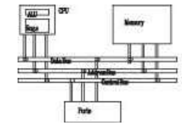
Intel x86 Instruction Set Architecture

Computer Organization and Assembly Languages
Yung-Yu Chuang

with slides by Kip Irvine

Data Transfers Instructions

MOV instruction

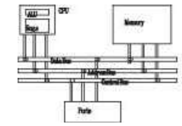


- Move from source to destination. Syntax:

  MOV *destination, source*

- Source and destination have the same size
- No more than one memory operand permitted
- CS, EIP, and IP cannot be the destination
- No immediate to segment moves

MOV instruction



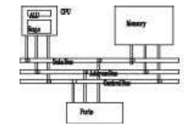
```
.data
count BYTE 100
wVal  WORD 2
.code
   mov bl,count
   mov ax,wVal
   mov count,al

   mov al,wVal            ; error
   mov ax,count           ; error
   mov eax,count          ; error
```

Exercise . . .

Explain why each of the following **MOV** statements are invalid:

```
.data
bVal   BYTE    100
bVal2  BYTE    ?
wVal   WORD    2
dVal   DWORD   5
.code
   mov ds,45            ; a.
   mov esi,wVal         ; b.
   mov eip,dVal         ; c.
   mov 25,bVal          ; d.
   mov bVal2,bVal       ; e.
```

Memory to memory



```
.data
var1 WORD ?
var2 WORD ?
.code
mov ax, var1
mov var2, ax
```

Copy smaller to larger

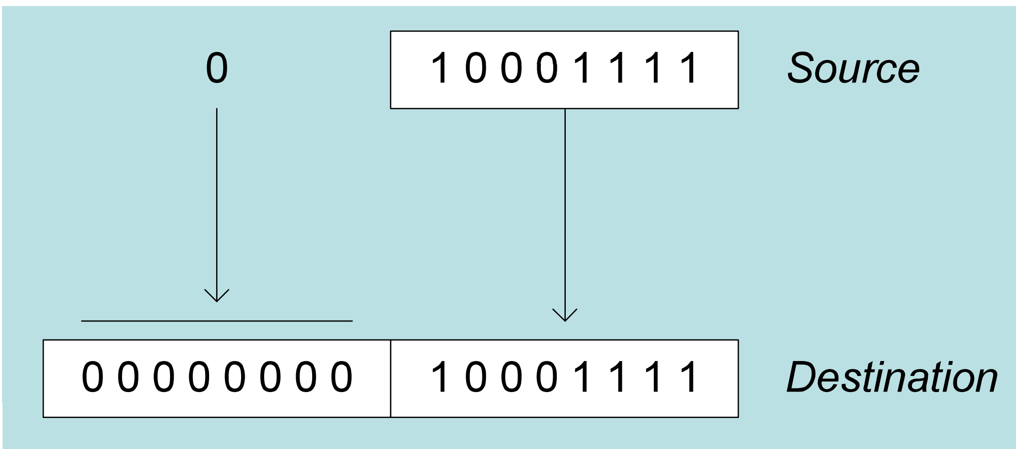
```
.data
count WORD 1
.code
mov ecx, 0
mov cx, count
```

```
.data
signedVal SWORD -16 ; FFF0h
.code
mov ecx, 0           ; mov ecx, 0FFFFFFFFh
mov cx, signedVal
```

MOVZX and **MOVSX** instructions take care of extension for both sign and unsigned integers.

Zero extension

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



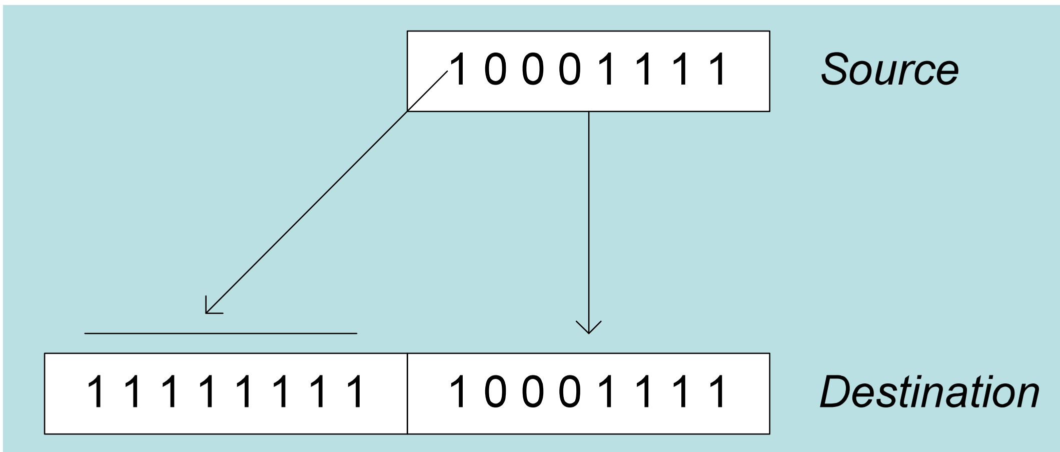

```
movzx r32,r/m8
movzx r32,r/m16
movzx r16,r/m8
```

```
mov bl,10001111b
movzx ax,bl            ; zero-extension
```

