 16-bit fields.

Segments provide a 2^{16} (64 KBytes) window into memory.

To generate a linear 20-bit address from a segment:offset address:

The segment is shifted left by 4 bits, and added to the offset field.

On the 80386 and later, the offset can be a 16-bit constant or a 32-bit constant.

Example
Linear address for Segment:Offset = 2222:3333 = 25553
Segment:offset address for Linear address = 25553:
Several possible combinations: 2222:3333 2000:5553 etc.

A Little 80x86 History

In 1976, when Intel began designing the 8086 processor, memory was very expensive. Personal computers at the time, typically had four thousand bytes of memory. Even when IBM introduced the PC five years later, 64K was still quite a bit of memory, one megabyte was a tremendous amount.

Intel's designers felt that 64K memory would remain a large amount throughout the lifetime of the 8086. The only mistake they made was completely underestimating the lifetime of the 8086. They figured it would last about five years, like their earlier 8080 processor.

They had plans for lots of other processors at the time, and "86" was not a suffix on the names of any of those. Intel figured they were set. Surely one megabyte would be more than enough to last until they came out with something better. Unfortunately, Intel didn't count on the IBM PC and the massive amount of software to appear for it. By 1983, it was very clear that Intel could not abandon the 80x86 architecture.

They were stuck with it, but by then people were running up against the one megabyte limit of 8086. So Intel gave us the 80286. This processor could address up to 16 megabytes of memory. Surely more than enough. The only problem was that all that wonderful software written for the IBM PC (MS DOS) was written in such a way that it couldn't take advantage of any memory beyond one megabyte.

80x86 Real Mode Memory Addressing

Utilization of segments:

SS:

Stack segment,

stack area

CS:

