Topic 4
Lecture 4a
Data Transfer

CSCI 150

Assembly Language / Machine Architecture

Prof. Dominick Atanasio

Chapter Overview

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

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 dereferenced by the assembler if it is surrounded by square braces

```
section .data
var1 db 10h

section .text
mov al, [var1]                    ; AL = 10h
```

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

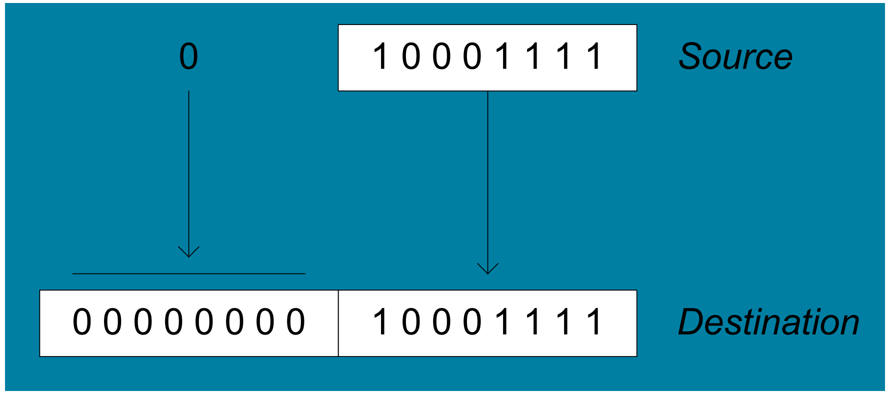
```
section .data
    count   db  100
    wVal    dw  2

section .text
    mov bl, [count]
    mov ax, [wVal]
    mov [count], al

    mov al, wVal            ; error: address doesn't fit
    mov ax, count           ; error: address doesn't fit
    mov eax, count
```

NASM

- NASM, by design, chooses not to remember the types of variables you declare.
- NASM will deliberately remember nothing about the symbol wVal except where it begins (its address)
- You must explicitly tell the assembler the size of the memory location
  - mov WORD [wVal], 2 ; the value 2 will occupy the 16 bits
  - mov WORD [wVal], al ; results in an error because al is only one byte
  - mov BYTE [wVal], 2 ; the value 2 is loaded into a single byte at that address

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.

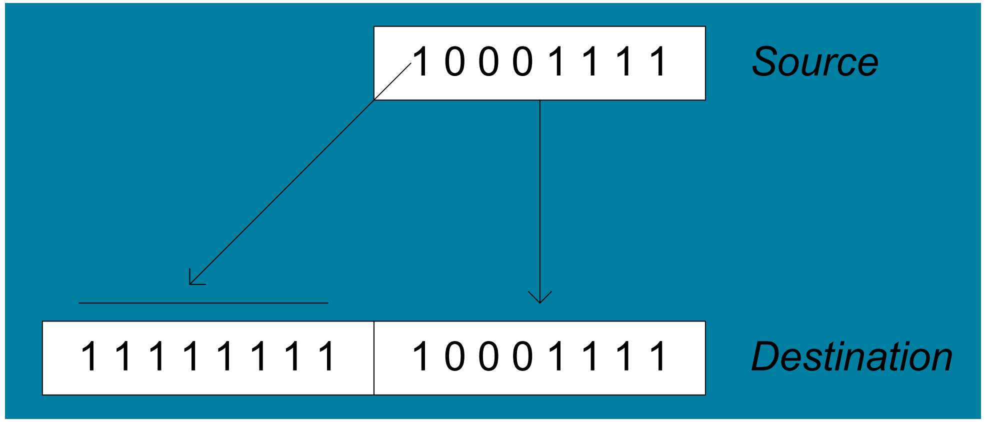



```
mov bl, 1000_1111b
movzx ax, bl            ; zero-extension
```

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,1000_ 1111b

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.

```
section .data
var1: dw 1000h
var2: dw 2000h

section .text
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 not allowed
```

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.

```
section .data
arrayB: db 10h, 20h, 30h, 40h

section .text
mov al, [arrayB + 1]                    ; AL = 20h
```

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

Direct-Offset Operands (cont) (1 of 2)

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.

Direct-Offset Operands (cont)

```
section .data
arrayW: dw 1000h, 2000h, 3000h
arrayD:   dd 1, 2, 3, 4
section .text
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?

Your turn. . .

Begin your in-class assignment.

Evaluate this . . .

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

```
section .data
myBytes: db 80h, 66h, 0A5h
```

- What is your evaluation of the following code?

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

Evaluate this . . . (cont)

- How about the following code. Is anything missing?

```
section .data
        myBytes: db 80h, 66h, 0A5h
section .text
        movzx ax, BYTE [myBytes]
        mov  bl, [myBytes + 1]
        add  ax, bx
        mov  bl, [myBytes+2]
        add  ax, bx                    ; AX = sum
```

Yes: Move zero to BX before the "*mov bl, [myBytes + 1]*" instruction to clear the BX register.

What's Next (1 of 5)

- Data Transfer Instructions
- **Addition and Subtraction**
- Data-Related Operators and Directives
- Indirect Addressing
- JMP and LOOP 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
  - Preserves the state of the carry flag CF
- INC *destination*
  - Logic: *destination* ← *destination* + 1
- DEC *destination*
  - Logic: *destination* ← *destination* – 1

INC and DEC Examples

