

CSC 3210

Computer Organization and Programming

Chapter 4: Data Transfers, Addressing, and Arithmetic

Dr. Zulkar Nine

mnine@gsu.edu

Georgia State University

Spring 2021

Outline

- **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

Data Transfer 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

Listing File

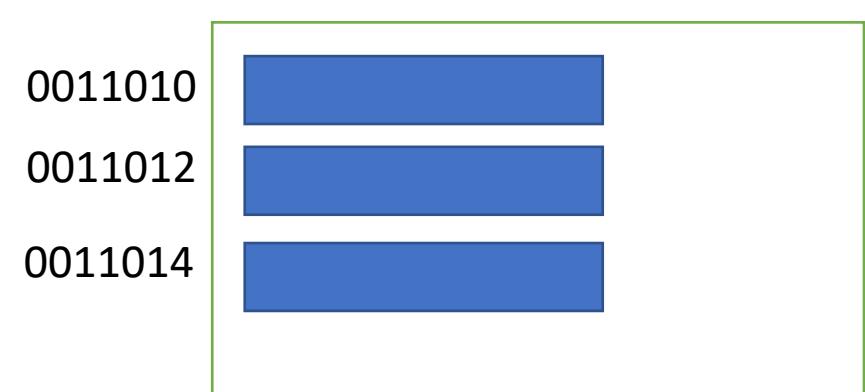
```
00000000 .data
00000000 00000000 sum DWORD 0
00000000 .code
00000000 main proc
00000000 B8 00000008      mov eax,8
00000005 83 C0 04          add eax,4
00000008 A3 00000000 R    mov sum, eax
```

Data Transfer Instructions: Instruction Operand Notation

Operand	Description	
<i>reg8</i>	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL	MOV reg,reg MOV mem,reg MOV reg,mem MOV mem,imm MOV reg,imm
<i>reg16</i>	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP	
<i>reg32</i>	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP	
<i>reg</i>	Any general-purpose register	
<i>sreg</i>	16-bit segment register: CS, DS, SS, ES, FS, GS	
<i>imm</i>	8-, 16-, or 32-bit immediate value	MOVZX <i>reg32,reg/mem8</i> MOVZX <i>reg32,reg/mem16</i> MOVZX <i>reg16,reg/mem8</i>
<i>imm8</i>	8-bit immediate byte value	
<i>imm16</i>	16-bit immediate word value	
<i>imm32</i>	32-bit immediate doubleword value	
<i>reg/mem8</i>	8-bit operand, which can be an 8-bit general register or memory byte	MOVZX <i>reg/mem8</i> MOVZX <i>reg/mem16</i> MOVZX <i>reg/mem32</i>
<i>reg/mem16</i>	16-bit operand, which can be a 16-bit general register or memory word	
<i>reg/mem32</i>	32-bit operand, which can be a 32-bit general register or memory doubleword	
<i>mem</i>	An 8-, 16-, or 32-bit memory operand	

Data Transfer Instructions

- Register to Register
- Register to Memory , Vice versa



How to see variables in VS

Data Transfer Instructions: Direct Memory Operands

- A **direct memory operand** is a named reference to storage in memory

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

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

alternate format

Data Transfer Instructions: Direct Memory Operands

- The **named reference (label)** is automatically dereferenced by the **assembler**

```
.data  
var1 BYTE 10h
```

- Suppose **var1** were located at offset 10400h.
- The following instruction copies its value into the **AL** register:

```
mov al, var1
```

- It would be **assembled** into the following machine instruction:

A0 00010400

Data Transfer Instructions: Direct Memory Operands

- The **named reference (label)** is automatically dereferenced by the **assembler**

A0 00010400

Listing File

- The **first byte** in the machine instruction is the **opcode**.
- The **remaining part** is the **32-bit hexadecimal address of var1**.

```
00000000 .data
00000000 00000000 sum DWORD 0
00000000 .code
00000000 main proc
00000000 B8 00000008    mov eax, 8
00000005 83 C0 04      add eax, 4
00000008 A3 00000000 R  mov sum, eax
```

Data Transfer Instructions: MOV Instruction

- Move from source to destination.
- Syntax:

MOV *destination, source*

- MOV instruction **formats**:

MOV reg,reg
MOV mem,reg
MOV reg,mem
MOV mem,imm
MOV reg,imm

- Both operands must be the same size.
- Both operands cannot be memory operands.
- The (IP, EIP, or RIP) cannot be a destination operand.

Ex:

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

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

Data Transfer Instructions: MOV Instruction

- 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  
    mov eax,wVal  
    mov eip,dVal  
    mov 25,bVal  
    mov bVal2,bVal
```

- Both operands must be the same size.
- Both operands cannot be memory operands.
- The (IP, EIP, or RIP) cannot be a destination operand.

Data Transfer Instructions: MOV Instruction

- **Memory to Memory (problem):**
 - A single **MOV** instruction cannot be used to move data directly from one memory location to another.
 - Instead, you must move
 - the **source** operand's value to a **register**
 - before assigning its value to a memory operand:

Ex:

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

Data Transfer Instructions: MOV Instruction

- Overlapping Values

- The same 32-bit register can be modified using differently sized data.
 - When **oneWord** is moved to **AX**, it **overwrites** the existing value of **AL**.
 - When **oneDword** is moved to **EAX**, it **overwrites** **AX**.
 - When 0 is moved to **AX**, it **overwrites** the lower half of **EAX**.

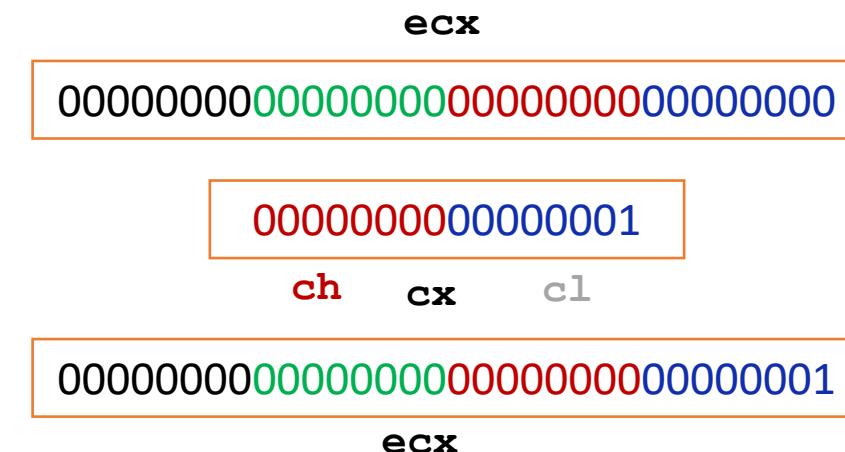
```
.data  
oneWord WORD 1234h  
oneDword DWORD 12345678h
```

