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
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

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

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

Data Transfer Instructions

- Register to Register

- Register to Memory , Vice versa

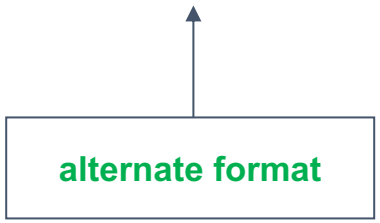
How to see variables in VS

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
```



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.

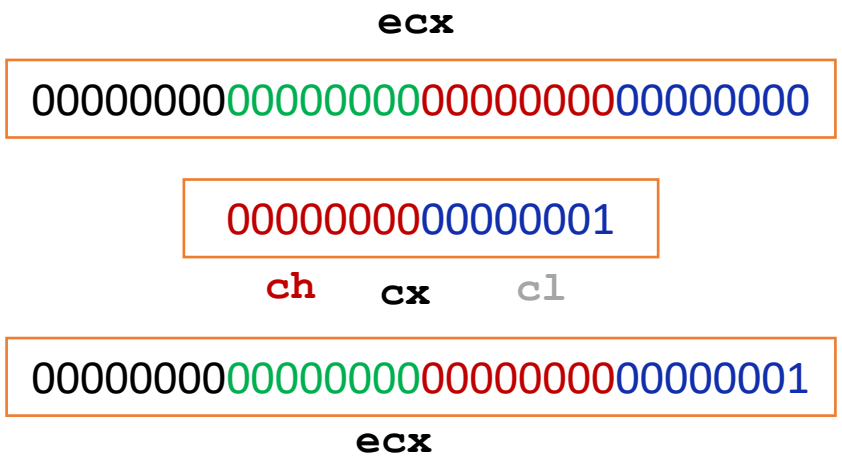
```
.data
oneWord WORD 1234h
oneDword DWORD 12345678h
```

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

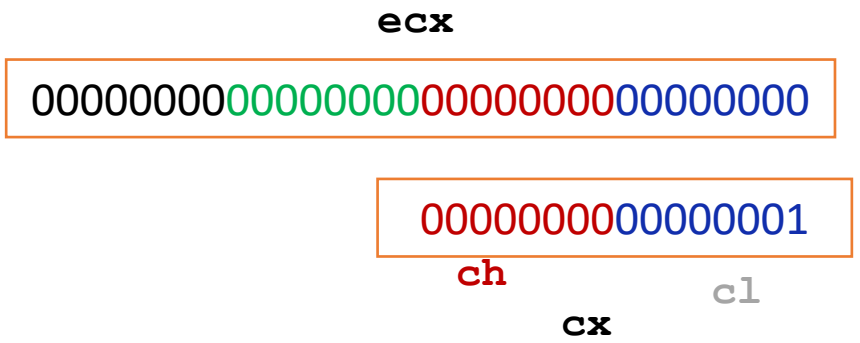
Data Transfer Instructions: Zero & Sign Extension

- **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
cl
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)
```

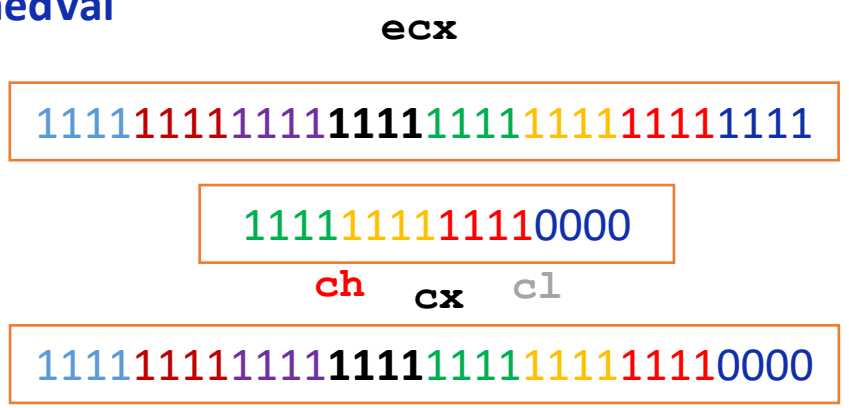
Data Transfer Instructions: Zero & Sign Extension