The destination must be a register.

Sign extension

The **MOVSX** instruction fills the upper half of the destination with a copy of the source operand's sign bit.




```
mov bl,10001111b
movsx ax,bl             ; sign extension
```

The destination must be a register.

MOVZX MOVSX

From a smaller location to a larger one

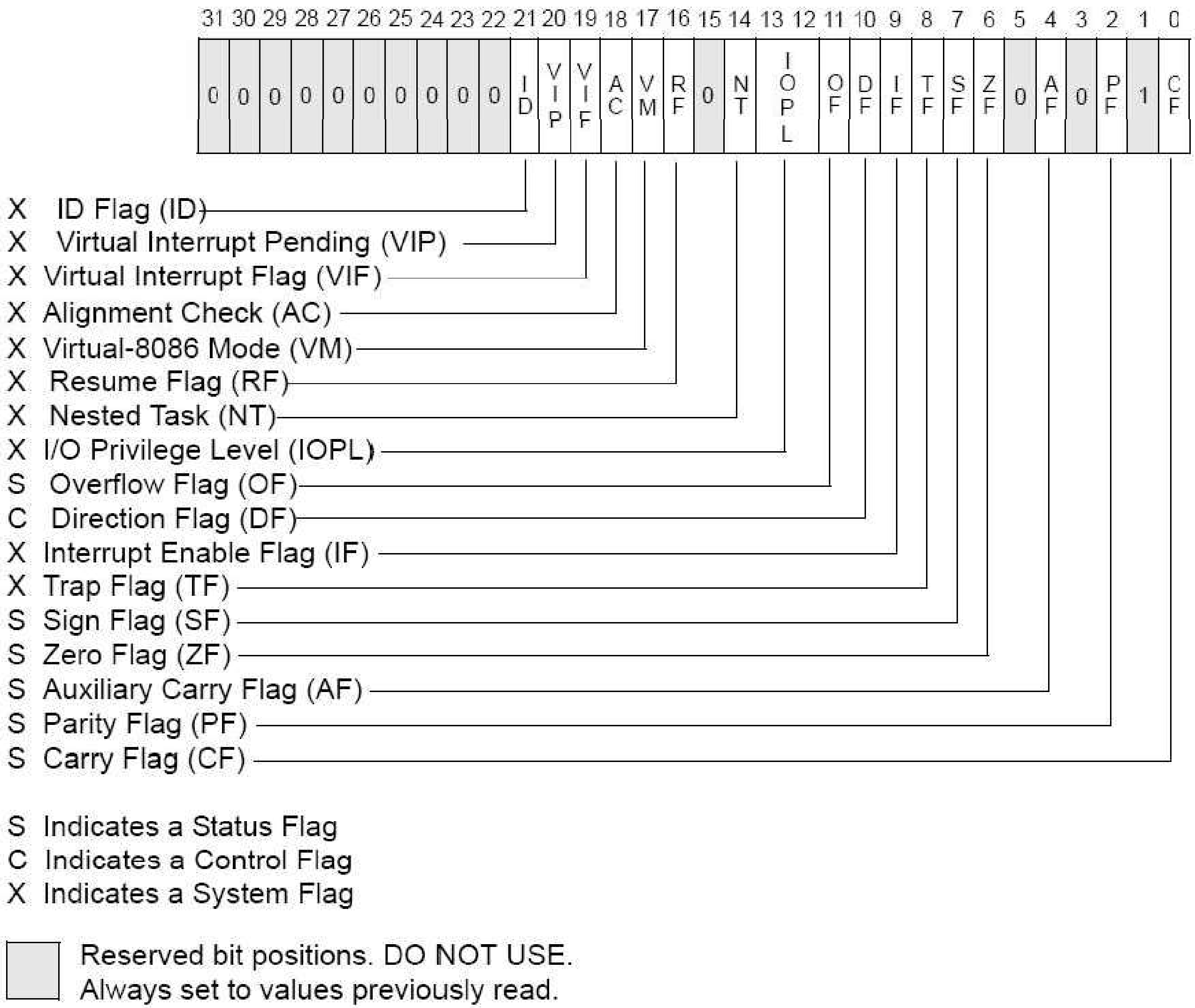
```
mov   bx,  0A69Bh
movzx eax, bx        ; EAX=0000A69Bh
movzx edx, bl        ; EDX=0000009Bh
movzx cx,  bl        ; EAX=009Bh

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

LAHF/SAHF (load/store status flag from/to AH)

