

# CSC 3210

## Computer Organization and Programming

### Lab Work 2

Dr. Zulkar Nine

[mnine@gsu.edu](mailto:mnine@gsu.edu)

Georgia State University

Fall 2021

# Lab Work 2 Instructions

- Lab 2(a): Run a sample assembly language code (3 points)
- Lab 2(b): Math problems (2 points)
- Lab 2(c): Registers and Memory related Problems (5 points)

Due Date: posted on iCollege

# Lab 2(a)

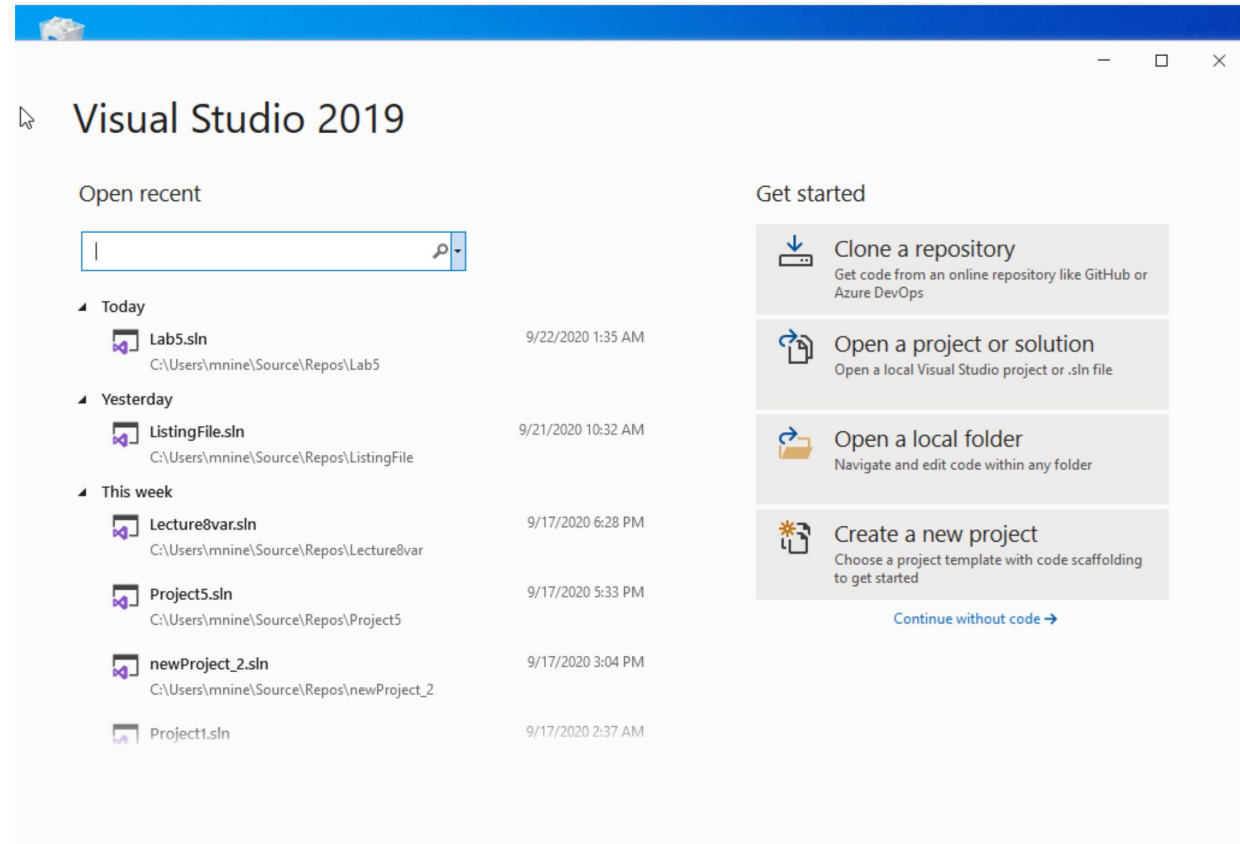
Run a sample assembly code in Microsoft visual Studio

# Lab 2(a) Instructions

- Follow the instructions to run the first program in assembly language.
- Take screenshot/screenshots showing the code and the register contents.
- Submit to the iCollege

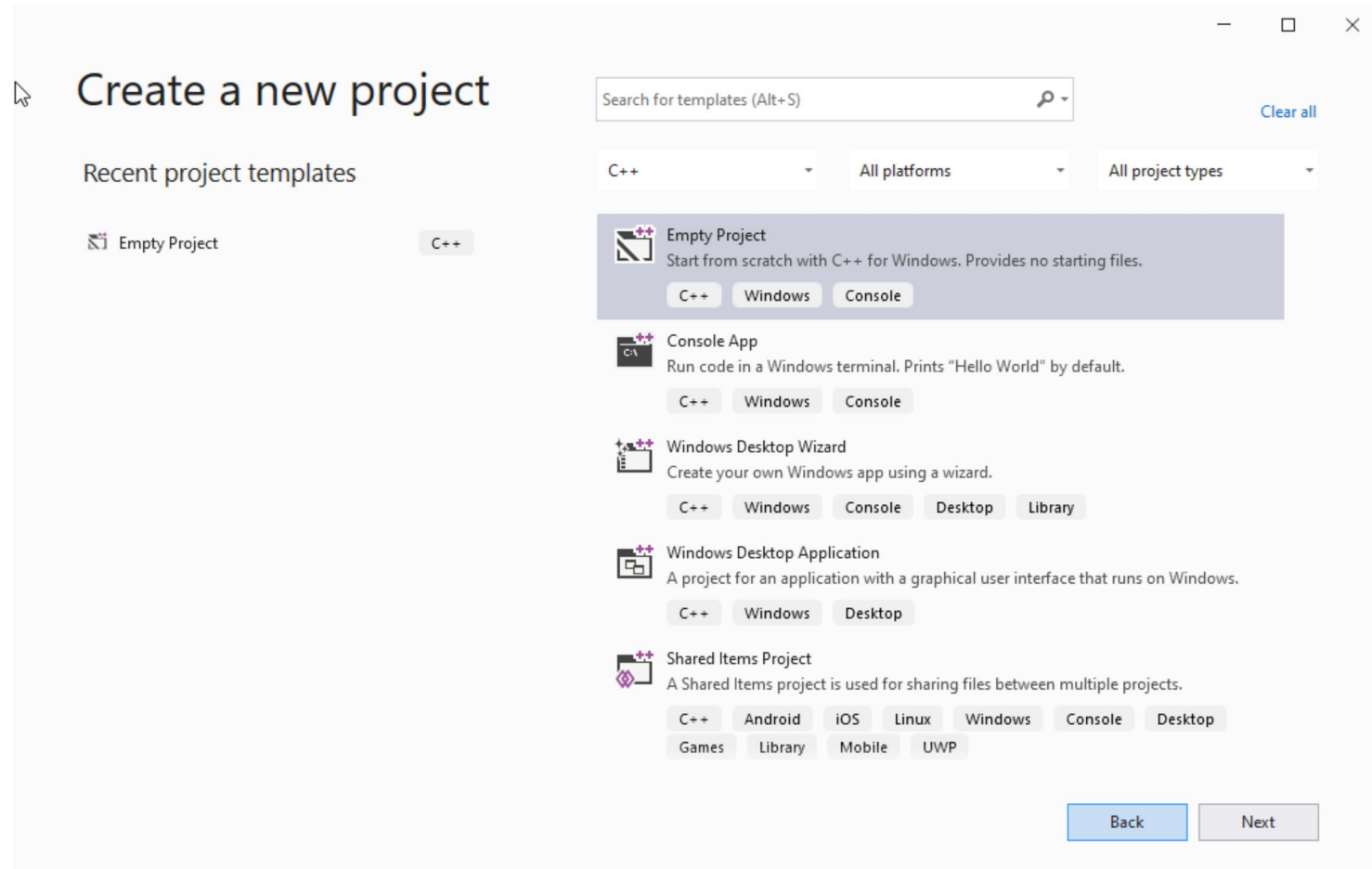
# Create a new Project

- (1) Start Visual Studio
- (2) Click Create a new Project



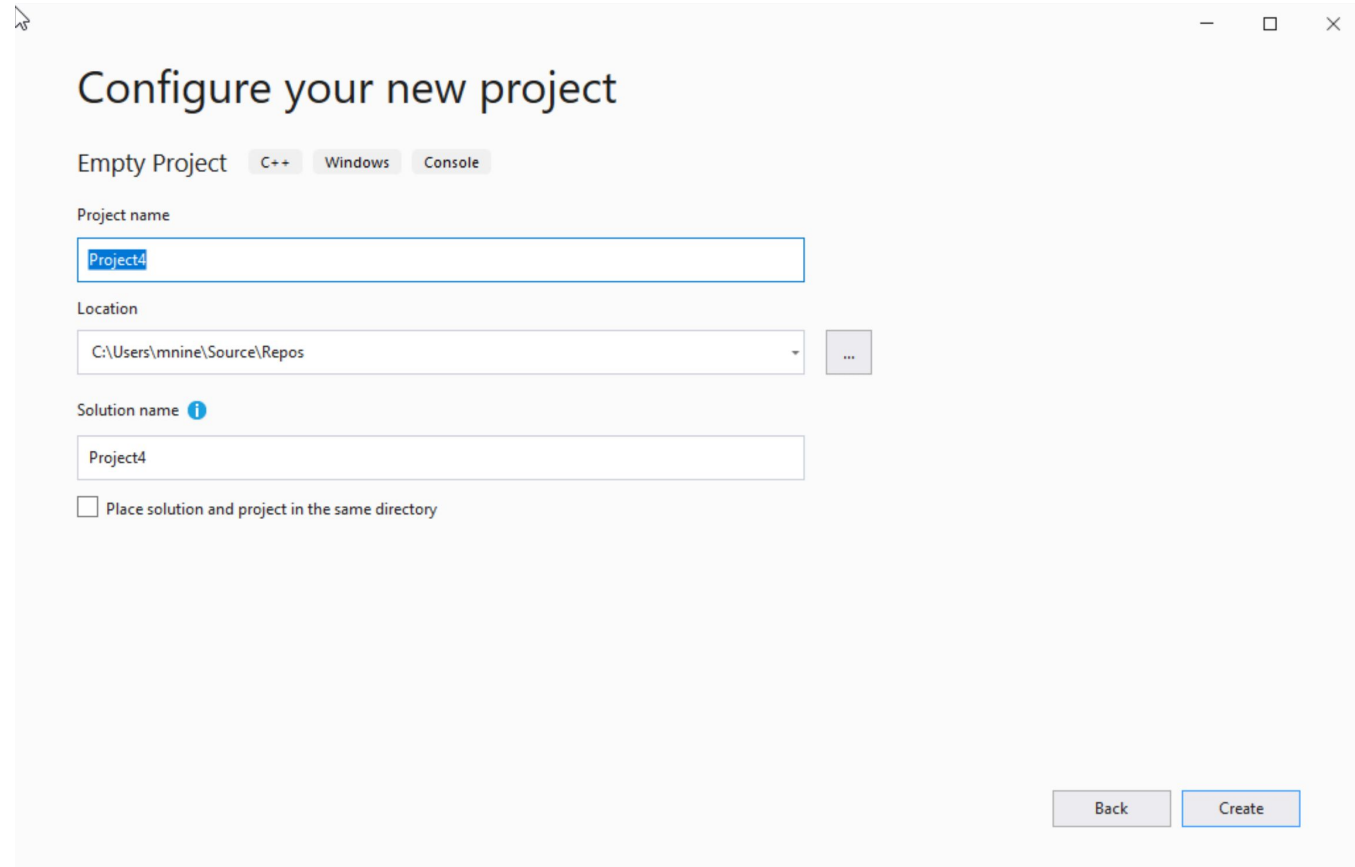
# Create a new Project

- (1) Select C++ as language
- (2) Select Empty Project
- (3) Click Next



# Create a new Project

- (1) You can change the project name as you like
- (1) Also you can change the project location
- (2) Click Next



The screenshot shows the 'Configure your new project' dialog box in Visual Studio. The title bar includes standard window controls. The main heading is 'Configure your new project'. Below it, there are three tabs: 'Empty Project' (selected), 'C++', 'Windows', and 'Console'. The 'Project name' field contains 'Project4'. The 'Location' field shows the path 'C:\Users\mnine\Source\Repos' with a dropdown arrow and a browse button ('...'). The 'Solution name' field, which has an information icon, also contains 'Project4'. At the bottom, there is a checkbox labeled 'Place solution and project in the same directory' which is currently unchecked. At the bottom right, there are two buttons: 'Back' and 'Create'.

