National University of Sciences and Technology

School of Electrical Engineering and Computer Science

Department of Electrical Engineering

Lab Engineer: Ms. Bareera Anam

|  |  |
| --- | --- |
| **CS-235: Computer Organization and Assembly Language** | |
| **Faculty Member** | **Semester** |
|  |  |
| **Class/Section** | **Date** |
|  |  |

**Lab 1: Programming in Assembly Language using MASM**

Muhammad Faseeh  
456267  
BSCS 13-E

**Lab 1: Programming in Assembly Language using MASM**

**Objective**

The aim of the first lab is familiarization with MASM and writing and testing our first program in Assembly Language.

**Introduction to MASM**

The Microsoft Macro Assembler (MASM) is an x86 assembler for Microsoft Windows that uses the Intel syntax. Assembly language is a great tool to understand how a computer works and with the help of MASM you will be able to assemble and run your programs written in Assembly language.

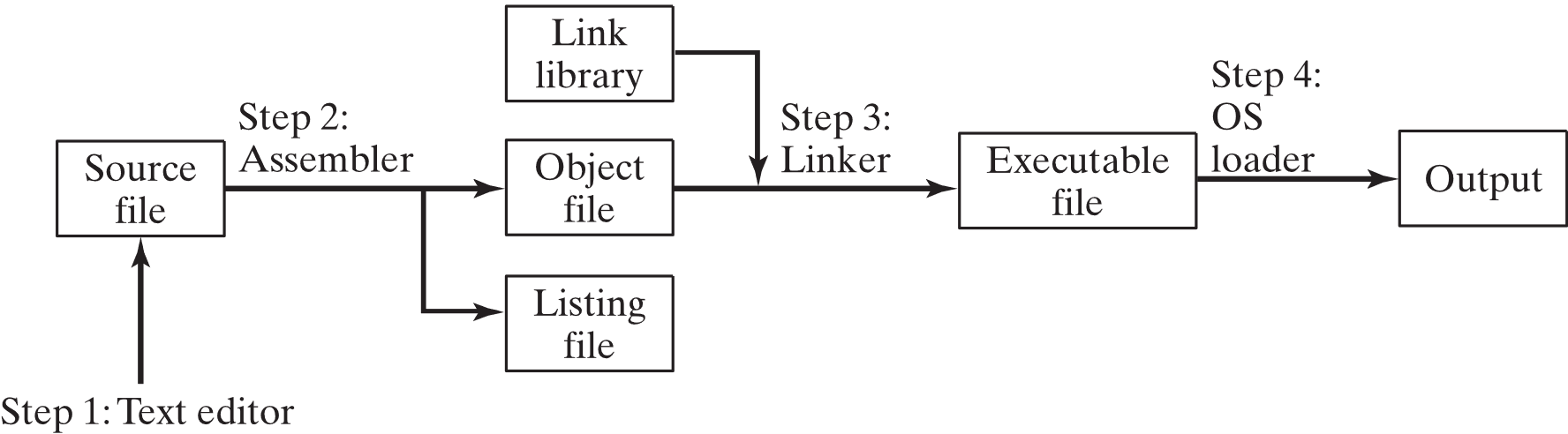
**Writing Assembly Language Programs**

You can write Assembly language programs in any text editor e.g. Notepad etc. However, you have to make sure that you save your programs with an extension of asm i.e. if you name your file example then it should be saved by going to saveAs and then typing example.asm in the file name and selecting All Files in the file Type.

Once you have installed MASM on your PC and written your program then you have to assemble and link your programs before they can be executed.

**Assemble – Link – Execute Cycle**

*Source: Assembly Language for x86 Systems, Kip Irvine*

The process of editing, assembling, linking, and executing assembly language programs is summarized in Figure below. Following is a detailed description of each step.

*Step 1:* A programmer uses a text editor to create an ASCII text file named the *source file*.