- **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,FFFFFFFFh
mov  cx,signedVal          ; ECX = FFFFFFF0h (-16)
```



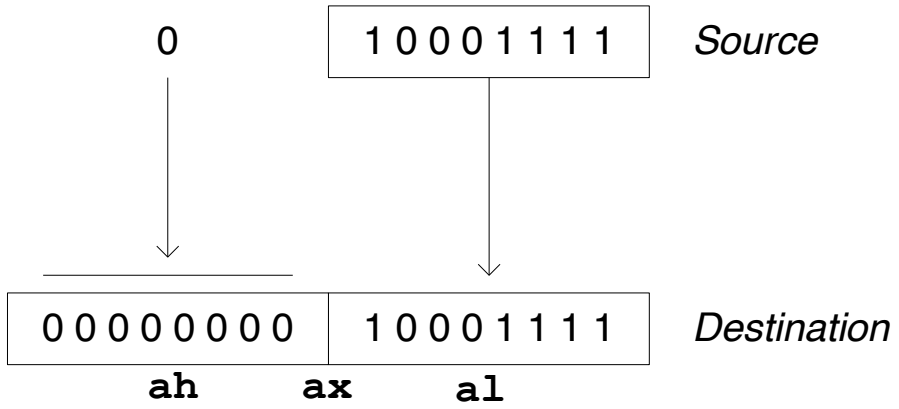
- **MOVZX** and **MOVSX** instructions to deal with both unsigned and signed integers.

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:

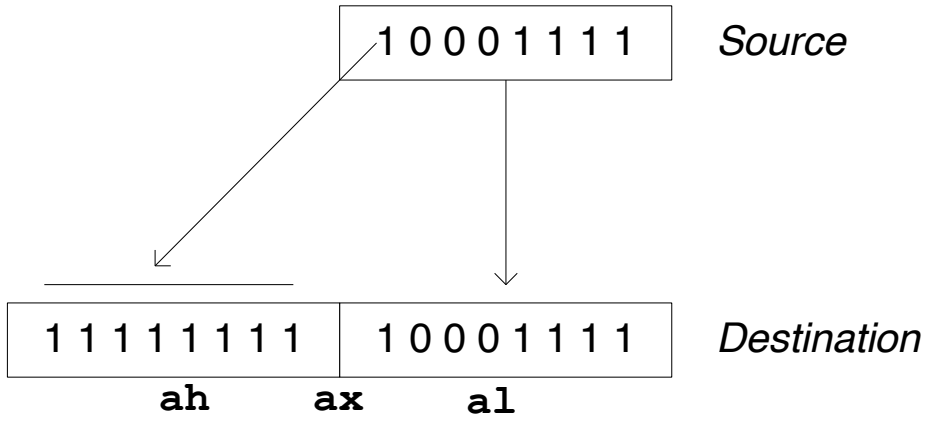
```
.data
byte1  BYTE 9Bh
word1  WORD 0A69Bh
.code
movzx  eax,word1               ; EAX = 0000A69Bh
movzx  edx,byte1               ; EDX = 0000009Bh
movzx  cx,byte1                ; CX  = 009Bh
```

Data Transfer Instructions: **Sign Extension (MOVSX)**

- The MOVSX instruction
- It fills the upper half of the destination with a copy of the source operand's sign bit.

```
MOVSX  reg32,reg/mem8
MOVSX  reg32,reg/mem16
MOVSX  reg16,reg/mem8
```

```
mov bl,10001111b
movsx ax,bl      ; sign extension
```



The destination must be a register.

Data Transfer Instructions: **Sign Extension (MOVSX)**

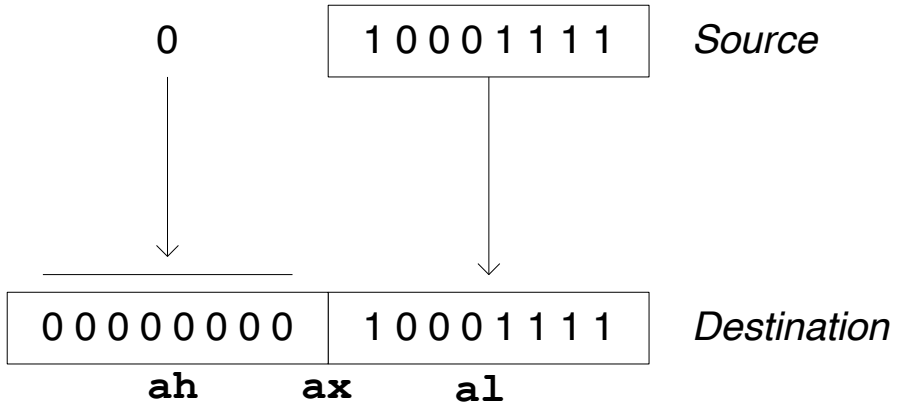
- In the following example,
  - the hexadecimal value moved to BX is A69B,
  - so the leading "A" digit tells us that the highest bit is set.

```
mov     bx, A69Bh
movsx eax,bx            ; EAX = FFFFA69Bh
movsx edx,bl            ; EDX = FFFFFF9Bh
movsx cx,bl             ; CX = FF9Bh
```

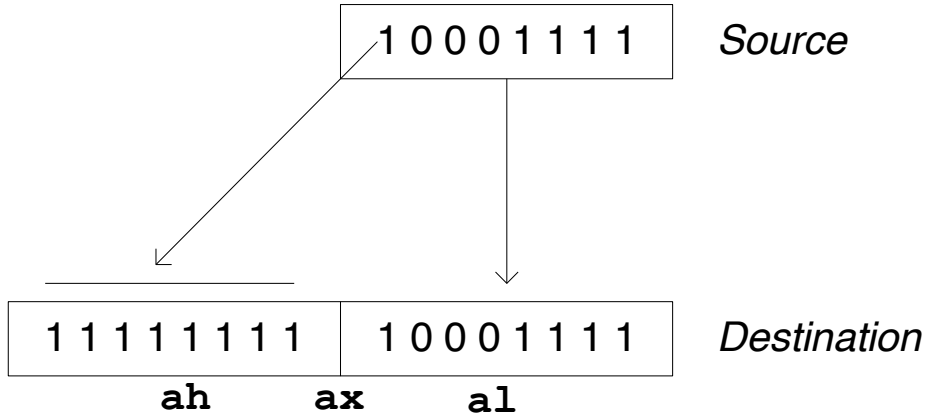
Data Transfer Instructions: Zero & Sign Extension

- MOVZX vs. MOVSX

MOVZX



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,val1
xchg ax,val2
mov val1,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
```