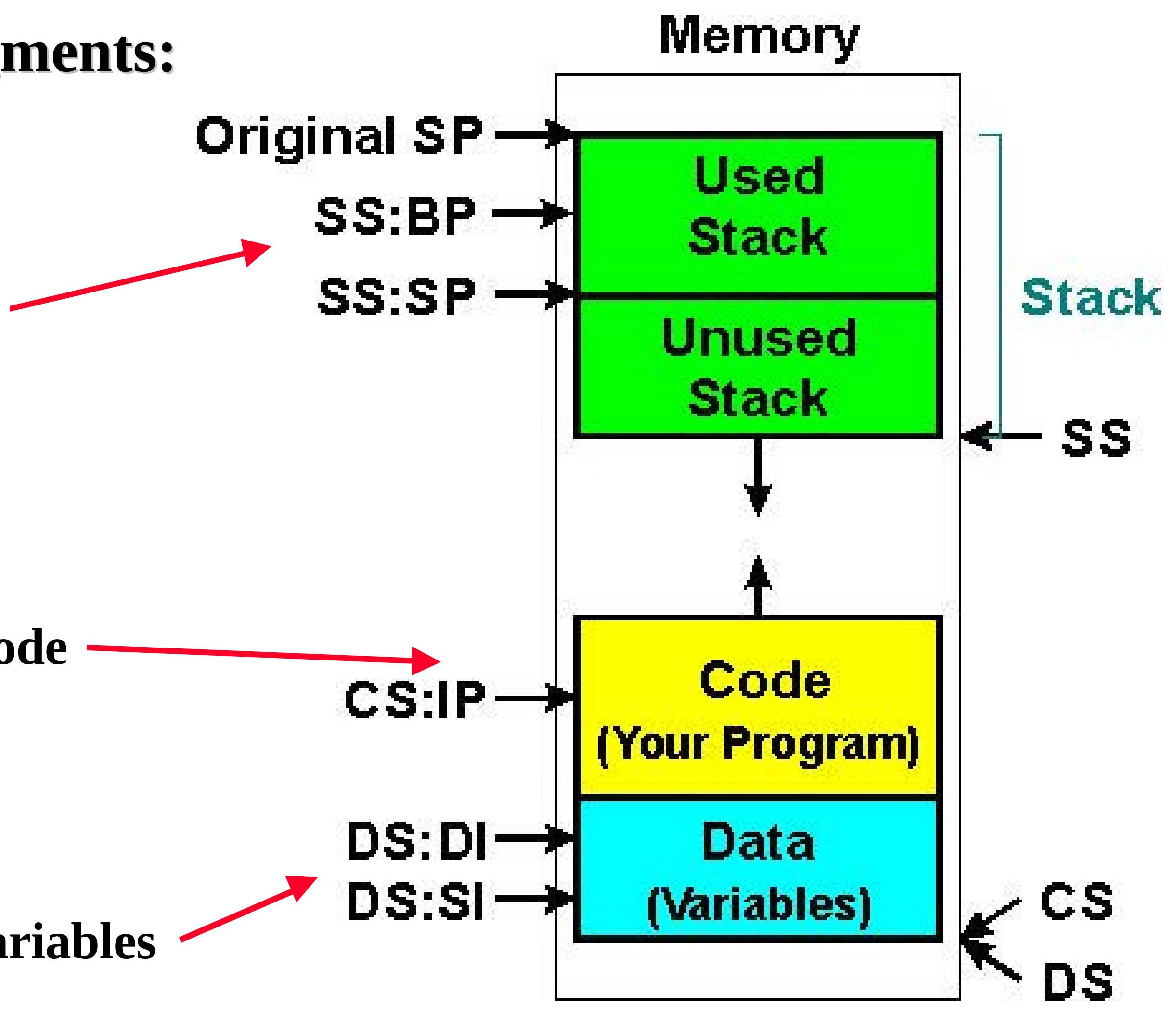
Code segment,

Program code

DS:

Default segment,

Data and variables



EECC250 - Shaaban

80x86 Instruction Properties

- Each instruction can only access memory once:

  MOV VAR1,VAR2 not valid

- For 2-operand instructions, size of operands must match:
  - Compare 8-bit number to 8-bit number
  - Compare 16-bit number to 16-bit number

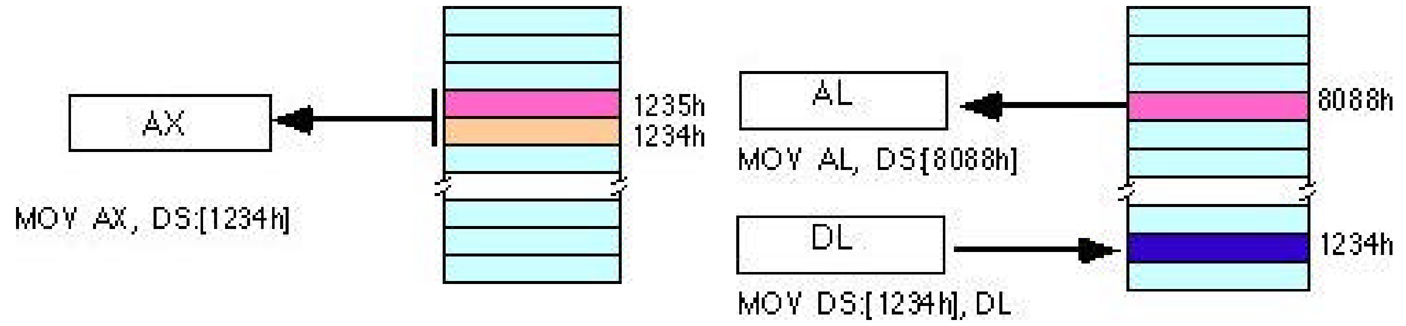
  CMP AH,AX not valid

- Destination operand (usually the first) must be a register or memory address:

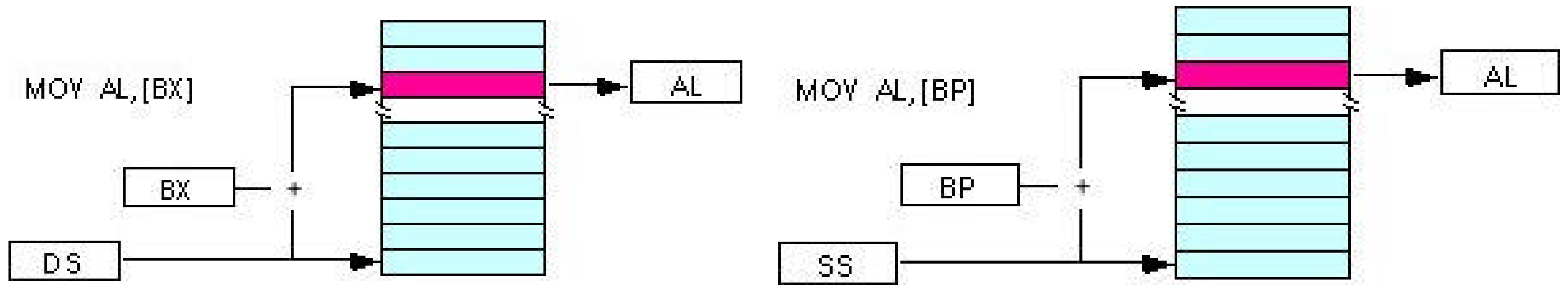
  MOV 1234,AX not valid

- Number storage in memory: LSB in low memory.

80x86 Addressing Modes



Direct (uses default segment, DS)

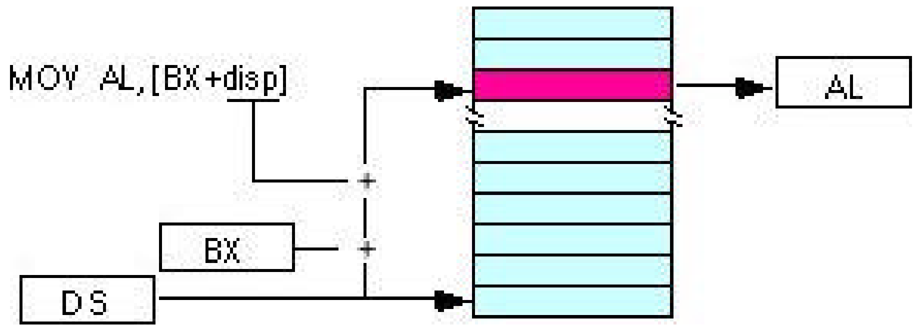


mov	**al, cs:[bx]**
mov	**al, ds:[bp]**
mov	**al, ss:[si]**
mov	**al, es:[di]**

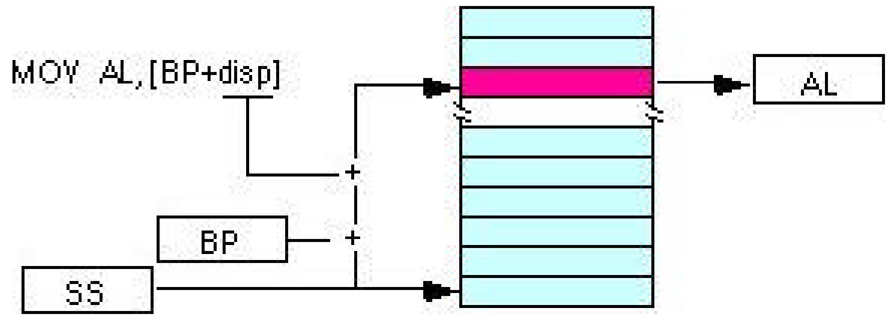
Register Indirect
(or Base Relative)

mov	**al, [bx]**
mov	**al, [bp]**
mov	**al, [si]**
mov	**al, [di]**

80x86 Addressing Modes



```
mov    al, disp[bx]
mov    al, disp[bp]
mov    al, disp[si]
mov    al, disp[di]
```

