LAB 02

COMPUTER ORGANIZATION AND ASSEMBLY LANG(COAL)

_____   _____   ___

STUDENT NAME   ROLL NO   SEC

_____

SIGNATURE & DATE

MARKS AWARDED: _____

_____

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES (NUCES), KARACHI

Lab Session 02

Objectives:

- Debugging of programs
- Basic elements of Assembly language
- Defining Data
- Intrinsic Data Types

Steps Involved in Creating and Running a Program:

ASSEMBLER:

It converts the assembly language to machine language (Object Code) May contain unresolved references (i.e. file contains some or all of complete program)

LINKER:

A program that combines object files to create a single "executable" file. Major functional difference is that all references are resolved. (i.e. Program contains all parts needed to run) A program that loads executable files into memory, and may initialize some registers (e.g. IP) and starts it going.

DEBUGGER:

A program that loads but controls the execution of the program. To start/stop execution, to view and modify state variables.

STEPS IN CREATING & RUNNING CODE:
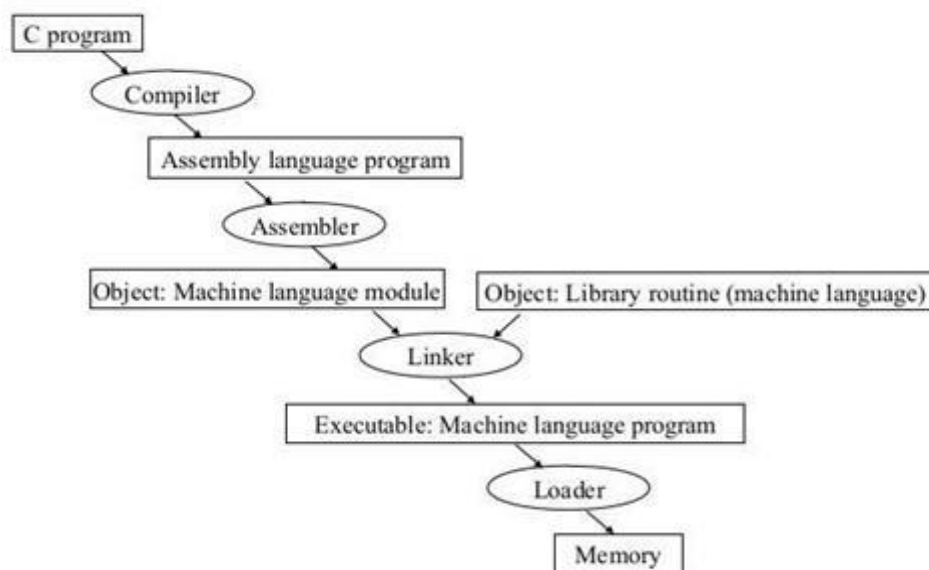
SECTION 1: DEBUGGING OUR PROGRAM

We have seen how to configure Visual Studio 2019 for Assembly Language and tested it with a sample program. The output of our sample program was displayed using a console window but it is usually more desirable to watch the step by step execution of our program with each line of code using breakpoints.

Let us briefly define the keywords relevant to debugging in Visual Studio and then we will cover an example for understanding.

DEBUGGER

The (Visual Studio) debugger helps us observe the run-time behavior of our program and find problems. With the debugger, we can break execution of our program to examine our code, examine and edit variables, view registers, see the instructions created from our source code, and view the memory space used by our application.

BREAKPOINT

A breakpoint is a signal that tells the debugger to temporarily suspend execution of your program at a certain point. When execution is suspended at a breakpoint, your program is said to be in break mode.

CODE STEPPING

One of the most common debugging procedures is stepping: executing code one line at a time. The Debug menu provides three commands for stepping through code:

- Step Into (By pressing F11)
- Step Over (By pressing F10)
- Step Out (Shift+F11)

SINGLE STEPPING

To see the values of internal registers and memory variables during execution, let us use an example. Copy the following code onto your Test.asm file.k

