Computer Organization and Assembly Language

Lecture 06:

Data Transfer, Add & SUB (Flags)

Lecture Outline

- Data Transfer Instructions
  - Operand Types
  - Instruction Operand Notation
  - Direct Memory Operands
  - MOV Instruction
  - Zero & Sign Extension
  - XCHG Instruction
  - Direct-Offset Instructions
- Addition and Subtraction
  - Which Flags are affected?

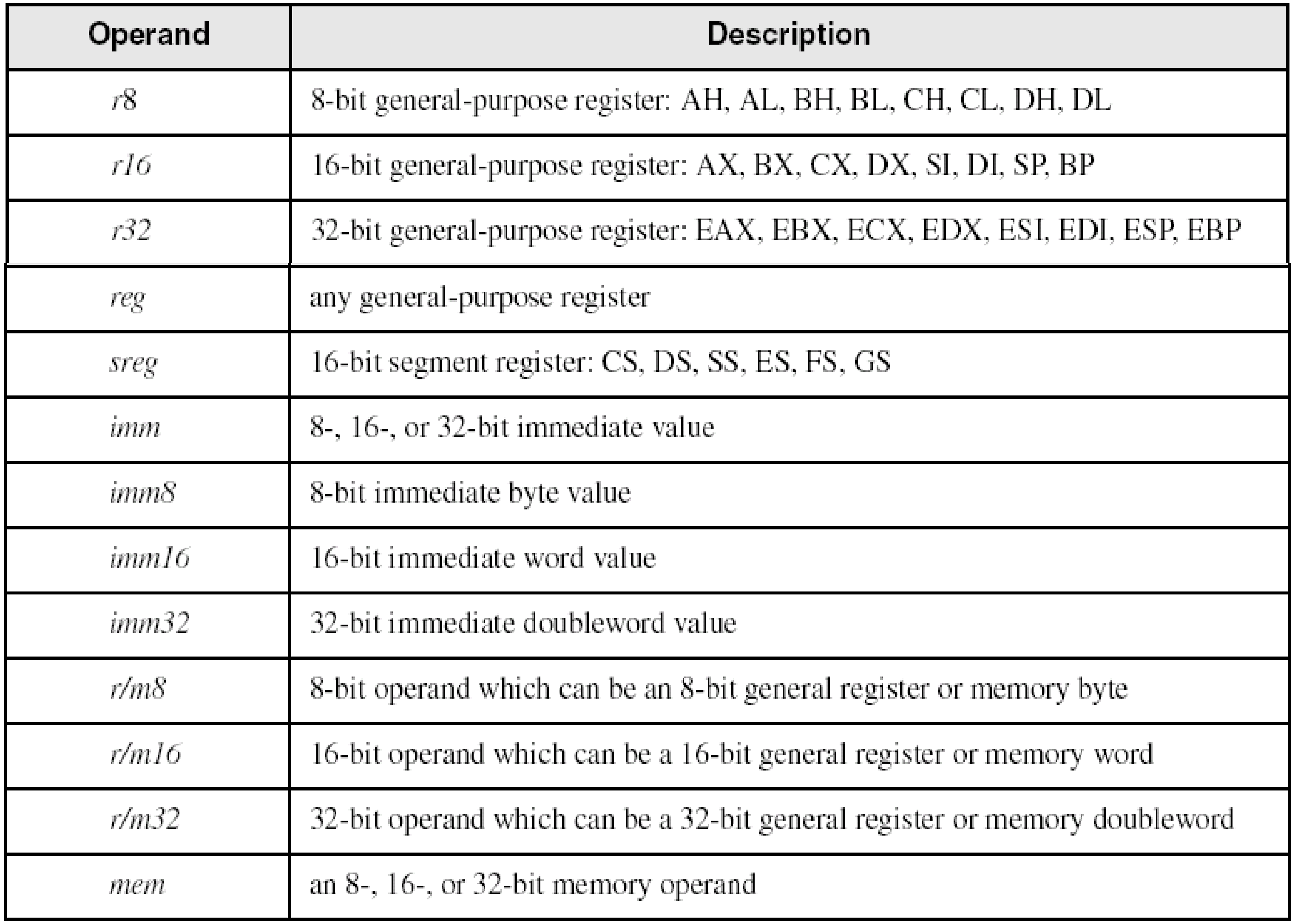
Data Transfer Instructions

- Operand Types
- Instruction Operand Notation
- Direct Memory Operands
- MOV Instruction
- Zero & Sign Extension
- XCHG Instruction
- Direct-Offset Instructions