```
.code  
mov eax, 0  
mov ax, oneWord           ; EAX = 00001234h  
mov eax, oneDword         ; EAX = 12345678h  
mov ax, 0                 ; EAX = 12340000h
```

Data Transfer Instructions: Zero & Sign Extension

- **Sign extension problem:**
 - MOV cannot directly copy data from a **smaller** operand to a **larger** one
 - **Workarounds:**
 - Suppose **count** (unsigned, 16 bits) must be moved to **ECX** (32 bits).
 - **Trick:** Set **ECX** to **zero** and move **count** to **CX**:

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



- What happens if we try the same approach with a signed integer equal to -16?

ecx

00000000000000000000000000000000

0000000000000001

ch

c1

cx

Data Transfer Instructions: Zero & Sign Extension

- **Sign extension problem:**

- What happens if we try the same approach with a signed integer equal to -16?

```
.data  
signedVal SWORD -16 ; FFF0h (-16)  
.code  
mov ecx,0  
mov cx,signedVal ; ECX = 0000FFF0h (+65,520)
```


Data Transfer Instructions: Zero & Sign Extension

- **Sign extension problem:**

```
.data  
signedVal SWORD -16           ; FFF0h (-16)  
.code  
mov ecx,0  
mov cx,signedVal            ; ECX = 0000FFF0h (+65,520)
```

- If we had filled **ECX** first with FFFFFFFFh and then copied **signedVal** to **CX**:

```
mov ecx,FFFFFFFh  
mov cx,signedVal           ; ECX = FFFFFFF0h (-16)
```

ecx

11111111111111111111111111111111

11111111111111110000

ch cx cl

11111111111111111111111111111111

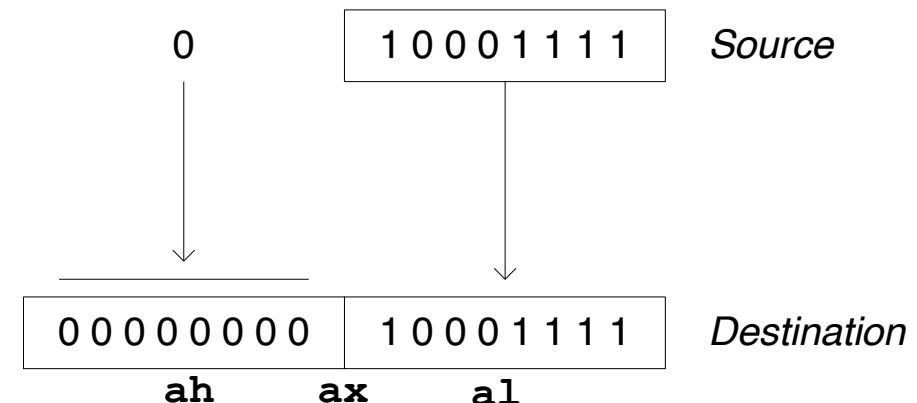
- **MOVZX** and **MOVSX** instructions to deal with both unsigned and signed integers.

Data Transfer Instructions: Zero Extension (MOVZX)

- The **MOVZX** instruction
- 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
```

MOVZX *reg32, reg/mem8*
MOVZX *reg32, reg/mem16*
MOVZX *reg16, reg/mem8*



The destination must be a register.

Data Transfer Instructions: Zero Extension (MOVZX)

The following examples use registers for all operands, showing all the size variations:

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

The following examples use memory operands for the source and produce the same results:

```
.data
byte1  BYTE  9Bh
word1  WORD  0A69Bh
.code
movzx  eax, word1       ; EAX = 0000A69Bh
movzx  edx, byte1       ; EDX = 0000009Bh
movzx  cx, byte1        ; CX  = 009Bh
```

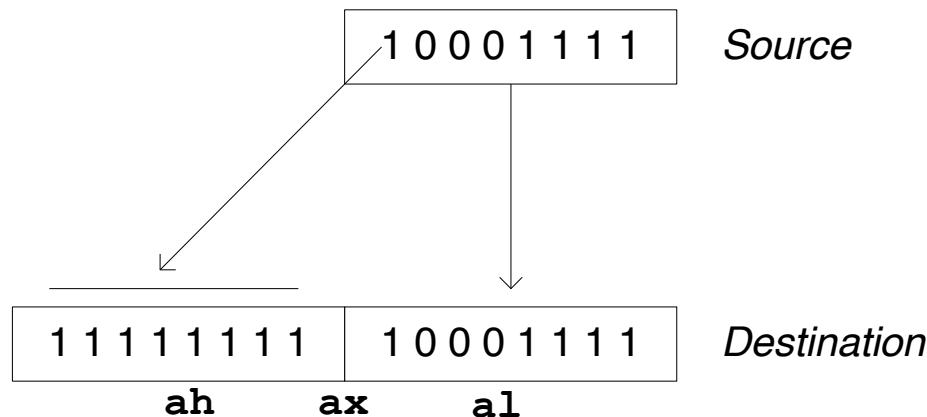
Data Transfer Instructions: Sign Extension (MOV_{SX})

- The **MOV_{SX}** instruction
- It fills the upper half of the destination with a copy of the **source operand's sign bit**.