1. Right-click on line 6 to insert a breakpoint.
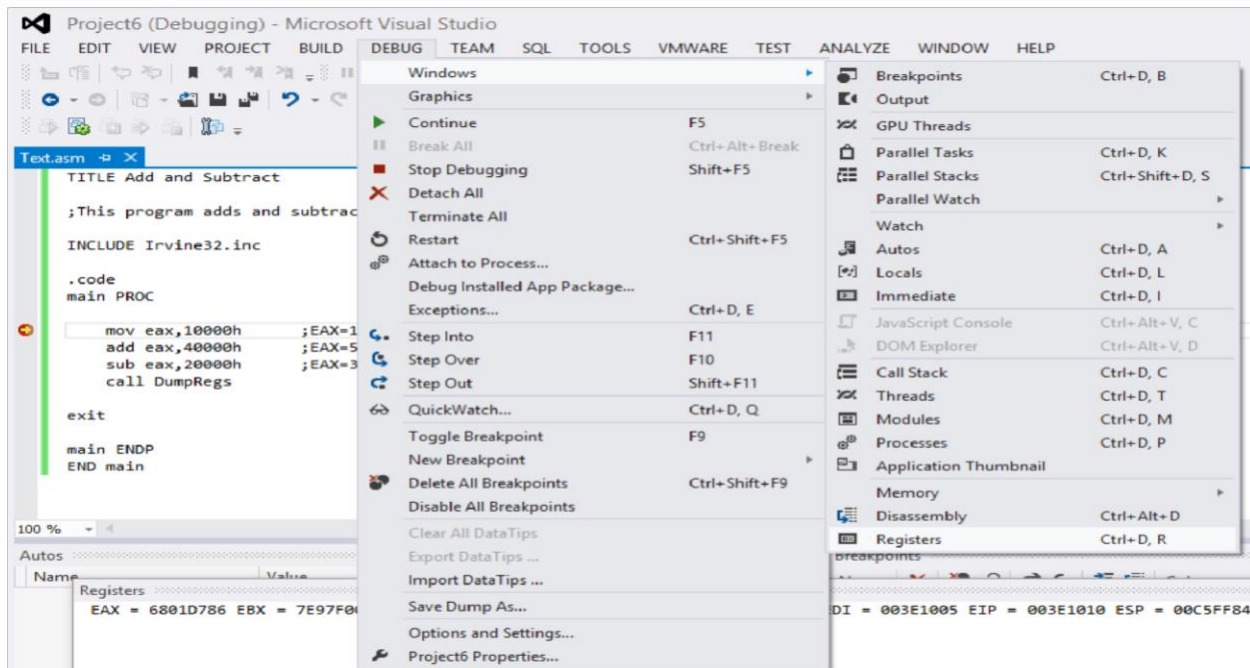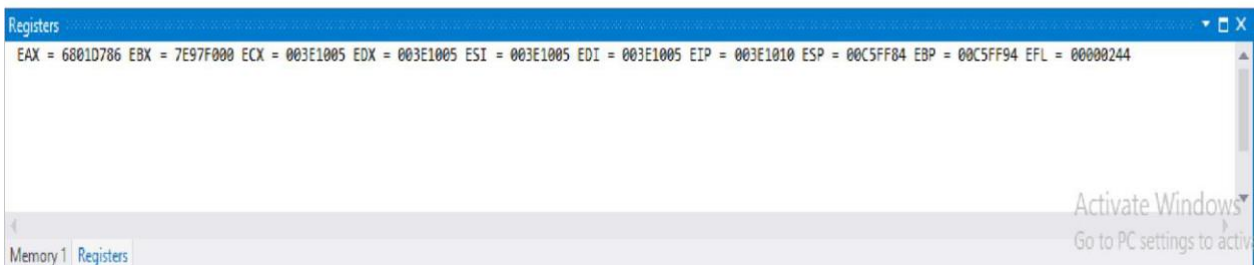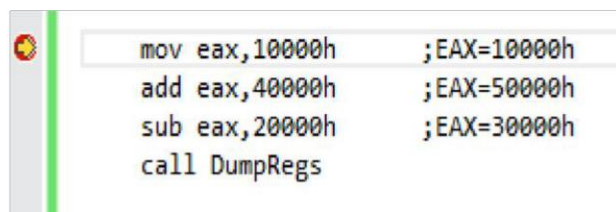
2. Click on **Debug** tab from the toolbar, select **Start Debugging** OR press **F10** to start stepping over the code.
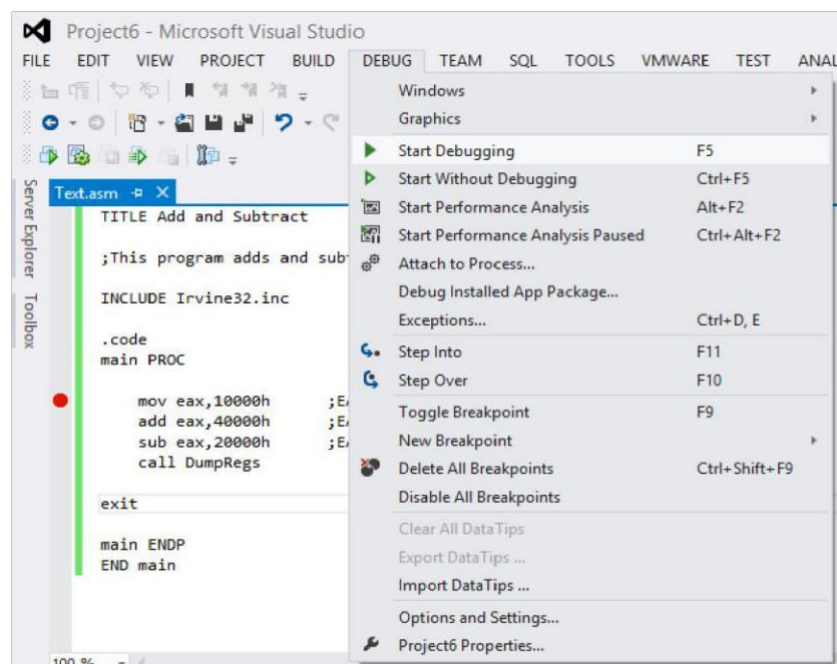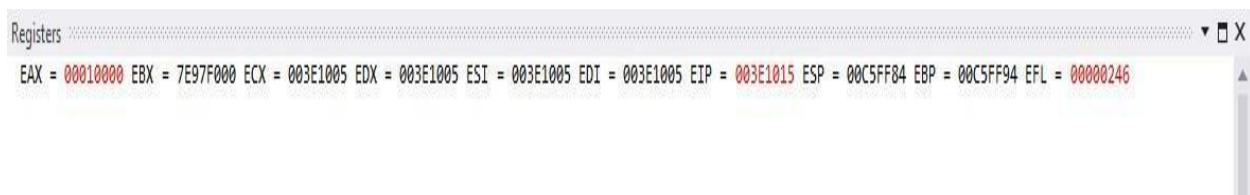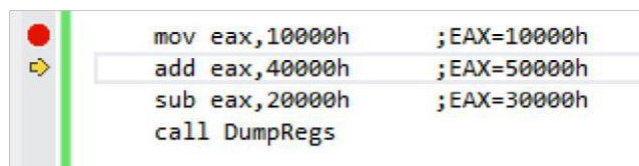