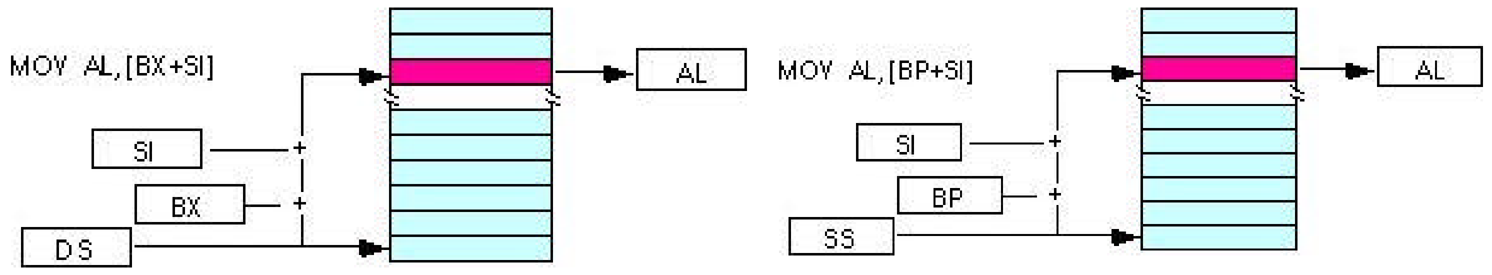


```
mov    al, ss:disp[bx]
mov    al, es:disp[bp]
mov    al, cs:disp[si]
mov    al, ss:disp[di]
```

Indexed
***or* Base-relative Direct**

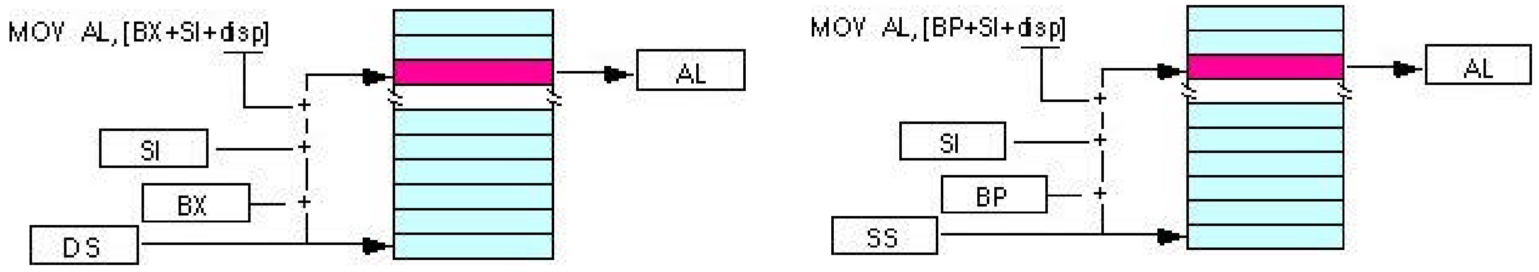
Note that Intel still refers to these addressing modes as based addressing and indexed addressing. Intel's literature does not differentiate between these modes with or without the constant. There is very little consensus on the use of these terms in the 80x86 world.

80x86 Addressing Modes



Based Indexed

```
mov    al, [bx][si]
mov    al, [bx][di]
mov    al, [bp][si]
mov    al, [bp][di]
```



Based Indexed Plus Displacement

```
mov    al, disp[bx][si]
mov    al, disp[bx+di]
mov    al, [bp+si+disp]
mov    al, [bp][di][disp]
```

Addressing Modes Samples

Register	MOV AX, BX	Move to AX the 16-bit value in BX
Register	MOV AX, DI	Move to AX the 16-bit value in DI
Register	MOV AH, AL	Move to AH the 8-bit value in AL
Immediate	MOV AH, 12H	Move to AH the byte value 12H
Immediate	MOV AX, 1234H	Move to AX the value 1234H
Immediate	MOV AX, CONST	Move to AX the constant defined as CONST
Immediate	MOV AX, OFFSET x	Move to AX the address (offset) of variable x
Direct	MOV AX, [1234H]	Move to AX value at memory location 1234H (uses default segment, DS)
Direct	MOV AX, x	Move to AX the value of M[x] (uses default segment, DS)
Direct	MOV x, AX	Move to M[x] the value of AX (uses default segment, DS)
Register Indirect	MOV AX, [DI]	Move to AX the value at M[DI] (uses default segment, DS)
Register indirect	MOV [DI], AX	Move to M[DI] the value AX (uses default segment, DS)

