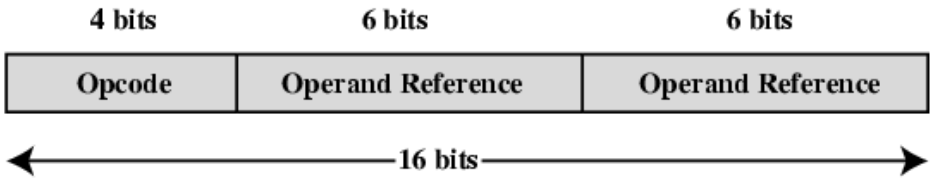
Instructions:

Almost every instruction in assembly/low language has two parts. One is **opcode** the other is address **references.**

If we break down a 16-bit instruction, then we may get following components.



Opcode: This is a unique pattern which identify "what operation is to be done".

Operand references: This is the unique address of a memory location "where the operation is needed to be done".

For example,

If `add a,b` is an instruction, then `add` is opcode, which is telling the CPU to add something. `a,b` are the memory locations whose values will be added in the result of the operation.

Depending upon the architecture, we may have a third reference where the result of the operation will be stored or just one reference. In case of one reference, accumulator register "ac" will be used implicitly to hold the temporary data during arithmetic operations.

Add operation can be executed as:

One reference:

```
mov a           ;move value of a into accumulator register "ax".
add b           ;add value of b into value of "ax" and keep in it.
stor c          ;copy the value of accumulator register in "c variable".
```

Two References:

```
add a,b         ;add value of b into a and keep the result in a.
stor c,a        ;copy the value of a into c.
```

Three References:

```
add c,a,b       ;add values of a and b, and store the result in c.
```

Register and their types:

Register are small but fastest memory locations located inside CPU. They are used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU. There are various types of registers used for different purposes. Mostly used Registers named as AC or **Accumulator**, Data Register or DR, the AR or **Address Register**, **Program Counter** (PC), **Memory Data Register** (MDR).

Registers are grouped into several categories as follows:

- **Four general-purpose registers**, AX, BX, CX, and DX.
- **Four special-purpose registers**, SP, BP, SI, and DI.
- **Four segment registers**, CS, DS, ES, and SS.
- **The instruction pointer**, IP (sometimes referred to as the program counter).
- **The status flag register**, FLAGS.

General Purpose Registers:

32 bits registers:

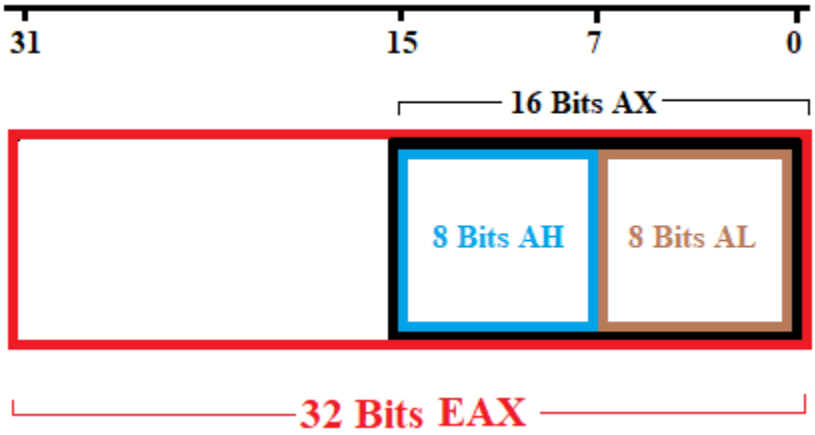
EAX, EBX, ECX, EDX are the 32, contains 4 bytes.

16 bits registers:

AX, BX, CX, DX are the 16 bit registers, contain lower two bytes of 32 bit registers.

8 bit registers:

AL, AH, BH, BL, CH, CL, DH, DL are the 8 bit register. The "H" and "L" suffix on the 8 bit registers stand for high byte and low byte of 16 bits registers.



