Week 4

Chapter 4: Data Transfers, Addressing, and Arithmetic

Class 10

Chapter Overview

- **Data Transfer Instructions**
- Addition and Subtraction
- Data-Related Operators and Directives
- Indirect Addressing
- JMP and LOOP Instructions
- 64-Bit Programming

Book's Page 96 to 111

Data Transfer Instructions

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

Operand Types

- 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
reg8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
reg16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
reg32	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
reg/mem8	8-bit operand, which can be an 8-bit general register or memory byte
reg/mem16	16-bit operand, which can be a 16-bit general register or memory word
reg/mem32	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

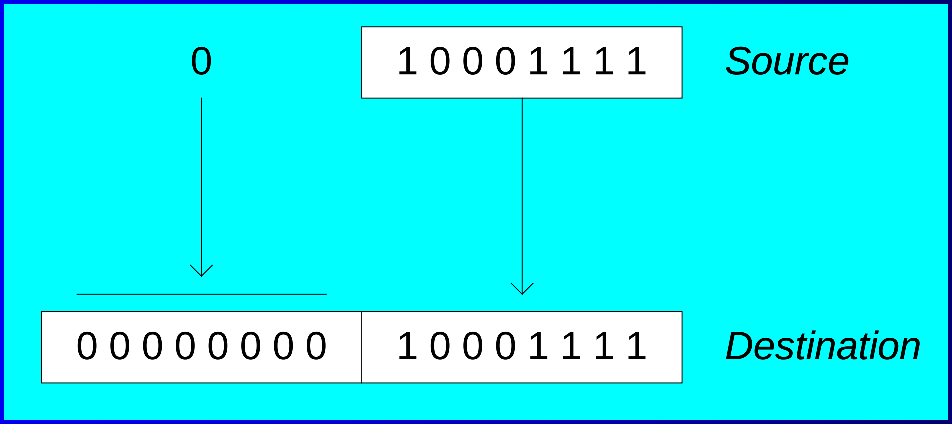
```
.data
count BYTE 100
wVal  WORD 2
.code
   mov bl,count
   mov ax,wVal
   mov count,al

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

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
```

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.

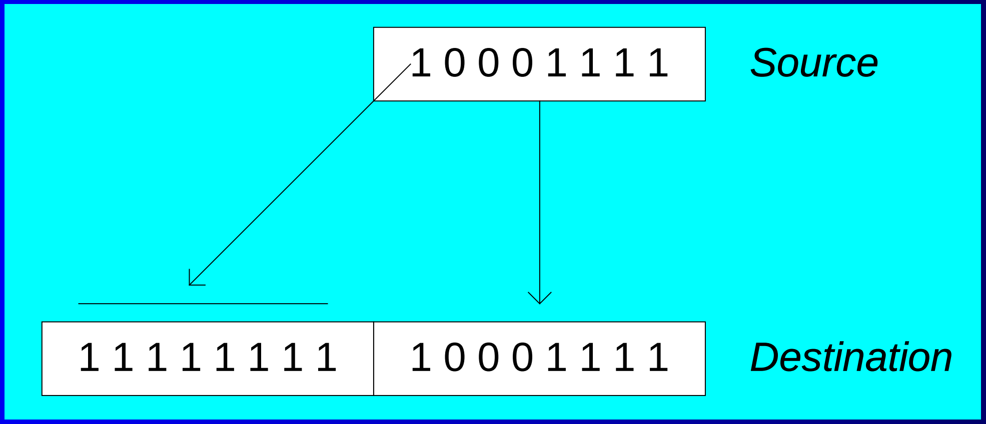



```
mov bl,10001111b
movzx ax,bl                ; zero-extension
```

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?