```
.data
saveflags BYTE ?
.code
lahf
mov  saveflags, ah
...
mov  ah, saveflags
sahf
```

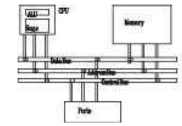
S,Z,A,P,C flags are copied.

EFLAGS



31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE.
Always set to values previously read.

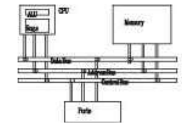
XCHG Instruction



XCHG exchanges the values of two operands. At least one operand must be a register. No immediate operands are permitted.

```
.data
var1 WORD 1000h
var2 WORD 2000h
.code
xchg ax,bx          ; exchange 16-bit regs
xchg ah,al          ; exchange 8-bit regs
xchg var1,bx        ; exchange mem, reg
xchg eax,ebx        ; exchange 32-bit regs

xchg var1,var2      ; error 2 memory operands
```

Exchange two memory locations



```
.data
var1 WORD 1000h
var2 WORD 2000h
.code
mov  ax, val1
xchg ax, val2
mov  val1, ax
```

Arithmetic Instructions

Addition and Subtraction

- **`INC`** and **`DEC`** Instructions
- **`ADD`** and **`SUB`** Instructions
- **`NEG`** Instruction
- Implementing Arithmetic Expressions
- Flags Affected by Arithmetic
  - Zero
  - Sign
  - Carry
  - Overflow

INC and **DEC** Instructions

- Add 1, subtract 1 from destination operand
  - operand may be register or memory
- **INC** *destination*
  - Logic: *destination* $\leftarrow$ *destination* + 1
- **DEC** *destination*
  - Logic: *destination* $\leftarrow$ *destination* – 1

INC and DEC Examples

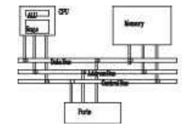
```
.data
myWord  WORD 1000h
myDword DWORD 10000000h
.code
   inc myWord            ; 1001h
   dec myWord            ; 1000h
   inc myDword           ; 10000001h

   mov ax,00FFh
   inc ax                ; AX = 0100h
   mov ax,00FFh
   inc al                ; AX = 0000h
```

Exercise...

Show the value of the destination operand after each of the following instructions executes:

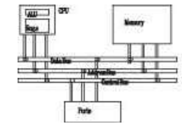
```
.data
myByte BYTE 0FFh, 0
.code
   mov al,myByte       ; AL = FFh
   mov ah,[myByte+1] ; AH = 00h
   dec ah              ; AH = FFh
   inc al              ; AL = 00h
   dec ax              ; AX = FEFF
```

ADD and **SUB** Instructions



- **`ADD`** *destination, source*
  - Logic: *destination* $\leftarrow$ *destination* + source
- **`SUB`** *destination, source*
  - Logic: *destination* $\leftarrow$ *destination* – source
- Same operand rules as for the **`MOV`** instruction

ADD and SUB Examples



```
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code                        ; ---EAX---
  mov eax,var1               ; 00010000h
  add eax,var2               ; 00030000h
  add ax,0FFFFh              ; 0003FFFFh
  add eax,1                  ; 00040000h
  sub ax,1                   ; 0004FFFFh
```

NEG (negate) Instruction

Reverses the sign of an operand. Operand can be a register or memory operand.