Addressing Modes Samples

Base-relative	MOV AX, [BX]	Move to AX the value M[BX] (uses default segment, DS)
Base-relative	MOV [BX], AX	Move to M[BX] the value AX (uses default segment, DS)
Base-relative	MOV AX, [BP]	Move to AX the value of M[BP] (uses stack segment, SS)
Base-relative	MOV [BP], AX	Move to M[BP] the value of AX (uses stack segment, SS)
Base-relative Direct (or indexed)	MOV AX, tab[BX]	Move to AX the value M[tab+BX] (uses default segment,DS)
Base-relative Direct (or indexed)	MOV tab[BX], AX	Move to M[tab+BX] the value A (uses default segment, DS)
Based Indexed	MOV AX, [BX+DI]	Move to AX the value M[BX+DI] (uses default segment, DS)
Based Indexed	MOV [BX+DI], AX	Move to M[BX+DI] the value AX (uses default segment, DS)
Based Indexed Plus Displacement	MOV AX, [BX+DI+1234H]	Move to AX the value pointed to by BX+DI+1234H (uses DS)

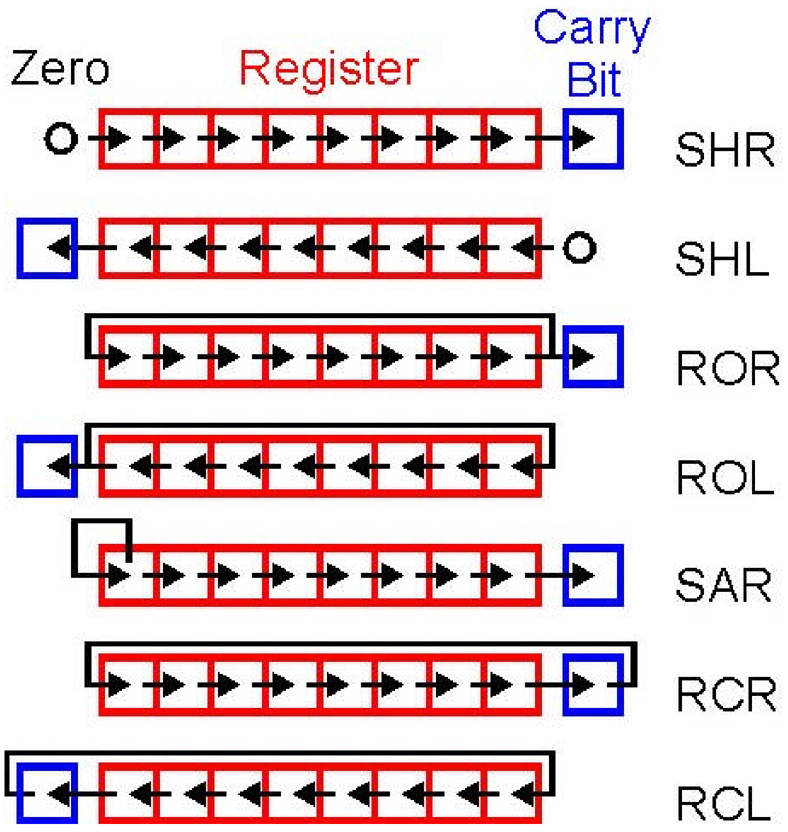
80x86 Instructions

- **Logic Operations:**

NOT OR XOR AND

TEST (Non-destructive AND)

- **Shifting/Rotate Operations:**



SHR : Shift Right (insert zero)

SHL == SAL : Shift Left (insert zero)

ROR : Rotate Right

ROL : Rotate Left

SAR : Shift Arithmetic Right (preserve high-bit)

RCR : use 9th/17th carry flag

RCL : use 9th/17th carry flag

Note: Carry flag (CF) always receives the shifted-out result
Note: Register CL can be used for shifts > 1 bit.

