EE 213 Computer Organization and Assembly Language

Week # 3, Lecture # 9

4th Muharram ul Haram, 1440 A.H

14th September 2018

Minds open...

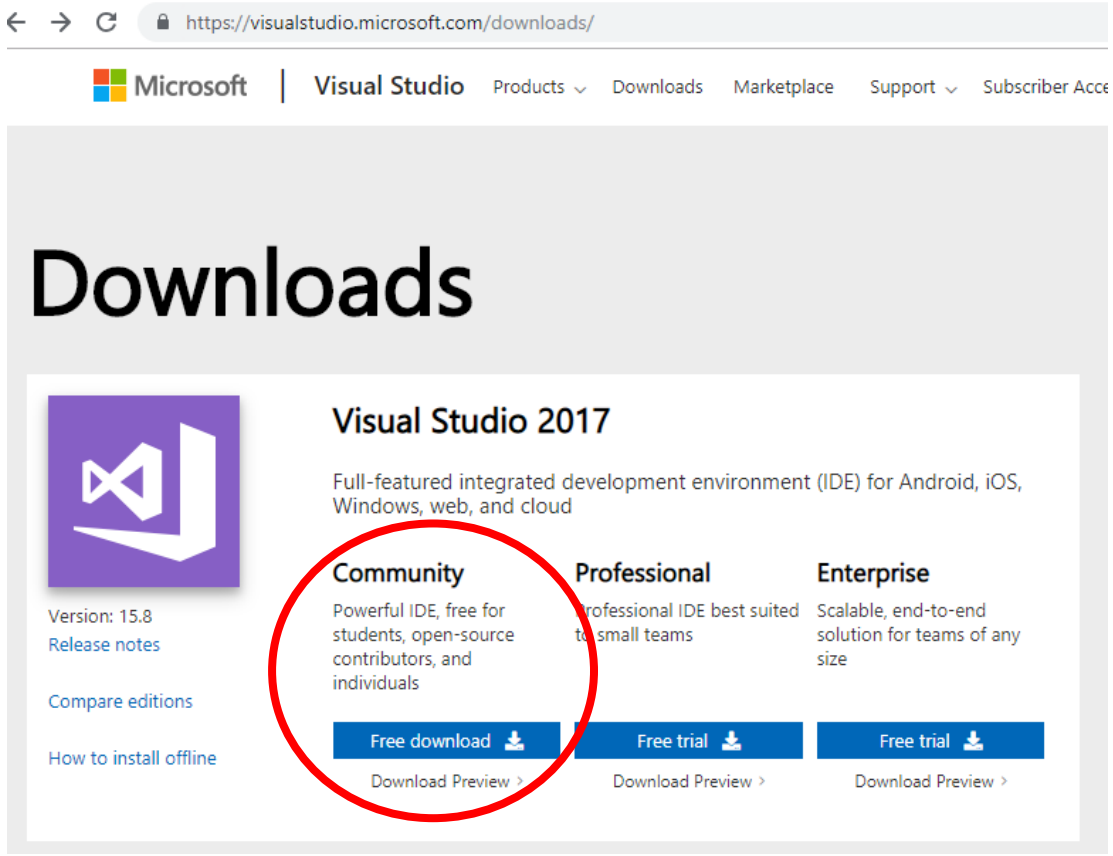
... Laptops closed

This presentation helps in delivering the lecture.
Take notes, interact and read text book to learn and gain knowledge.

