COAL LAB MANUALS

Department of Computer Science

U.E.T LAHORE

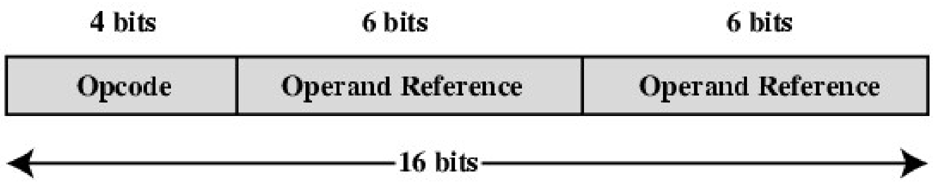
Contents

Lab 1

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". Operands can be immediate value, variable, memory address or register.

For example,

If `add a,b` is an instruction, then `add` is a nemonic which represent unique opcode of add operation, which is telling the CPU to add something. `a,b` are the operands 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 "ax" will be used implicitly to hold the temporary data during arithmetic operations.

Add operation can be executed as:

One reference:

```
mov a          ;copy value of a into accumulator register "ax".
add b          ;add value of b into value of "ax" and keep in it.
sto 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.
sto 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:

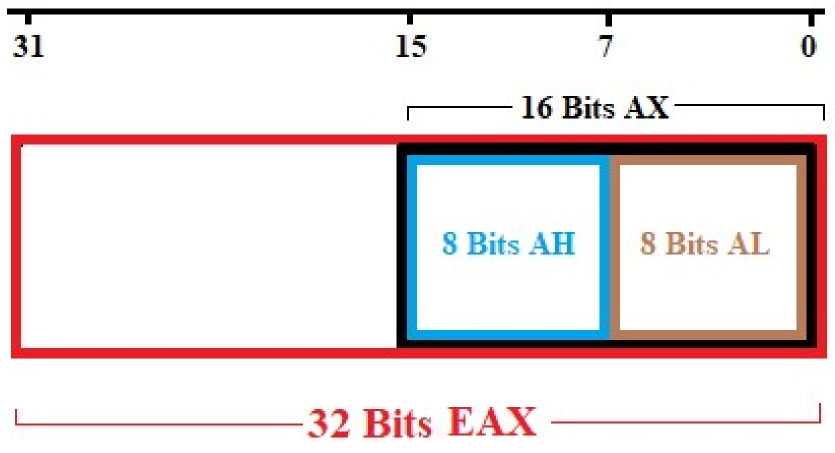
EAX, EBX, ECX, EDX are the 32 bit registers, contains 4 bytes.

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 registers. The "H" and "L" suffix on the 8 bit registers stand for high byte and low byte of 16 bits registers.



<u>EAX, AX, AH, AL</u>:

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.

<u>EBX, BX, BH, BL</u>:
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.

<u>ECX, CX, CH, CL</u>:
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.

<u>EDX, DX, DH, DL</u>:
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 all of the other general-purpose registers.

Special Purpose Register:

- <u>SP</u> is the <u>stack pointer</u>, 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.

- <u>BP</u> is the <u>base pointer</u>, which can be used for indirect addressing similar to BX.

- <u>SI</u> is the <u>source index</u>, 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.

- <u>DI</u> is the <u>destination index</u>, 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

Word is a 16-bit long memory storage unit.

Getting Started with Visual Studio:

For Lab:

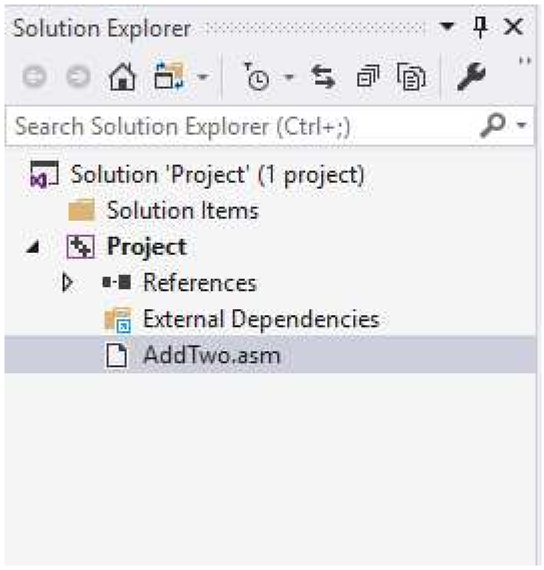
Click here and follow the link to download Visual Studio Project, extract it anywhere in computer.

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

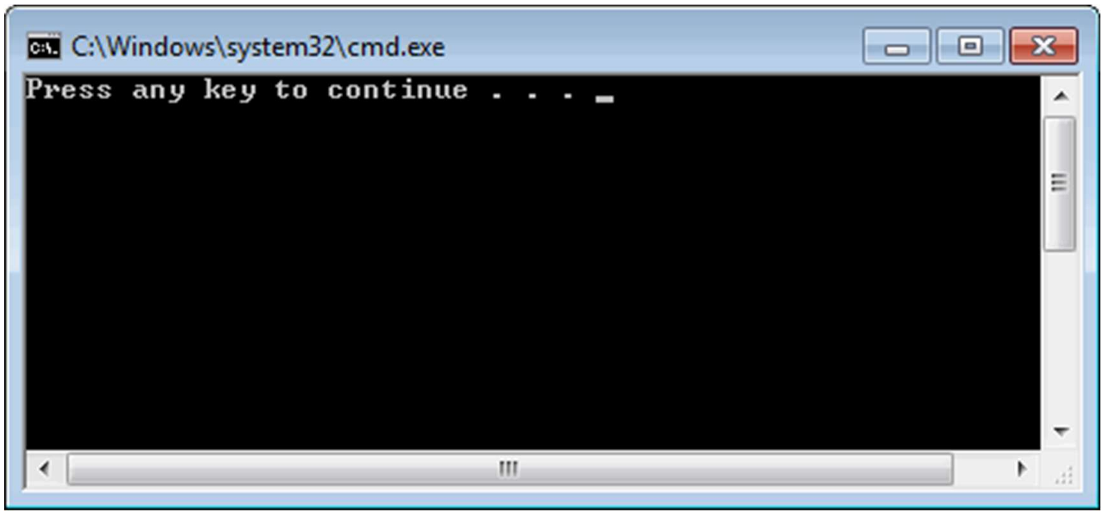
Now you will build (assemble and link) the sample program. Select **Build Project** from the Build menu. In the Output window for Visual Studio at the bottom of the screen, you should see messages similar to the following, indicating the build progress:

```
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:



```
Press any key to continue . . . _
```

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

Press any key to close the Console window.

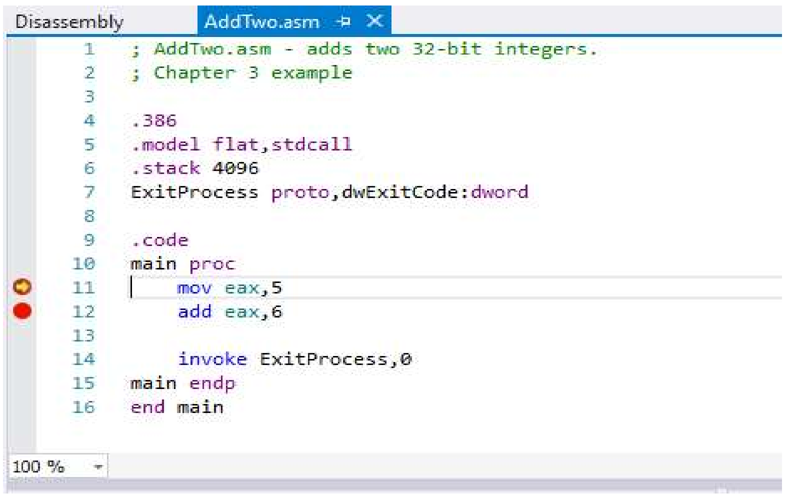
Running the Sample Program in Debug Mode

Setting a BreakPoint

If you set a breakpoint in a program, you can use the debugger to execute the program a full speed (more or less) until it reaches the breakpoint. At that point, the debugger drops into single-step mode.

1. In our sample program, click the mouse along the border to the left of the **mov eax,5** statement. A large red dot should appear in the margin.
2. Select *Local Window Debugger*. The program should run, and pause on the line with the breakpoint, showing the Yellow arrow.
3. Press *Continue* until the program finishes.

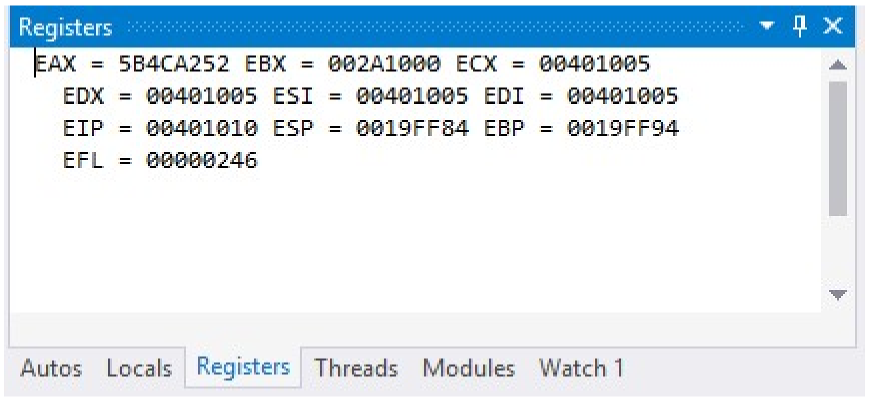
You can remove a breakpoint by clicking its red dot with the mouse. Take a few minutes to experiment with the Debug menu commands. Set more breakpoints and run the program again.



```
; 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
```