MOV_{SX} *reg32, reg/mem8*
MOV_{SX} *reg32, reg/mem16*
MOV_{SX} *reg16, reg/mem8*

mov bl,10001111b

movsx ax,bl ; sign extension



The destination must be a register.

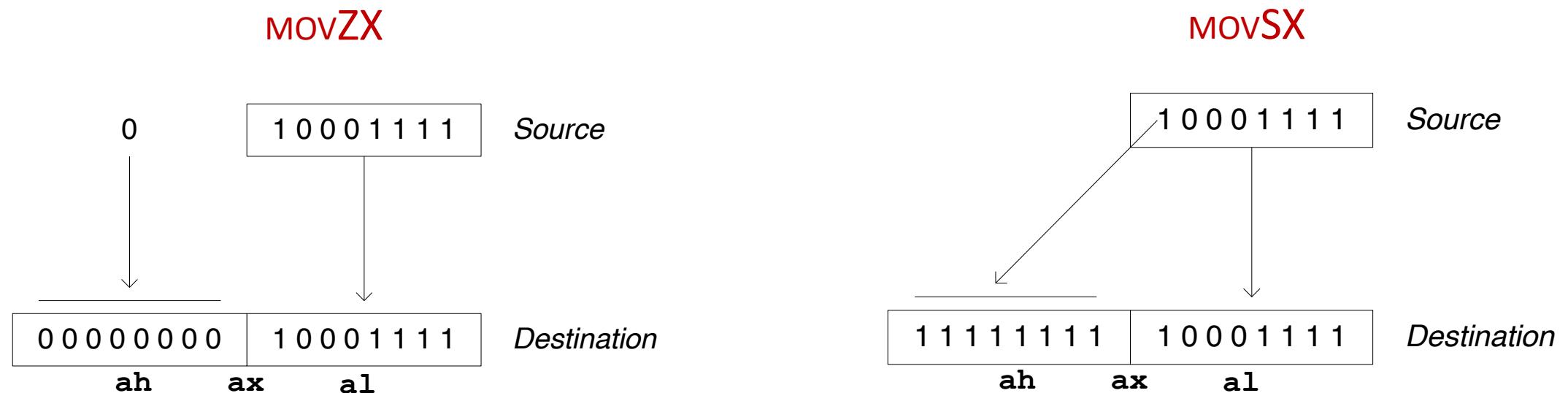
Data Transfer Instructions: Sign Extension (MOVSX)

- In the following example,
 - the hexadecimal value moved to BX is A69B,
 - so the leading “A” digit tells us that the highest bit is set.

```
mov    bx, A69Bh
movsx eax,bx          ; EAX = FFFFA69Bh
movsx edx,bl          ; EDX = FFFFFFF9Bh
movsx cx,bl           ; CX = FF9Bh
```

Data Transfer Instructions: Zero & Sign Extension

- **MOVZX vs. MOVSX**



Data Transfer Instructions: XCHG Instruction

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

XCHG *reg, reg*
XCHG *reg, mem*
XCHG *mem, reg*

```
xchg ah,al      ; exchange 8-bit regs  
xchg ax,bx      ; exchange 16-bit regs  
xchg eax,ebx    ; exchange 32-bit regs  
xchg var1,bx ; exchange 16-bit mem op with BX
```

```
xchg var1,var2 ; error: two memory operands
```


Xchange items

Data Transfer Instructions: XCHG Instruction

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

XCHG *reg, reg*
XCHG *reg, mem*
XCHG *mem, reg*

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

Data Transfer Instructions: 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           ; AL = 10h  
mov al, arrayB+1         ; AL = 20h  
  
mov al, [arrayB+1]       ; alternative notation
```

Why doesn't arrayB+1 produce 11h?

Data Transfer Instructions: Direct-Offset Operands

```
.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] ; ??
```


Data Transfer Instructions: Direct-Offset Operands

```
.data  
arrayW WORD 1000h,2000h,3000h  
arrayD DWORD 1,2,3,4  
.code  
mov ax, [arrayD+0] ; AX = 2000h  
mov ax, [arrayD+4] ; AX = 3000h  
mov eax, [arrayD+8] ; EAX = 00000002h
```

Will the following statements assemble?

```
mov ax,[arrayW-2] ; ??  
mov eax,[arrayD+16] ; ??
```


Data Transfer Instructions: Direct-Offset Operands

- Write a program that rearranges the values of three **doubleword** values in the following array as:

1,2,3 -----→ 3, 1, 2.

LAB

```
.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]
```

XCHG *reg, reg*
XCHG *reg, mem*
XCHG *mem, reg*

- **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
```


Data Transfer Instructions: Direct-Offset Operands

- 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?

movsx ax, myBytes

add ax, [myBytes+1]

add ax, [myBytes+2]

- Any other possibilities?

Data Transfer Instructions: Direct-Offset Operands

- 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?

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

- Any other possibilities?