```
.data
valB BYTE -1
valW WORD +32767
.code
   mov al,valB          ; AL = -1
   neg al               ; AL = +1
   neg valW             ; valW = -32767
```

Implementing Arithmetic Expressions

HLL compilers translate mathematical expressions into assembly language. You can do it also. For example:

Rval = -Xval + (Yval – Zval)

```
Rval DWORD ?
Xval DWORD 26
Yval DWORD 30
Zval DWORD 40
.code
   mov eax,Xval
   neg eax               ; EAX = -26
   mov ebx,Yval
   sub ebx,Zval          ; EBX = -10
   add eax,ebx
   mov Rval,eax          ; -36
```

Exercise ...

Translate the following expression into assembly language. Do not permit Xval, Yval, or Zval to be modified:

```
Rval = Xval - (-Yval + Zval)
```

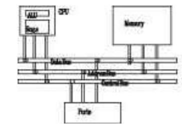
Assume that all values are signed doublewords.

```
mov ebx,Yval
neg ebx
add ebx,Zval
mov eax,Xval
sub eax,ebx
mov Rval,eax
```

Flags Affected by Arithmetic

- The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations
  - based on the contents of the destination operand
- Essential flags:
  - Zero flag – destination equals zero
  - Sign flag – destination is negative
  - Carry flag – unsigned value out of range
  - Overflow flag – signed value out of range
- The **`MOV`** instruction never affects the flags.

Zero Flag (ZF)



Whenever the destination operand equals Zero, the Zero flag is set.

```
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
```

A flag is set when it equals 1.

A flag is clear when it equals 0.

Sign Flag (SF)

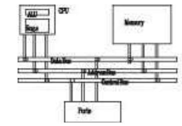
The Sign flag is set when the destination operand is negative. The flag is clear when the destination is positive.

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

The sign flag is a copy of the destination's highest bit:

```
mov al,0
sub al,1                    ; AL=11111111b, SF=1
add al,2                    ; AL=00000001b, SF=0
```

Carry Flag (CF)



- Addition and CF: copy carry out of MSB to CF
- Subtraction and CF: copy inverted carry out of MSB to CF
- **`INC/DEC`** do not affect CF
- Applying **`NEG`** to a nonzero operand sets CF

Exercise . . .

For each of the following marked entries, show the values of the destination operand and the Sign, Zero, and Carry flags:

```
mov ax,00FFh
add ax,1          ; AX= 0100h  SF= 0 ZF= 0 CF= 0
sub ax,1          ; AX= 00FFh  SF= 0 ZF= 0 CF= 0
add al,1          ; AL= 00h    SF= 0 ZF= 1 CF= 1
mov bh,6Ch
add bh,95h        ; BH= 01h    SF= 0 ZF= 0 CF= 1

mov al,2
sub al,3          ; AL= FFh    SF= 1 ZF= 0 CF= 1
```

Overflow Flag (OF)

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

```
; Example 1
mov al,+127
add al,1                ; OF = 1,   AL = ??

; Example 2
mov al,7Fh              ; OF = 1,   AL = 80h
add al,1
```

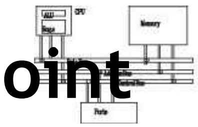
The two examples are identical at the binary level because 7Fh equals +127. To determine the value of the destination operand, it is often easier to calculate in hexadecimal.

A Rule of Thumb

- When adding two integers, remember that the Overflow flag is only set when . . .
  - Two positive operands are added and their sum is negative
  - Two negative operands are added and their sum is positive

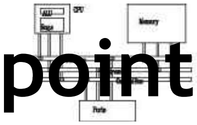
```
What will be the values of OF flag?
   mov al,80h
   add al,92h          ; OF =

   mov al,-2
   add al,+127         ; OF =
```

Signed/Unsigned Integers: Hardware Viewpoint



- All CPU instructions operate exactly the same on signed and unsigned integers
- The CPU cannot distinguish between signed and unsigned integers
- YOU, the programmer, are solely responsible for using the correct data type with each instruction

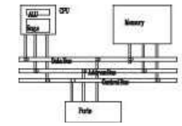
Overflow/Carry Flags: Hardware Viewpoint



- How the **`ADD`** instruction modifies OF and CF:
  - CF = (carry out of the MSB)
  - OF = (carry out of the MSB) XOR (carry into the MSB)

- How the **`SUB`** instruction modifies OF and CF:
  - NEG the source and ADD it to the destination
  - CF = INVERT (carry out of the MSB)
  - OF = (carry out of the MSB) XOR (carry into the MSB)

Auxiliary Carry (AC) flag



- AC indicates a carry or borrow of bit 3 in the destination operand.
- It is primarily used in binary coded decimal (BCD) arithmetic.