80x86 Logic & Rotates Example

```
mov ax,3                  ;   Initial register values
mov bx,5                  ;

or ax,9                   ;   ax <- ax | 00000010      (bitwise OR)
and ax,10101010b          ;   ax <- ax & 10101010      (bitwise AND)
xor ax,0FFh               ;   ax <- ax ^ 11111111      (bitwise XOR)
neg ax                    ;   ax <- (-ax)              (2's complement)
not ax                    ;   ax <- (~ax)              (bitise inversion)

shl ax,1                  ;   logical shift left by 1 bit
shr ax,1                  ;   logical shift right  by 1 bit
rol ax,1                  ;   rotate left (LSB=MSB)
ror ax,1                  ;   rotate right (MSB=LSB)

mov cl,3                  ;   Use CL to shift 3 bits
shr ax,cl                 ;   Divide AX by 8
shl bx,cl                 ;   Multiply BX by 8
```

80x86 Instructions

- Simple Math Operations:

  ADD : Add : A+=B
  SUB : Subtract : A-=B
  NEG : A=-A
  INC : A++ (Does not affect carry flag)
  DEC : A-- (Does not affect carry flag)

- Compare:

  CMP : Same as SUB, but does not write result
  (only sets flags) i.e., CMP A,B performs A-B
  eg: CMP 5,4 performs 5-4=1 : NZ PL NC NV

- Conditional jumps:

  JE,JL,JG,JNE,etc..

  Short jumps (signed, 8-bit displacement).

- Unconditional JUMP:

  JMP

  Allows long displacement

Stack & Procedure Instructions

- **PUSH - Place element on top of stack**
- **POP - Remove element from top of stack**

Example:

```
PUSH AX   ; Place AX on the stack
PUSH BX   ; Place BX on the stack
           ...
       modify contents of   Registers AX & BX
           …
POP BX    ; Restore original  value of BX
POP AX    ; Restore original  value of AX
```

- **CALL - Call a subroutine**
- **RET - Return from a subroutine**

Example:

```
PrintRec PROC NEAR
           ...
   Print value of a record
           …
RET
PrintRec ENDP
main PROC FAR
           ..
Calculate Scores
           …
CALL PrintRec
Continue Execution HERE
           …
CALL DOSXIT
main ENDP
```

68000 & *8086* Example

The following two programs implement the product of two vectors X, Y of size 100. This process is described as SUM Xi*Yi for i=1 to 100. Pseudocode of this calcualtion:

```
    sum <- 0
    for i<- 1 to 100 do
        sum <- sum + X(i) * Y(i)
    endfor
```

In 68000 assembly code

```
; both Xs and Ys  are16-bits and the result is 32-bit.
        MOVEQ.L  #99, D0      ; DO is the index i
        LEA      P, A0        ; Array X starts at P
        LEA      Q, A1        ; Array Y starts at Q
        CLR.L    D1           ;
LOOP:   MOVE     (A0)+, D2    ; get X(i)
        MULS     (A1)+, D2    ; get Y(i) and dose the multiply X(i)*Y(i)
        ADD.L    D2, D1       ; sum <- sum + X(i)*Y(i)
        DBF      D0, LOOP     ; test for zero; decrement and loop
```

68000 & *8086* Example

The same program coded in8086 assembly:

```
; this is 8-bit by 8-bit multiply only. TO do 16-bit by 16-bit the code
; will be much more complicated.
        MOV      AX, 2000H    ; Setup the data segment register
        MOV      DS, AX       ; load the data segment base
        MOV      CX, 100      ; index i
        LEA      BX, P        ; load X address
        LEA      SI, Q        ; Load Y address
        MOV      DX, 0000H    ; sum <- 0
LOOPA:  MOV      AL, [BX]     ; get X(i)
        IMUL     [SI]         ; X(i)*Y(i)
        ADD      DX, AX       ; sum <- sum + X(i)*Y(i)
        INC      BX           ; next X
        INC      SI           ; next Y
        LOOP     LOOPA
```

Data Movement Instructions

- These instructions include:
  - mov, xchg, lds, lea, les, lfs, lgs, lss, push, pusha, pushad, pushf, pushfd, pop, popa, popad, popf, popfd, lahf, and sahf.
- The MOV Instruction:

  The mov instruction takes several different forms:

```
mov    reg, reg
mov    mem, reg
mov    reg, mem
mov    mem, immediate data
mov    reg, immediate data
mov    ax/al, mem
mov    mem, ax/al
mov    segreg, mem16
mov    segreg, reg16
mov    mem16, segreg
mov    reg16, segreg
```

No memory to memory move operation

The XCHG Instruction

- The xchg (exchange) instruction swaps two values. The general form is:

```
xchg    operand1, operand2
```

