

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

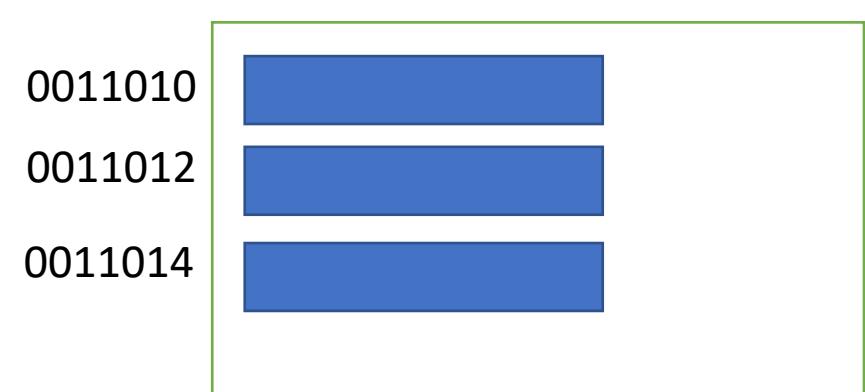
```
00000000 .data
00000000 00000000 sum DWORD 0
00000000 .code
00000000 main proc
00000000 B8 00000008      mov eax,8
00000005 83 C0 04          add eax,4
00000008 A3 00000000 R    mov sum, eax
```

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**.

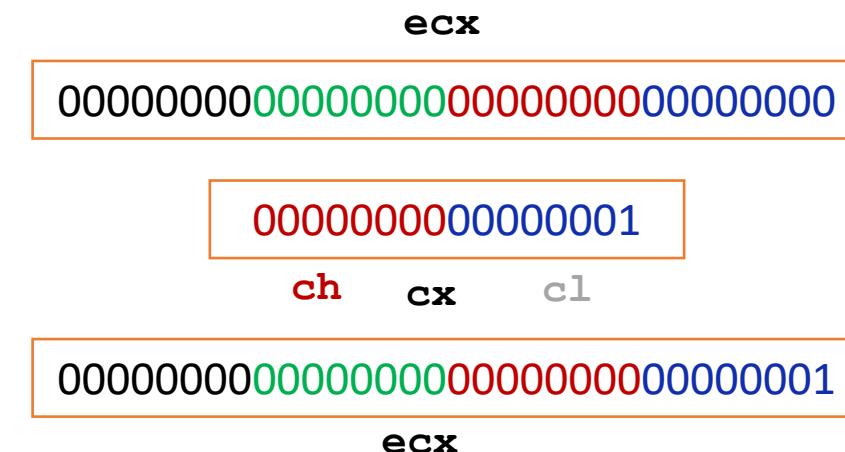
```
.data  
oneWord WORD 1234h  
oneDword DWORD 12345678h
```

```
.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

111111111111111111111111111111110000

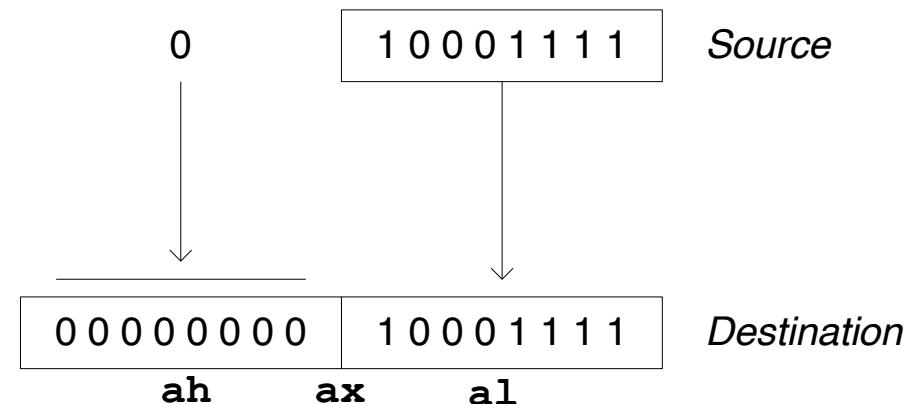
- **MOVZX** and **MOVSX** instructions to deal with both unsigned and signed integers.

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
```

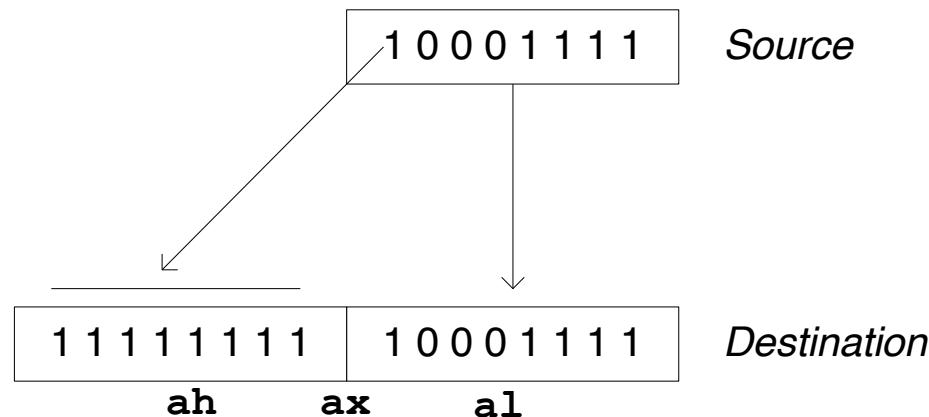
Data Transfer Instructions: Sign Extension (MOV_{SX})

- 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.