```
section .data
myWord:   dw 1000h
myDword: dd 10000000h

section .text
        inc WORD [myWord]           ; 1001h
        dec WORD [myWord]           ; 1000h
        inc DWORD [myDword]         ; 10000001h

        mov ax, 00FFh
        inc ax                 ; AX = 0100h
        mov ax, 00FFh
        inc al                 ; AX = 0000h – only works on lower byte
```

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

```
section .data
myByte: db 0FFh, 0

section .text
    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

```
section .data
var1:	dw 10000h
var2:	dw 20000h
section .text
	mov eax, [var1]		; 00010000h
	add eax, [var2]		; 00030000h
	add ax, 0FFFFh		; 0003FFFFh
	add eax, 1		; 00040000h
	sub ax, 1		; 0004FFFFh
```

NEG (negate) Instruction

Performs 2’s compliment on the operand. Operand can be a register or memory operand.

```
section .data
valB db -1
valW dw +32767
section .text
    mov al, [valB]          ; AL = -1
    neg al                  ; AL = +1
    neg word [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.

```
section .data
valB: db 1, 0
valC: db -128

section .text
    neg [valB]          ; CF = 1, OF = 0
    neg [valB + 1]      ; CF = 0, OF = 0
    neg [valC]          ; CF = 1, OF = 1
```

Implementing Arithmetic Expressions (You Try)

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

rVal = -xVal + (yVal – zVal) (assume 32-bit variables)

```
section .data
rVal:    dd 0
xVal:    dd 26
yVal:    dd 30
zVal:    dd 40

section .text
         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:

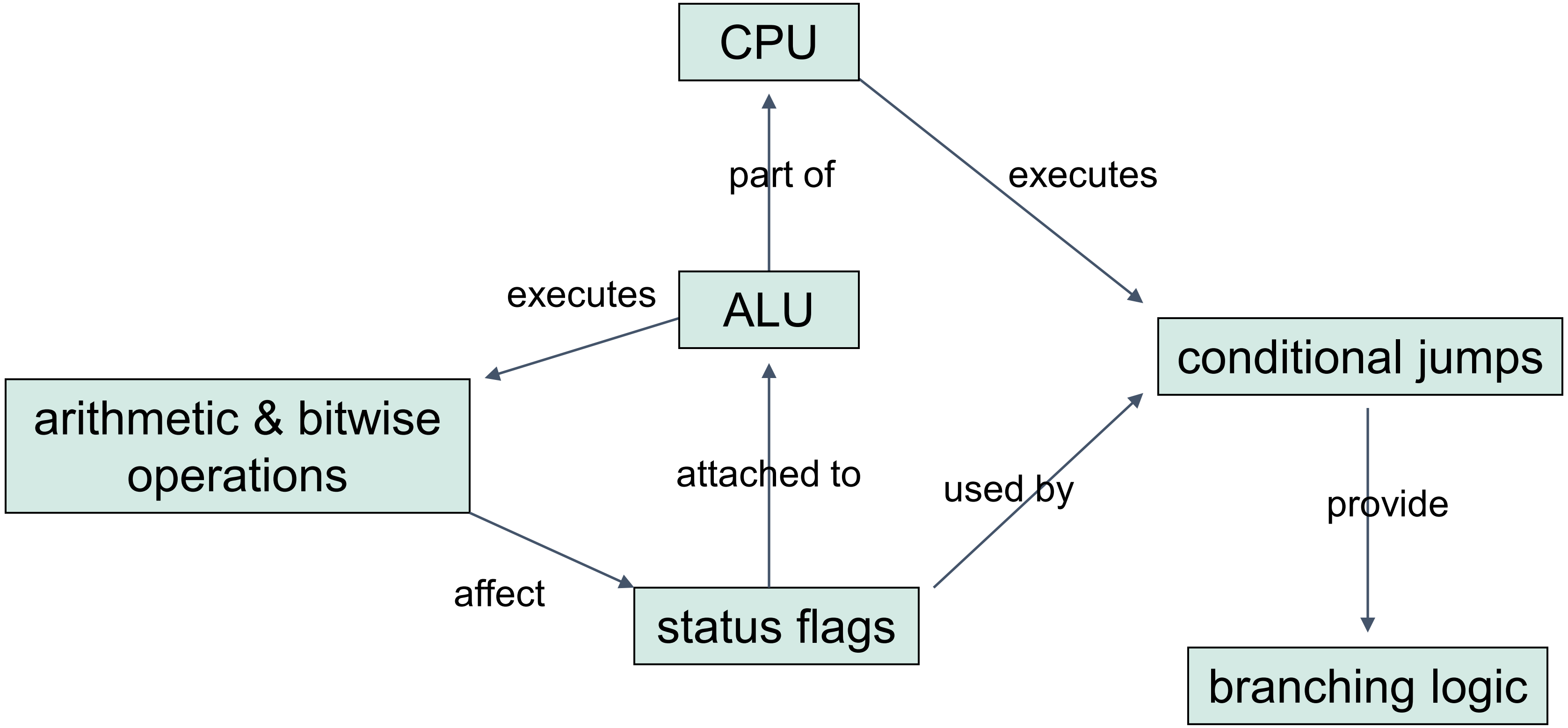
rVal = xVal - (-yVal + zVal)

Assume that all values are signed 32-bit values and rVal is an uninitialized variable.

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.

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 affects OF and CF:
  - CF = (carry out of the MSB)
  - OF = CF XOR MSB
- How the SUB instruction affects OF and CF:
  - CF = INVERT (carry out of the MSB)
  - negate the source and add it to the destination
  - OF = CF XOR 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
```

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
```

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

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

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

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

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

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            ; CF = 0     OF = 0
```

46 69 6E 61 6C