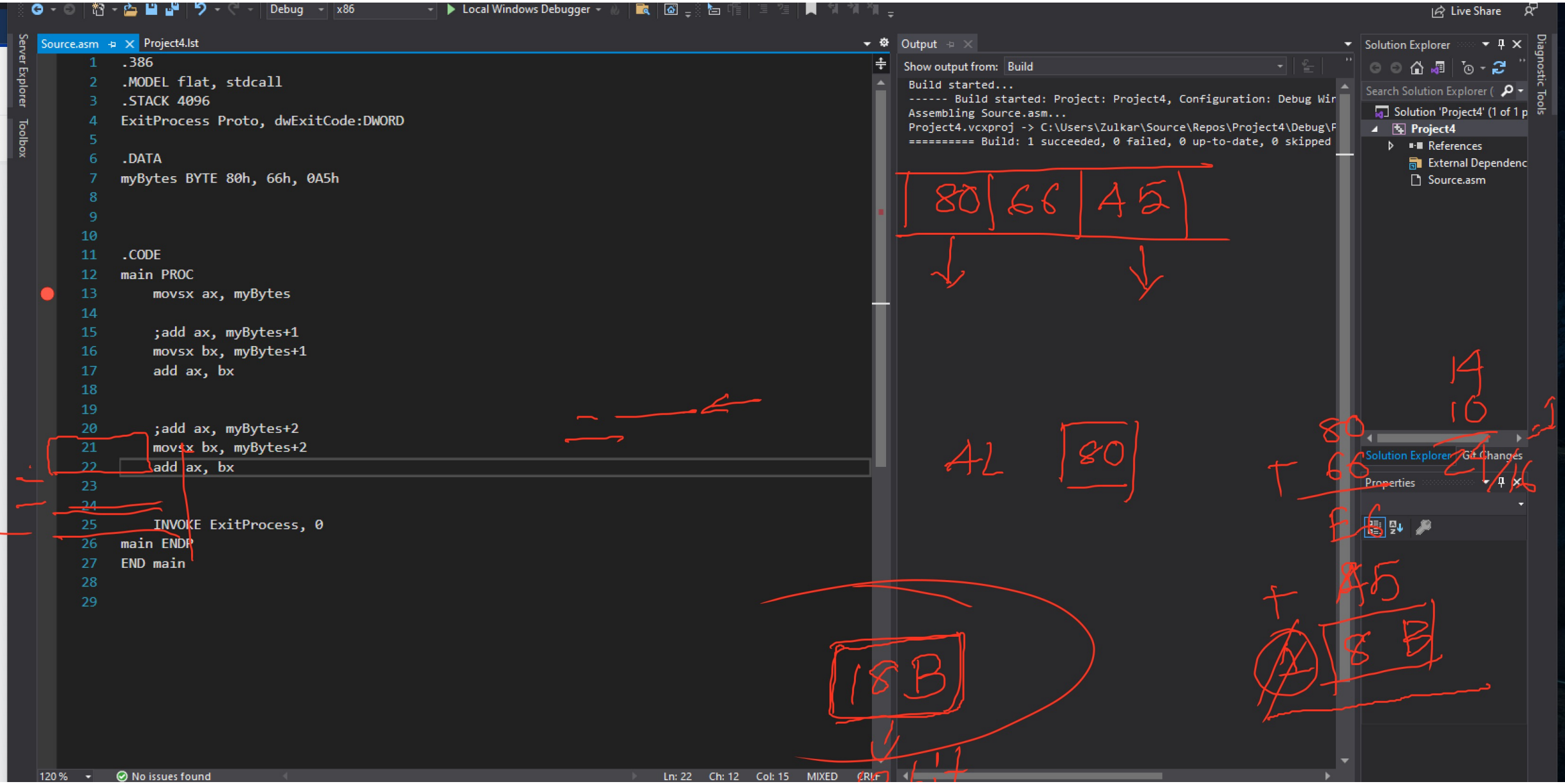
- What is your evaluation of the following code?

```
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?


Debug
x86
Local Windows Debugger
Live Share
Source.asm
Project4.lst
.386
.MODEL flat, stdcall
.STACK 4096
ExitProcess Proto, dwExitCode:DWORD
.DATA
myBytes BYTE 80h, 66h, 0A5h
.CODE
main PROC
movsx ax, myBytes
;add ax, myBytes+1
movsx bx, myBytes+1
add ax, bx
;add ax, myBytes+2
movsx bx, myBytes+2
add ax, bx
INVOKE ExitProcess, 0
main ENDP
END main
Output
Show output from: Build
Build started...
------ Build started: Project: Project4, Configuration: Debug Wir
Assembling Source.asm...
Project4.vcxproj -> C:\Users\Zulkar\Source\Repos\Project4\Debug\F
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped
Solution Explorer
Search Solution Explorer (
Solution 'Project4' (1 of 1 p
Project4
References
External Dependenc
Source.asm
Diagnostic Tools
Server Explorer
Toolbox
Classes
120 %
No issues found
Ln: 22
Ch: 12
Col: 15
MIXED
CRLF
Solution Explorer
Git Changes
Properties

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

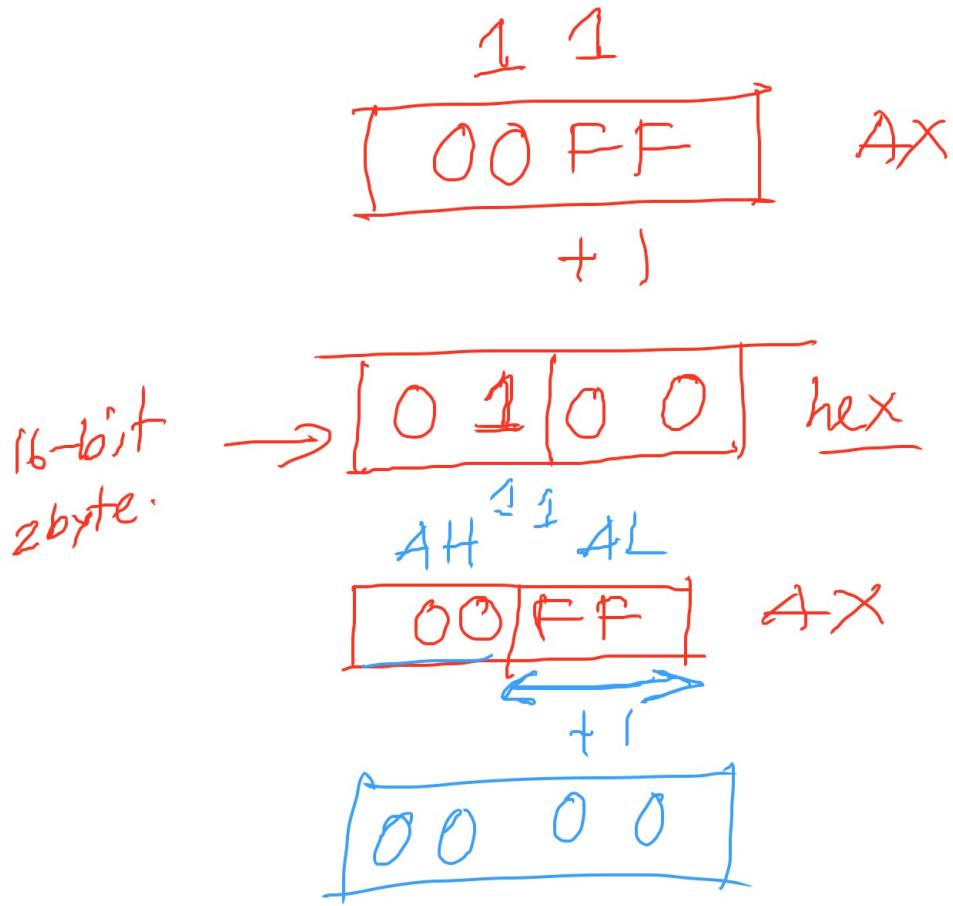
Addition and Subtraction: INC and DEC Examples

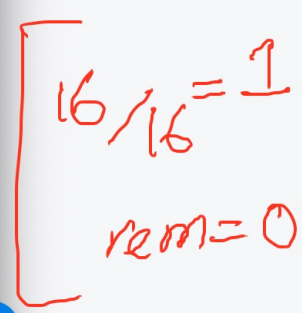
```
.data
myWord  WORD 1000h
myDword DWORD 10000000h
.code
        inc myWord
        dec myWord
        inc myDword