*Step 2:* The assembler (file ML.exe)reads the source file and produces an *object file,* a machine-language translation of the program. Optionally, it produces a *listing file*. If any errors occur, the programmer must return to Step 1 and fix the program.

*Step 3:* The linker (file Link32.exe) reads the object file and checks to see if the program contains any calls to procedures in a link library. The linker copies any required procedures from the link library, combines them with the object file, and produces the *executable file*. Optionally, the linker can produce a *map file*.

*Step 4:* The operating system loader utility reads the executable file into memory and branches theCPU to the program’s starting address, and the program begins to execute.

At the end the useful files generated are as follows:

*Example.obj*

*Example.lst*

*Example.exe*

Example.exe is the executable file that can now be run by typing example on the DOS prompt and pressing enter.

**How to use MASM**

**Step 1**

Open Notepad copy example code given below to Notepad and Save As the fie with *.asm* extension. Make sure you saved the file in the directory “C:\masm615” which is directory in which masm is installed .

**Step 2**

Open the command prompt by typing *cmd* in Run and change your current path to “C:\masm615”

By typing following commands in command prompt

**cd** c:\

**cd** masm615

or

**cd** c:\masm615

**Step 3**

Use make32.bat file to assembling and linking by typing following in command prompt

**make32** example

This will create following files

*Example.obj*

*Example.lst*

*Example.exe*

*Note: make32.bat is batch file containing list of commands for assembling , linking and setting r*