Handwritten notes in green ink:

```
mov bl, [myBytes+1]  
movsx bx, bl  
add ax, bx
```

Outline

- 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

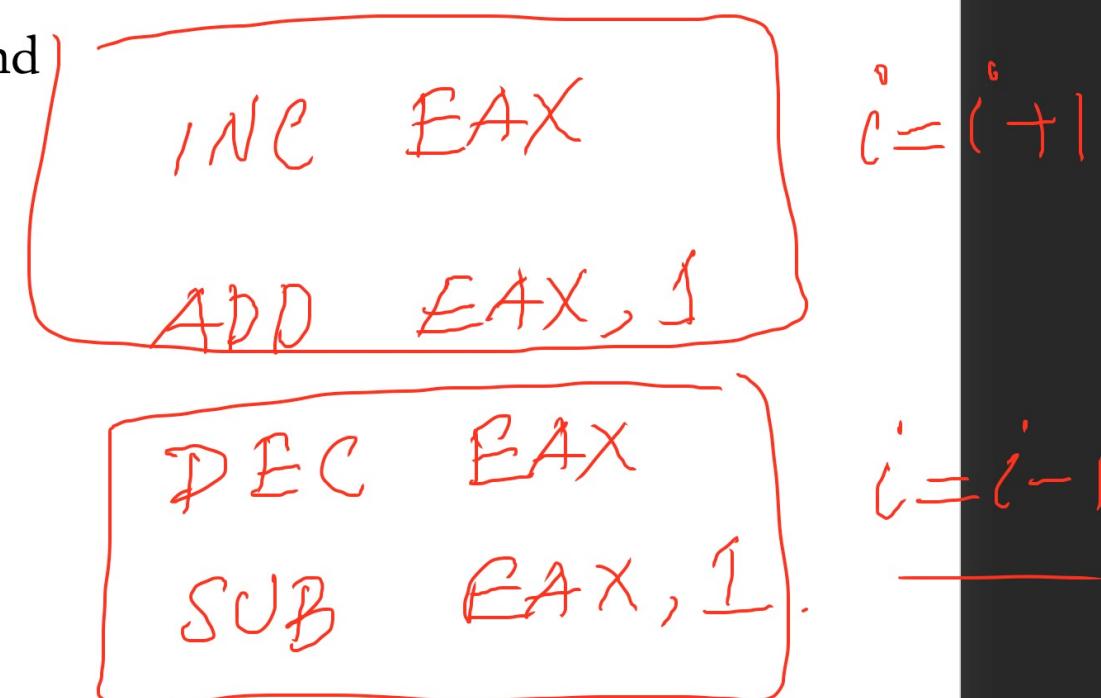
Addition and Subtraction: INC and DEC Instructions

- Add 1, subtract 1 from destination operand
 - operand may be register or memory
- The Syntax:

INC reg/mem

DEC reg/mem

- INC *destination*
 - Logic: $destination \leftarrow destination + 1$
- DEC *destination*
 - Logic: $destination \leftarrow destination - 1$



Addition and Subtraction: INC and DEC Examples

```
.data  
myWord WORD 1000h  
myDword DWORD 10000000h  
.code  
inc myWord  
dec myWord  
inc myDword  
  
mov ax,00FFh  
inc ax  
mov ax,00FFh  
inc al
```

Addition and Subtraction: INC and DEC Examples

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

.data

myByte BYTE FFh, 0

.code

```
    mov al,myByte  
    mov ah,[myByte+1]  
    dec ah  
    inc al  
    dec ax
```

Addition and Subtraction

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

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

reg, reg

mem, reg

reg, mem

mem, imm

reg, imm

ADD and SUB Examples

```
.data
var1 DWORD 10000h
var2 DWORD 20000h

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

reg, reg
mem, reg
reg, mem
mem, imm
reg, imm
```

Can you add registers of different sizes?

add eax,bx

Can you add mem to mem?

add var1,var2

Addition and Subtraction

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

NEG (negate) Instruction

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

NEG *reg*

NEG *mem*

Ex.

```
.data  
valB SBYTE -1  
valW WORD +32  
.code  
    mov al, valB  
    neg al  
    neg valW
```

;valB BYTE -1 is also acceptable, why?

; AL = -1
; AL = +1
; valW = -32

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

Addition and Subtraction

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

Implementing Arithmetic Expressions

- HLL compilers translate mathematical expressions into assembly language.
- You can do it also.
- For example: Do not permit Xval, Yval, or Zval to be modified

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

```
.data
    Rval SDWORD ?
    Xval SDWORD 26
    Yval SDWORD 30
    Zval SDWORD 40

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

Implementation

- Help
- You
- For

Source.asm* Project4.lst

```

1 .386
2 .MODEL flat, stdcall
3 .STACK 4096
4 ExitProcess Proto, dwExitCode:DWORD
5
6 .DATA
7 Rval SDWORD ?
8 Xval SDWORD 26
9 Yval SDWORD 30
10 Zval SDWORD 40
11
12
13 .CODE
14 main PROC
15     mov eax, Yval
16     sub eax, Zval
17
18     sub eax, Xval
19
20     mov Rval, eax
21
22
23     INVOKE ExitProcess, 0
24 main ENDP
25 END main
26
27

```

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

* cannot alter variables

* inputs → same size ✓

* At least one register. ✓

Implementing Arithmetic Expressions

- 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.