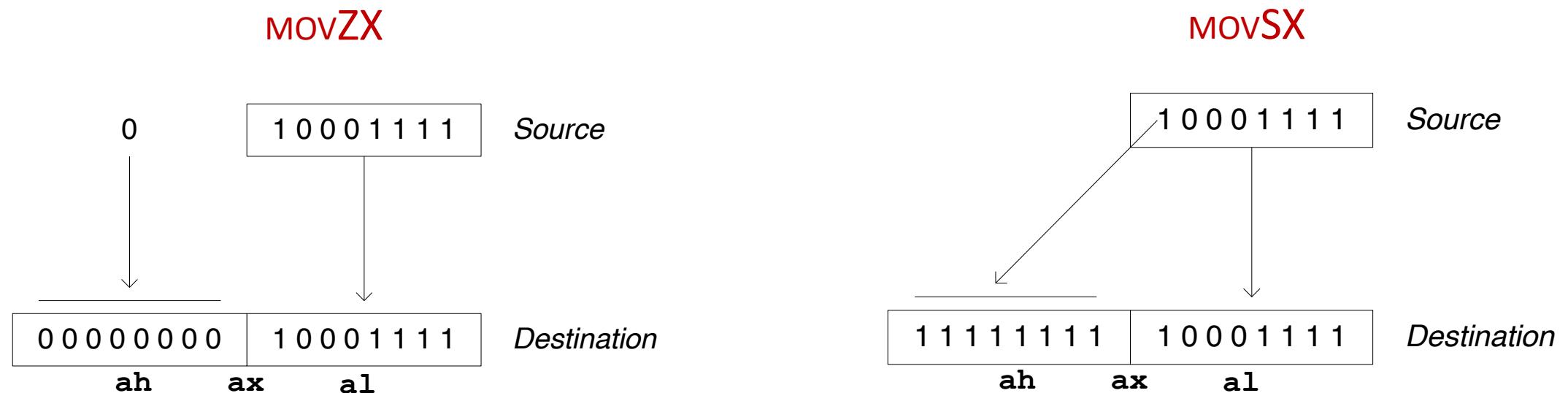
Data Transfer Instructions: Sign Extension (MOVSX)