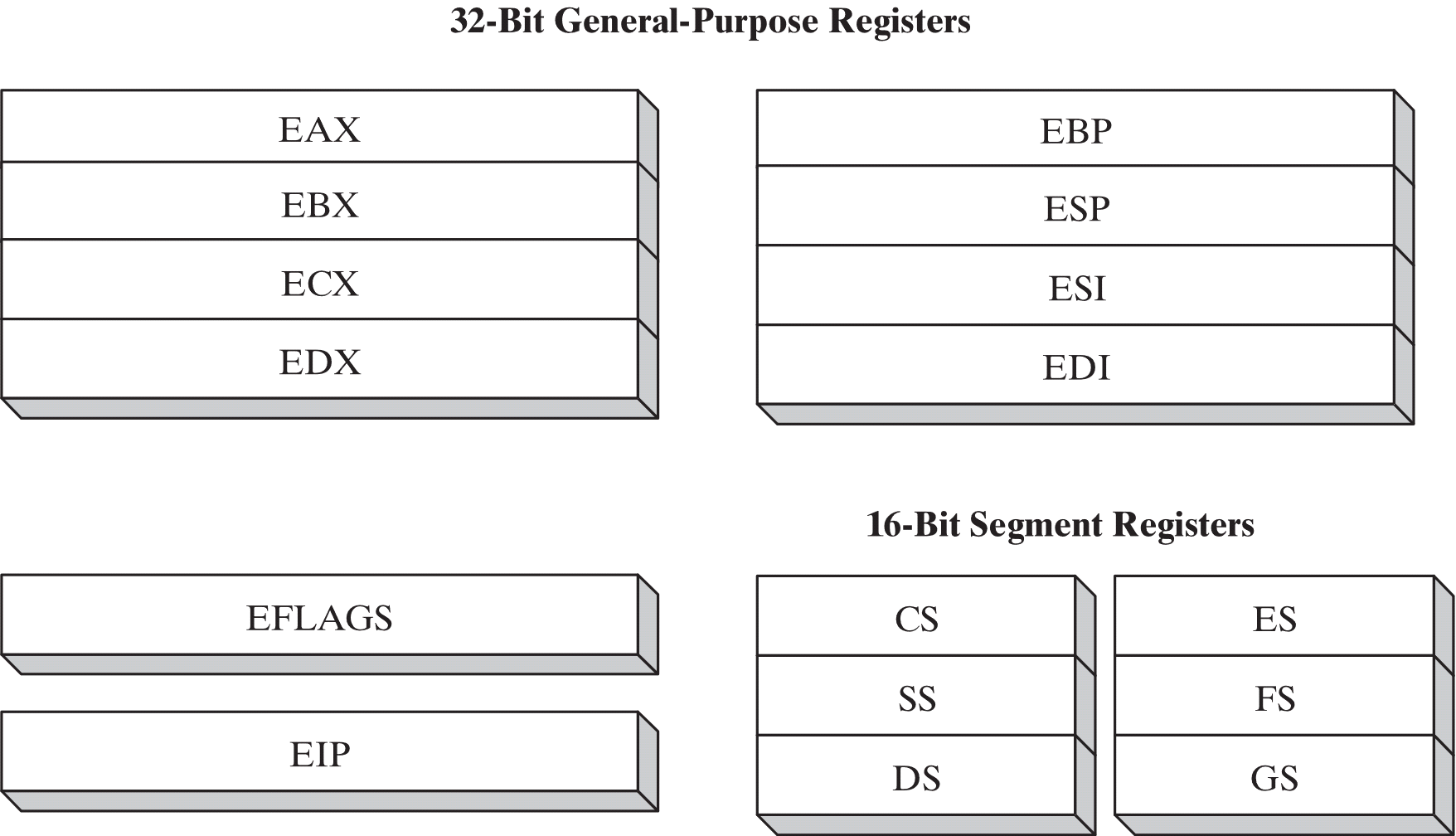
**Step 4**

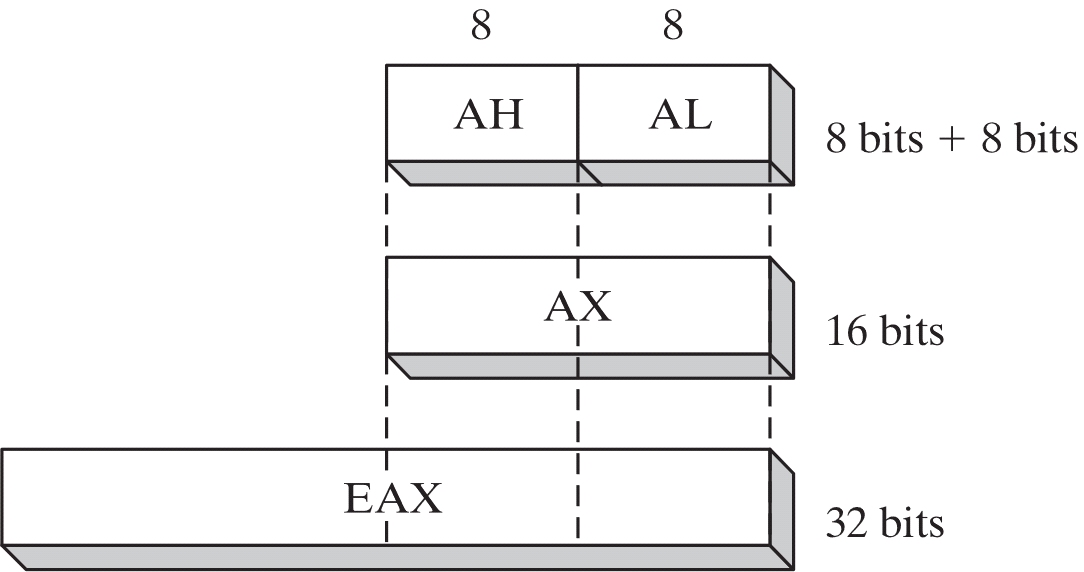
Run the exe file by typing following in command prompt

Example.exe

This will show the output of your code.

**CPU Registers**

Registers are special memory locations on the CPU. One important difference between older and later processors is that the pre-386 processors are 16-bit instead of 32-bit.There are 8 32-bit general purpose registers. The first 4, eax, ebx, ecx, and edx can also be accessed using 16 or 8-bit names. ax gets the first 16 bits of eax, al gets the first 8bits, and ah gets bits 9-16. The fig below shows all the general purpose and special purpose registers and their sizes.