# Create a new Project

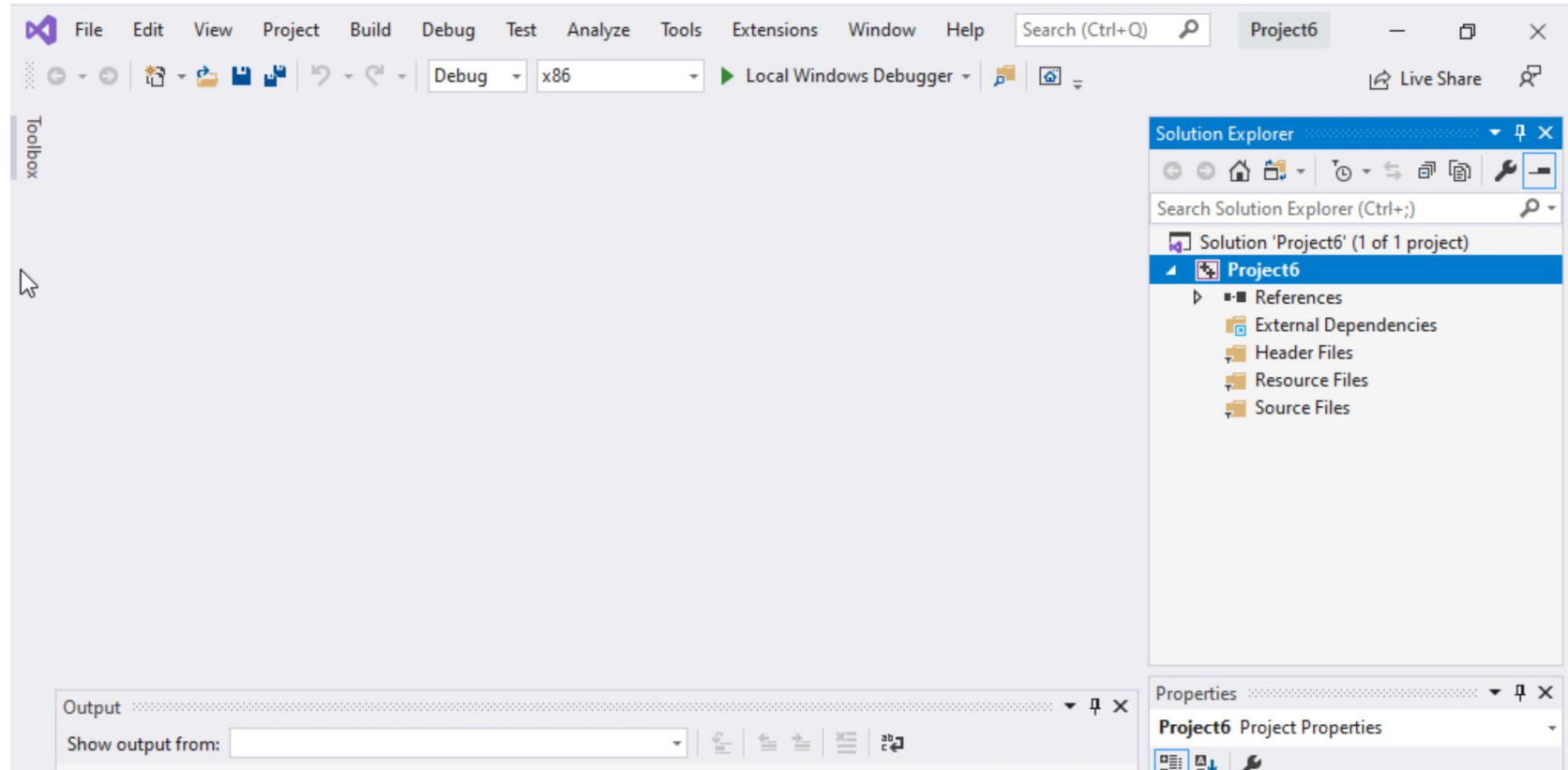
Delete the

Following folders:

Header files

Resources Files, and

Source Files





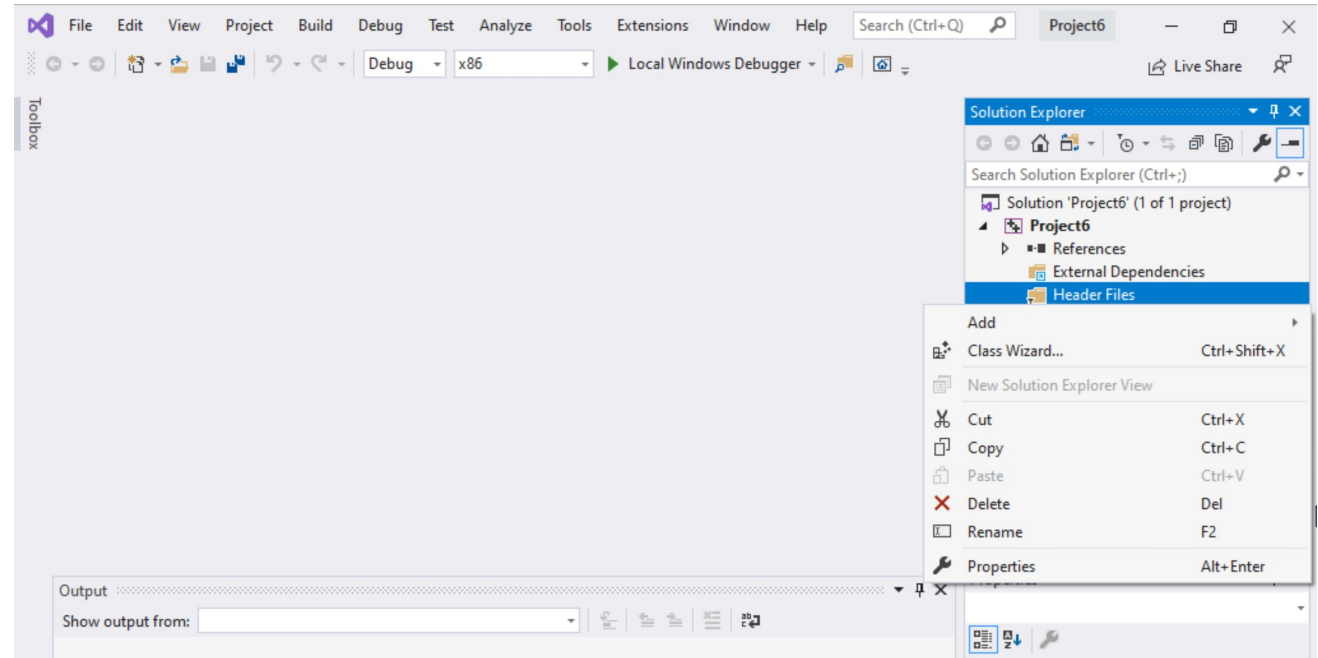
# Create a new Project

To delete :

Select the folders

Right click on it

Select delete



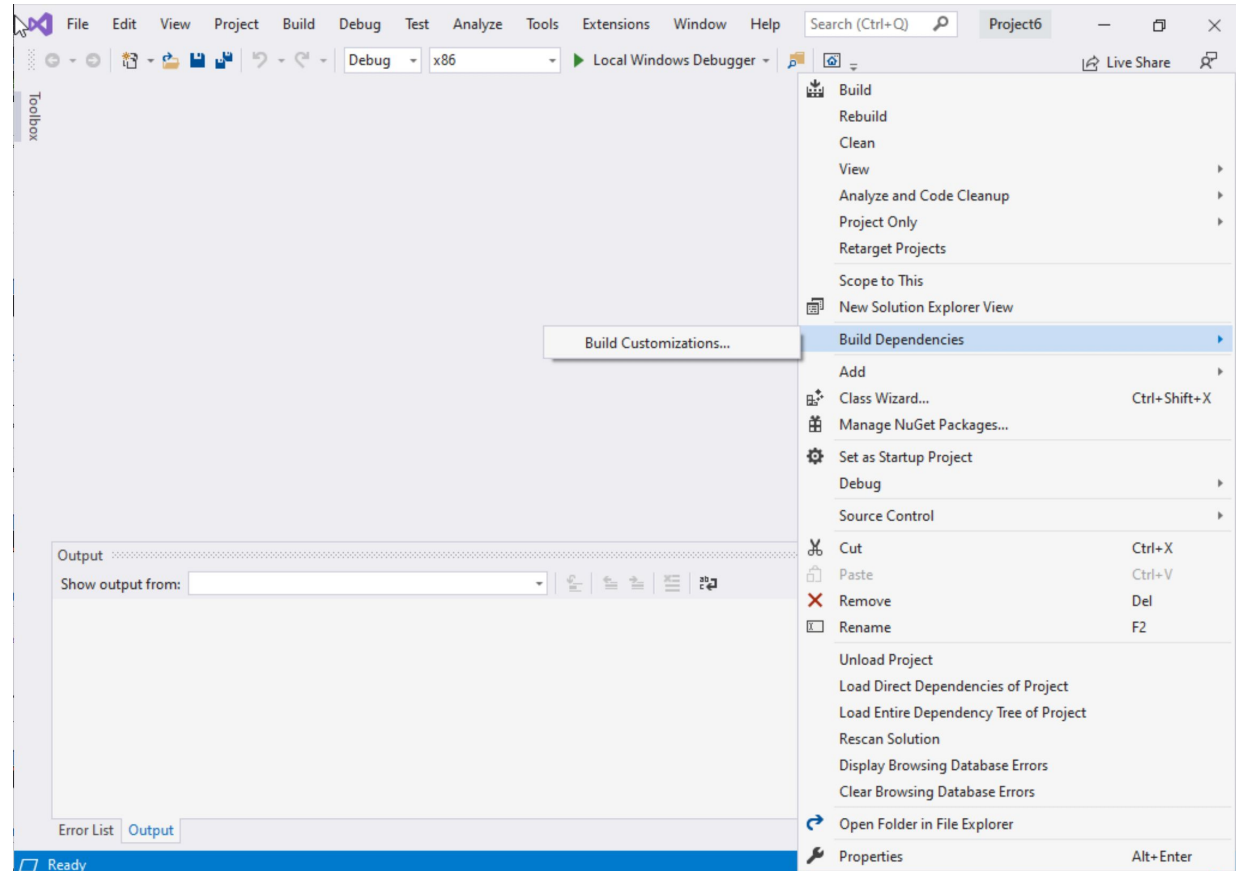
# Create a new Project

Select Project Name on solution explorer

Right click on it

Go to Build Dependencies

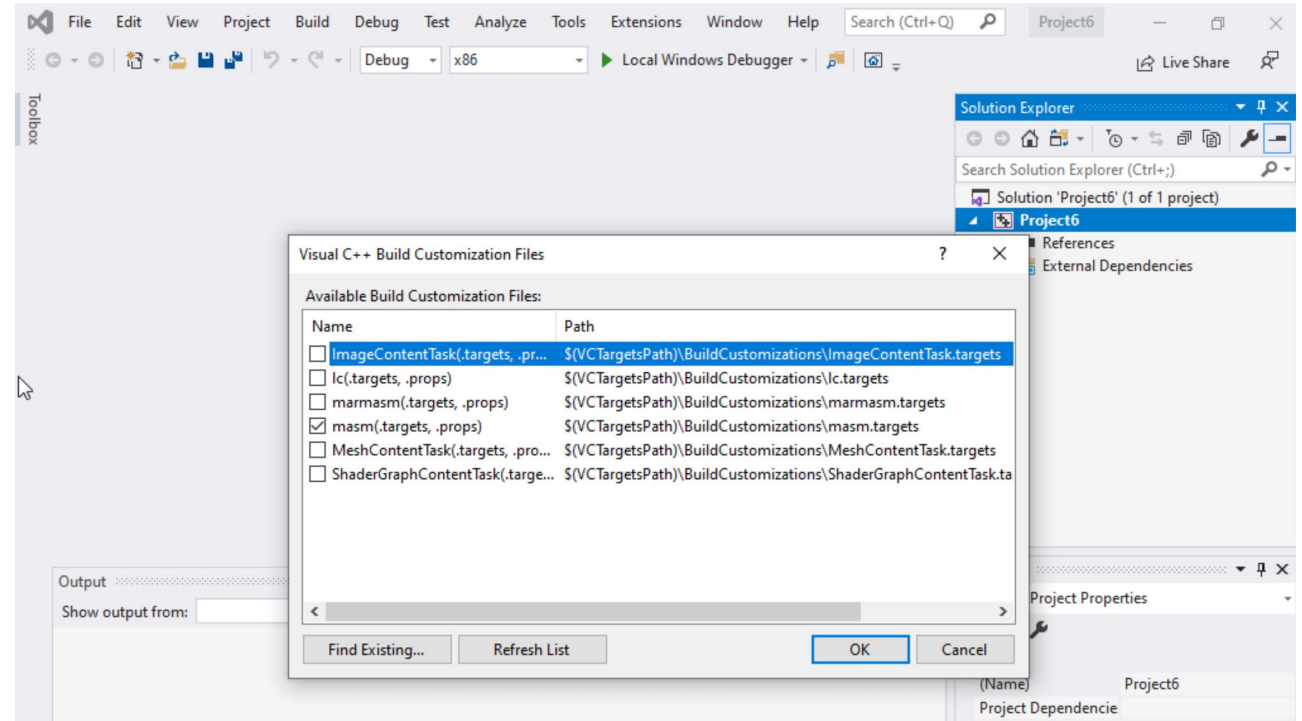
Click on Build Customizations



# Create a new Project

Select mash(.target, .props)