```
.data  
Rval SDWORD ?  
Xval SDWORD 26  
Yval SDWORD 30  
Zval SDWORD 40
```

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

Previous Example:

Rval = -Xval + (Yval – Zval)

Addition and Subtraction

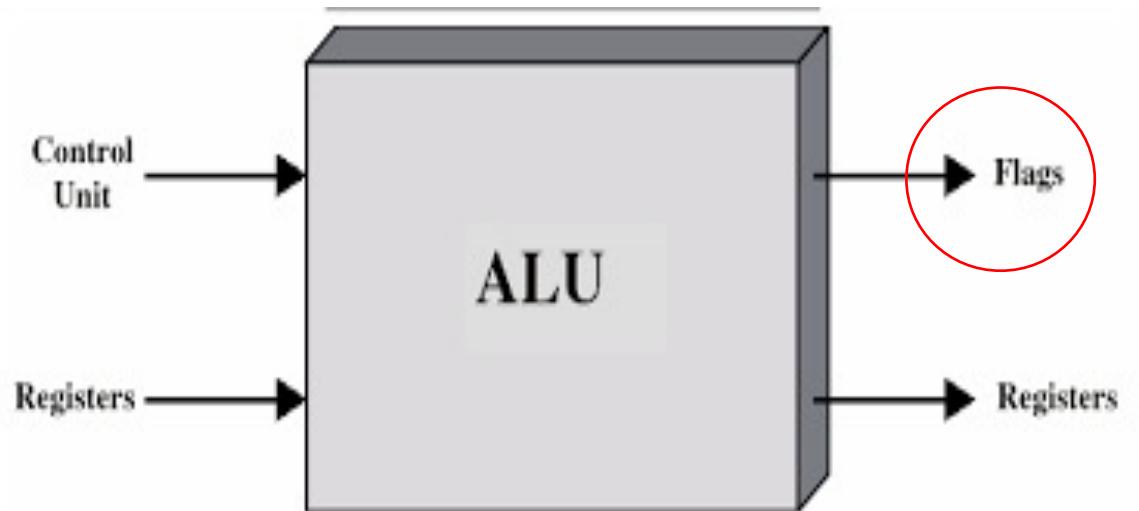
- INC and DEC Instructions
- ADD and SUB Instructions
- NEG Instruction
- Implementing Arithmetic Expressions
- Flags Affected by Arithmetic
 - Zero
 - Sign
 - Carry
 - Overflow

Flags: Why?

- When executing **arithmetic instructions**, we often want to know something about the **result**:
 - Is it **negative**, **positive**, or **zero**?
 - Is it **too large** or **too small** to fit into the destination operand?
- Answers to such questions can help detect **calculation errors** that might otherwise cause erratic program behavior.
- The **values** of CPU status flags is used to check the **outcome** of arithmetic operations.
 - Status flag values are also used to activate **conditional branching instructions**, the basic tools of program logic.

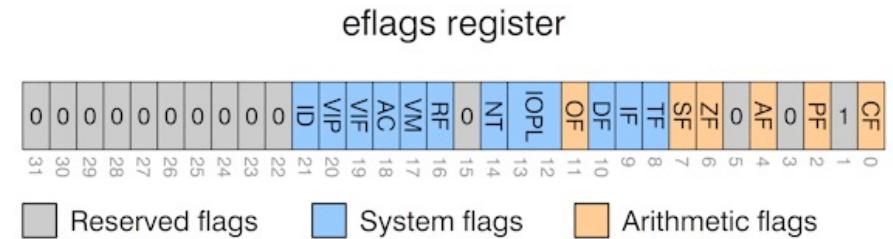
Flags: Flags Affected by Arithmetic/Bitwise

- The **ALU** has a number **of status flags** that reflect the outcome of
 - arithmetic operations
 - bitwise operations
- **based** on the contents of the
- destination operand



Flags: Flags Affected by Arithmetic/Bitwise

- 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**
 - **Visual Studio Flags?**

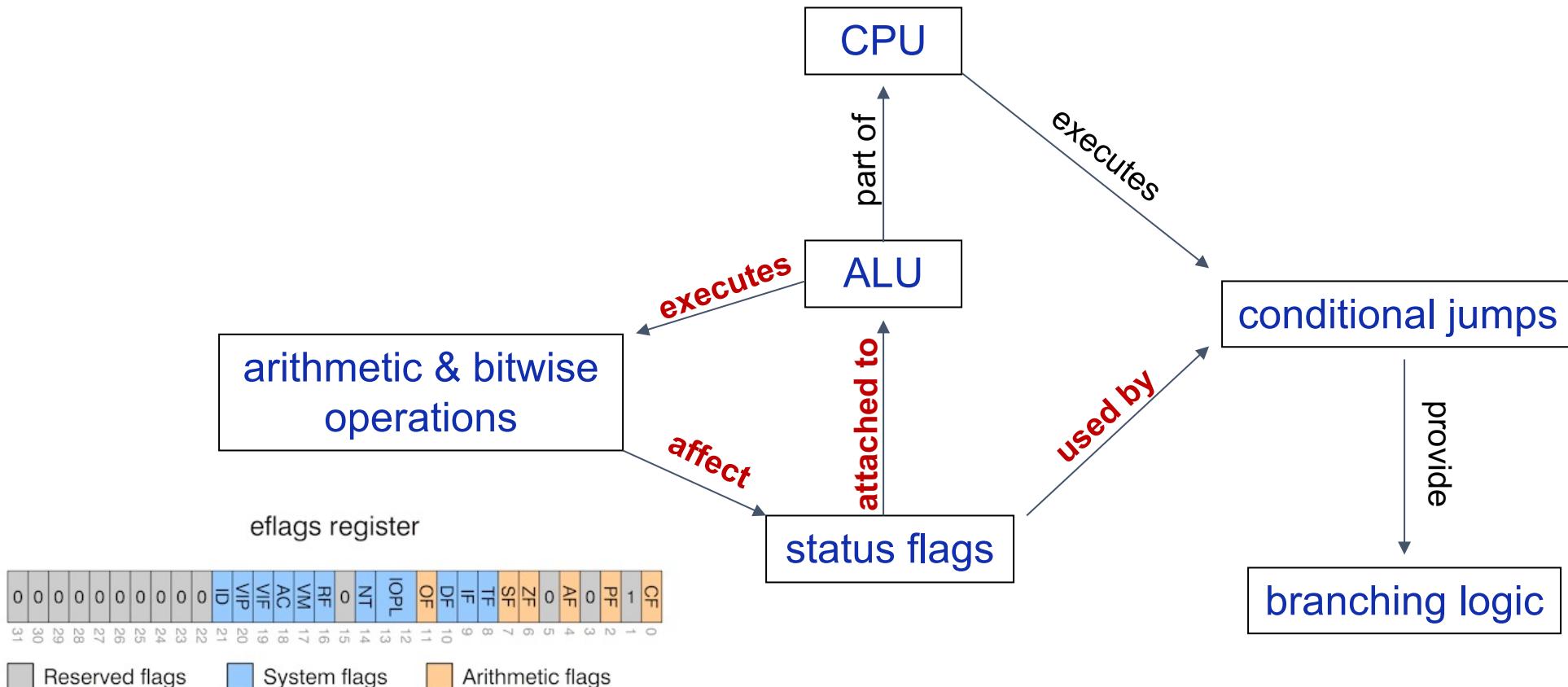


Flag Name	Overflow	Direction	Interrupt	Sign	Zero	Aux Carry	Parity	Carry
Symbol	OV	UP	EI	PL	ZR	AC	PE	CY