*Source: Assembly Language for x86 Systems, Kip Irvine*

**Example Program**

**TITLE** Add two registers (example.asm)

; The comments are given after the semi colon on a line

; This program adds 32-bit unsigned

; integers and stores the sum in the ecx register

**Include** irvine32.inc

**.data**

;variable declarations go here

**.code**

**Main** **Proc**

;instructions go here

**Mov** **eax**, 30 ;Assembly Language is NOT case sensitive

**Mov** **ebx**, 20

**Add** **ecx**, **eax**

**Add** **ecx**, **ebx**

**Call** dumpregs ;displays the result on the screen by displaying

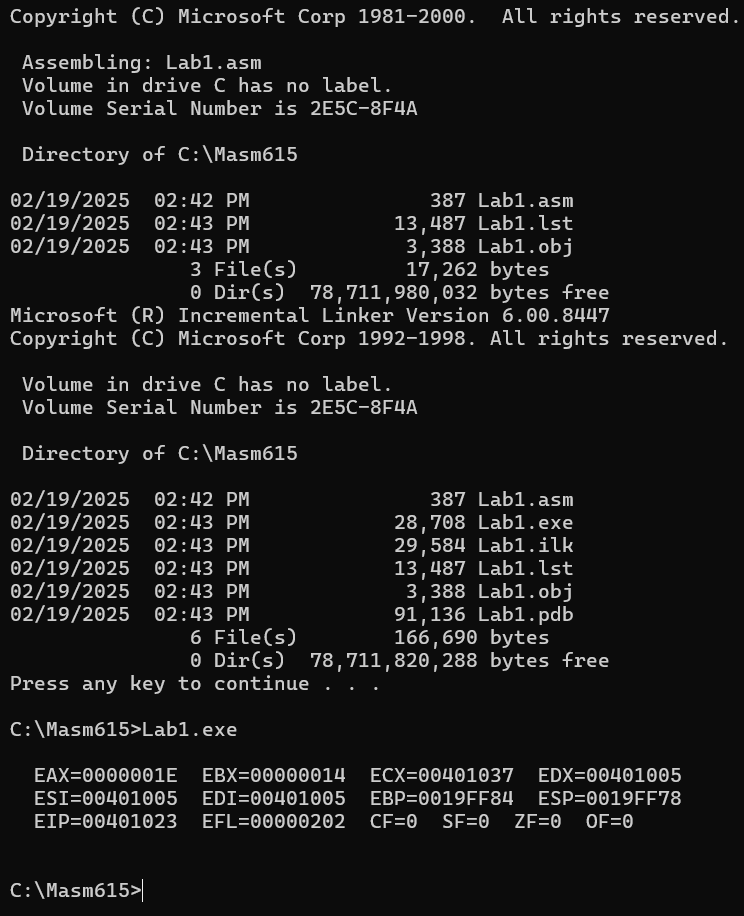
;all register values

**Exit**

**Main endp**

**Exercises**

**Exercise 1:** Write the assembly language program given in the Lab handout, assemble it and run it as per the instructions in the handout.



**Exercise 2:** Note down the contents of registers EAX, EBX and ECX as displayed by the program:

**EAX=0000001E EBX=00000014 ECX=00401037**

**Exercise 3:** Check the contents of registers against the instructions.

**EAX:**

**Mov** **eax**, 30

**The value 30 (decimal) is moved into the eax register.**

**eax becomes 0x1E (30 in hexadecimal).**

**This matches the content.**

**EBX:**

**Mov** **ebx**, 20

**The value 20 (decimal) is moved into the ebx register.**

**ebx becomes 0x14 (20 in hexadecimal).**

**This matches the content.**

**ECX:**

**Add** **ecx**, **eax**

**This instruction adds the value in eax (0x1E, or 30) to ecx.**

**Add** **ecx**, **ebx**

**This instruction adds the value in ebx (0x14, or 20) to ecx.**

**The ecx value comes out 00401037 which is not equal to 50 in decimal.**

**Exercise 4:** Do the contents of register ECX match the expected result? If not, what step needs to be taken?

**If ecx is not set to an initial value before performing the addition, it could contain an unpredictable or random value. As a result, when executing Add ecx, eax, the sum might be affected by this unknown value.**