        mov ax,00FFh
        inc ax
        mov ax,00FFh
        inc al
```

Addition and Subtraction: INC and DEC Examples

```
.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* ← *destination* + source
- SUB destination, source
  - Logic: *destination* ← *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                                   ; ---EAX---
        mov eax,var1                    ; 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

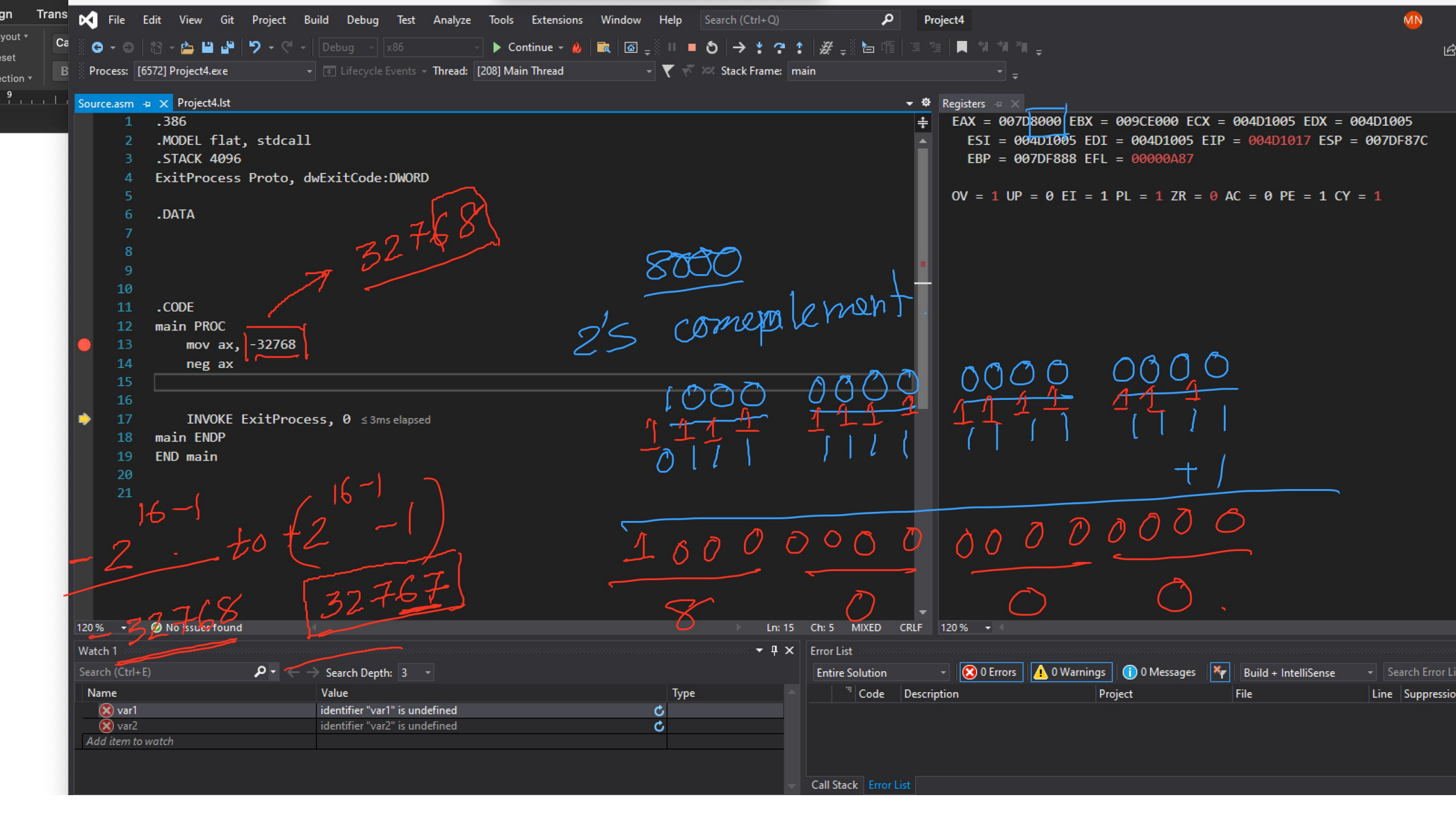
- **Reverses** the **sign** of an operand.
- Operand can be a register or memory operand.

```
NEG reg
NEG mem
```