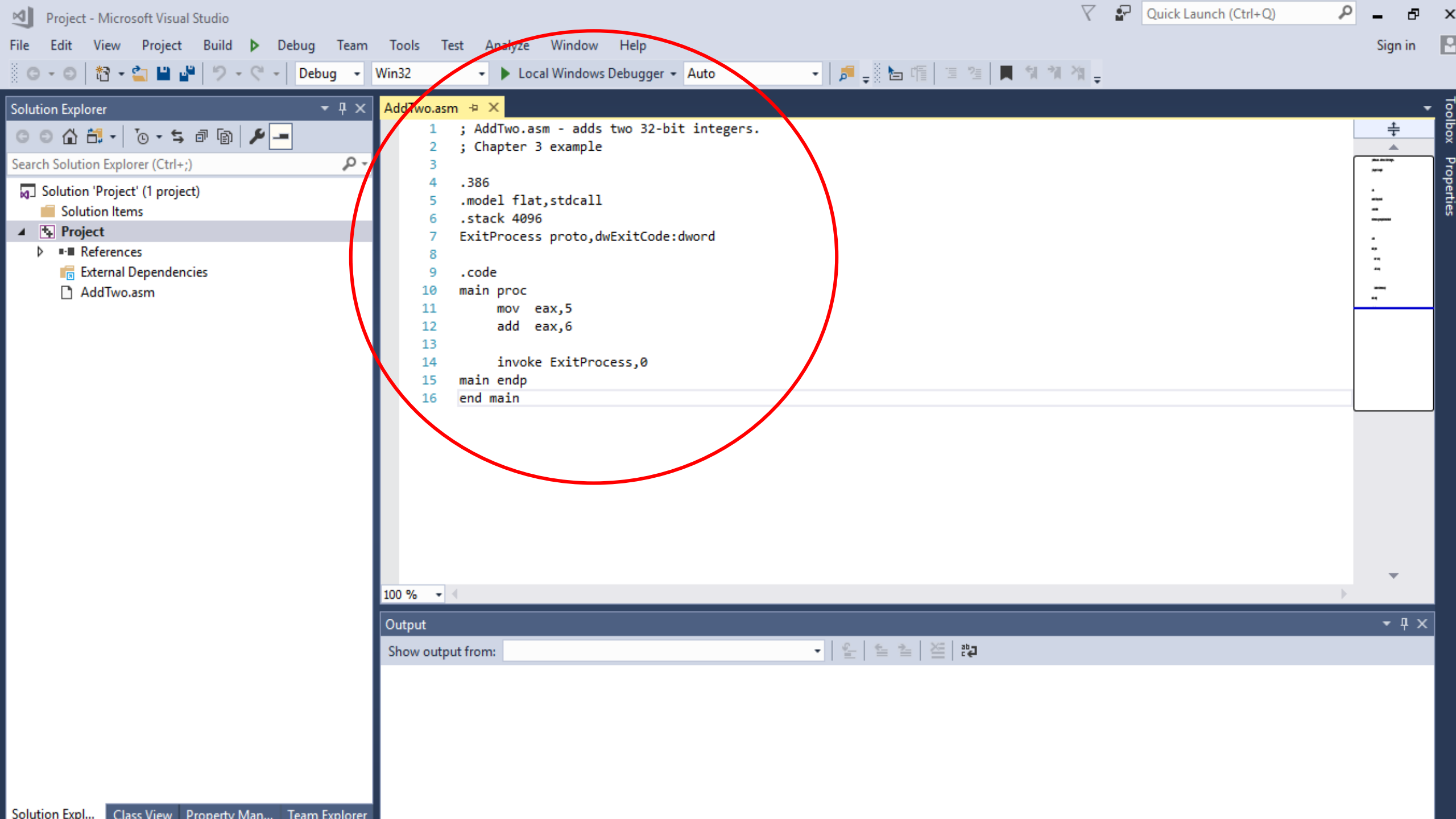
Today's Topics

- Homework: Visual Studio 2017 installation and assemble Irvine code
- Writing Assembly Programs

Install VS 2017 and Try x86 Assembly @ home



- Getting Started with MASM and Visual Studio 2017
  http://kipirvine.com/asm/gettingStartedVS2017/index.htm


Project - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Sign in
Debug
Win32
Local Windows Debugger
Auto
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Project' (1 project)
Solution Items
Project
References
External Dependencies
AddTwo.asm
AddTwo.asm
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 %
Output
Show output from:
Solution Expl...
Class View
Property Man...
Team Explorer
Toolbox
Properties

Chapter # 2

2.7 Key Terms

32-bit mode
64-bit mode
address bus
application programming interface (API)
arithmetic logic unit (ALU)
auxiliary carry flag
basic program execution registers
BIOS (basic input–output system)
bus
cache
carry flag
central processor unit (CPU)
clock
clock cycle
clock generator
code cache
control flags
control unit
instruction pointer
interrupt flag
Level-1 cache
Level-2 cache
machine cycle
memory storage unit
MMX registers
motherboard
motherboard chipset
operating system (OS)
overflow flag
parity flag
PCI (peripheral component interconnect)
PCI express
process
process ID
programmable interrupt controller (PIC)
programmable interval timer/counter
data bus
data cache
device drivers
direction flag
dynamic RAM
EFLAGS register
extended destination index
extended physical addressing
extended source index
extended stack pointer
fetch-decode-execute
flags register
floating-point unit
general-purpose registers
instruction decoder
instruction execution cycle
instruction queue
programmable parallel port
protected mode
random access memory (RAM)
read-only memory (ROM)
real-address mode
registers
segment registers
sign flag
single-instruction, multiple-data (SIMD)
static RAM
status flags
system management mode (SMM)
Task Manager
virtual-8086 mode
wait states
XMM registers
zero flag

3.1.1 First Assembly Language Program

```
1: main PROC
2:    mov eax,5                    ; move 5 to the eax register
3:    add eax,6                    ; add 6 to the eax register
4:
5:    INVOKE ExitProcess,0         ; end the program
6: main ENDP
```

Let's go through the program one line at a time: Line 1 starts the **main** procedure, the entry point for the program. Line 2 places the integer 5 in the **eax** register. Line 3 adds 6 to the value in EAX, giving it a new value of 11. Line 5 calls a Windows service (also known as a function) named **ExitProcess** that halts the program and returns control to the operating system. Line 6 is the ending marker of the main procedure.