EAX,AX,AH,AL :
Called as the "accumulator''; some of the operations, such as MUL and DIV, require that one of the operands be in the accumulator. Some other operations, such as ADD and SUB, may be applied to any of the registers (that is, any of the eight general- and special-purpose registers) but are more efficient when working with the accumulator.

EBX,BX,BH,BL :
Called as the "base'' register; it is the only general-purpose register which may be used for indirect addressing. For example, the instruction MOV [BX], AX causes the contents of AX to be stored in the memory location whose address is given in BX.

ECX,CX,CH,CL :
Called as the "count'' register. The looping instructions (LOOP, LOOPE, and LOOPNE), the shift and rotate instructions (RCL, RCR, ROL, ROR, SHL, SHR, and SAR), and the string instructions (with the prefixes REP, REPE, and REPNE) all use the count register to determine how many times they will repeat.

EDX,DX,DH,DL :
Called as the "data'' register; it is used together with AX for the word-size MUL and DIV operations, and it can also hold the port number for the IN and OUT instructions, but it is mostly available as a convenient place to store data, as are all of the other general-purpose registers.

Special Purpose Register:

SP is the stack pointer, indicating the current position of the top of the stack. You should generally never modify this directly, since the subroutine and interrupt call-and-return mechanisms depend on the contents of the stack.

BP is the base pointer, which can be used for indirect addressing similar to BX.

SI is the source index, used as a pointer to the current character being read in a string instruction (LODS, MOVS, or CMPS). It is also available as an offset to add to BX or BP when doing indirect addressing; for example, the instruction MOV [BX+SI], AX copies the contents of AX into the memory location whose address is the sum of the contents of BX and SI.

DI is the destination index, used as a pointer to the current character being written or compared in a string instruction (MOVS, STOS, CMPS, or SCAS). It is also available as an offset, just like SI

A word is either 16-bit or 32-bit long depending upon the architecture. You can get clear idea of word in theory class.

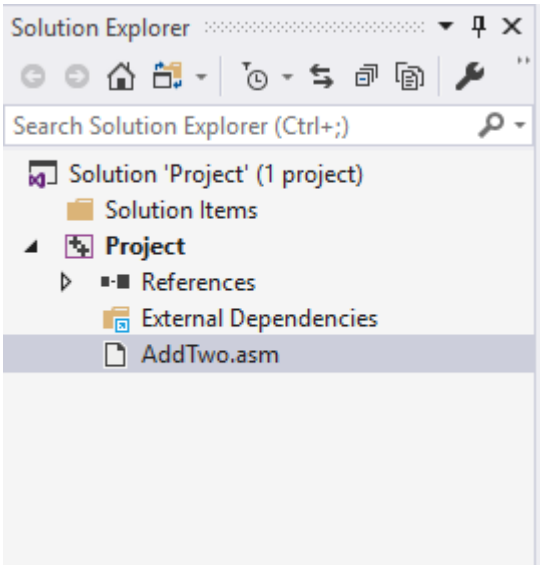
Getting Started with Visual Studio:

Opening a Project

Visual Studio requires assembly language source files to belong to a *project*, which is a kind of container. A project holds configuration information such as the locations of the assembler, linker, and required libraries. A project has its own folder, and it holds the names and locations of all files belonging to it.

Do the following steps, in order:

1. Start Visual Studio.
2. To begin, open our sample Visual Studio project file by selecting **File/Open/Project** from the Visual Studio menu.
3. Navigate to your working folder where you unzipped our project file, and select the file named **Project.sln**.
4. Once the project has been opened, you will see the project name in the Solution Explorer window. You should also see an assembly language source file in the project named AddTwo.asm. Double-click the file name to open it in the editor.



You should see the following program in the editor window:

```
; AddTwo.asm - adds two 32-bit integers.
; Chapter 3 example

.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
```

In the future, you can use this file as a starting point to create new programs by copying it and renaming the copy in the Solution Explorer window.