- 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
```

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

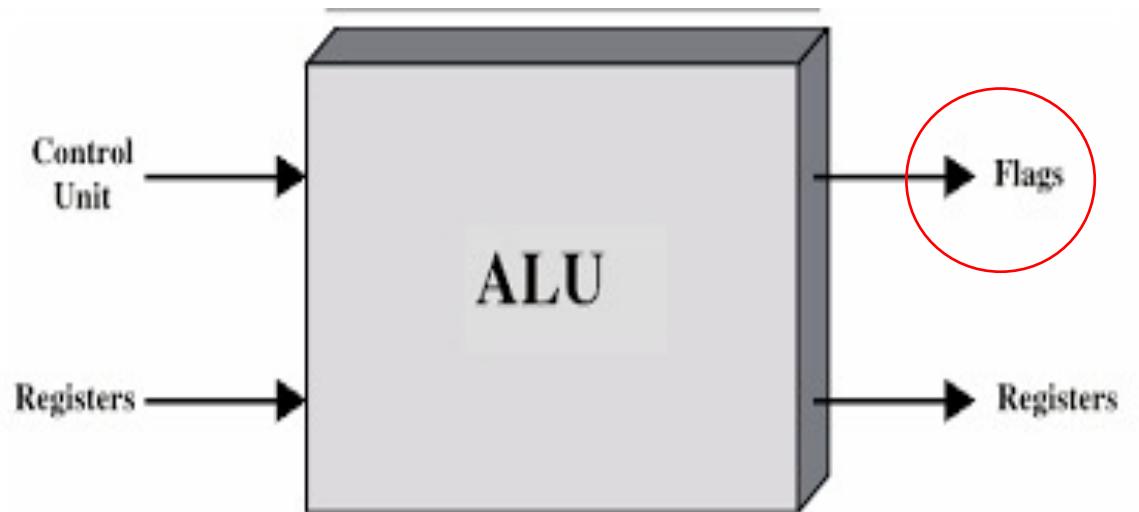
- INC and DEC Instructions
- ADD and SUB Instructions
- NEG Instruction
- Implementing Arithmetic Expressions
- Flags Affected by Arithmetic
 - Zero
 - Sign
 - Carry
 - Overflow

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.

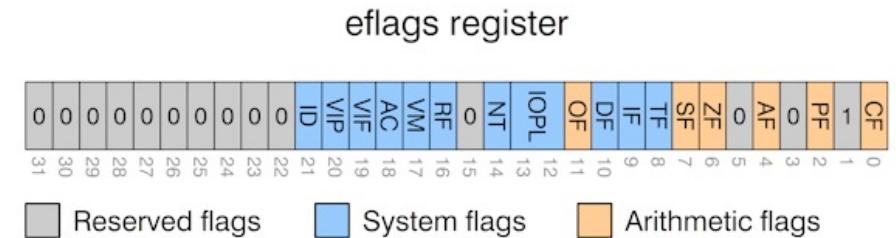
Flags: Flags Affected by Arithmetic/Bitwise

- 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?**

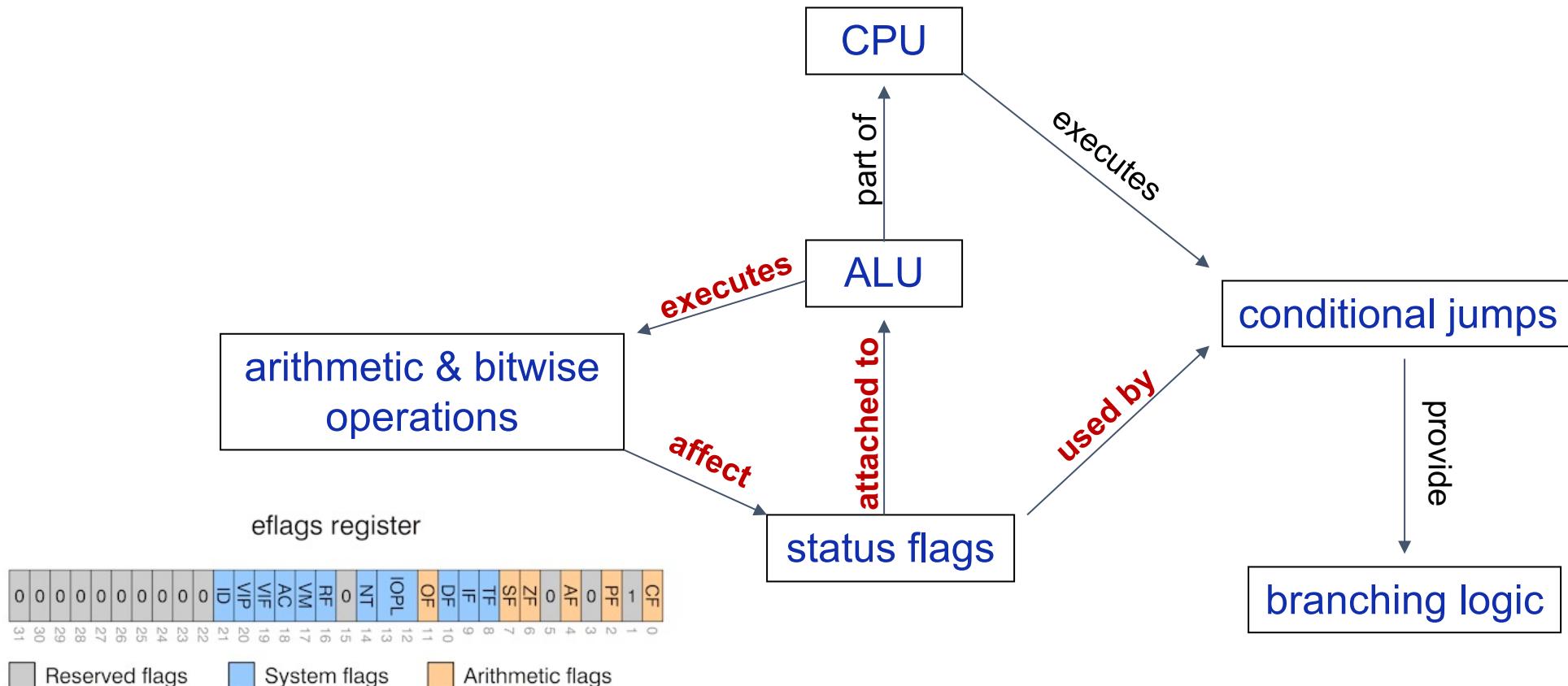


Flag Name	Overflow	Direction	Interrupt	Sign	Zero	Aux Carry	Parity	Carry
Symbol	OV	UP	EI	PL	ZR	AC	PE	CY

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...

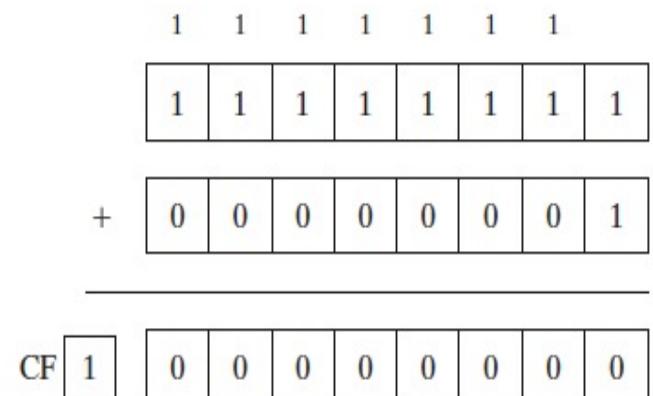
- A flag is **set** when it equals **1**.
- A flag is **clear** when it equals **0**.

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)
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: 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: 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

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

- **Overflow and Carry Flags**

- 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 ?
```

a larger signed integer
is subtracted from a
smaller one

Flags: NEG Instruction

- The **NEG instruction** produces an invalid result
 - If the destination operand cannot be stored correctly.

Ex1.

- If we move 128 to AL and try to negate it, the correct value (128) will not fit into AL.
- The Overflow flag is set, indicating that AL contains an invalid value:

```
mov al,-128          ; AL = 10000000b
neg al              ; AL = 10000000b, OF = 1
```

Flags: NEG Instruction

- The **NEG instruction** produces an invalid result
 - If the destination operand cannot be stored correctly.

Ex2.

- If 127 is negated, the result is valid and the Overflow flag is clear:

mov al,+127	; AL = 0111111b
neg al	; AL = 10000001b, OF = 0

Flags: NEG Instruction

- The **NEG instruction** produces an invalid result
 - If the destination operand cannot be stored correctly.

Ex2.