Operand Types

- Three basic types of operands:
  - Immediate – a constant integer (8, 16, or 32 bits)
    - value is encoded within the instruction
  - Register – the name of a register
    - register name is converted to a number and encoded within the instruction
  - Memory – reference to a location in memory
    - memory address is encoded within the instruction, or a register holds the address of a memory location

Instruction Operand Notation



Operand	Description
r8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
r16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
r32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
r/m8	8-bit operand which can be an 8-bit general register or memory byte
r/m16	16-bit operand which can be a 16-bit general register or memory word
r/m32	32-bit operand which can be a 32-bit general register or memory doubleword
mem	an 8-, 16-, or 32-bit memory operand

Direct Memory Operands

- A direct memory operand is a named reference to storage in memory
- The named reference (label) is automatically dereferenced by the assembler

```
.data
var1 BYTE 10h
.code
mov al,var1                        ; AL = 10h
mov al,[var1]                      ; AL = 10h
```

alternate format

MOV Instruction

- Move from source to destination. Syntax:

  MOV *destination,source*
- No more than one memory operand permitted
- CS, EIP, and IP cannot be the destination
- No immediate to segment moves

```
.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
```

Drill . . .

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        immediate move to DS not permitted
   mov esi,wVal     size mismatch
   mov eip,dVal     EIP cannot be the destination
   mov 25,bVal      immediate value cannot be destination
   mov bVal2,bVal   memory-to-memory move not permitted
```

1. `mov ds, 45`

   ✗ **Invalid because**: You cannot **move an immediate value directly into a segment register** (DS, ES, etc.) like this. You must first move it into a general-purpose register (e.g., AX) and then into the segment register:

   ✓ Correct form (if allowed in your mode):

```
mov ax, 45
mov ds, ax
```

---

2. `mov esi, wVal`

   ✗ **Possibly invalid** depending on `wVal`'s type. If `wVal` is a **word (16-bit)** variable, and `ESI` is a **32-bit register**, this causes a **size mismatch**.

   ✓ Fix: Either change `wVal` to be 32-bit (e.g., `dword wVal`) or use a 16-bit register like `SI`.

   Example fixes:

```
mov esi, dword ptr [wVal]     ; if wVal is 32-bit
mov si, word ptr [wVal]       ; if wVal is 16-bit
```

---

3. `mov eip, dVal`

❌ **Invalid because**: You cannot **directly move a value into** `EIP` (instruction pointer). The only way to change `EIP` is through **jumps, calls, or returns**.

✅ Alternatives:

```
jmp dVal              ; if dVal is a label or memory address
call dVal             ; to branch to a subroutine
```

---

4. `mov 25, bVal`

❌ **Invalid because**: `25` is an **immediate constant and cannot be the destination** of a `MOV` instruction. Only registers or memory can be destinations.

✅ If you meant to move `bVal` into memory address `25`, you'd need:

```
mov byte ptr [25], bVal
```

---

5. `mov bVal2, bVal`

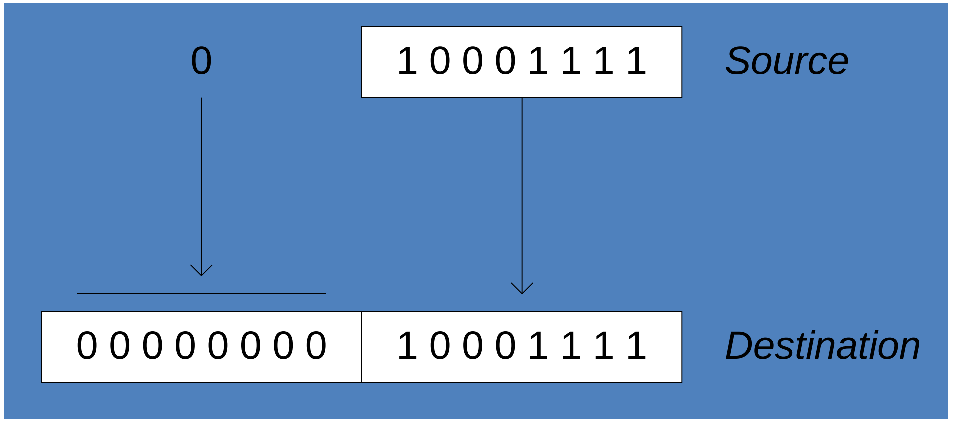
✗ **Invalid if both are variables** (i.e., memory-to-memory). x86 does **not allow memory-to-memory transfers** with MOV.

✓ Fix: Use a register as an intermediate:

```
mov al, bVal        ; assuming bVal is byte
mov bVal2, al
```

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.

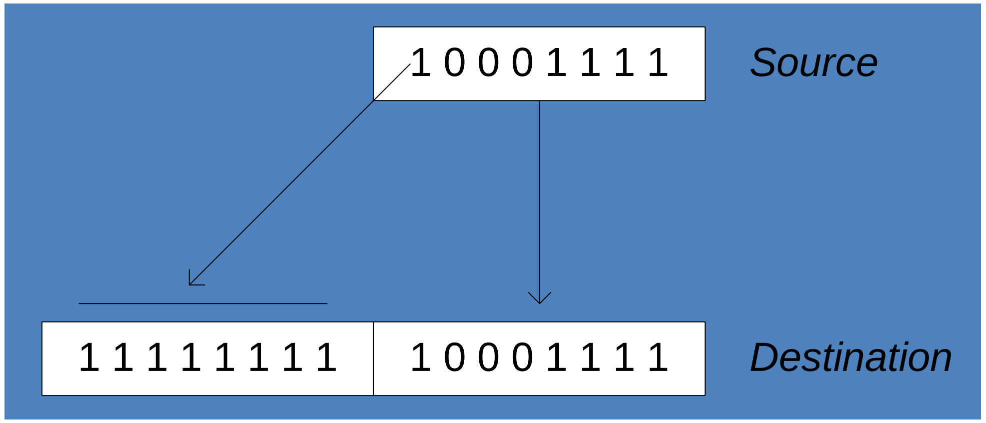


```
mov bl,10001111b
movzx ax,bl                         ; zero-extension
```

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.

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: two memory operands
```

Direct-Offset Operands

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.

```
.data
arrayB BYTE 10h,20h,30h,40h
.code
mov al,arrayB+1                 ; AL = 20h
mov al,[arrayB+1]               ; alternative notation
```

Q: Why doesn't arrayB+1 produce 11h?

Direct-Offset Operands (cont)

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.

```
.data
arrayW  WORD 1000h,2000h,3000h
arrayD  DWORD 1,2,3,4
.code
mov ax,[arrayW+2]              ; AX = 2000h
mov ax,[arrayW+4]              ; AX = 3000h
mov eax,[arrayD+4]             ; EAX = 00000002h
```

```
; Will the following statements assemble?
mov ax,[arrayW-2]          ; ??
mov eax,[arrayD+16]        ; ??
```

What will happen when they run?

Drill . . .

Write a program that rearranges the values of three doubleword values in the following array as: 3, 1, 2.

```
.data
arrayD DWORD 1,2,3
```

- Step1: copy the first value into AX and exchange it with the value in the second position.

```
mov eax,arrayD
xchg eax,[arrayD+4]
```

- Step 2: Exchange EAX with the third array value and copy the value in EAX to the first array position.

```
xchg eax,[arrayD+8]
mov  arrayD,eax
```

Evaluate this . . .

- We want to write a program that adds the following three bytes:

```
.data
myBytes BYTE 80h,66h,0A5h
```

- What is your evaluation of the following code?

```
mov al,myBytes
add al,[myBytes+1]
add al,[myBytes+2]
```

- What is your evaluation of the following code?

```
mov ax,myBytes
add ax,[myBytes+1]
add ax,[myBytes+2]
```

Evaluate this . . . (cont)

```
.data
myBytes BYTE 80h,66h,0A5h
```

- How about the following code. Is anything missing?

```
movzx ax,myBytes
mov   bl,[myBytes+1]
add   ax,bx
mov   bl,[myBytes+2]
add   ax,bx                        ; AX = sum
```

Yes: Move zero to BX before the MOVZX instruction.

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* ← *destination* + 1
- DEC *destination*
  - Logic: *destination* ← *destination* – 1

Flag Register	Full Name	Applies To	Set When...	Example	Decimal	Binary
CF	Carry Flag	Unsigned Ops	Carry out or borrow occurs	0xFF + 1 = 0x00, CF = 1	255	11111111
OF	Overflow Flag	Signed Ops	Signed result is too big or small	127 + 1 = -128, OF = 1		

```
  11111111   (0xFF)
+ 00000001   (0x01)
-----------
1 00000000   → 9 bits!
```

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
```

AX: [AH][AL]
$\uparrow$ $\uparrow$
8 8 bits

mov ax, 00FFh ; AX = 00FFh → AH = 00h, AL = FFh
inc al ; AL = FFh → 00h (overflow)

Drill ...

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* ← *destination* + source
- SUB destination, source
  - Logic: *destination* ← *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
```

Suppose AX contains –32,768 and we apply NEG to it. Will the result be valid?