3.1.9 Directives

A *directive* is a command embedded in the source code that is recognized and acted upon by the assembler. Directives do not execute at runtime, but they let you define variables, macros, and procedures. They can assign names to memory segments and perform many other housekeeping tasks related to the assembler. Directives are not, by default, case sensitive. For example, **.data**, **.DATA**, and **.Data** are equivalent.

The following example helps to show the difference between directives and instructions. The DWORD directive tells the assembler to reserve space in the program for a doubleword variable. The MOV instruction, on the other hand, executes at runtime, copying the contents of **myVar** to the EAX register:

```
myVar  DWORD 26
mov    eax,myVar
```

Although all assemblers for Intel processors share the same instruction set, they usually have different sets of directives. The Microsoft assembler's REPT directive, for example, is not recognized by some other assemblers.

Defining Segments One important function of assembler directives is to define program sections, or *segments*. Segments are sections of a program that have different purposes. For example, one segment can be used to define variables, and is identified by the .DATA directive:

```
.data
```

The .CODE directive identifies the area of a program containing executable instructions:

```
.code
```

The .STACK directive identifies the area of a program holding the runtime stack, setting its size:

```
.stack 100h
```

Appendix A contains a useful reference for directives and operators.

3.1.10 Instructions

An *instruction* is a statement that becomes executable when a program is assembled. Instructions are translated by the assembler into machine language bytes, which are loaded and executed by the CPU at runtime. An instruction contains four basic parts:

- Label (optional)
- Instruction mnemonic (required)
- Operand(s) (usually required)
- Comment (optional)

This is how the different parts are arranged:

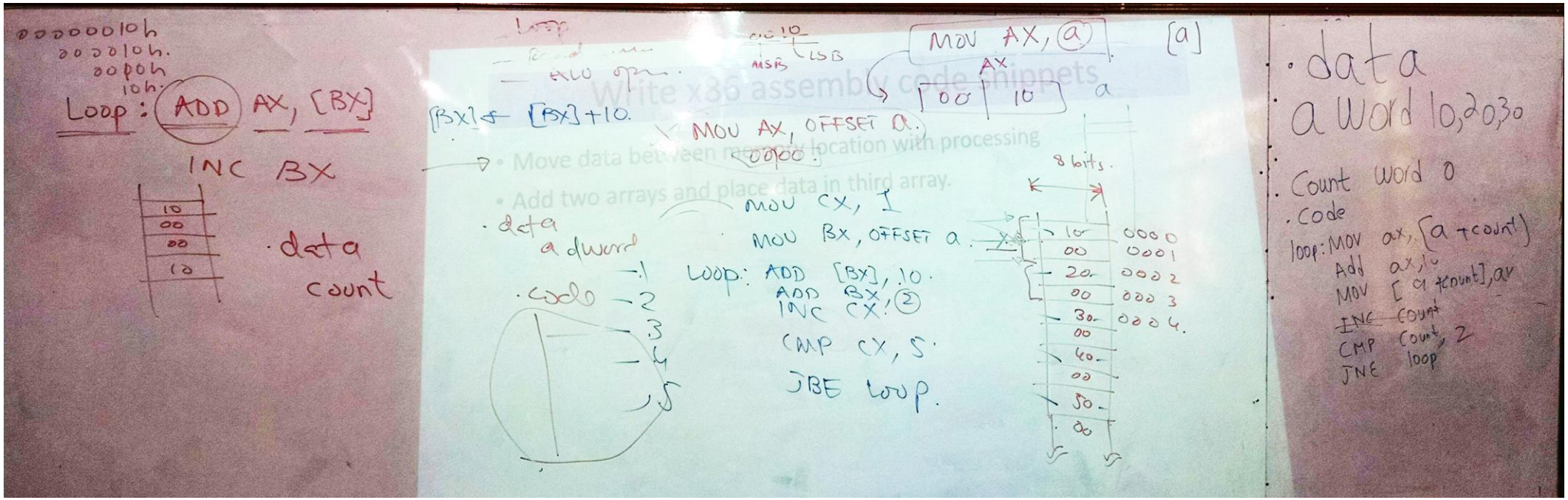
```
[label:] mnemonic [operands] [;comment]
```

Mnemonic	Description
MOV	Move (assign) one value to another
ADD	Add two values
SUB	Subtract one value from another
MUL	Multiply two values
JMP	Jump to a new location
CALL	Call a procedure

Example	Operand Type
96	*Integer literal*
2 + 4	Integer expression
eax	Register
count	Memory

Write x86 assembly code snippets

- Add 10 to existing elements to memory



Write x86 assembly code snippets

- Add two arrays and place data in third array.

```
.data
    array1 dword 50 dup (10)
    array2 dword 50 dup (20)
    array3 dword 50 dup (?)
```

- Write manipulation code yourself.