- If 127 is negated, the result is valid and the Overflow flag is clear:

mov al,+127	; AL = 0111111b
neg al	; AL = 10000001b, OF = 0

Outline

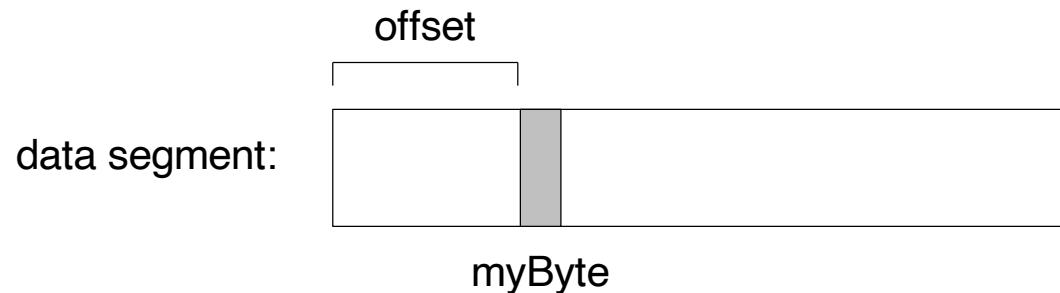
- Data Transfer Instructions
- Addition and Subtraction
- **Data-Related Operators and Directives**
- Indirect Addressing
- JMP and LOOP Instructions
- 64-Bit Programming

Operators and Directives

- **Operators** and **directives** are not **executable instructions**;
 - They are interpreted by the assembler.
 - OFFSET Operator
 - PTR Operator
 - TYPE Operator
 - LENGTHOF Operator
 - SIZEOF Operator

OFFSET Operator

- **OFFSET** returns the offset of the data label,
it represents the distance in bytes, of a label from the beginning of its enclosing segment
 - **Protected mode:** 32 bits
 - **Real mode:** 16 bits

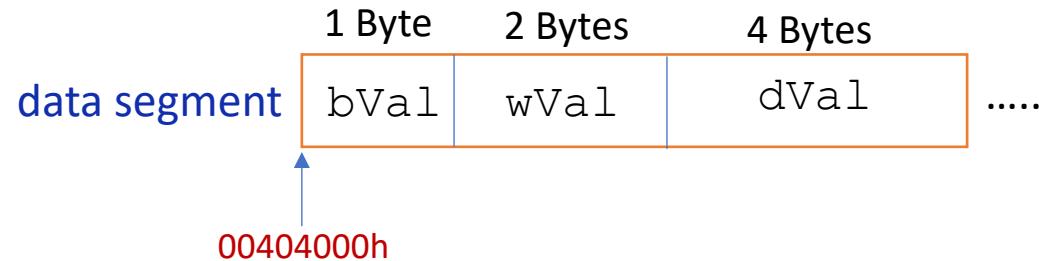


The **Protected-mode programs** we use only a single segment (**flat** memory model).

OFFSET: Examples

- Let's assume that the **data segment begins at 00404000h**:

```
.data  
bVal    BYTE  ?  
wVal    WORD  ?  
dVal    DWORD ?  
dVal2   DWORD ?
```



```
.code  
mov  esi,OFFSET bVal  
mov  esi,OFFSET wVal  
mov  esi,OFFSET dVal  
mov  esi,OFFSET dVal2
```

```
; ESI = 00404000  
; ESI = 00404001  
; ESI = 00404003  
; ESI = 00404007
```

This is a memory address
not a memory value

OFFSET: Examples

- OFFSET can also be applied to a **direct-offset operand**
- Suppose **myArray** contains five **16-bit words**
- The following **MOV** instruction
 - **Obtains the offset of myArray,**
 - Adds 4,
 - **Moves the resulting address to ESI.**
 - We can say that **ESI** points to the third integer in the array:

```
.data
myArray WORD 1,2,3,4,5
```

```
.code
mov esi,OFFSET myArray + 4      ESI = 00404000 + 4
```

OFFSET: Relating to C/C++

- The **value returned by OFFSET is a pointer**.
- Compare the following code written for both **C++** and **assembly** language:

```
// C++ version:  
  
char array[1000];  
char * p = array;
```

```
; Assembly language:  
  
.data  
array BYTE 1000 DUP(?)  
.code  
mov esi,OFFSET array
```

PTR Operator

- Overrides the default type of a label (variable).
- Provides the flexibility to access PART of a variable.
- Used in combination with BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD, or TBYTE
- Ex. PTR operator makes it possible to move the **low-order**

```
.data  
myDouble  DWORD  12345678h  
.code  
mov  ax,myDouble ; error
```

- But the **WORD PTR operator** makes it possible to move the low-order word (**5678h**) to AX:

```
mov  ax,WORD PTR myDouble
```

Why wasn't **1234h** moved into **AX**?

PTR Operator

Why wasn't 1234h moved into AX?

- x86 processors use the **little endian storage format**
- **low-order byte is stored at the variable's starting address**
- For example, the **doubleword 12345678h** would be stored as:

```
mov ax,WORD PTR myDouble  
(5678h)
```

byte	offset
78	0000
56	0001
34	0002
12	0003

When integers are loaded from memory into registers, the bytes are automatically re-reversed into their correct positions

PTR Operator: Example1

```
.data  
myDouble  DWORD  12345678h
```

doubleword	word	byte	offset
12345678	5678	78	0000 myDouble
		56	0001 myDouble + 1
	1234	34	0002 myDouble + 2
		12	0003 myDouble + 3

Memory Layout

```
mov al, BYTE PTR myDouble          ; AL = 78h  
mov al, BYTE PTR [myDouble+1]       ; AL = 56h  
mov al, BYTE PTR [myDouble+2]       ; AL = 34h  
mov ax, WORD PTR myDouble         ; AX = 5678h  
mov ax, WORD PTR [myDouble+2]       ; AX = 1234h
```