NEG Instruction and the Flags

The processor implements NEG using the following internal operation:

```
SUB 0,operand
```

Any nonzero operand causes the Carry flag to be set.

```
.data
valB BYTE 1,0
valC SBYTE -128
.code
   neg valB                       ; CF = 1, OF = 0
   neg [valB + 1]                 ; CF = 0, OF = 0
   neg valC                       ; CF = 1, OF = 1
```

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
```

Drill …

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

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

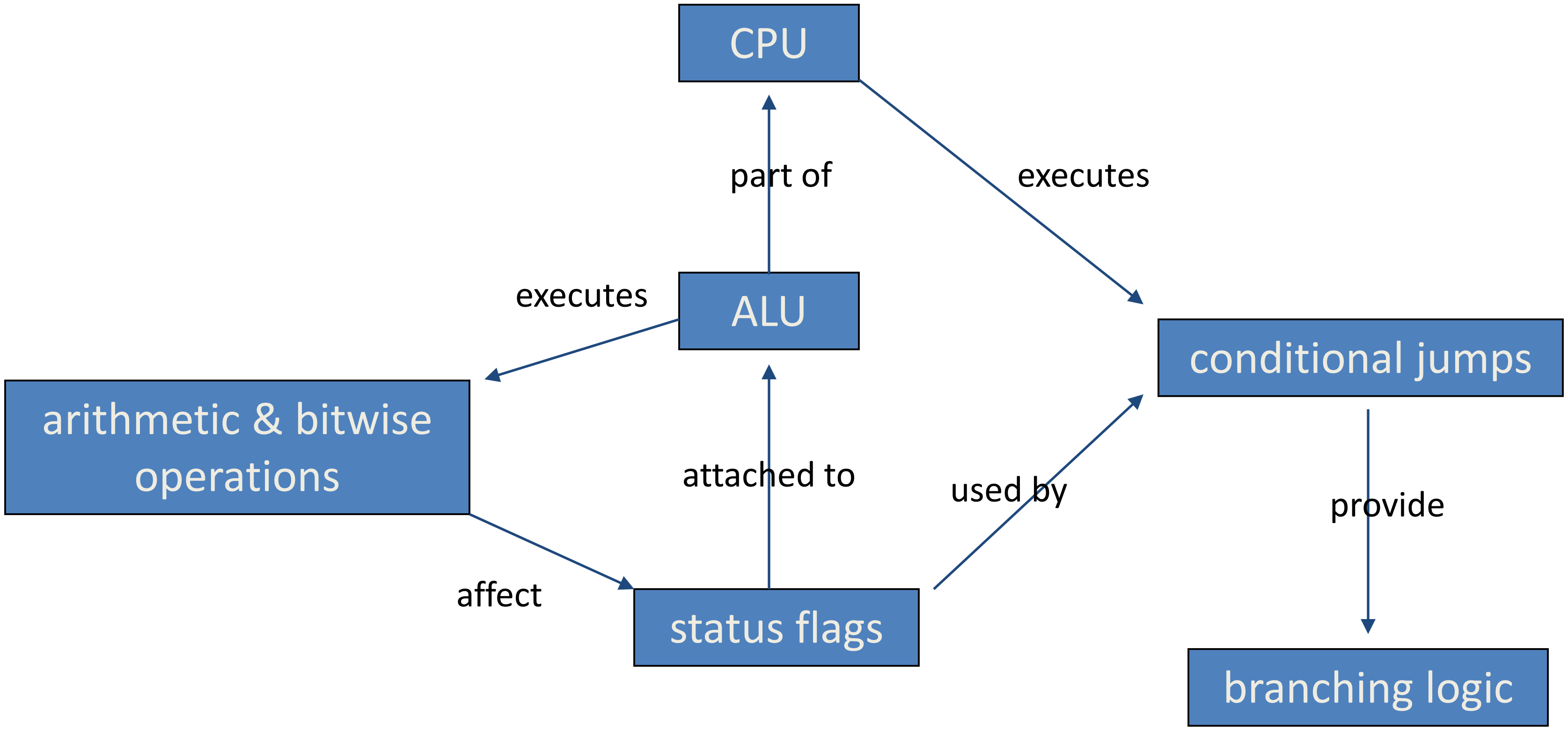
Assume that all values are signed doublewords.