- There are four specific forms of this instruction on the 80x86:

```
xchg    reg, mem
xchg    reg, reg
xchg    ax, reg16
xchg    eax, reg32     (Available only on 80386 and later processors)
```

The LEA Instruction

- **The lea (Load Effective Address) loads the specified 16 or 32 bit general purpose register with the effective address of the specified memory location. lea takes the form:**

```
lea     dest, source
```

- **The specific forms on the 80x86 are**

```
lea     reg16, mem
lea     reg32, mem   (Available only on 80386 and later processors)
```

Examples:

```
lea     ax, [bx]
lea     bx, 3[bx]
lea     ax, 3[bx]
lea     bx, 4[bp+si]
lea     ax, -123[di]
```

Conversion Instructions

- The 80x86 instruction set provides several conversion instructions. They include movzx, movsx, cbw, cwd, cwde, cdq, bswap, and xlat.
- Most of these instructions sign or zero extend values, the last two convert between storage formats and translate values via a lookup table. These instructions take the general form:
- movsx

  Sign extend an eight bit value to a sixteen or thirty-two bits, or sign extend a sixteen bit value to a thirty-two bits. This instruction uses a mod-reg-r/m byte to specify the two operands. The allowable forms for this instruction are

```
movsx   reg16, mem8
movsx   reg16, reg8
movsx   reg32, mem8
movsx   reg32, reg8
movsx   reg32, mem16
movsx   reg32, reg16
```

Arithmetic Instructions

- The 80x86 provides many arithmetic operations: addition, subtraction, negation, multiplication, division/modulo (remainder), and comparing two values. The instructions provided are:

  add, adc, sub, sbb, mul, imul, div, idiv, cmp, neg, inc, dec, xadd, cmpxchg, and some miscellaneous conversion instructions: aaa, aad, aam, aas, daa, and das

- The generic forms for these instructions are:

```
add     dest, src                   dest := dest + src
adc     dest, src                   dest := dest + src + C
SUB     dest, src                   dest := dest - src
sbb     dest, src                   dest := dest - src - C
mul     src                         acc := acc * src
imul    src                         acc := acc * src
imul    dest, src1, imm_src         dest := src1 * imm_src
imul    dest, imm_src               dest := dest * imm_src
imul    dest, src                   dest := dest * src
div     src                         acc := xacc /-mod src
idiv    src                         acc := xacc /-mod src
cmp     dest, src                   dest - src (and set flags)
neg     dest                        dest := - dest
inc     dest                        dest := dest + 1
dec     dest                        dest := dest - 1
```

The Multiplication Instructions: MUL, IMUL, AAM

The multiply instructions take the following forms:

Unsigned Multiplication:

mul	reg	
mul	mem	

Signed (Integer) Multiplication:

imul	reg	
imul	mem	
imul	reg, reg, immediate	80286 and later
imul	reg, mem, immediate	80286 and later
imul	reg, immediate	80286 and later
imul	reg, reg	80386 and later
imul	reg, mem	80286 and later

- **The mul instruction, with an eight bit operand, multiplies the al register by the operand and stores the 16 bit result in ax.**
- **If you specify a 16 bit operand, then mul and imul compute:**

  dx:ax := ax * operand16
- **If you specify a 32 bit operand, then mul and imul compute the following:**

  edx:eax := eax * operand32

The Division Instructions: DIV, IDIV, and AAD

- The 80x86 divide instructions perform a 64/32 division (80386 and later only), a 32/16 division or a 16/8 division. These instructions take the form:

```
div    reg    For unsigned division
div    mem
idiv   reg    For signed division
idiv   mem
aad           ASCII adjust for division
```

- The div instruction computes an unsigned division. If the operand is an eight bit operand, div divides the ax register by the operand leaving the quotient in al and the remainder (modulo) in ah. If the operand is a 16 bit quantity, then the div instruction divides the 32 bit quantity in dx:ax by the operand leaving the quotient in ax and the remainder in .
- With 32 bit operands (on the 80386 and later) div divides the 64 bit value in edx:eax by the operand leaving the quotient in eax and the remainder in edx.