PTR Operator: Example2

- PTR can also be used to **combine elements of a smaller data type** and **move them into a larger operand**.

```
.data  
wordList WORD 5678h,1234h  
.code  
mov eax, DWORD PTR wordList ; EAX = 12345678h
```

PTR Operator: Exercise

- Use the following data definitions for Questions 6:

myBytes BYTE 10h,20h,30h,40h

myWords WORD 8Ah,3Bh,72h,44h,66h

myDoubles DWORD 1,2,3,4,5

myPointer DWORD myDoubles

6. Fill in the requested register values on the right side of the following instruction sequence:

```
mov esi,OFFSET myBytes
mov ax,[esi] ; a. AX =
mov eax,DWORD PTR myWords ; b. EAX =
mov esi,myPointer
mov ax,[esi+2] ; c. AX =
mov ax,[esi+6] ; d. AX =
mov ax,[esi-4] ; e. AX =
```

TYPE Operator

- The TYPE operator returns **the size, in bytes, of a single element** of a data declaration.

```
.data  
var1 BYTE ?  
var2 WORD ?  
var3 DWORD ?  
var4 QWORD ?  
  
.code  
mov eax, TYPE var1           ; 1  
mov eax, TYPE var2           ; 2  
mov eax, TYPE var3           ; 4  
mov eax, TYPE var4           ; 8
```

LENGTHOF Operator

- The LENGTHOF operator counts the number of elements in a single data declaration.

.data	LENGTHOF
byte1 BYTE 10,20,30	; 3
array1 WORD 30 DUP (?),0,0	; 32
array2 WORD 5 DUP(3 DUP (?))	; 15
array3 DWORD 1,2,3,4	; 4
digitStr BYTE "12345678",0	; 9
.code	
mov ecx, LENGTHOF array1	; 32

SIZEOF Operator

The **SIZEOF** operator **returns** a value that is equivalent to **multiplying LENGTHOF by TYPE**.

.data	SIZEOF
byte1 BYTE 10,20,30	; 3
array1 WORD 30 DUP (?),0,0	; 64
array2 WORD 5 DUP (3 DUP (?))	; 30
array3 DWORD 1,2,3,4	; 16
digitStr BYTE "12345678",0	; 9
.code	
mov ecx, SIZEOF array1	; 64

Outline

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

Indirect Addressing

- **Indirect Operands**
 - Array Sum Example
- Indexed Operands
- Pointers

Indirect Addressing (1)

- Direct addressing becomes infeasible for array access.
- `Mov eax, arrayD`
- `Mov eax, [arrayD+4]`
- `Mov eax, [arrayD+8]`
- There is no way to parameterize the constant offset (e.g. 4)

Indirect Addressing (2)

- Indirect addressing can solve the issue
- Consider the example.

.data

Val1 BYTE 11h, 33h, 55h

- Store the address of the array in any 32-bit register.
 - **mov esi, OFFSET val1**
- Then put that register inside the bracket to dereference it.
 - **mov al, [esi]**
 - Now Al contains 11h

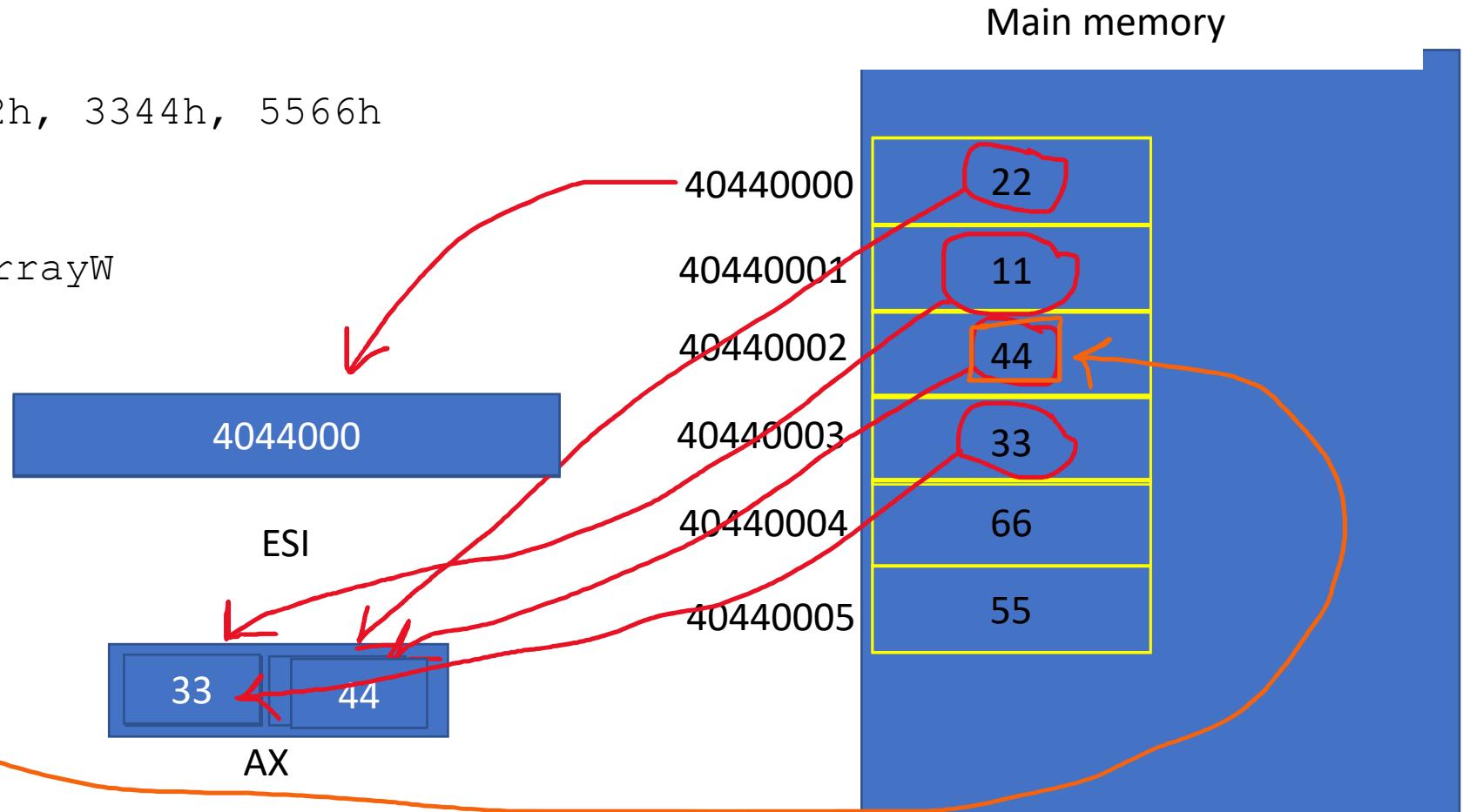
Indirect Addressing

```
.data  
arrayW WORD 1122h, 3344h, 5566h
```

```
mov esi,OFFSET arrayW
```

```
mov ax, [esi]  
add esi, 2  
mov ax, [esi]
```