Ex.

```
.data
valB SBYTE -1                    ;valB BYTE -1 is also acceptable, why?
valW WORD +32
.code
    mov al,valB                  ; AL = -1
    neg al                       ; AL = +1
    neg valW                     ; valW = -32
```

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


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Project4
Debug
x86
Continue
Process: [6572] Project4.exe
Lifecycle Events
Thread: [208] Main Thread
Stack Frame: main
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
11 .CODE
12 main PROC
13 mov ax, -32768
14 neg ax
15
16
17 INVOKE ExitProcess, 0 ≤ 3ms elapsed
18 main ENDP
19 END main
20
21
32768
8000
2's complement
1000 0000
1111 1111
0000 0000
1111 1111
+1
1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
8 0
0 0
16-1 16-1
-2 to +(2 -1)
-32768
32767
Registers
EAX = 007D8000 EBX = 009CE000 ECX = 004D1005 EDX = 004D1005
ESI = 004D1005 EDI = 004D1005 EIP = 004D1017 ESP = 007DF87C
EBP = 007DF888 EFL = 00000A87
OV = 1 UP = 0 EI = 1 PL = 1 ZR = 0 AC = 0 PE = 1 CY = 1
120 %
No issues found
Ln: 15 Ch: 5 MIXED CRLF
120 %
Watch 1
Search (Ctrl+E)
Search Depth: 3
Name Value Type
var1 identifier "var1" is undefined
var2 identifier "var2" is undefined
Add item to watch
Error List
Entire Solution
0 Errors
0 Warnings
0 Messages
Build + IntelliSense
Search Error Li
Code Description Project File Line Suppressio
Call Stack
Error List

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
```

windows10_fall21 [Running]

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl+Q) Project4 MN

Debug x86 Local Windows Debugger Live Share

Source.asm Project4.lst

```
1   .386
2   .MODEL flat, stdcall
3   .STACK 4096
4   ExitProcess Proto, dwExitCode:DWORD
5
6   .DATA
7
8   xval WORD 12h
9   yval DWORD 13h
10  zval BYTE 14h
11
12
13
14  .CODE
15  main PROC
16      mov eax, 10h
17
18      movsx ebx, xval
19      sub ebx, eax
20
21      ;sub yval, zval
22      movsx ecx, zval
23      sub yval, ecx
24
25      add yval, ebx
26
27      ;mov zval, yval
28      mov edx, yval
29      mov zval, dl
30
31      INVOKE ExitProcess, 0
32  main ENDP
33  END main
34
35
```

EAX = 10h

zval = (xval - zval) + (xval - eax)

xval

ebx

* inputs always same size
* Atleast one register.

Output

Show output from: Build

```
Build started...
------ Build started: Project: Project4, Configuration: Debug Win32 ------
Assembling Source.asm...
Project4.vcxproj -> C:\Users\Zulkar\Source\Repos\Project4\Debug\Project4.exe
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Solution Explorer

Search Solution Explorer

Solution 'Project4' (1 of 1

Project4

References

External Dependen

Source.asm

Solution Explorer Git Change

Properties

Server Explorer Toolbox

Implementing **Arithmetic Expressions**

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

Rval = Xval - (-Yval + Zval)

Previous Example:

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
```

Lecture-14-Chapter-4-Data-Transfer-Addressing-Arithmetic-with-annotations

You are screen sharing | Stop Share | Share

Mail - Md S Q Zulkar Nine - | Google Calendar - Week of | Homepage - COMPUTER O

View | Acrobat | Convert to SmartAct | Picture | Shapes | Text Box

nting Arithmetic Expres

e following expression into assembly language

permit Xval, Yval, or Zval to be modified:

Rval = Xval - (-Yval + Zval)

e 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,Xv
sub eax,ebx
mov Rval,eax
```

52

windows10_fall21 [Running]

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help | Search (Ctrl+Q) | Project4

Debug | x86 | Local Windows Debugger

Source.asm | Project4.lst

```
1  .386
2  .MODEL flat, stdcall
3  .STACK 4096
4  ExitProcess Proto, dwExitCode:DWORD
5
6  .DATA
7
8  Rval SDWORD ?
9  Xval SDWORD 26
10 Yval SDWORD 30
11 Zval SDWORD 40
12
13
14
15 .CODE
16 main PROC
17    ;sub zval, yval
18    neg yval
19    mov eax, yval
20    add eax, zval
21
22    sub xval, eax
23
24    mov eax, xval
25    mov rval, eax
26
27     INVOKE ExitProcess, 0
28 main ENDP
29 END main
30
31
```