Registers

You can view current values of different registers while debugging a program. Press Alt+Ctrl+G to open register window while program is debugging.



```
EAX = 5B4CA252 EBX = 002A1000 ECX = 00401005
  EDX = 00401005 ESI = 00401005 EDI = 00401005
  EIP = 00401010 ESP = 0019FF84 EBP = 0019FF94
  EFL = 00000246
```

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.

Structure of Assembly Program

Assembly programs have mainly two segments, one is data and other is code. Data part contains declared variables like words and strings etc. Code part contains code.

Single Line Comments: In assembly language, single line comments starts with semicolon ;

```
INCLUDE Irvine32.inc

.data
 ;Variables and Data

.code
main PROC ; Main Procedure/Function start

      ;Code

      exit   ; Exiting the Process

main ENDP ; Main Function/End

END main  ; Marks the END of the end of Program, and main identify entry procedure.
```

This a snippet of code which does nothing at all. But it contains the structure of assembly program. We would write code inside our "`main PROC`" (Main Procedure) which is entry point of the program.

LAB 1

Mnemonics	Syntax	Working	Example
mov	`mov op1,op2`	Copy data from operand 2 to operand 1.	`mov eax, 5` 5 is being moved into eax register.
add	`add op1, op2`	Add operand 2 into operand 1 and keep it stored in operand 1.	add eax, 6 integer 6 sums with eax and get store in eax.
mul	`Mul op1`	It multiplies operand to eax and store result in edx:ex	`Mul b` It multiplies b with the content of eax and store result in edx:eax
call	`call proc1`	Calling the procedure1	`Call writeint` It will print the value present in eax register in console.

(All arithmetic operations are performed in registers. Variable will only be used to store data. No processing can be done inside a variable.)

When two 32-bit integers are multiplied, output is 64-bit integer. In 32-bit architecture, 64-bit registers don't exist. Hence to accommodate 64-bit integer, upper 32-bits are moved to edx and lower 32-bits are moved to eax.

Procedures:

These are procedures present in Kip Irvine library of assembly Language.

writeint:

It prints the value of eax register on console.

readint:

It generates an interrupt and waits for the user to enter something in the console and copy the input to eax register.

Variable Declaration:

Syntax:

```
name   TYPE    value
```

Example:

```
num    dword  7
```

Above num is a variable of type DWORD, (dword has length 32-bit) and 7 is the value of num. If we don't want to initialize variable, we can use ? instead of 7.

You will know about all type of variables in coming labs.

Example 1: Add Two Numbers

```
INCLUDE Irvine32.inc

.data

.code
main PROC
      mov eax, 6        ; copy 6 into eax
      add eax, 10       ; add 10 into eax and store result in eax
      call writeint     ; print value of eax register on console

      call readint      ; Stopping Console from disappearing
      exit
main ENDP
END main
```

```
Result: +16
```

+ sign shows that the result is signed.

Example 2: Multiply Two Numbers

```
INCLUDE Irvine32.inc

.data
b dword 40                 ; declares 32-bit variable with value of 40
.code
main PROC
       mov eax, 12         ; Copy 12 to 32-bit register eax
       mul b               ; value of b is multiplied with value of eax and result is
                           ; stored in eax.

       call writeint       ; print value of eax register on console
       mov eax, edx        ; move the value of edx to eax so that it can be printed

       call writeint
       call readint        ; Stopping Console from disappearing
       exit
main ENDP
END main
```

Result: +480+0

Here +480 is the value of eax register and +0 is the value of edx.

Why edx is zero after multiply operation?

Because, eax (32-bit reigster) can hold a unsigned integer value of 2^{32} = 4294967295. So if the unsinged value of output after the multiplication is less than or equal to 4294967295, eax alone will be able to handle it and edx will remain zero.

Lab Tasks:

1. Write a program to solve this equation using only eax register (10 - 7) * (5 + 6) * 9. (*Note: Multiple variables can be used).*

2. Write a program to solve this equation without using any variable (4 * 5) - (3 + 7 * 21). (*Note: Multiple registers can be used*).