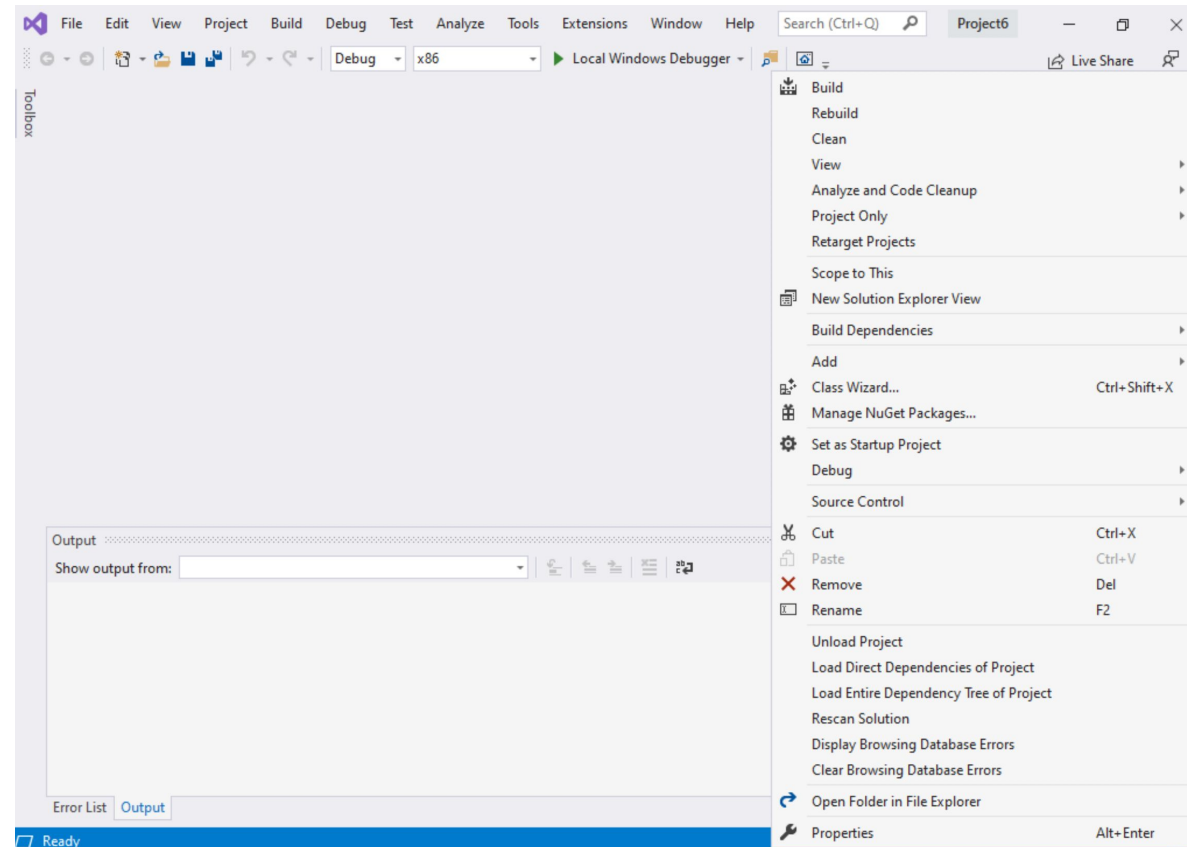
Click ok



# Create a new Project

Right click on the Project name in the solution explorer

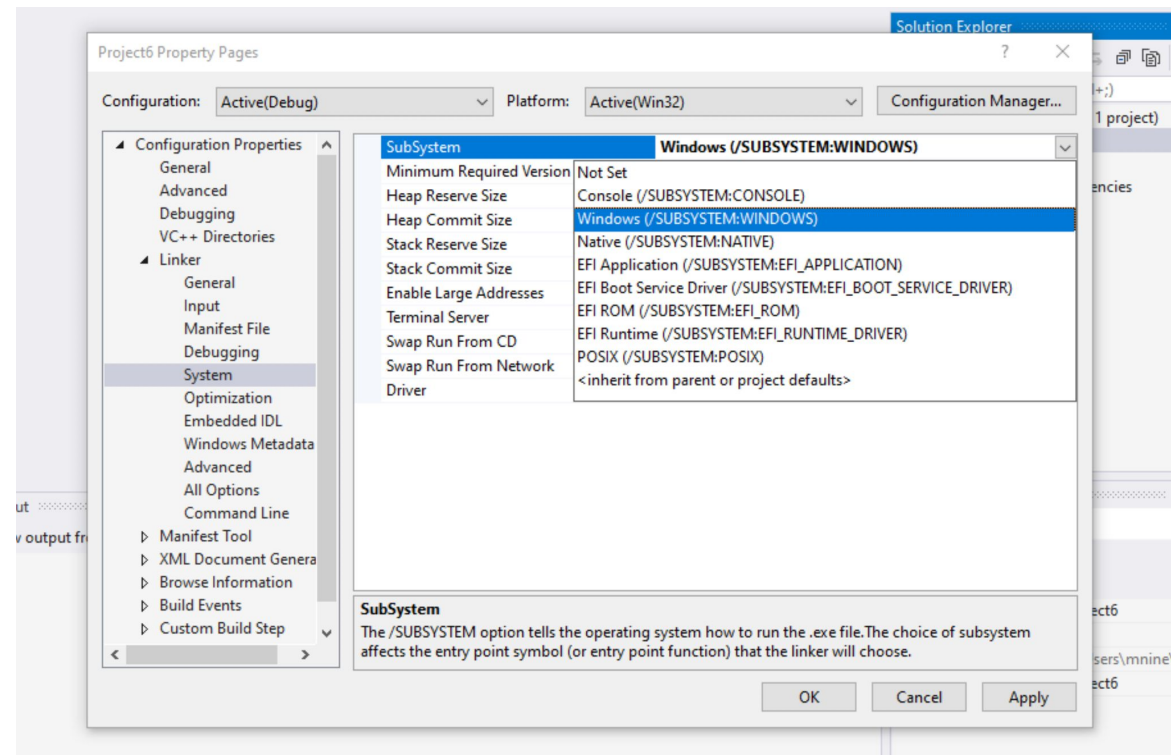
Click properties



# Create a new Project

Select Windows(/SUBSYSTEM:WINDOWS)

Click OK



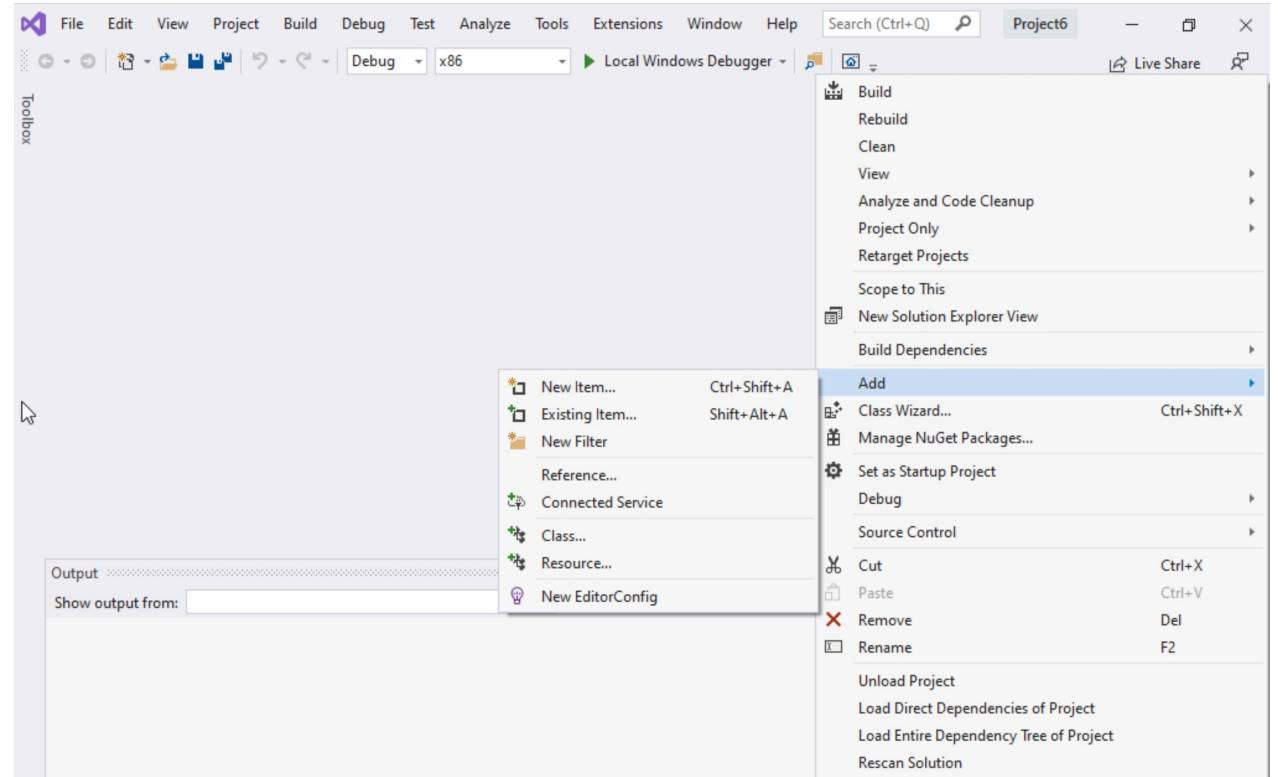
# Create a new Project

Select Project name on solution explorer

Right click on it

Expand Add

Choose New Item

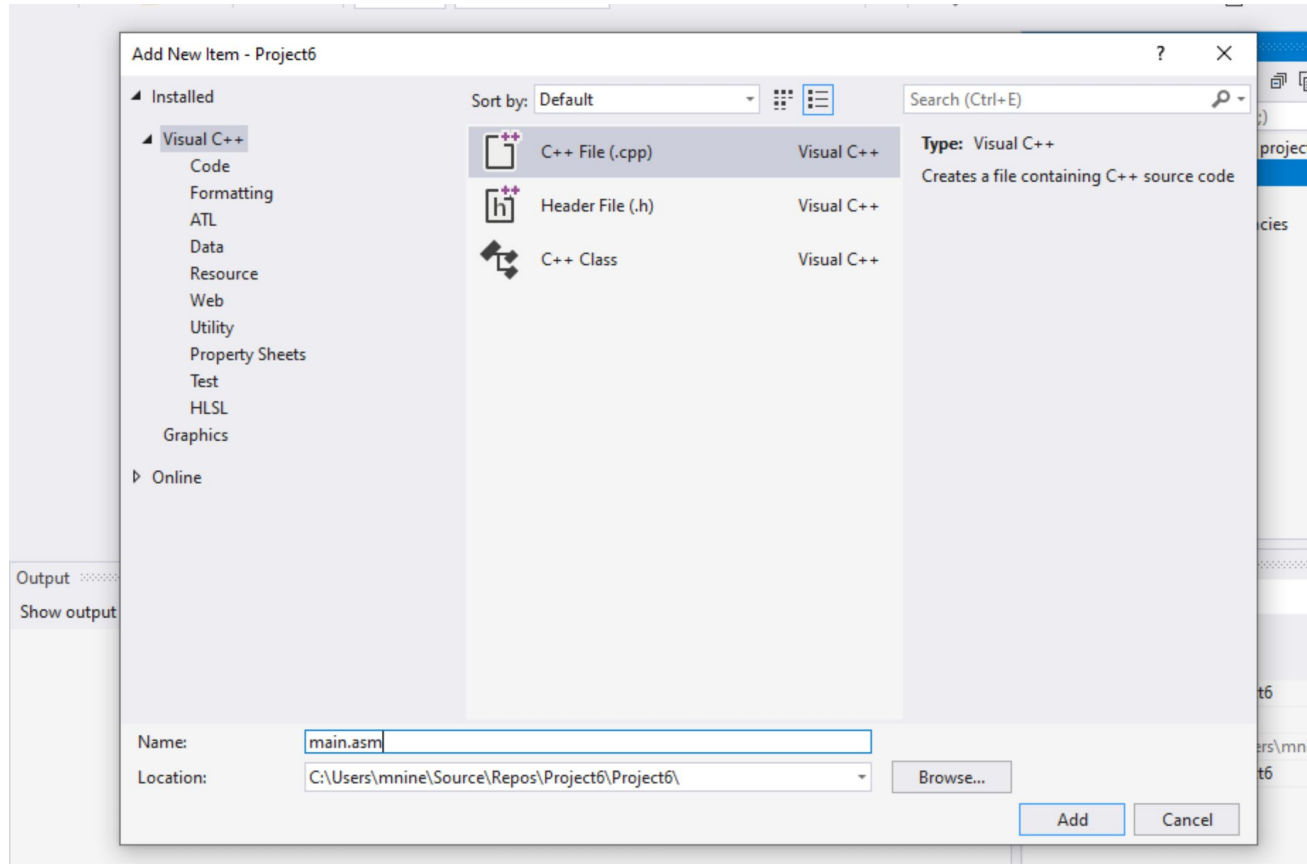


# Create a new Project

Select C++ File(.cpp)