Output

Show output from: Build

```
Build started...
------ Build started: Project: Project4, Configuration: Debug Win32 ------
Assembling Source.asm...
Project4.vcxproj -> C:\Users\Zulkar\Source\Repos\Project4\Debug\Project4.
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Rval = Xval - (-Xval + Zval)

EAX + Zval

EAX

120 % | No issues found | Ln: 25 | Ch: 14 | TABS | CRLF

Attendance!

Addition and Subtraction

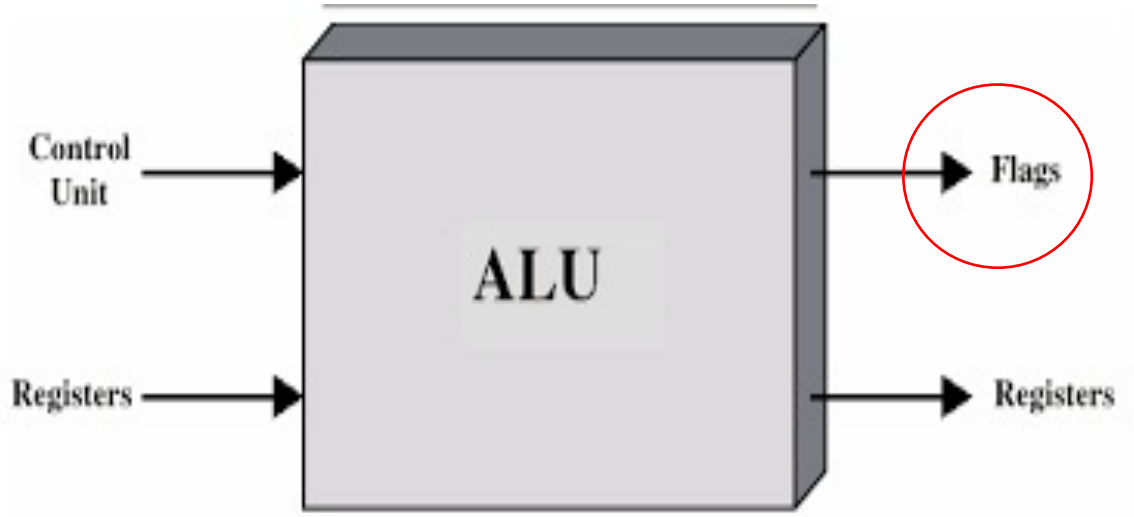
- INC and DEC Instructions
- ADD and SUB Instructions
- NEG Instruction
- Implementing Arithmetic Expressions
- Flags Affected by Arithmetic
  - Zero
  - Sign
  - Carry
  - Overflow

Flags: Why?

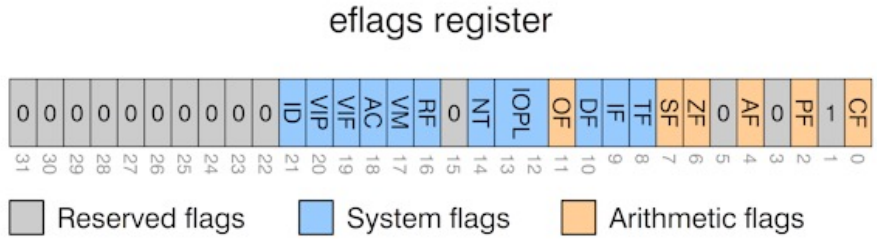
- When executing **arithmetic instructions**, we often want to know something about the result:
  - Is it negative, positive, or zero?
  - Is it **too large** or **too small** to fit into the destination operand?
- Answers to such questions can help detect calculation errors that might otherwise cause 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

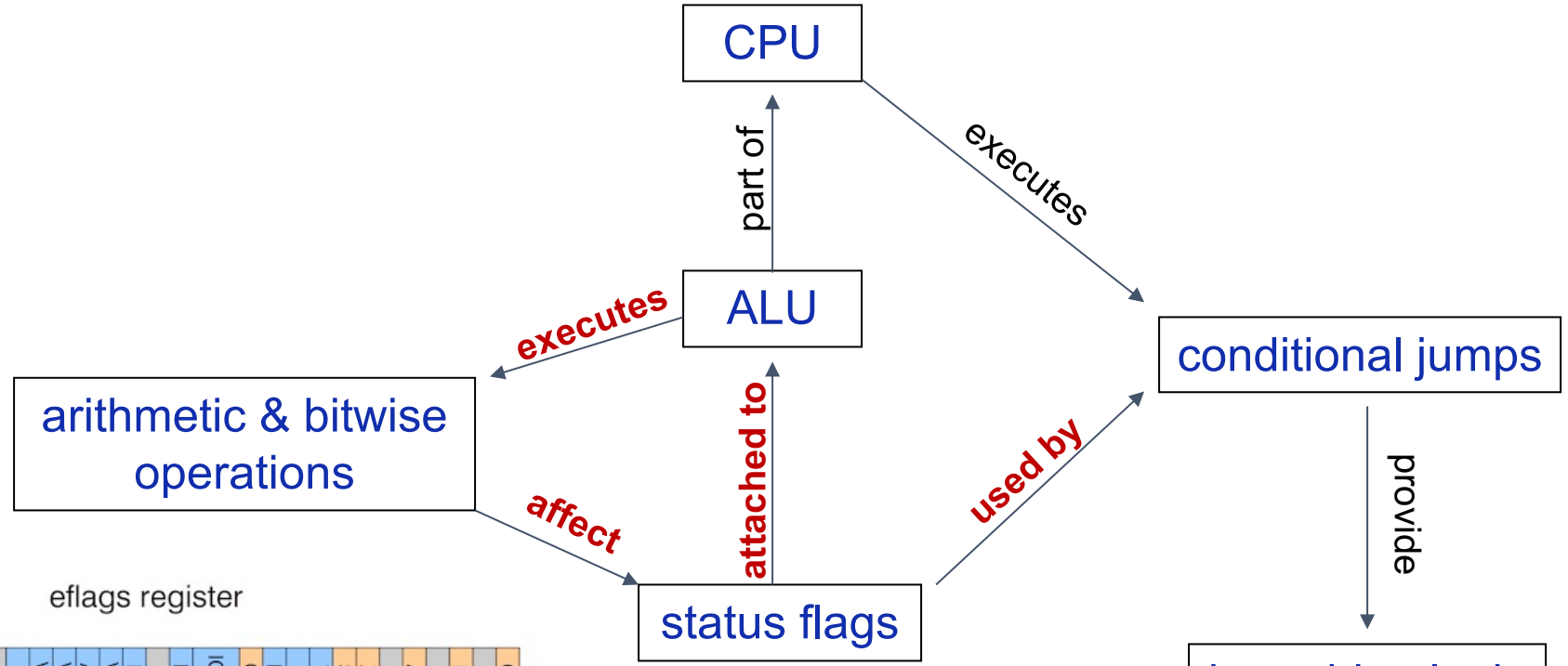
- **Visual Studio Flags?**

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**

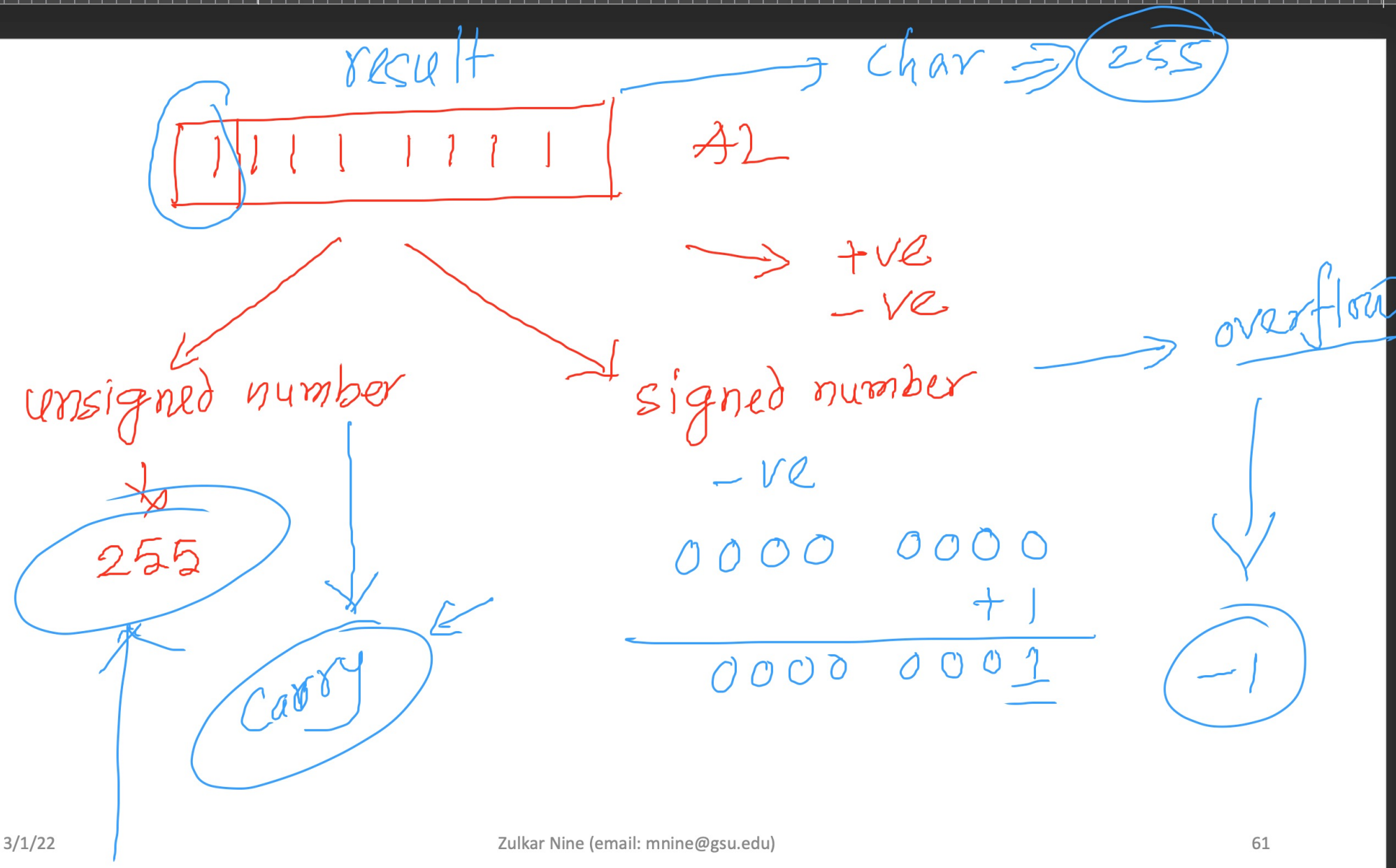
- **You must be currently debugging a program in order to see these menu options.**
- To display the CPU status flags **during a debugging session**,
  - select Windows from the Debug menu,
  - then select Registers from the Windows menu.
  - Inside the Registers window, right-click and select Flags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


result
char
255
A2
+ve
-ve
unsigned number
signed number
overflow
255
carry
-ve
0000 0000
+1
0000 0001
-1

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          ; ECX = 0, ZF = 1
mov eax,FFFFFFFFh
inc eax            ; EAX = 0, ZF = 1
inc eax            ; EAX = 1, ZF = 0
```