$$4044000+2 = 4044002$$



Attendance!

Indirect Addressing: Indirect Operands

- An **indirect operand** holds the address of a variable, usually an **array** or **string**.
- It can be dereferenced (just like a pointer).

```
.data  
vall BYTE 10h,20h,30h  
.code  
mov esi,OFFSET vall  
mov al,[esi] ; dereference ESI (AL = 10h) 🤔  
  
inc esi  
mov al,[esi] ; AL = 20h  
  
inc esi  
mov al,[esi] ; AL = 30h
```

Indirect Addressing: **Indirect Operands**

- Use **PTR** to clarify the size attribute of a memory operand.

```
.data  
myCount WORD 7,3,2  
  
.code  
mov esi,OFFSET myCount  
inc [esi]  
inc WORD PTR [esi]
```



Indirect Addressing: Array Sum Example

- Indirect operands are **ideal for traversing an array**.
- Note that the register in brackets must be incremented by a value that matches the array type.

```
.data  
arrayW WORD 1000h,2000h,3000h  
.code  
    mov esi,OFFSET arrayW  
    mov ax,[esi]  
    add esi,2          ; or: add esi,TYPE arrayW  
add ax,[esi]  
    add esi,2  
add ax,[esi]        ; AX = sum of the array
```

ToDo: Modify this example for an array of doublewords.

Indirect Addressing

- Indirect Operands
 - Array Sum Example
- **Indexed Operands**
- Pointers

Indexed Operands

- Instead of storing the address in ESI register (or any 32-bit register)
 - You can use these registers as counter.
- An **indexed operand** adds a constant to a register to generate an effective address.
- There are two notational forms:

[label + reg] *label[reg]*

- [var1 + esi] OR var1[esi]

Indexed Operands

- An **indexed operand** adds a constant to a register to generate an **effective address**.
- There are two notational forms:

[label + reg] label[reg]

```
.data  
arrayW WORD 1000h,2000h,3000h  
.code  
    mov esi,0  
    mov ax,[arrayW + esi]           ; AX = 1000h  
    mov ax,arrayW[esi]             ; alternate format  
    add esi,2  
    add ax,[arrayW + esi]  
etc.
```

ToDo: Modify this example
for an array of **doublewords**.

Indirect Addressing

- Indirect Operands
- Array Sum Example
- Indexed Operands
- **Pointers**

Pointers

- You can declare a pointer variable that contains the offset of another variable.

```
.data  
arrayW WORD 1000h,2000h,3000h  
myptrW DWORD arrayW  
.code  
    mov esi, myptrW  
    mov ax, [esi]           ; AX = 1000h
```

Alternate format:

```
myptrW DWORD OFFSET arrayW
```

Outline

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

JMP and LOOP Instructions

- JMP Instruction
- LOOP Instruction
 - LOOP Example
- Summing an Integer Array
- Copying a String

JMP and LOOP Instructions

- CPU loads and executes programs **sequentially**
- Current **instruction** might be a one that transfer control to a different **memory address**

offset	machine code	source code
00000000	B8 00000000	mov eax,0
00000005	EB 03	jmp addone
00000007	83 C0 02	add eax,2
0000000A		addone:
0000000A	40	inc eax
0000000B	83 C0 01	add eax,4294967295

- Assembly language programs use **conditional instructions** to implement high-level statements
 - IF statements and Loops.

JMP and LOOP Instructions

- There are two basic **types of transfers**:
 - **Unconditional Transfer**
 - **Conditional Transfer**
- **Unconditional Transfer**: Control is transferred to a new location **in all cases**
 - New address is loaded into the instruction pointer, EIP,
 - causing execution to continue at the new address.
 - The **JMP** instruction does this.