Name: main.asm

Click Add

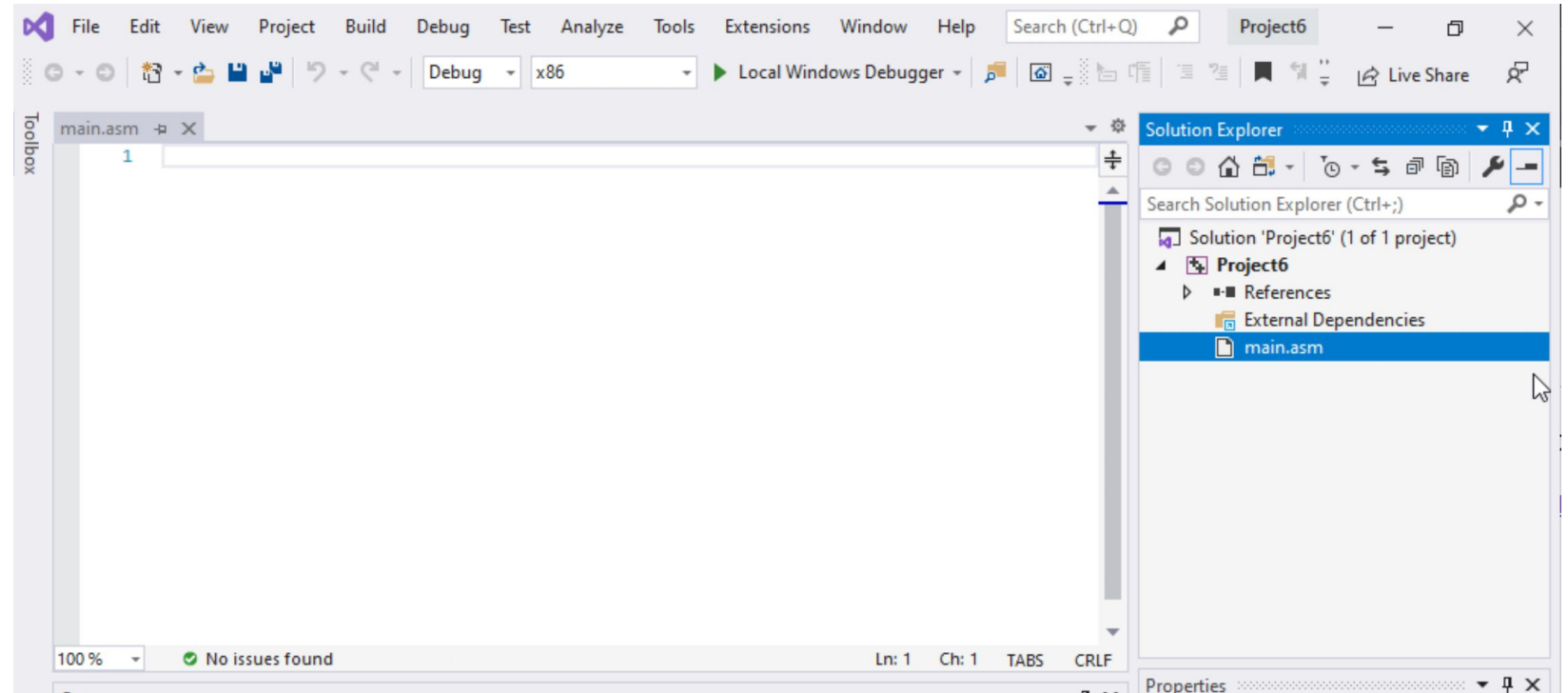


# Create a new Project

Select main.asm

Add your code

In the main.asm File.





# Sample Code

Type the sample code in the main.asm

```
.386
.model flat, stdcall
.stack 4096

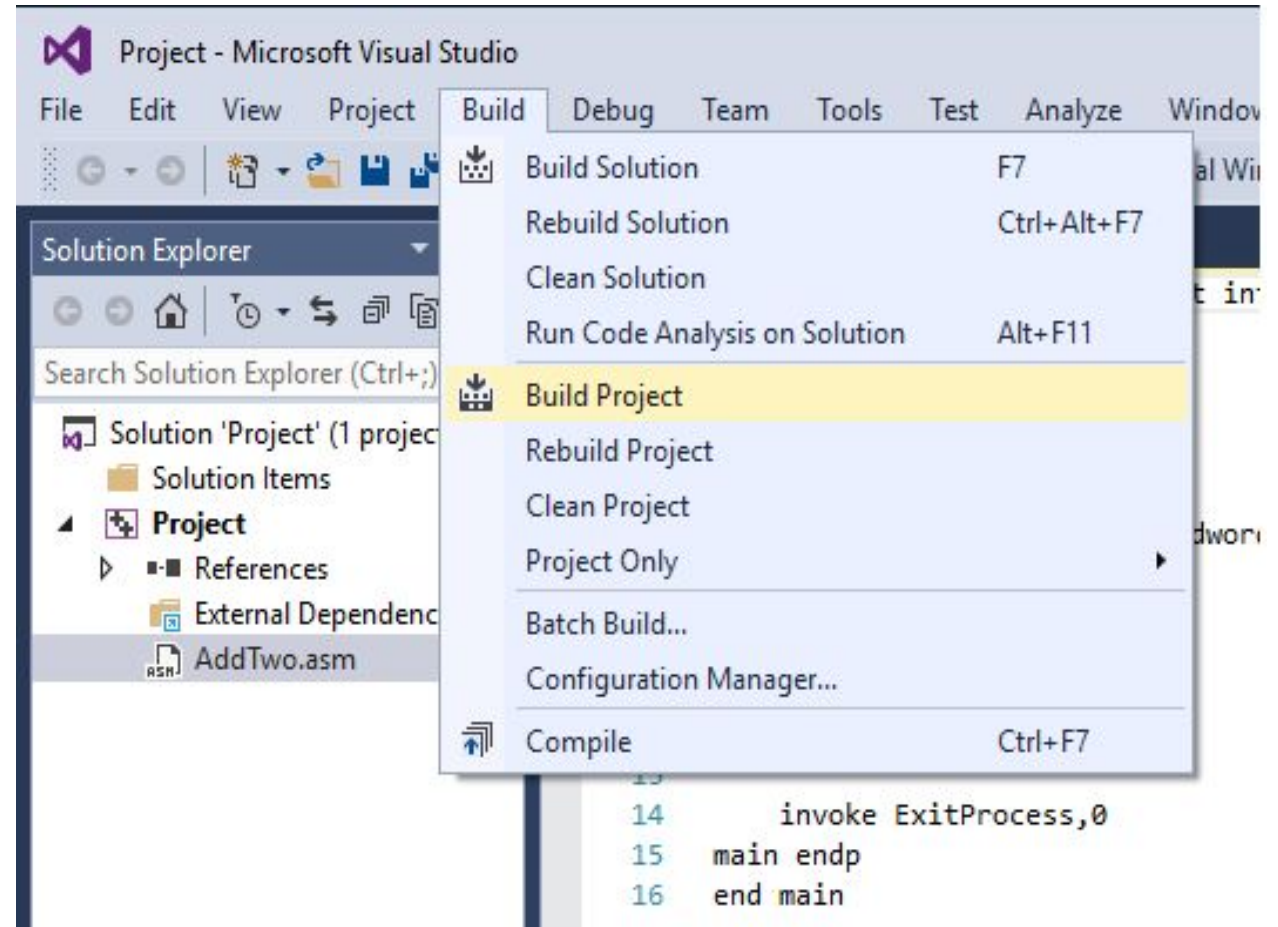
ExitProcess PROTO, dwExitCode:DWORD

.code
main PROC
    mov eax, 5
    add eax, 6

    INVOKE ExitProcess, 0
main ENDP
END main
```

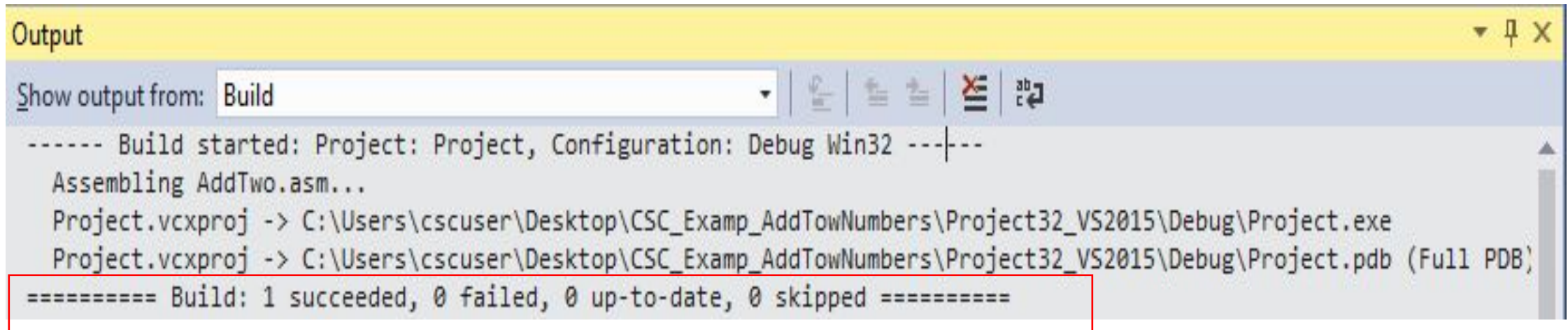
# Run the code

- Select **Build Project** (this will assemble and link your program) from the **Build** menu.
- This will **assemble** and **link** your program



# Run the code

- You should see messages like the following, indicating the build progress:

A screenshot of the Visual Studio Output window. The window has a yellow title bar labeled 'Output'. Below the title bar is a toolbar with icons for showing output from different sources (Build, Errors, Warnings, etc.). The main area of the window displays the following text:

```
----- Build started: Project: Project, Configuration: Debug Win32 ---|---  
Assembling AddTwo.asm...  
Project.vcxproj -> C:\Users\cscuser\Desktop\CSC_Examp_AddTowNumbers\Project32_VS2015\Debug\Project.exe  
Project.vcxproj -> C:\Users\cscuser\Desktop\CSC_Examp_AddTowNumbers\Project32_VS2015\Debug\Project.pdb (Full PDB)  
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

The last line of the output is highlighted with a red rectangular box.

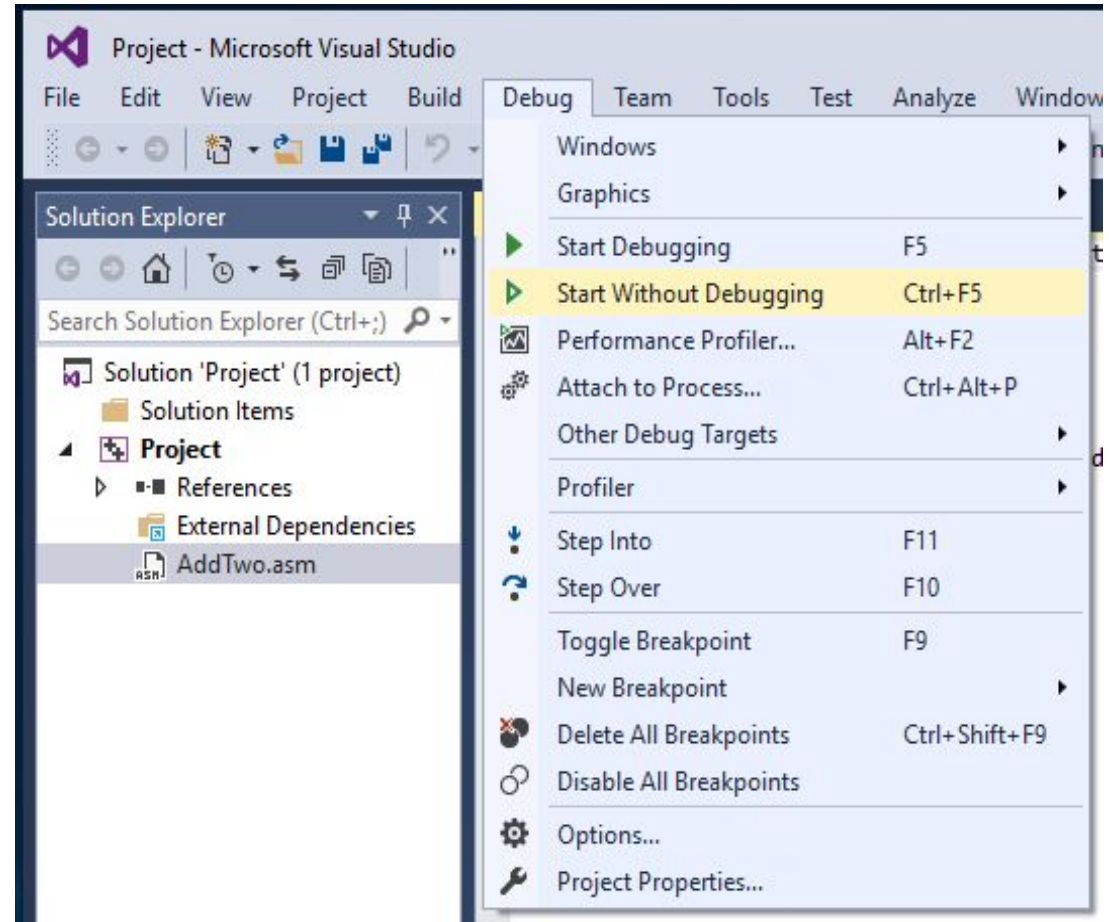
**Note:** if you see **1 failed** or more, then there must be at least one error that needs to be corrected

# Run the code

- **Run** the Program by selecting **Start without Debugging** from the **Debug menu**.
- Press any key to end running

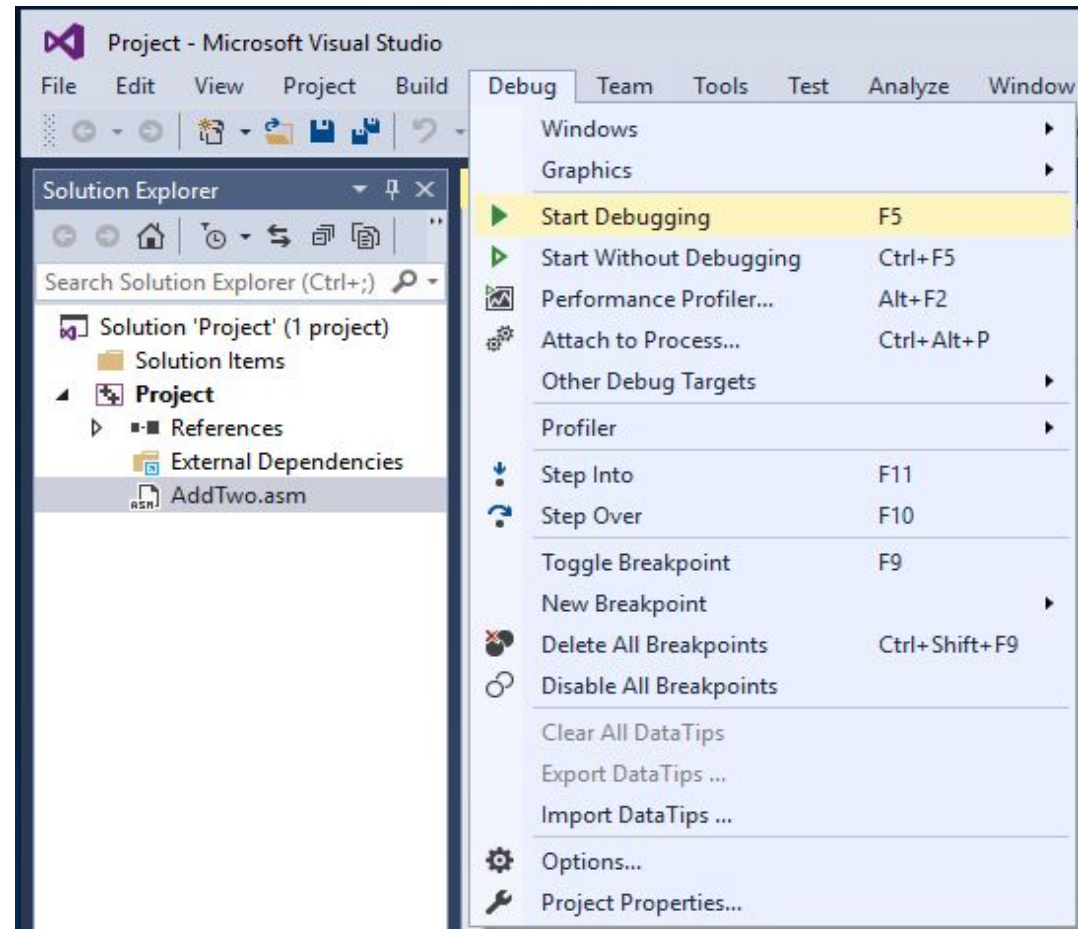
- You will not see the result  
as the result is in a register

- See next slide




# Debug the code

- Select **Start Debugging** from the **Debug** menu.



# Debug the code

- You should see the **register window** in the bottom:



```
AddTwo.asm  X
1  ; AddTwo.asm - adds two 32-bit integers.
2  ; Chapter 3 example
3
4  .386
5  .model flat,stdcall
6  .stack 4096
7  ExitProcess proto,dwExitCode:dword
8
9  .code
10 main proc
11 |   mov eax,5
12   add eax,6
13
14   invoke ExitProcess,0
15 main endp
16 end main
```

100 %

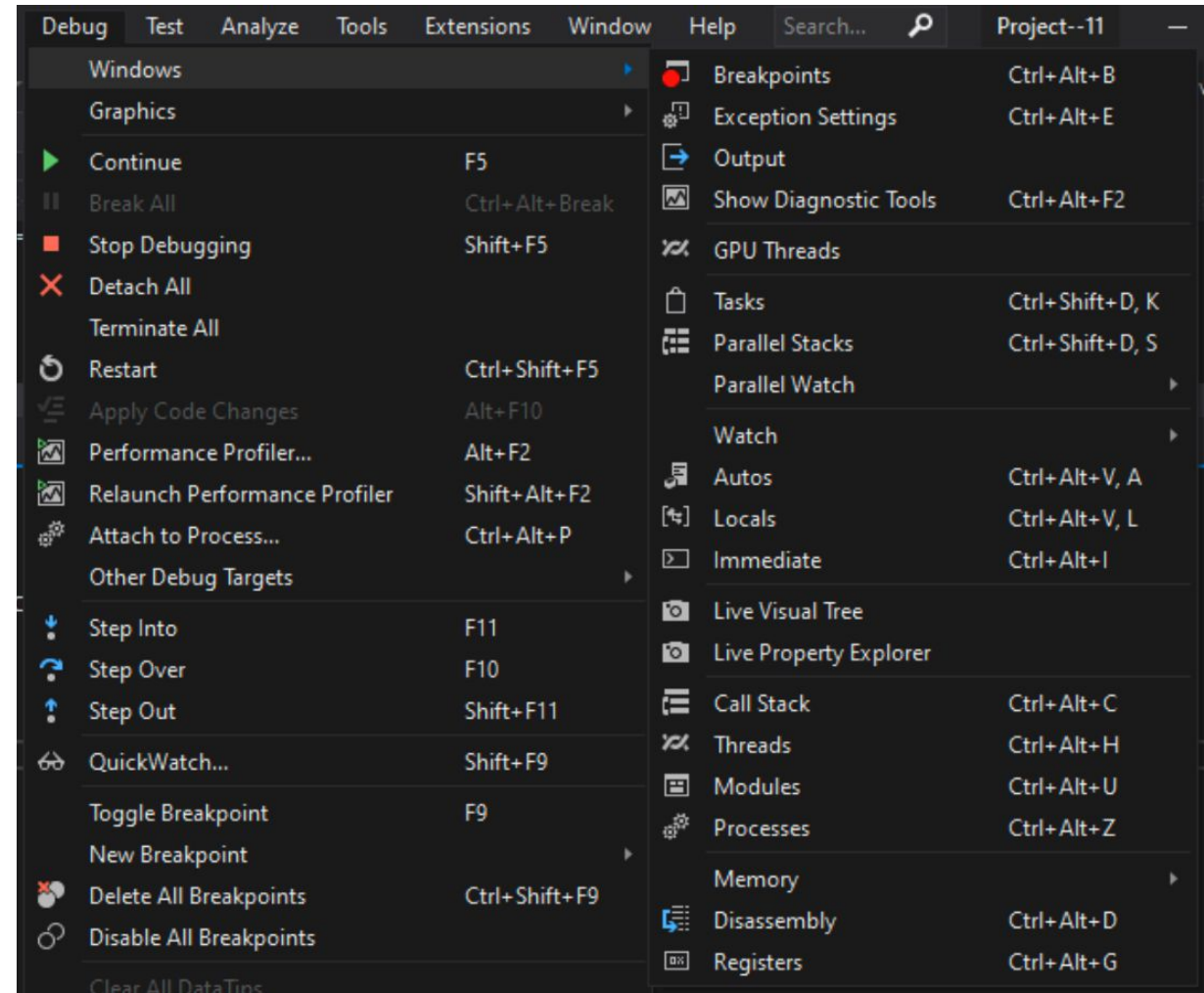
Registers

EAX = 4BD36093 EBX = 7FFDE000 ECX = 00401005 EDX = 00401005 ESI = 00401005 EDI = 00401005  
EIP = 00401010 ESP = 0019FF84 EBP = 0019FF94 EFL = 00000244



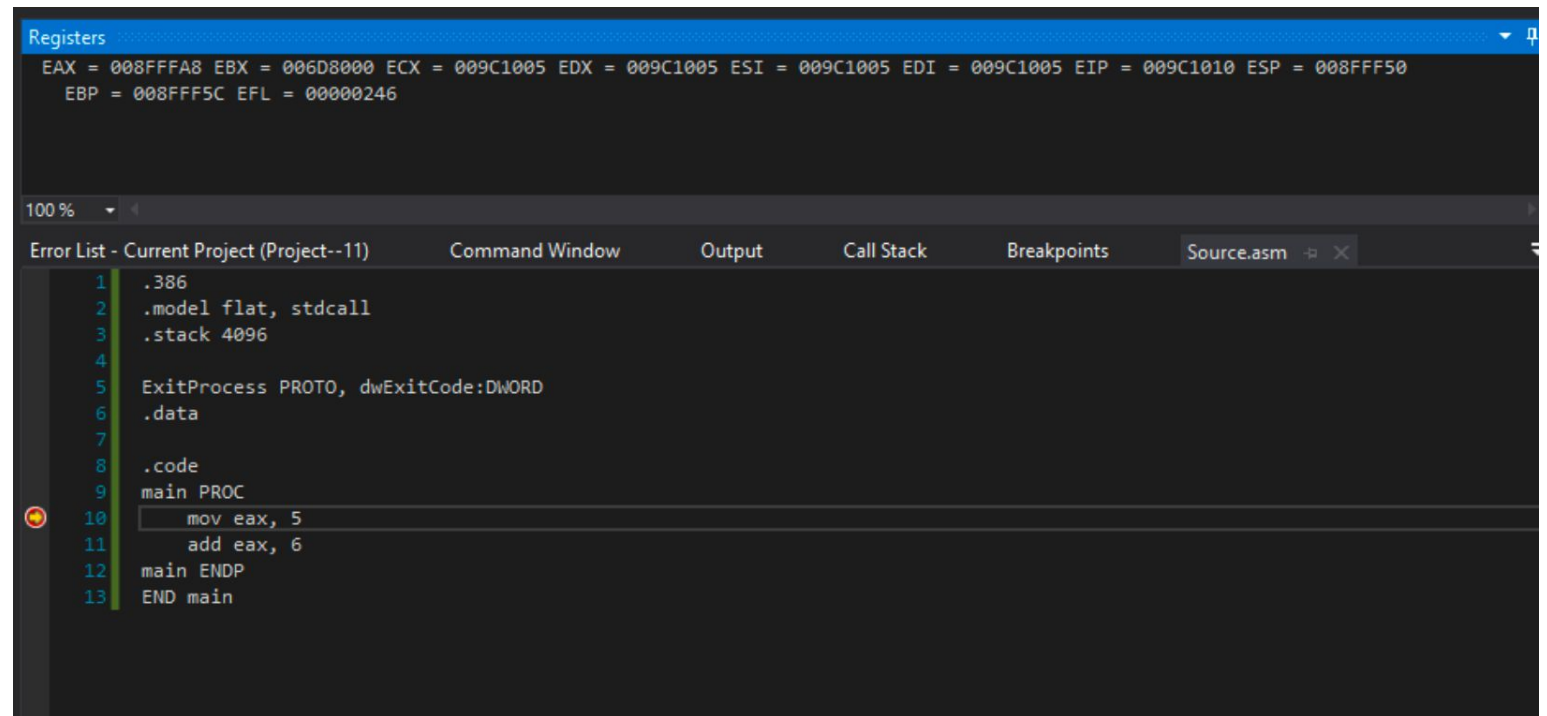
# Debug the code

- IF you don't see the register window
- Go to
- **Debug->Windows->Registers**



# Debug the code

- The register window should appear during the debugging



The screenshot displays a debugger interface with two main windows. The top window, titled 'Registers', shows the current state of various CPU registers: EAX = 008FFFA8, EBX = 006D8000, ECX = 009C1005, EDX = 009C1005, ESI = 009C1005, EDI = 009C1005, EIP = 009C1010, ESP = 008FFF50, EBP = 008FFF5C, and EFL = 00000246. Below this, a horizontal tab bar contains several options: 'Error List - Current Project (Project--11)', 'Command Window', 'Output', 'Call Stack', 'Breakpoints', and 'Source.asm'. The 'Source.asm' tab is currently selected, showing a list of assembly instructions. The instructions are numbered 1 through 13. Line 10, 'mov eax, 5', is highlighted with a green vertical bar, indicating the current instruction pointer. A small red circle with a yellow exclamation mark is visible in the left margin next to line 10.

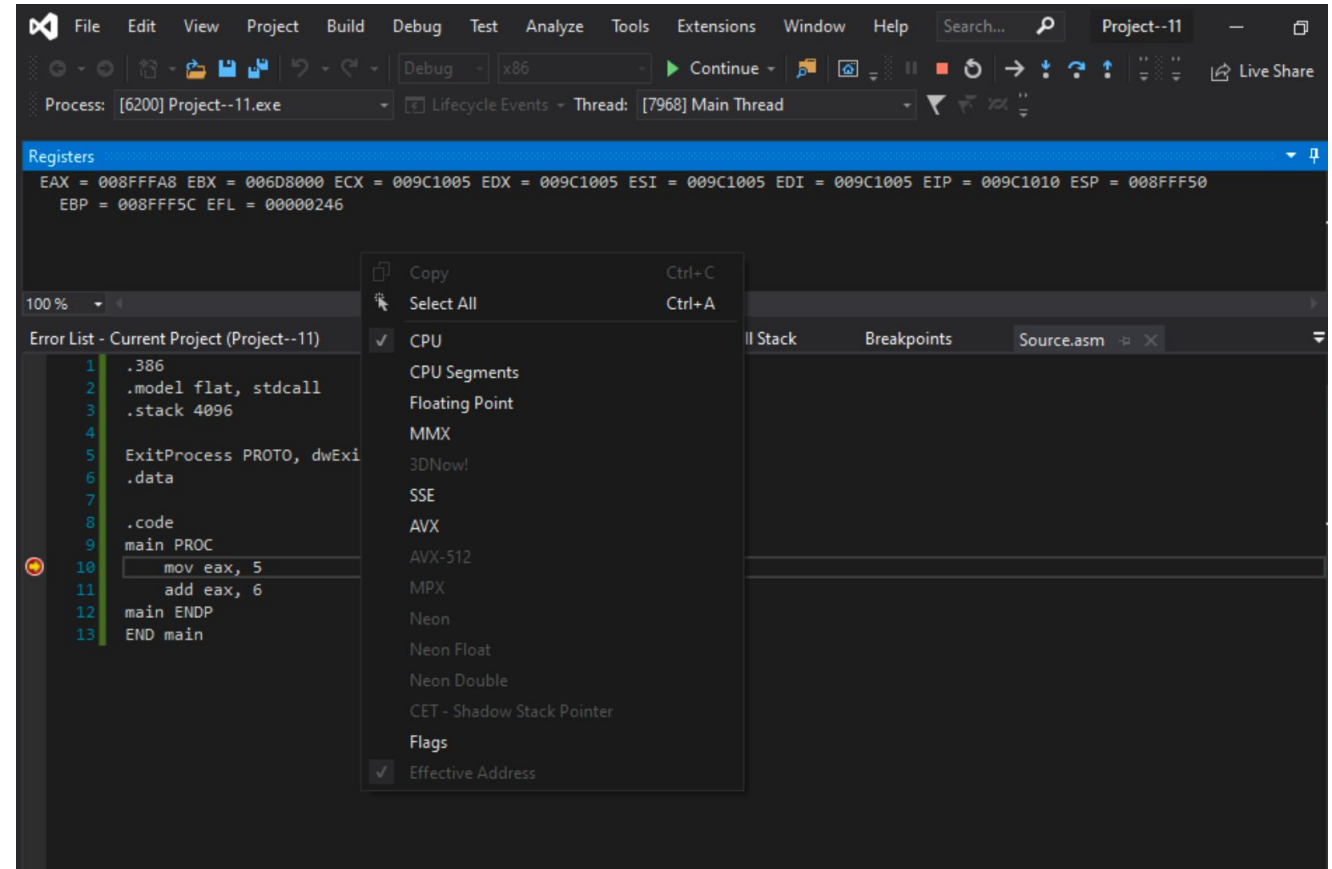
```
Registers
EAX = 008FFFA8 EBX = 006D8000 ECX = 009C1005 EDX = 009C1005 ESI = 009C1005 EDI = 009C1005 EIP = 009C1010 ESP = 008FFF50
EBP = 008FFF5C EFL = 00000246