Adding a File to a Project: If you ever need to add an .asm file to an open project, do the following: (1) Right-click the project name in the Visual Studio window, select Add, select Existing Item. (2) In the *Add Existing Item*dialog window, browse to the location of the file you want to add, select the filename, and click the Add button to close the dialog window.

Build the Program

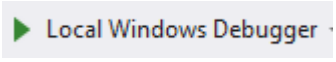
Now you will build (assemble and link) the sample program. Select **Build Project** from the Build menu. In the Output window for Visual Studio at the bottom of the screen, you should see messages similar to the following, indicating the build progress:

```
1>------ Build started: Project: Project, Configuration: Debug Win32 ------
1>  Assembling ..\Project32_VS2015\AddTwo.asm...
1>  Project.vcxproj -> ...\Project32_VS2015\Debug\Project.exe
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
```

If you do not see these messages, the project has probably not been modified since it was last built. No problem--just select **Rebuild Project** from the Build menu.

Run the Program

Select **Local Window Debugger**. The following console window should appear, although your window will be larger than the one shown here:



The "Press any key to continue..." message is automatically generated by Visual Studio.

Press any key to close the Console window.

Copying a Source File

One way to make a copy of an existing source code file is to use Windows Explorer to copy the file into your project directory. Then, right-click the project name in Solution Explorer, select Add, select Existing Item, and select the filename. Or you can copy and paste code from .asm file to any other text file to save the code using same project again and again.

Example 1:

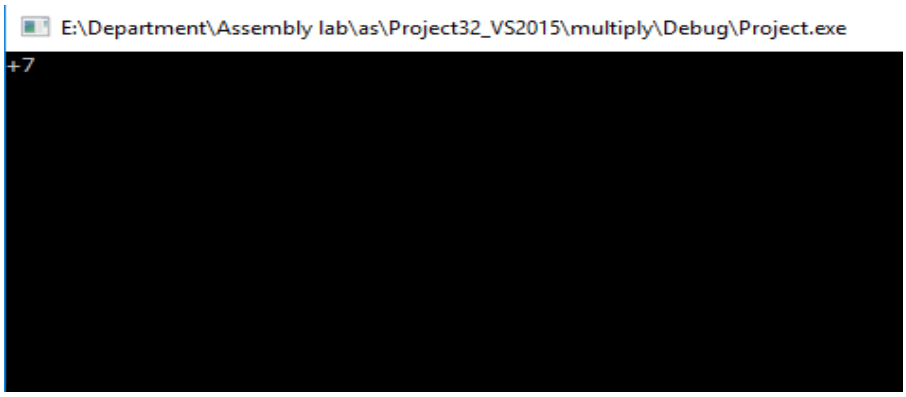
Add two numbers.

In **.data** variables are initialized. Firstly the variable name is written than mention their data type and value of the variable. If there is no value to assign the variable then the ? is used.

For example a dword 0

In **.code** all the commands are written. Registers are used to store the value. By default **eax** register use for the addition or subtraction. In addition 2 arguments are pass the **add** function.

```
1   ; AddTwo.asm - adds two 32-bit integers.
2
3
4   INCLUDE Irvine32.inc
5   .data
6
7
8   .code
9   main PROC
10
11  mov eax, 3    ; move 3 into eax register
12  add eax, 4    ; 4 will be added with the value of eax register(3)
13
14
15  call writeint    ; show the output on console
16  call readint
17  exit
18  main ENDP
19  END main
```



Example 2:

Multiply 2 numbers.

In multiplication, **mul** function accepts only one argument and it takes automatically **eax** register as a second argument.

```
4   INCLUDE Irvine32.inc
5   .data
6   a dword 0
7   b dword 0
8
9   .code
10  main PROC
11
12  mov eax, 3   ; move 3 into eax register
13  mov a, eax   ; move value of eax register into a variable
14  mov eax, 4   ;  move 4 into eax register
15
16  mul a        ; multiply a with eax register
17
18  call writeint   ; show the output on console
19  call readint
20  exit
21  main ENDP
22  END main
```

Lab Tasks:

(10 - 7) * (5 + 6) * 9

5 * (3 + 7 * 21) + 50