JMP Instruction (Unconditional Transfer)

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

	offset	machine code	source code
EIP=	00000005	EB 03	<code>jmp addone</code>
EIP=	00000007	83 C0 02	<code>add eax,2</code>
	0000000A		<code>addone:</code>
	0000000A	40	<code>inc eax</code>

JMP Instruction (Unconditional Transfer)

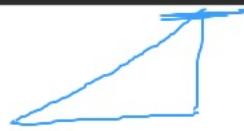
- JMP is **an unconditional jump** to **a label** that is **usually** within the same procedure.
- **Example:**

```
mov eax,0  
Jmp addone  
add eax,2  
addone:  
inc eax  
add eax, 4294967295 ; $2^{32} = 4294967296$ 
```

A **jump outside** the **current procedure** must be to a special type of label called **a global label** (see Section 5.5.2.3 for details).

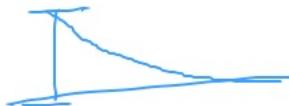
Try Visual Studio

```
for (i=0; i<10; i++) {  
    a=a+1;
```



10 times

```
}  
for (i=10; i>0; i--) {  
    a=a+1;
```



10 times.

```
}
```

LOOP Instruction (Conditional Transfer)

- **Conditional Transfer:** The program branches **if a certain condition is true**
 - A wide variety of conditional transfer **instructions** can be combined to create conditional logic structures.
- The **LOOP instruction** creates a counting loop
- Syntax: LOOP *target*
- Logic:
 1. ECX \leftarrow ECX – 1
 2. if ECX != 0, jump to *target*

The CPU interprets **true/false conditions** based on the contents of the **ECX** and Flags registers.

Source.asm* Project4.lst

```
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 EAX, 0
13     ; loop counter always goes to ECX register
14     mov ECX, 5
15
16 L1:
17     add EAX, 2
18     Loop L1
19
20     INVOKE ExitProcess, 0
21
22 main ENDP
23 END main
24
25
26
27
```

Build started...
----- Build started: Project: Proj
Assembling Source.asm...
Project4.vcxproj -> C:\Users\Zulkarnain\OneDrive\Desktop\Project4\Source.asm
===== Build: 1 succeeded, 0 failed

$ECX = ECX - 1$
check if ($ECX == 0$)
break the loop.
else continue loop.

LOOP: Example

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

```
    mov  ax,0  
    mov  ecx,5  
L1: add  ax,cx  
loop L1
```

Logic: $ECX \leftarrow ECX - 1$

if $ECX \neq 0$, jump to target

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	<u>How assembler calculates FB?</u>
0000000C	E2 FB	loop L1	
0000000E			

LOOP: Example (Conditional Transfer)

- The assembler calculates the **distance**, in bytes, between
 - the **offset** of the following instruction and
 - the **offset** of **the target label**.

offset	machine code	source code
00000009	66 03 C1	L1: add ax,cx
0000000C	E2 FB	loop L1
0000000E		

LOOP: Example (Conditional Transfer)

- When **LOOP** is assembled,
 1. The current location = **0000000E** (offset of the next instruction)
 2. Distance is
 - **-5 (FBh)**
 - **FBh** is added to the the current location,
 - causing a jump to location **00000009**:

Target Label location address = Offset of the next instruction + Distance
 $00000009 \leftarrow 0000000E + FB$
 3. The **relative offset** is added to **EIP** = **00000009**

LOOP: Example

If the **relative offset** is encoded in a single signed byte,

(a) what is the largest possible **backward** jump?

(b) what is the largest possible **forward** jump?

(a) -128

(b) +127

LOOP: Example

What will be the final value of AX?

10

How many times will the loop execute?

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

```
mov ecx, 0  
x2:  
    inc ax  
    loop x2
```

A common programming error is to inadvertently **initialize ECX to zero** before beginning a loop.

LOOP: Nested LOOP

```
for(i=10; i>0; i--) {
```

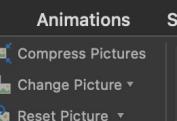
```
    a=a+1;
```

```
    for(j=5; j>0; j--) {
```

```
        a=a+b;
```

```
}
```

```
}
```



LOOP: Nested LOOP

```
for(i=10; i>0; i--) {
    a=a+1;
    for(j=5; j>0; j--) {
        a=a+1;
    }
}
```

3

Source.asm* | Project4.lst

```

7  a BYTE 0
8
9
10 .CODE
11 main PROC
12
13 ; Outer loop counter always goes to ECX register
14 mov ECX, 10
15
16 L1: ; outer loop starts here
17     inc a
18     ; keep a backup copy of the outer loop counter
19     mov edi, ecx
20
21     ; start the inner loop
22     mov ECX, 5
23     L2:
24         inc a
25     Loop L2 ; end of inner loop
26
27     ; restore the outer loop counter from edi
28     mov ecx, edi
29
30     Loop L1 ; end of outer loop
31
32
33
34 INVOKE ExitProcess, 0

```

ECX [10] → L1: ; outer loop starts here
 ECX = ECX - 1 → Loop L1 ; end of outer loop

Outerloop: 1, 2

Annotations:

- Yellow circle around the outer loop code (lines 14-31).
- Yellow box around the ECX register value in line 14.
- Yellow box around the ECX register value in line 29.
- Yellow arrow pointing from the ECX register value in line 14 to the ECX register value in line 29.
- Yellow box around the EDI register value in line 19.
- Yellow arrow pointing from the ECX register value in line 14 to the EDI register value in line 19.
- Yellow box around the ECX register value in line 22.
- Yellow arrow pointing from the ECX register value in line 14 to the ECX register value in line 22.
- Yellow box around the ECX register value in line 28.
- Yellow arrow pointing from the ECX register value in line 29 to the ECX register value in line 28.
- Yellow box around the ECX register value in line 31.
- Yellow arrow pointing from the ECX register value in line 29 to the ECX register value in line 31.
- Yellow box around the INVOKE ExitProcess, 0 instruction in line 34.