**To avoid this issue, we can explicitly assign ecx a value of 0 at the beginning of the program, ensuring that the additions start from a defined state.**

**Exercise 5:** Modify the source code to get the right result in the register ECX, re-assemble, and re-run the program.

TITLE Add two registers (example.asm)

; This program demonstrates how to add two 32-bit unsigned integers

; and store the result in the ECX register. The values are stored in

; the EAX and EBX registers before performing the addition.

INCLUDE irvine32.inc ; Include the Irvine32 library for I/O functions

.data

; Data section (no variables needed for this program)

.code

Main PROC

; Move values into registers

Mov eax, 30 ; Load 30 into the EAX register

Mov ebx, 20 ; Load 20 into the EBX register

Mov ecx, 0 ; Initialize ECX to 0

; Perform addition

Add ecx, eax ; Add EAX value (30) to ECX

Add ecx, ebx ; Add EBX value (20) to ECX

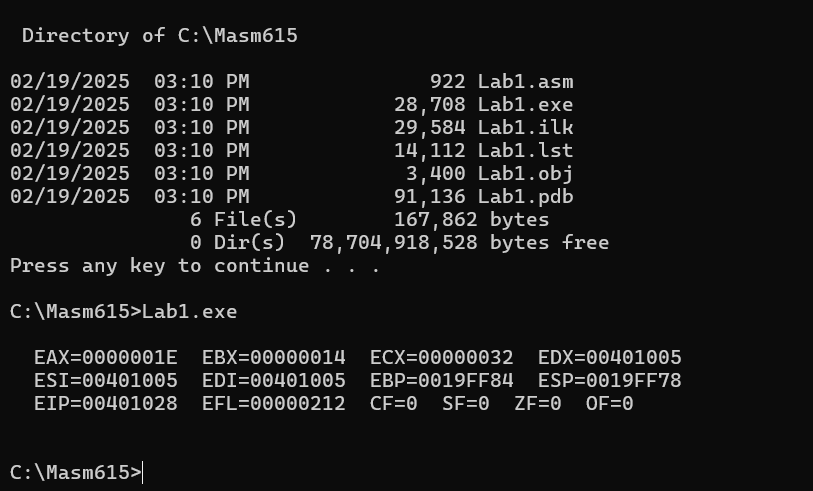
; Display register values on screen

Call dumpregs ; Irvine32 procedure to show register contents

Exit ; Exit the program

Main ENDP

END Main ; End of program, specifies the entry point



**Exercise 6:** Verify that the contents of the ECX register are now correct.

**The first operation, Add ecx, eax, adds the value stored in EAX (which is 30 or 0x1E) to ECX (which was initially set to 0). As a result, ECX is updated to 30 (or 0x1E).**

**The second operation, Add ecx, ebx, adds the value in EBX (which is 20 or 0x14) to the current value of ECX (which is now 30 or 0x1E). This updates ECX to 50 (or 0x32).**

**Expected Outcome**:

After these additions, the final value of the ECX register should be 0x32 (50 in decimal), which is the correct sum of the two operands (30 + 20).

**Verification via dumpers**:

The dumpregs function is used to display the contents of all registers, including ECX. Upon execution of the program with the explicit initialization of ECX, the expected value of ECX (50, or 0x32) is displayed, confirming that the register contents are now correct and match the intended result.