The **MOV** instruction never affects the flags. (**Why?**)

Flags: Map

- The ALU has a number **of status flags** that reflect the outcome of arithmetic operations and bitwise operations.



Flags: Displaying CPU Flags in Visual Studio

- **You must be currently debugging a program in order to see these menu options.**
- To display the CPU status flags **during a debugging session**,
 - select **Windows** from the Debug menu,
 - then select **Registers** from the **Windows** menu.
 - Inside the **Registers window**, right-click and select **Flags** from the dropdown list.

Each flag is assigned a value of **0 (clear)** or **1 (set)**.

Flags: Unsigned and Signed Integers Operations

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

CF vs. OF

Flags: Unsigned Operations, **Zero flag (ZF)**

- The **Zero flag (ZF)** is **set** when the result of an **operation** produces **zero** in the destination operand.

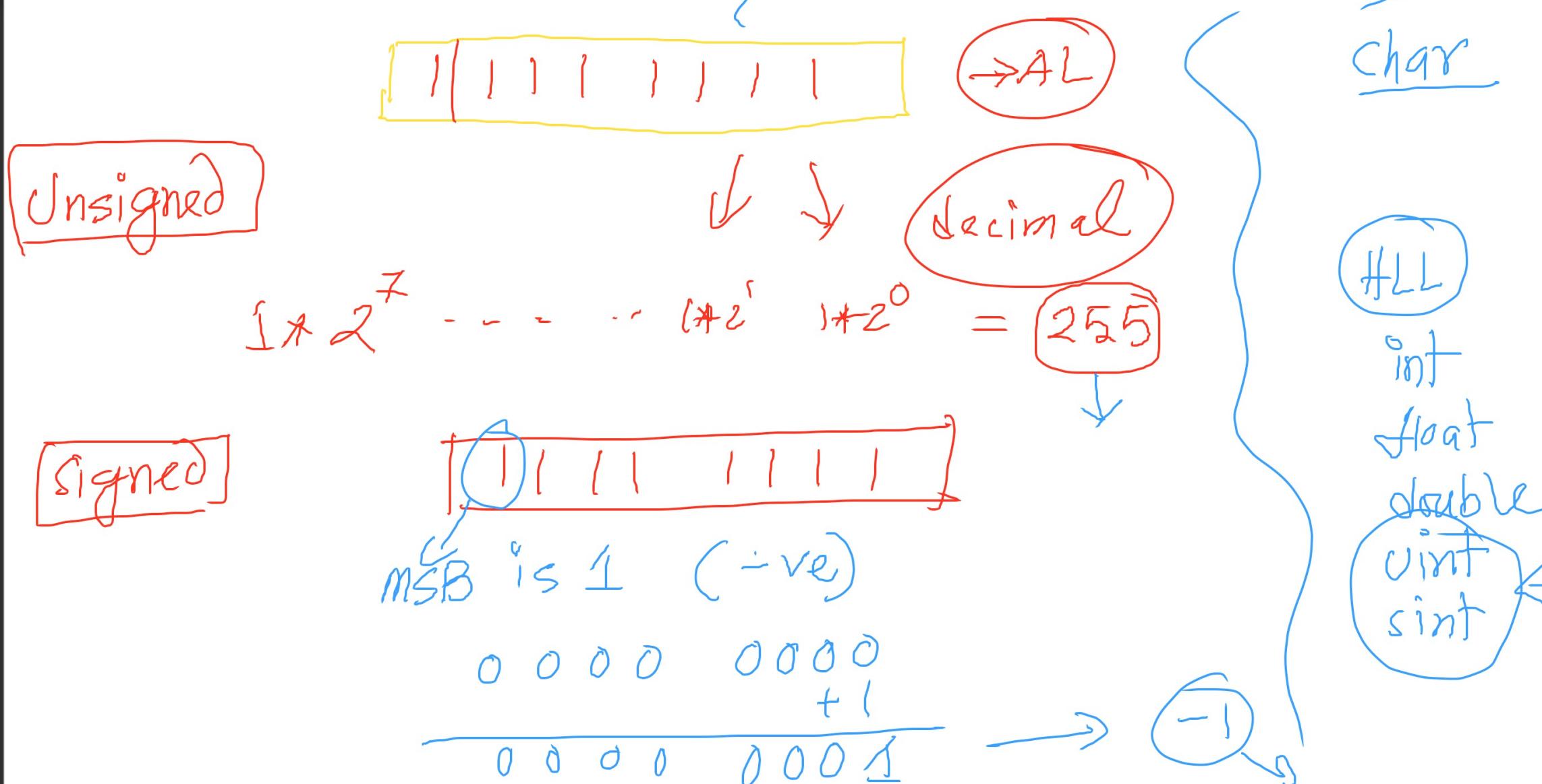
```
mov ecx,1  
sub ecx,1  
mov eax,FFFFFFFh  
inc eax           ; EAX = 0, ZF = 1  
inc eax           ; EAX = 1, ZF = 0
```

; ECX = 0, ZF = 1

Based on the contents of
the **destination operand**

Remember...

- A flag is **set** when it equals **1**.
- A flag is **clear** when it equals **0**.

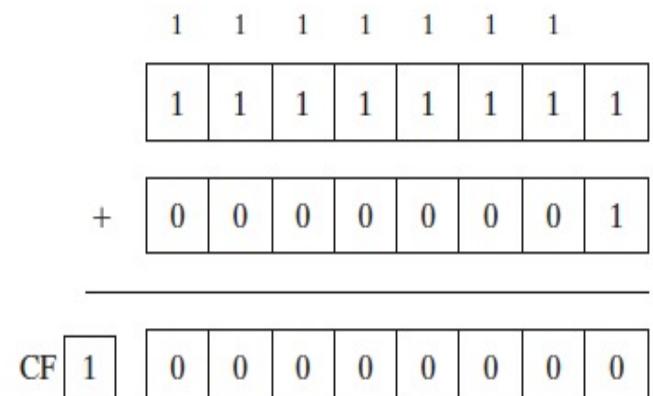


Flags: Unsigned Operations, **Carry flag (CF)**

- The **Carry flag (CF)** 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**).
- Addition .Vs Subtraction (**borrow**)
- Addition and the Carry Flag**

FIGURE 4-3 Adding 1 to 0FFh sets the Carry flag.