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 EAX 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]
```

- Any other possibilities?

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.

What's Next

- Data Transfer Instructions
- **Addition and Subtraction**
- Data-Related Operators and Directives
- Indirect Addressing
- JMP and LOOP Instructions
- 64-Bit Programming

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

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
```

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
```

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

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

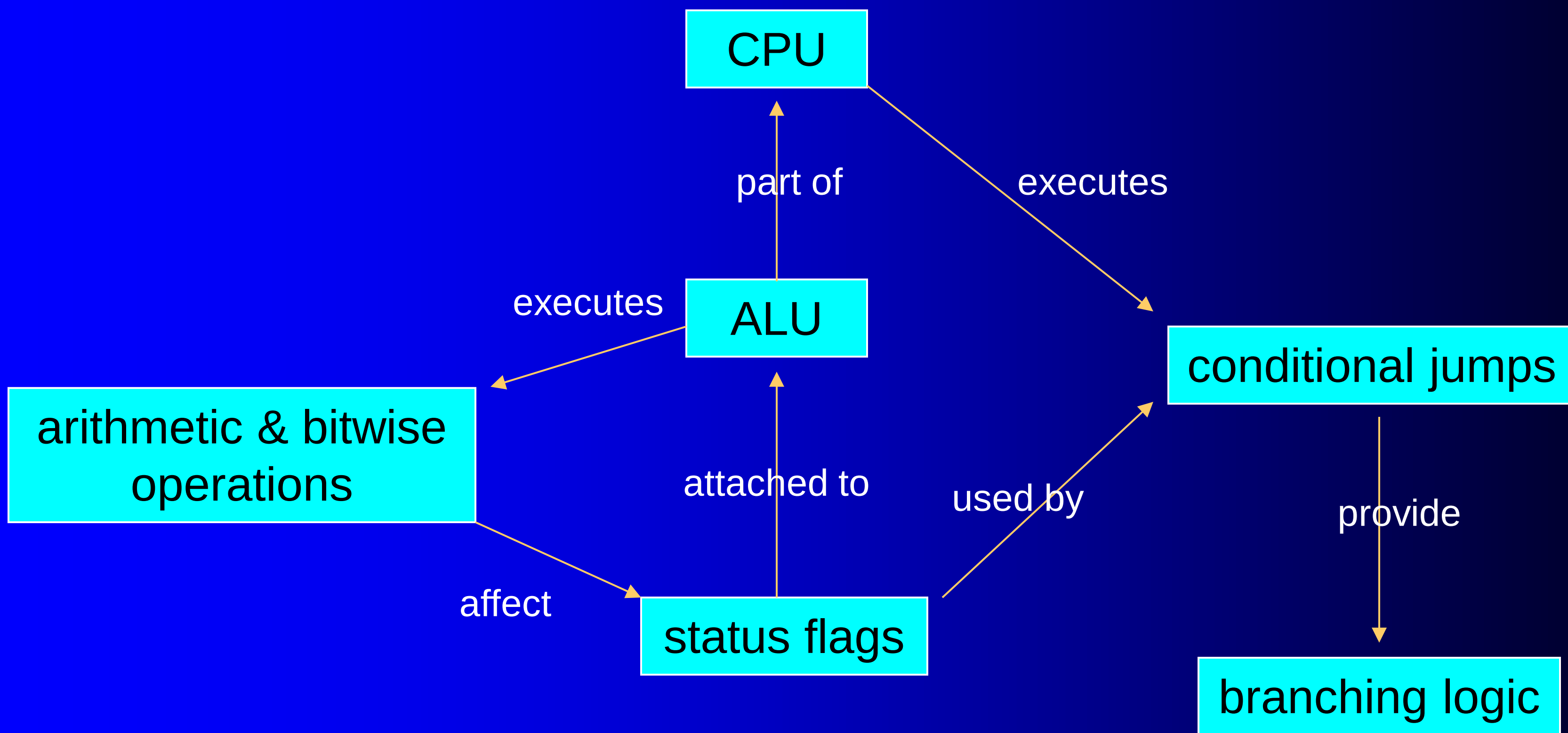
Assume that all values are signed doublewords.

```
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.