100 %
Error List - Current Project (Project--11) Command Window Output Call Stack Breakpoints Source.asm
1 .386
2 .model flat, stdcall
3 .stack 4096
4
5 ExitProcess PROTO, dwExitCode:DWORD
6 .data
7
8 .code
9 main PROC
10 mov eax, 5
11 add eax, 6
12 main ENDP
13 END main
```



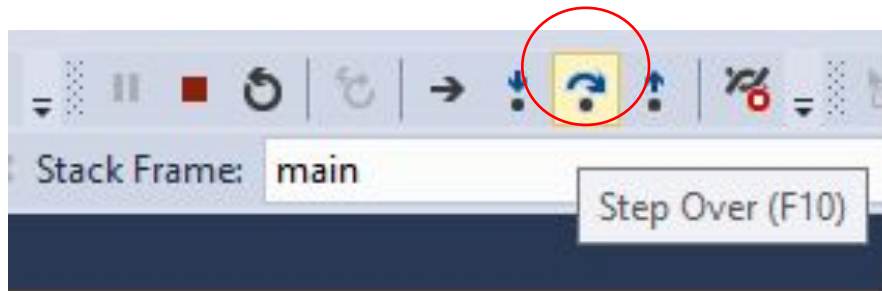
# Debug the code

- To see the EFLAGS
- **Right click on the register window and select “Flags”**

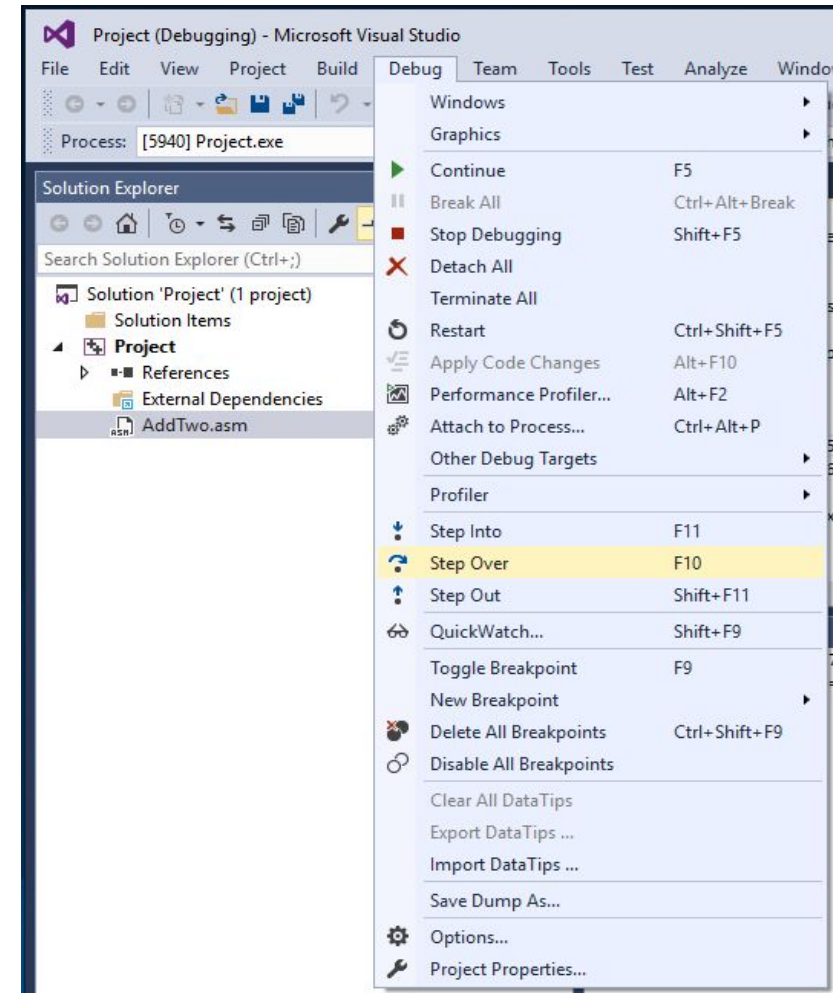


# Debug the code

- Select **Step Over** from the **Debug** menu.
  - Depending on how Visual Studio was configured,
  - Either the Fn+F10 function key or the Shift+F8 keys will execute the **Step Over command**.
  - You can also use the button to stepover:

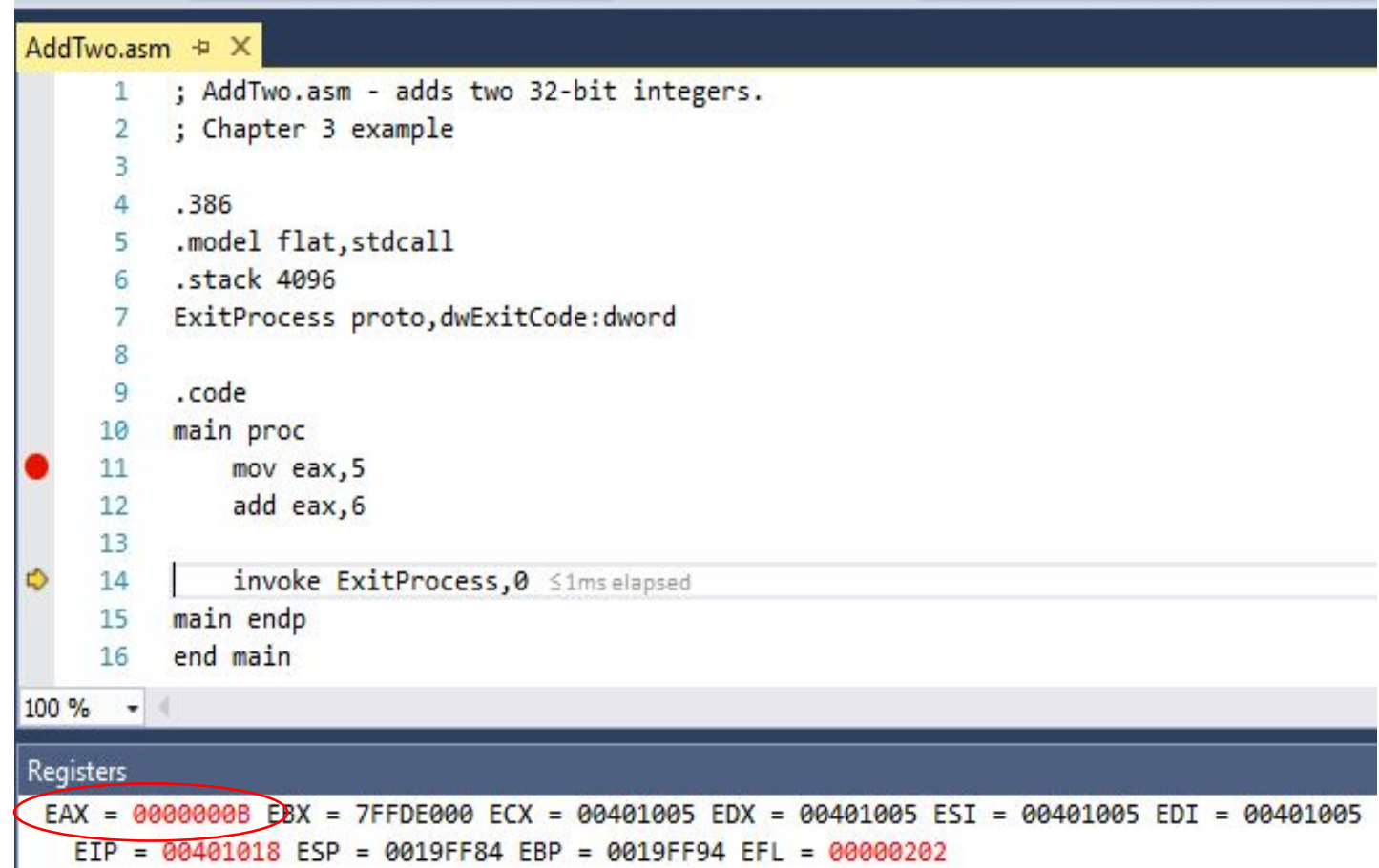


- After you reach the “invoke ExistProcess,0”,
  - Do not hit step over



# Debug the code

- After you reach the “invoke ExitProcess,0”
- Look at the **eax** register content and verify that its content is 11 (Which is **B**?).



The screenshot shows a debugger window with the title bar 'AddTwo.asm'. The assembly code is displayed in a list with line numbers 1 through 16. A red dot on line 11 indicates the current instruction pointer. A yellow arrow on line 14 points to the instruction 'invoke ExitProcess,0', with a tooltip showing '≤ 1ms elapsed'. Below the code, the 'Registers' window shows the current state of the CPU registers. The EAX register is highlighted with a red circle and contains the value 0000000B.

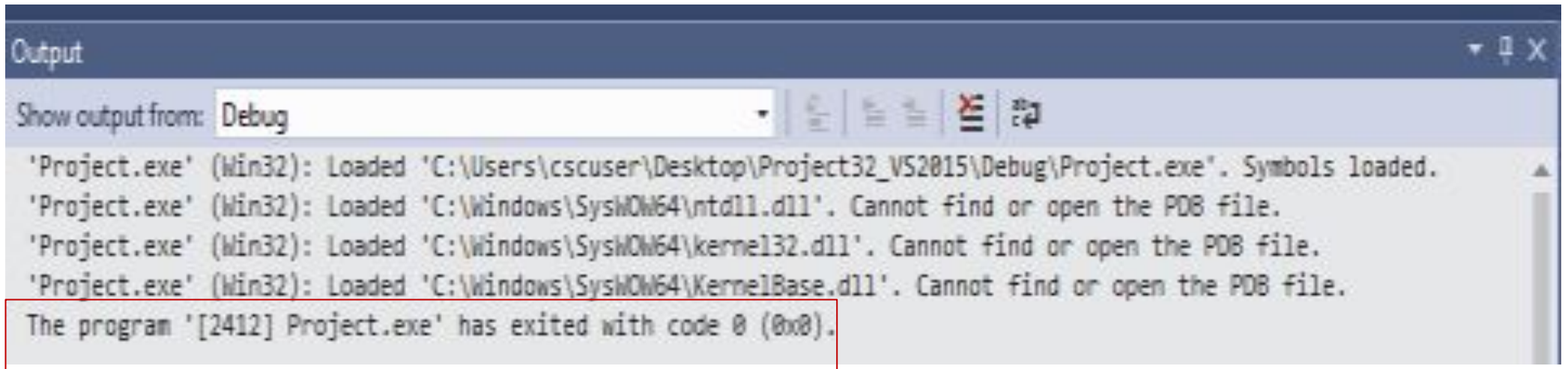
```
1 ; AddTwo.asm - adds two 32-bit integers.
2 ; Chapter 3 example
3
4 .386
5 .model flat,stdcall
6 .stack 4096
7 ExitProcess proto,dwExitCode:dword
8
9 .code
10 main proc
11     mov eax,5
12     add eax,6
13
14     invoke ExitProcess,0 ≤ 1ms elapsed
15 main endp
16 end main
```

Registers

EAX = 0000000B EBX = 7FFDE000 ECX = 00401005 EDX = 00401005 ESI = 00401005 EDI = 00401005  
EIP = 00401018 ESP = 0019FF84 EBP = 0019FF94 EFL = 00000202

# Debug the code

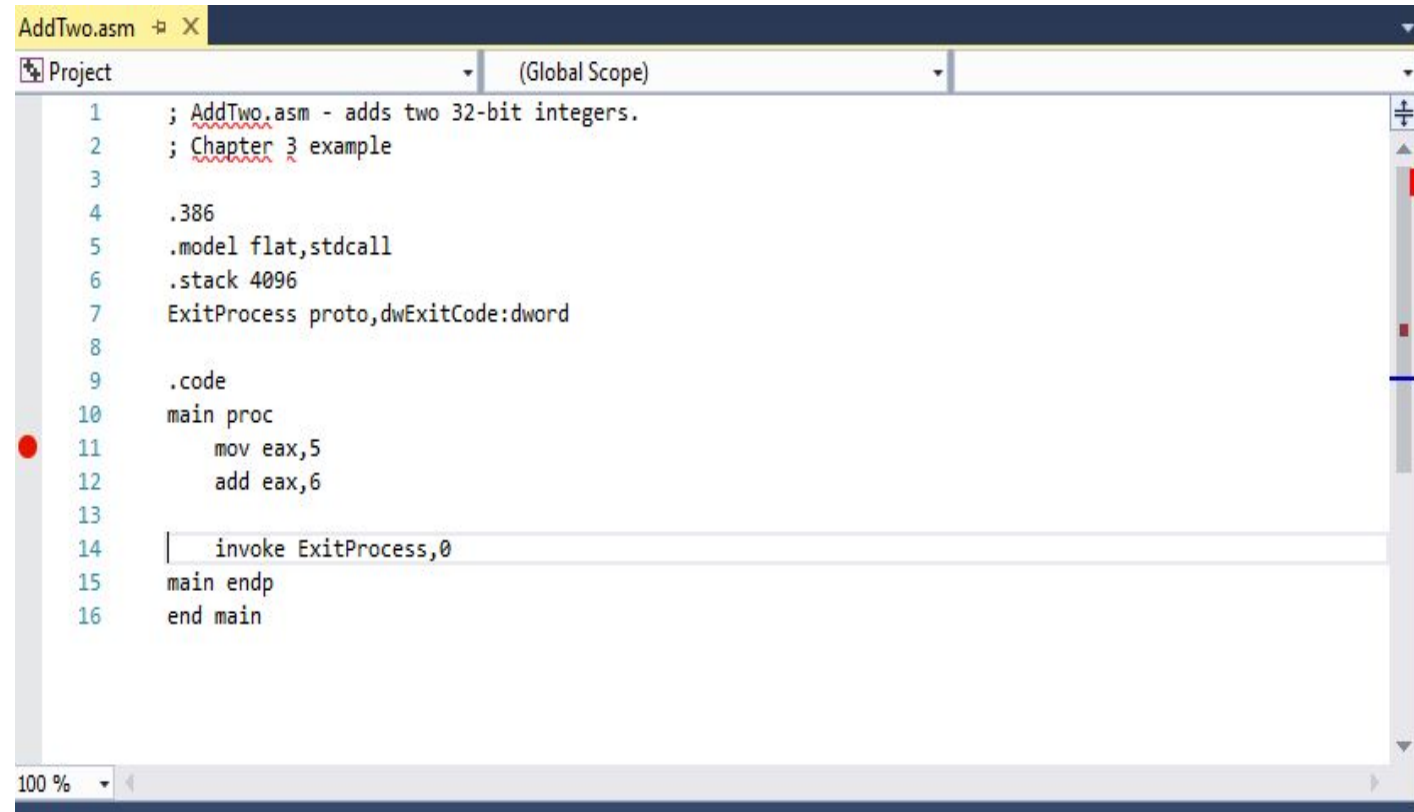
- When you hit step over, the program will end, as there is nothing to execute after:  
“invoke ExistProcess,0”