```
mov al, 0FFh  
add al, 1      ; CF = 1, AL = 0
```



Flags: Unsigned Operations, **Carry flag (CF)**

- **Subtraction** and the **Carry (and borrow) Flag**
- A **subtract** operation sets the **Carry flag** when
 - a **larger unsigned** integer is **subtracted** from a **smaller** one.

FIGURE 4–4 Subtracting 2 from 1 sets the Carry flag.

```
mov al,1  
sub al,2 ; AL = FFh, CF = 1
```

	0	0	0	0	0	0	0	1	(1)
+	1	1	1	1	1	1	1	0	(-2)
<hr/>									
CF	1	1	1	1	1	1	1	1	(FFh)

Flags: Unsigned Operations, **Carry flag (CF)**

- **Subtraction** and the **Carry (and borrow) Flag**
- A **subtract** operation sets the **Carry flag** when
 - a **larger unsigned** integer is **subtracted** from a **smaller one**.

; Try to go below zero:

```
mov al, 0  
sub al, 1 ; CF = 1, AL = FF
```

Flags: Unsigned Operations, **Parity**

- **Parity**
 - The Parity flag (**PF**) is **set** when
 - The least significant byte of the **destination** has an even number of 1 bits.
 - The following ADD and SUB instructions alter the parity of AL:
 - mov al,10001100b
 - add al,**00000010b** ; AL = **10001110**, **PF = 1**
 - sub al,10000000b ; AL = **00001110**, **PF = 0**

Flags: Signed Operations, **Sign Flag (SF)**

- The **Sign flag (SF)** 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
```

Flags: Signed Operations, **Sign Flag (SF)**

- The **Sign flag (SF)** is **set** when the destination operand is **negative**.
- 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
```

Flags: Unsigned and Signed Integers Operations

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

mov ax, 7ff0h					
add al, 10h	; AX=	SF=	ZF=	CF=	OF=
add ah, 1	; AX=	SF=	ZF=	CF=	OF=
add ax, 2	; AL=	SF=	ZF=	CF=	OF=

8000 → 32768

Attendance!

Flags: Signed Operations, **Overflow Flag (OF)**

The **Overflow flag (OF)** 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.

Flags: Signed Operations, Overflow Flag (OF)

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

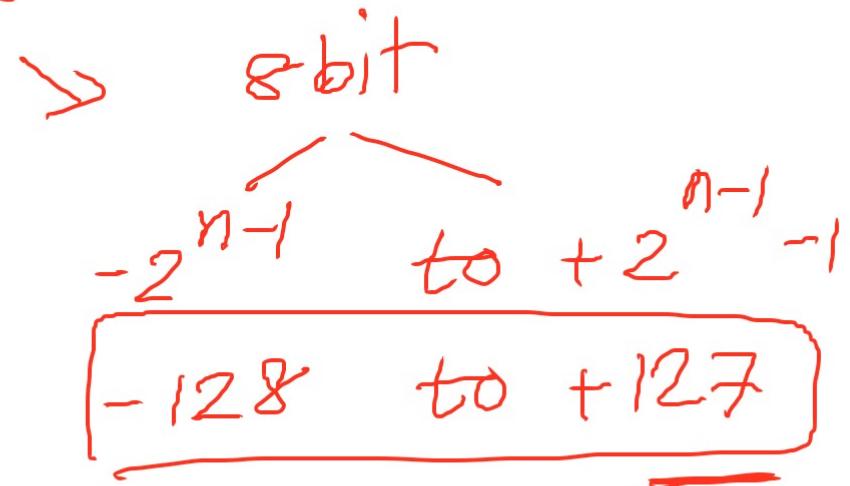
[; Example 1
mov al, +127
add al, 1

→ 128

; OF = 1, AL = ??

; Example 2
mov al, 7Fh
add al, 1

; OF = 1, AL = 80h



- 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.

Flags: Unsigned and Signed Integers Operations

Use the following data for Questions 4-5:

```
.data  
val2 WORD 8000h  
val4 WORD 7FFFh
```

1. If **val2** is incremented by 1 using the ADD instruction, what will be the values of the **Carry** and **Sign** flags?
2. If **val4** is incremented by 1 using the ADD instruction, what will be the values of the **Overflow** and **Sign** flags?

Flags: Unsigned and Signed Integers Operations

- **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
```

Flags: Unsigned and Signed Integers Operations

- **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
```

~~MSB~~ 80 -
~~100 0000~~
 92
~~1001 0010~~
 msb.

Signed
 80 → negative
 92 → negative
 + 12
 17 / 16 = 1
 rem = 1

~~MSB~~ 12
~~1001 0010~~

Flags: Unsigned and Signed Integers Operations

- **Overflow and Carry Flags**

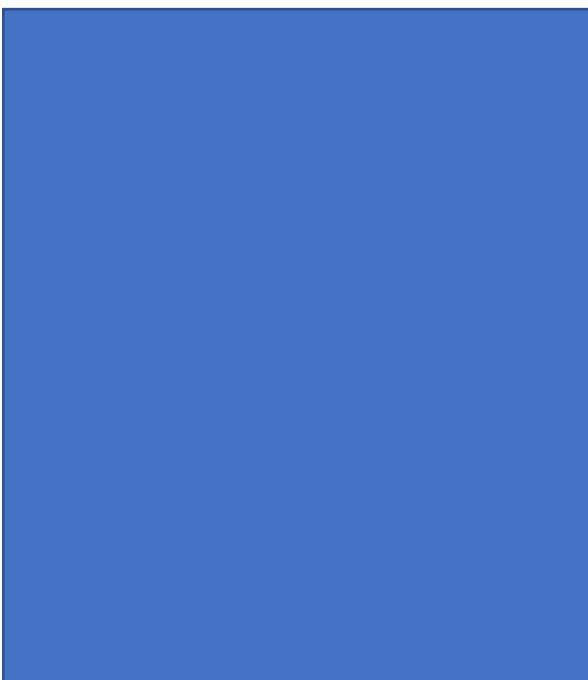
- What will be the values of the given **flags** after each operation?

```
mov al,-128  
neg al
```

```
mov ax,8000h  
add ax,2
```

```
mov ax,0  
sub ax,2
```

```
mov al,-5  
sub al,+125
```



1
0
0

a larger signed integer
is subtracted from a
smaller one

A = 10
B = 11

Registers

EAX = 00A6FD7E EBX = 00656000 ECX = 00961005 EDX = 00961005
 ESI = 00961005 EDI = 00961005 EIP = 00961014 ESP = 00A6FD94
 EBP = 00A6FDA0 EFL = 00000A16

OV = 1 JP = 0 EI = 1 PL = 0 ZR = 0 AC = 1 PE = 1 CY = 0

signed

FB (-Ve)
83 (-Ve)
7F (+Ve)

$23/16 = 1$
 $\text{rem} = 7$

Source.asm

```

1 .386
2 .MODEL flat, stdcall
3 .STACK 4096
4 ExitProcess Proto, dwExitCode:DWORD
5
6 .DATA
7
8
9
10 .CODE
11 main PROC
12     mov al, -5
13     sub al, +125
14     INVOKE ExitProcess, 0
15 main ENDP
16 END main
17
18
19

```

Handwritten Calculations:

z's complement of 125: $11111010 + 1 = 10000010$

$-5 - (125) = -5 + (-125)$

Watch 1

Name	Type	Value
var1		identifier "var1" is undefined
var2		identifier "var2" is undefined

Error List

Code	Description
Entire Solution	0 Errors, 0 Warnings, 0 Messages

Source.asm Project4.list

```

1 .386
2 .MODEL flat, stdcall
3 .STACK 4096
4 ExitProcess Proto, dwExitCode:DWORD
5
6 .DATA
7
8
9
10 .CODE
11 main PROC
12     mov al, -5
13     sub al, +125
14     INVOKE ExitProcess, 0
15 main ENDP
16 END main
17
18
19

```

$-5 \rightarrow (FB)$

$\begin{array}{r} 000000101 \\ 11111010 \\ +1 \\ \hline 11111011 \end{array}$ z's complement

$(F) \quad B$

(-5)

$-5 - (125) = -5 + (-125)$

$\begin{array}{r} 125 \\ \hline 0111101 \\ +1 \\ \hline 10000010 \end{array}$

$\begin{array}{r} 83 \\ \hline 10000010 \\ +1 \\ \hline 10000011 \end{array}$

$83 (+)$

$\begin{array}{r} 1 \\ 7E \\ +ve \end{array}$

$23/16 = 1$
 $rem = 7$

$A = 10$
 $B = 11$

EAX = 00A6FDDE EBX = 00656000 ECX = 00961005 EDX = 00961005
ESI = 00961005 EDI = 00961005 EIP = 00961014 ESP = 00A6FD94
EBP = 00A6FDA0 EFL = 00000A16

OV = 1 UP = 0 EI = 1 PL = 0 ZR = 0 AC = 1 PE = 1 CY = 0

singed

FB (-ve)
83 (-ve)

$\begin{array}{r} 15 \\ 8 \\ \hline 23/16 = 1 \\ rem = 7 \end{array}$

73