LOOP: 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 **3** times, and **the inner loop 2 times**.

```
.data  
count DWORD ?  
.code  
    mov ecx, 3 ; set outer loop count  
    L1:  
        mov count, ecx ; save outer loop count  
        mov ecx, 2 ; set inner loop count  
        L2: .  
            .  
            loop L2 ; repeat the inner loop  
            mov ecx, count ; restore outer loop count  
            loop L1 ; repeat the outer loop
```

JMP and LOOP Instructions

- JMP Instruction
- LOOP Instruction
 - LOOP Example
- **Summing an Integer Array**
- Copying a String

Array: 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  
→ 1    mov edi,OFFSET intarray      ; address of intarray  
2    mov ecx,LENGTHOF intarray   ; loop counter  
3    mov ax,0                      ; zero the accumulator  
4    L1:  
5    add ax,[edi]                  ; add an integer  
6    add edi,TYPE intarray        ; point to next integer  
7    loop L1                      ; repeat until ECX = 0
```

1. **Assign** the array's **address** to a register that will serve as an indexed operand.
2. **Initialize** the loop counter to **the length of the array**.
3. **Assign** zero to **the register that accumulates the sum**.
4. **Create** a label **to mark the beginning of the loop**.
5. In the loop body, **add** a single array element **to the sum**.
6. **Point** to the **next array element**.
7. **Use** a LOOP instruction to **repeat the loop**.

Array: Summing an Integer Array

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

bytes indirect addressing → access array items

```

.data
intarray WORD 100h, 200h, 300h, 400h
.code
1    mov edi, OFFSET intarray      ; address of intarray
2    mov ecx, LENGTHOF intarray   ; loop counter
3    mov ax, 0                     ; zero the accumulator
4    L1:                           ; label for loop
5    add ax, [edi]                 ; add an integer
6    add edi, TYPE intarray       ; point to next integer
7    loop L1                      ; repeat until ECX = 0

```

int a[] = ?
 sum = 0
 for(i=0; i<4; i++)
 sum += a[i]

- Assign the array's address to a register that will serve as an indexed operand.
- Initialize the loop counter to the length of the array.
- Assign zero to the register that accumulates the sum.
- Create a label to mark the beginning of the loop.
- In the loop body, add a single array element to the sum.
- Point to the next array element.
- Use a LOOP instruction to repeat the loop.

Source.asm* Project4.lst

```
4 ExitProcess Proto, dwExitCode:DWORD
5
6 .DATA
7 intarray WORD 100h, 200h, 300h, 400h
8
9
10 .CODE
11 main PROC
12
13 ; initialize index in esi
14 mov esi, 0
15
16 ; initialize the loop counter
17 mov ecx, Lengthof intarray
18
19 ; initialize sum
20 mov AX, 0
21
22 L1:
23     add AX, intarray[esi]
24
25     add esi, TYPE intarray
26
27 Loop L1
28
29
30
31     INVOKE ExitProcess, 0
```

Indexed operand
esi, 0
intarray[esi]

Array: Summing an Integer Array

What changes would you make to the program on the previous slide if you were summing a **doubleword** array?

```
intarray WORD 100h,200h,300h,400h
```

```
mov ax, 0
```

```
add ax, [edi]
```

JMP and LOOP Instructions

- JMP Instruction
- LOOP Instruction
 - LOOP Example
- Summing an Integer Array
- **Copying a String**

Copying a String

- Use a **loop** to **copy** a string, represented as an **array of bytes** with a **null terminator value**.

```
source    BYTE    "This is the source string", 0
```

- The target string must have enough available space to receive the copied characters, **including the null byte at the end** 😐:

```
target    BYTE    SIZEOF source DUP (0)
```

The **SIZEOF** operator **returns** a value that is equivalent to multiplying **LENGTHOF** by **TYPE**.

- **Indexed addressing** works well for this type of operation because the same index register references both strings.

```
mov    esi, 0           ; index register
mov    al, source[esi]   ; get char from source
mov    target[esi], al
```

Copying a String



- Use a **loop** to **copy** a string, represented as an **array of bytes** with a **null terminator value**.

- The target string must have enough available space to receive the copied characters, **including the null byte at the end** 😐:

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

The **SIZEOF** operator returns a value that is equivalent to multiplying **LENGTHOF** by **TYPE**.

- **Indexed addressing** works well for this type of operation because the same index register references both strings.

mov esi, 0 ; index register
mov al, source[esi] ; get char from source
mov target[esi], al

125

(MOV target[esi], source[esi])

Copying a String

- 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)

.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 it in the target
        inc    esi                      ; move to next character
        loop   L1                      ; repeat for entire string
```

good use of
SIZEOF

Copying a **String**

Rewrite the program shown in the previous slide, using
indirect addressing rather than indexed addressing.

Summary

- Data Transfer
 - MOV – data transfer from source to destination
 - MOVSX, MOVZX, XCHG
- Operand types
 - direct, direct-offset, indirect, indexed
- Arithmetic
 - INC, DEC, ADD, SUB, NEG
 - Sign, Carry, Zero, Overflow flags
- Operators
 - OFFSET, PTR, TYPE, LENGTHOF, SIZEOF, TYPEDEF
- JMP and LOOP – branching instructions