```
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 – 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
- The MOV instruction never affects the flags.

Concept Map



You can use diagrams such as these to express the relationships between assembly language concepts.

This diagram illustrates the relationship between key components within a Central Processing Unit (CPU) and how they work together to execute instructions, particularly focusing on arithmetic and bitwise operations and conditional jumps.
Here's a breakdown of the diagram:

1. CPU (Central Processing Unit):

•This is the brain of the computer, responsible for executing instructions. It's at the top of the diagram, indicating its central role.

2. ALU (Arithmetic Logic Unit):

•Connected to the CPU with the label "part of," the ALU is a fundamental component of the CPU.

•Its primary function is to execute "arithmetic & bitwise operations."

3. Arithmetic & Bitwise Operations:

•This box represents the set of operations that the ALU can perform.

•**Arithmetic operations** include addition, subtraction, multiplication, division, etc.

•**Bitwise operations** include logical operations like AND, OR, NOT, XOR, and shift/rotate operations (like the ones we discussed earlier).

4. Status Flags:

•The ALU is "attached to" the status flags.

•"arithmetic & bitwise operations" "affect" the status flags.

•**Status flags** are special bits within the CPU that store information about the result of the last ALU operation. Common flags include:

- **Zero Flag (Z):** Set if the result of the operation is zero.
- **Carry Flag (C):** Set if there was a carry-out from the most significant bit (in addition) or a borrow (in subtraction).
- **Sign Flag (S):** Set if the result is negative (the most significant bit is 1 in signed arithmetic).
- **Overflow Flag (V or OF):** Set if a signed arithmetic operation resulted in an overflow.

5. Conditional Jumps:

•The CPU "executes" "conditional jumps."

•"conditional jumps" "used by" the "ALU." This is slightly inaccurate phrasing; rather, the *results of ALU operations* (reflected in the status flags) determine whether a conditional jump is taken.

•**Conditional jumps** are instructions that allow the program's execution flow to change based on certain conditions. These conditions are often the state of the status flags. For example, "jump if zero" will cause the program to jump to a different instruction if the Zero Flag is set.

6. Branching Logic:

•"conditional jumps" "provide" "branching logic."

•**Branching logic** refers to the ability of a program to execute different sequences of instructions based on conditions. Conditional jumps are the fundamental mechanism for implementing branching logic (e.g., if-else statements, loops) in assembly language.

In Simple Terms:
Imagine the ALU as a calculator inside the CPU. When you perform a calculation (arithmetic or bitwise operation), the calculator not only gives you the result but also sets some little indicators (status flags) based on that result (e.g., was the result zero? was there a carry?). The CPU can then look at these indicators when it encounters a "conditional jump" instruction. Based on what the indicators say, the CPU might decide to jump to a different part of the program instead of just executing the next instruction in sequence. This is how programs can make decisions and follow different paths, creating more complex and intelligent behavior.

Zero Flag (ZF)

The Zero flag is set when the result of an operation produces zero in the destination operand.

```
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
```

Remember...

- 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
```

Signed and Unsigned Integers
A 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 and Carry Flags
A Hardware Viewpoint

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

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

MSB = Most Significant Bit (high-order bit)

XOR = eXclusive-OR operation

NEG = Negate (same as SUB 0,operand)

Carry Flag (CF)

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
```

Drill . . .

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 the Overflow flag?
   mov al,80h
   add al,92h                 ; OF = 1

   mov al,-2
   add al,+127                ; OF = 0
```

Drill . . .

What will be the values of the given flags after each operation?

```
mov al,-128
neg al                ; CF = 1    OF = 1

mov ax,8000h
add ax,2              ; CF = 0    OF = 0

mov ax,0
sub ax,2              ; CF = 1    OF = 0

mov al,-5
sub al,+125           ; OF = 1
```

Summary

- Data Transfer Instructions
  - Operand Types
  - Instruction Operand Notation
  - Direct Memory Operands
  - MOV Instruction
  - Zero & Sign Extension
  - XCHG Instruction
  - Direct-Offset Instructions

Summary *(cont.)*

- Add and Subtract
  - Flags affected by the above operations

```
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
```

Reference

Most of the Slides are taken from Presentation:

Chapter 4

Assembly Language for Intel-Based Computers, 4th Edition

Kip R. Irvine