```
mov al, oFh
add al, 1        ; AC = 1
```

Parity (PF) flag

- PF is set when LSB of the destination has an even number of 1 bits.

```
mov al, 10001100b
add al, 00000010b; AL=10001110, PF=1
sub al, 10000000b; AL=00001110, PF=0
```

Jump and Loop

`JMP` and `LOOP` Instructions

- Transfer of control or branch instructions
  - unconditional
  - conditional
- `JMP` Instruction
- `LOOP` Instruction
- `LOOP` Example
- Summing an Integer Array
- Copying a String

JMP Instruction

- **JMP** is an unconditional jump to a label that is usually within the same procedure.
- Syntax: **JMP** *target*
- Logic: EIP $\leftarrow$ *target*
- Example:

```
top:
   .
   .
   jmp top
```

LOOP Instruction

- The **LOOP** instruction creates a counting loop
- Syntax: **LOOP** *target*
- Logic:
  - ECX ← ECX – 1
  - if ECX != 0, jump to *target*
- Implementation:
  - The assembler calculates the distance, in bytes, between the current location and the offset of the target label. It is called the relative offset.
  - The relative offset is added to EIP.

`LOOP` Example

The following loop calculates the sum of the integers 5 + 4 + 3 +2 + 1:

```
offset     machine code     source code
00000000   66 B8 0000          mov   ax,0
00000004   B9 00000005         mov   ecx,5

00000009   66 03 C1         L1:add   ax,cx
0000000C   E2 FB               loop L1
0000000E
```

When **`LOOP`** is assembled, the current location = 0000000E. Looking at the **`LOOP`** machine code, we see that –5 (FBh) is added to the current location, causing a jump to location 00000009:

$$00000009 \leftarrow 0000000E + FB$$

Exercise . . .

If the relative offset is encoded in a single byte,

(a) what is the largest possible backward jump?

(b) what is the largest possible forward jump?

(a) -128
(b) $+127$

Average sizes of machine instructions are about 3 bytes, so a loop might contain, on average, a maximum of 42 instructions!

Exercise . . .

What will be the final value of AX?

10

```
   mov ax,6
   mov ecx,4
L1:
   inc ax
   loop L1
```

How many times will the loop execute?

4,294,967,296

```
   mov ecx,0
X2:
   inc ax
   loop X2
```

Nested Loop

If you need to code a loop within a loop, you must save the outer loop counter's ECX value. In the following example, the outer loop executes 100 times, and the inner loop 20 times.

```
.data
count DWORD ?
.code
   mov ecx,100          ; set outer loop count
L1:
   mov count,ecx        ; save outer loop count
   mov ecx,20           ; set inner loop count
L2:...
   loop L2              ; repeat the inner loop
   mov ecx,count        ; restore outer loop count
   loop L1              ; repeat the outer loop
```

Summing an Integer Array

The following code calculates the sum of an array of 16-bit integers.

```
.data
intarray WORD 100h,200h,300h,400h
.code
   mov edi,OFFSET intarray     ; address
   mov ecx,LENGTHOF intarray   ; loop counter
   mov ax,0                    ; zero the sum
L1:
   add ax,[edi]                ; add an integer
   add edi,TYPE intarray       ; point to next
   loop L1                ; repeat until ECX = 0
```

Copying a String

good use of SIZEOF

The following code copies a string from source to target.

```
.data
source  BYTE  "This is the source string",0
target  BYTE  SIZEOF source DUP(0),0

.code
   mov  esi,0              ; index register
   mov  ecx,SIZEOF source   ; loop counter
L1:
   mov  al,source[esi]    ; get char from source
   mov  target[esi],al    ; store in the target
   inc  esi               ; move to next char
   loop L1              ; repeat for entire string
```

Conditional Processing

Status flags - review

- The Zero flag is set when the result of an operation equals zero.
- The Carry flag is set when an instruction generates a result that is too large (or too small) for the destination operand.
- The Sign flag is set if the destination operand is negative, and it is clear if the destination operand is positive.
- The Overflow flag is set when an instruction generates an invalid signed result.
- Less important:
  - The Parity flag is set when an instruction generates an even number of 1 bits in the low byte of the destination operand.
  - The Auxiliary Carry flag is set when an operation produces a carry out from bit 3 to bit 4