```
Output
Show output from: Debug
'Project.exe' (Win32): Loaded 'C:\Users\cscuser\Desktop\Project32_VS2015\Debug\Project.exe'. Symbols loaded.
'Project.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'. Cannot find or open the PDB file.
'Project.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'. Cannot find or open the PDB file.
'Project.exe' (Win32): Loaded 'C:\Windows\SysWOW64\KernelBase.dll'. Cannot find or open the PDB file.
The program '[2412] Project.exe' has exited with code 0 (0x0).
```

# Debug the code

- **Another way** to start a debugging session is
  - **set a breakpoint** on a program statement by clicking the mouse in **the vertical gray bar** just to the left of the code window.
    - A **large red dot** will mark the breakpoint location.
    - Then you can run the program by selecting **Start Debugging** from the **Debug menu**.



The screenshot shows the Visual Studio IDE with a file named 'AddTwo.asm' open. The 'Project' window on the left shows '(Global Scope)'. The main editor window displays the assembly code for 'AddTwo.asm'. A red dot, representing a breakpoint, is placed in the vertical gray bar to the left of line 11. The code is as follows:

```
1 ; AddTwo.asm - adds two 32-bit integers.
2 ; Chapter 3 example
3
4 .386
5 .model flat,stdcall
6 .stack 4096
7 ExitProcess proto,dwExitCode:dword
8
9 .code
10 main proc
11     mov eax,5
12     add eax,6
13
14     invoke ExitProcess,0
15 main endp
16 end main
```

If you try to set a breakpoint on a **non-executable line**, Visual Studio will just move the breakpoint forward to **the next executable line** when you run the program.

# Some Practice Problems

Don't need to turn-in these problems

# Examples

- What is the Most significant bit of the following binary number?
- 1000011010
- Solution: 1

# Examples

- What is the minimum number of binary bits to represent a decimal number 435644?

Solution:

- (1) Convert the decimal number to binary: 1101010010110111100
- (2) Count the number of bits: 19



# Examples

- What is the minimum number of binary bits to represent the hexadecimal number 6A445?

Solution :

(1) Convert the Hexadecimal number to binary:

0110 1010 0100 0100 0101

(2) Drop any leading zeros from the left side of the number:

110 1010 0100 0100 0101

(3) Count the number of bits: 19 bits

# Examples

- What is the 8-bit binary representation of the following decimal number?

- 43

Solution:

- (1) Convert 43 into binary : 101011
- (2) Make the number 8-bit by adding 2 leading zeros: 0010 1011
- (3) Perform 2's complement on 0010 1011
  - (3.1) Flip the bits: 1101 0100
  - (3.2) add 1 to it:  $1101\ 0100 + 1 = 1101\ 0101$

# Examples

- Perform the following hexadecimal subtraction using 2's complement.
- $A15F - 6ABC$

Solution:

(1) Compute the 2's complement of 6ABC:

(1.1) Reverse all the digits (subtract each digit from 15)

15 15 15 15

(-)	6	A	B	C
<hr/>				
9	5	4	3	

# Examples

- (1.2) add 1 to it:  $9543 + 1 = 9544$
- Add  $A15F + 9544$

Carry: 1 0 0 1

A 1 5 F

9 5 4 4

3 6 A 3

$$F + 4 = 19; \frac{19}{16} = 1, \text{rem } 3$$

$$1 + 5 + 4 = 10; 10 \text{ is } A \text{ in Hex}$$

$$0 + 10 + 9 = 19; \frac{19}{16} = 1, \text{rem } 3$$

Ignore the last carry.

# Examples

- What is the decimal representation of following signed numbers?
- 10110111

Solution: MSB is 1, so the number is negative

- (1) Perform a 2's complement: 1011 0111
  - (1.1) flip the bits : 0100 1000
  - (1.2) add 2 to it:  $0100\ 1000 + 1 = 0100\ 1001$
- (2) Convert the number into decimal: 73
- (3) Sign bit was 1, so the number is -73

# Examples

- How many selector bits required for a four input multiplexer?

Solution:

A multiplexer uses selector bits to select specific input and let that input available to the output.

When a multiplexer has two inputs, it can enumerate the inputs as 0 and 1.

When a multiplexer has 4 inputs, it needs two bits to enumerate them.

Selector bits - 00 – input 1

01 – input 2

10 – input 3

11 – input 4

# Lab 2(b)

Submit these Math Problems to iCollege

# Lab 2(b) : Submission

- Solve the Problems provided in slide 41 to 42.
- You can do your work in a text editor (Microsoft word, open office, etc.)
- Or you can do it in a piece of paper, then scan or take a picture of the paper.
- Convert them into pdf and submit in the icollege.



# Lab 2(b) Problem 1

- Design a 8-bit full adder. Draw the block diagram. [\[Hint: Lecture\]](#)

# Lab 2(b) Problem 2

- Draw the circuit for the following Boolean expression:
- $P = (X \text{ and } Y) \text{ or } (\text{not } X \text{ or } Z)$       [\[Hint: Lecture\]](#)

# Registers and Memory

*A review*

# Registers

- Registers are storage locations inside the processor.
- A register can be accessed more quickly than a memory location .
- Different registers serve different purposes.
- Some of them are described below:

## 1 32-bit General-Purpose Registers

EAX
EBX
ECX
EDX

EBP
ESP
ESI
EDI

## 2 16-bit Segment Registers

CS	ES
SS	FS
DS	GS