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**.

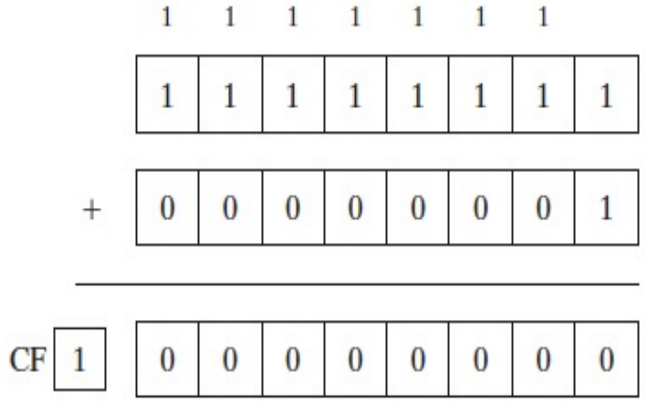
Flags: Unsigned Operations, **Carry flag (CF)**

- 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

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

Figure 4–3 Adding 1 to 0FFh sets the Carry flag.



Flags: Un

- The **Carry** value that is
- Addition .V
- **Addition a**

mov
add

windows10_fall21 [Running]

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl+Q) Project4

Debug x86 Continue

Process: [4960] Project4.exe Lifecycle Events Thread: [5596] Main Thread Stack Frame: main

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
11
12 .CODE
13 main PROC
14    mov al, 0FFh
15     add al, 1
16
17    INVOKE ExitProcess, 0 ≤1ms elapsed
18 main ENDP
19 END main
20
21
```

Registers

```
EAX = 010FFF00 EBX = 00F2E000 ECX = 00C91005 EDX = 00C91005 ESI = 00C9100
  EIP = 00C91014 ESP = 010FFED4 EBP = 010FFEE0 EFL = 00000257

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

AL

MSB

1 1 1 1 1 1 1 1 ⇒

carry: unsigned.

255 +1 = 256

8 bit: 0 - 2 - 1

0 - 255

-ve 00000000 +1

00000001

-128 to +127

Watch 1 Search (Ctrl+E) Search Depth: 3

Name	Value	Type
var1	identifier "var1" is undefined	
var2	identifier "var2" is undefined	
Add item to watch		

Error List: Entire Solution 0 Errors 0 Warnings 0 Messages Build + Intel

Code	Description	Project	File

Call Stack Error List

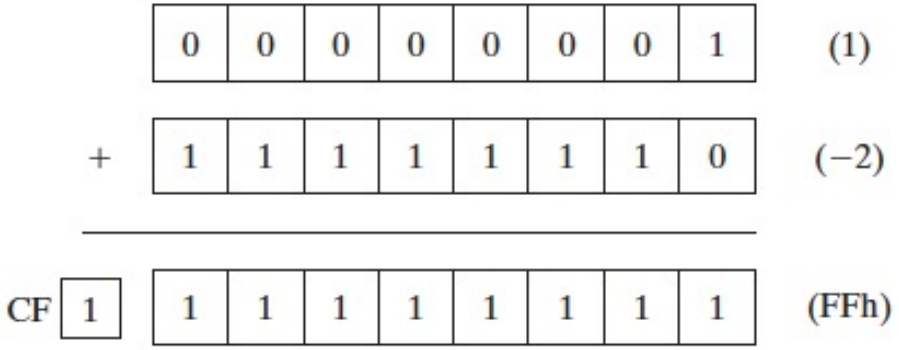
120 % No issues found Ln: 17 Ch: 1 TABS CRLF 120 %

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.

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

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



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.