## 4 EFLAGS

## 3 EIP

# General-Purpose Registers (8 registers)

- There are eight general purpose registers

## 1) EAX –

- All major calculations take place in EAX,
- making it similar to a dedicated **accumulator register**.

## 2) EDX –

- The data register is the an extension to the accumulator.
- It is most useful for storing data related to the accumulator's current calculation.

# General-Purpose Registers

- There are eight general purpose registers

## 3) ECX –

- Like the variable **i** in high-level languages, the **count register** is the **universal loop counter**.

## 4) EBX –

- In 16-bit mode, the base register **was useful** as a pointer.
- Now, it is **completely free for extra storage space**.

# General-Purpose Registers

- There are eight general purpose registers

## 5) ESP –

- **ESP** is the sacred stack pointer.
- With the important **PUSH**, **POP**, **CALL**, and **RET** instructions requiring it's value,
  - ◆ there is never a good reason to use the stack pointer for anything else.

## 6) EBP –

- In functions that **store parameters or variables on the stack**,
  - ◆ the **base pointer** holds the location of the current stack frame.
- In other situations, however, **EBP** is a free data-storage register.

# General-Purpose Registers

- There are eight general purpose registers

## 7) EDI –

- Every **loop** must store its result somewhere,
- and the **destination index** points to that place.

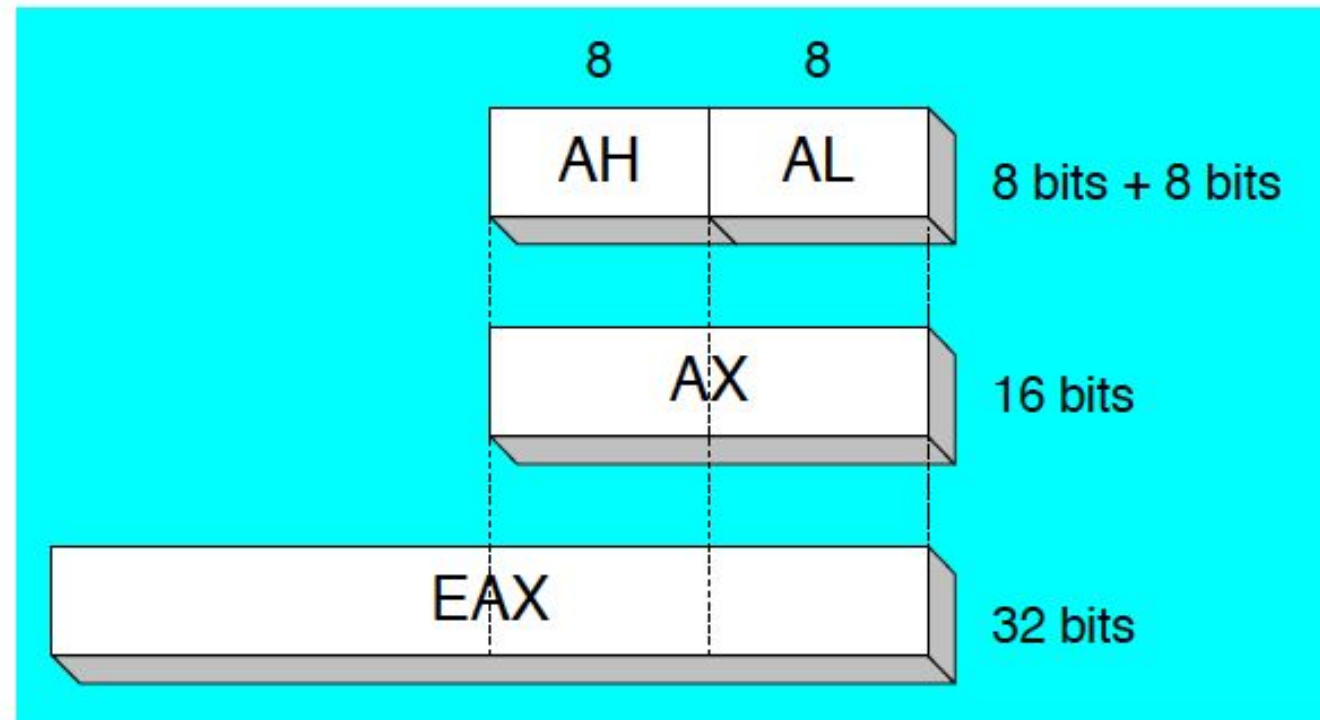
## 8) ESI –

- In **loops** that process data,
- the **source index** holds the location of the input data stream.



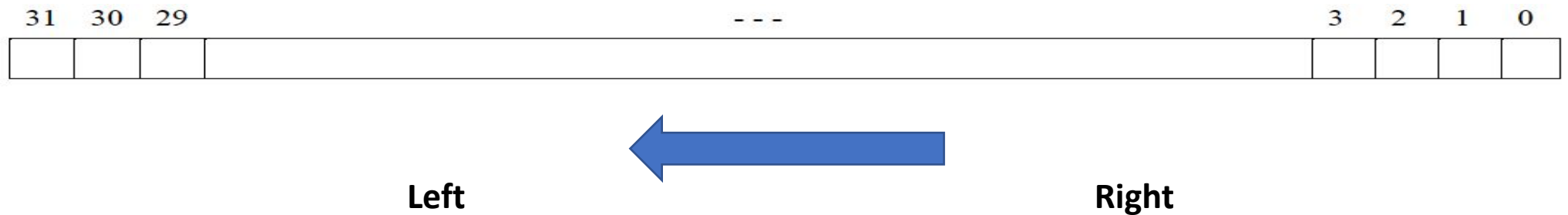
# General-Purpose Registers

- **Accessing** Parts of Registers



# General-Purpose Registers

- Bits are stored in any registers right to left
  - Ex: 32-bit Register (referring to bits storage not bytes)



# General-Purpose Registers

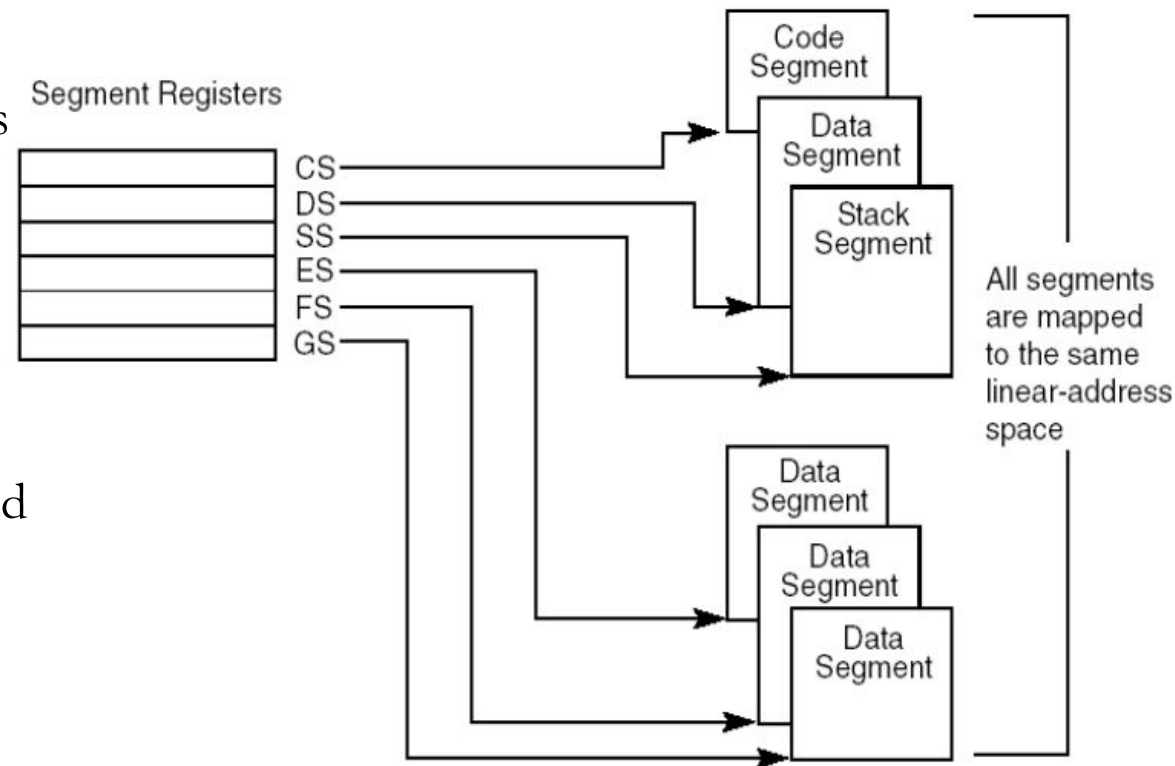
- Bytes are stored right to left too



Byte4 Byte3 Byte2 Byte1  
← ← ← ←  
**A2****B1****C3**D4h

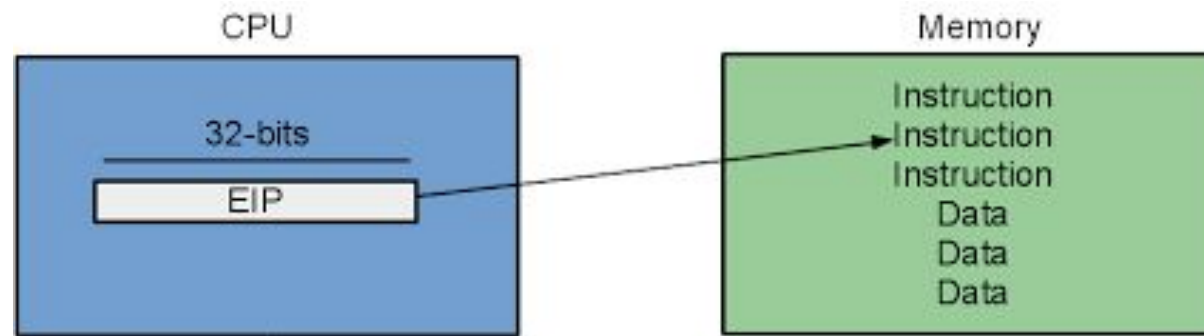
# Segment Registers

- There are **six** segment registers
  - 1) **CS – code segment:** hold program instructions
  - 2) **DS – data segment:** hold variables
  - 3) **SS – stack segment:** holds local function variables and function parameters.
  - 4) **ES, FS, GS - additional segments:** can be used in a similar way as the other segment registers.



# Instruction Pointer Register

- **EIP** – instruction pointer (also called **program counter**):
  - contains the address of the next instruction to be executed .

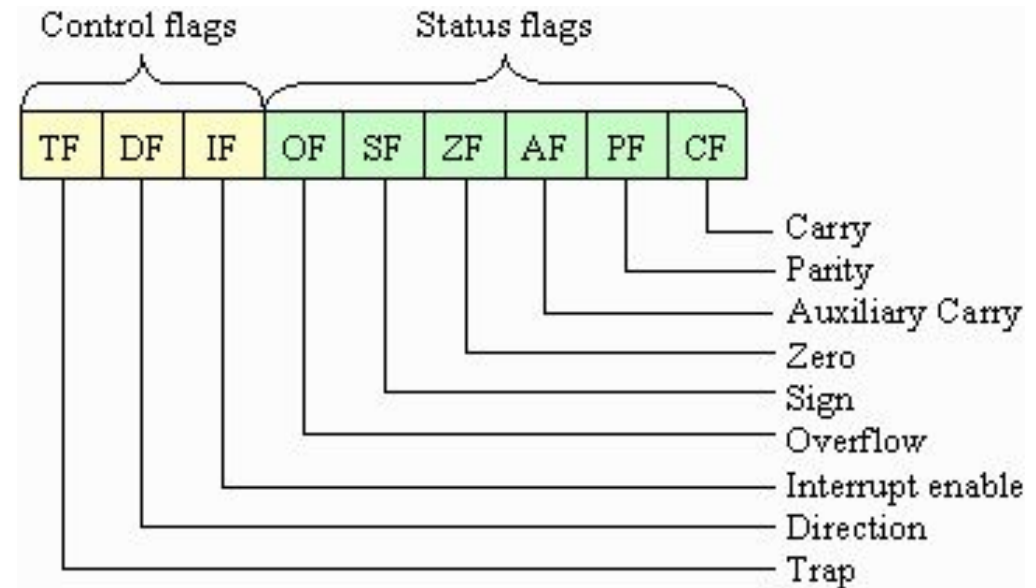


It is also known as PC (Program Counter)

# EFLAGS Register

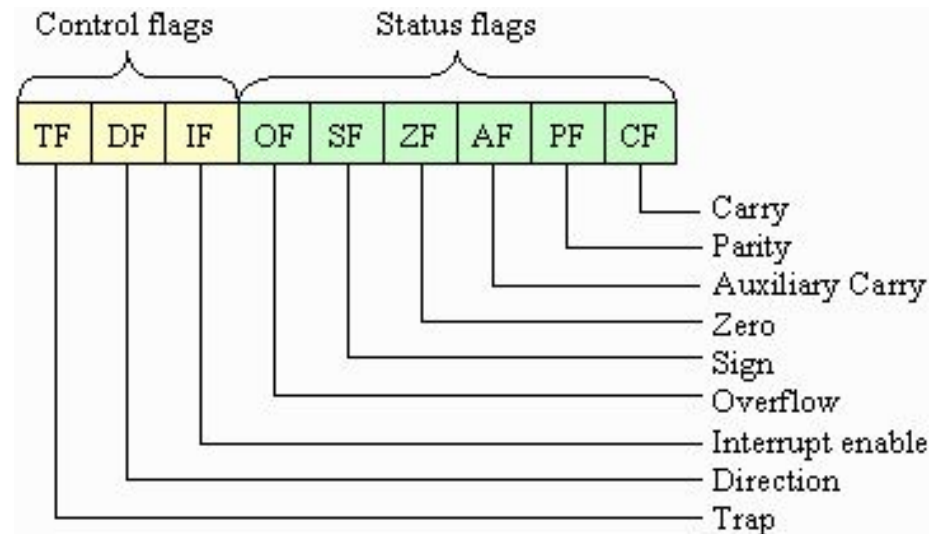
- **EFLAGS**- a register consists of **individual binary bits** that
  - control the **operation** of the CPU or
  - reflect **the outcome** of some CPU **operation**.

- Thus, we have **status** and **control** flags
  - Each flag is a single binary bit



# EFLAGS Register

- **Status flags** record certain information about the most recent **arithmetic** or **logical** operation.
  - **Carry**
    - **unsigned** arithmetic out of range
  - **Overflow**
    - **signed** arithmetic out of range
  - **Sign**
    - **result** is negative
  - **Zero**
    - **result** is zero



A flag is **set** when it equals 1; it is **clear** (or reset) when it equals 0.

**Note:** Control flags are out of the scope of this class

# Some Practice Examples

Don't need to turn-in these problems



# Practice Problems

1. Which flag is set when the result of an unsigned arithmetic operation is too large to fit into the destination?

Carry

2. Which flag is set when the result of a signed arithmetic operation is either too large or too small to fit into the destination?

Overflow

3. Which flag is set when an arithmetic or logical operation generates a negative result?

Sign

# Practice Examples

**4. Show the EDX register and the size and position of the DH, DL, and DX within it.**

# Lab 2(c)

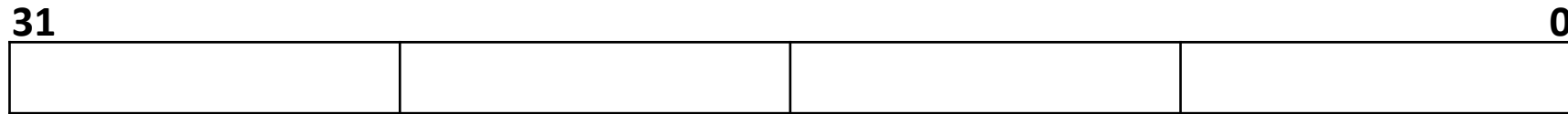
Submit these Problems

# Lab 2(c) : Submission

- Solve the Problems provided in slide 61 to 62.
- You can do your work in a text editor (Microsoft word, open office, etc.)
- Or you can do it in a piece of paper, then scan or take a picture of the paper.
- Convert them into pdf and submit in the iCollege.

# Lab 2(c) Problems

2. Store the following value in EAX register: 12784569h



4 bytes

# Lab 2(c) Problems

6. For each **add instruction in this exercise**, assume that **EAX** contains the given contents before the instruction is executed.

- Give the contents of **EAX** as well as the values of the **CF**, **OF**, **SF**, **ZF** after the instruction is executed.
- All numbers are in hex. (Hint: add eax, 40 will add 40 to the contents of register eax and stores the result back in eax)

Contents of EAX (Before)	Instruction	Contents of EAX (After)	CF	OF	SF	ZF
00000040	add eax, 40					
FFFFFF40	add eax, 40					
00000040	add eax, -40					

Assume that numbers are signed integers