3. Click on **Debug** tab than select Windows after that open menu and select **Registers** option.

4. Breakpoint set on 1st instruction

Press **F10** again to execute next line.
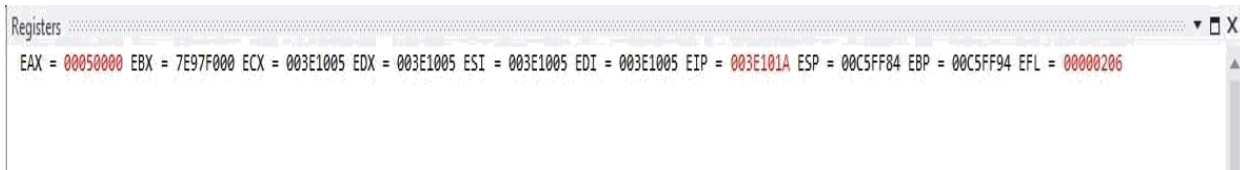

Again press **F10** key for next instruction execution.

```
  ●      mov eax,10000h        ;EAX=10000h
         add eax,40000h        ;EAX=50000h
  ⇨      sub eax,20000h        ;EAX=30000h
         call DumpRegs
```

Registers

EAX = 00050000 EBX = 7E97F000 ECX = 003E1005 EDX = 003E1005 ESI = 003E1005 EDI = 003E1005 EIP = 003E101A ESP = 00C5FF84 EBP = 00C5FF94 EFL = 00000206

Press **F10** again, the program will not terminate after executing the current instruction and as soon as it reaches the line with a call to **DumpRegs**

Registers

EAX = 00030000 EBX = 7E97F000 ECX = 003E1005 EDX = 003E1005 ESI = 003E1005 EDI = 003E1005 EIP = 003E101F ESP = 00C5FF84 EBP = 00C5FF94 EFL = 00000206
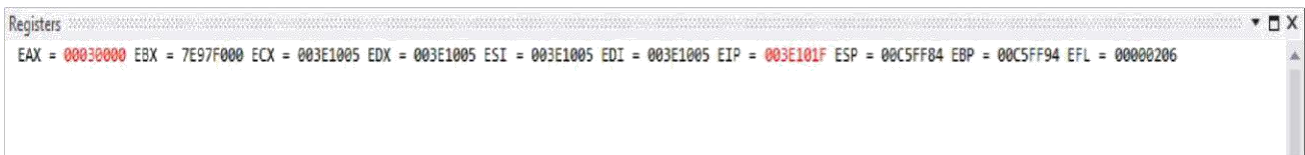
SECTION 3: INTRODUCTION TO REGISTERS

To speed up the processor operations, the processor includes some internal memory storage locations, called **Registers**. The registers store data elements for processing without having to access the memory.
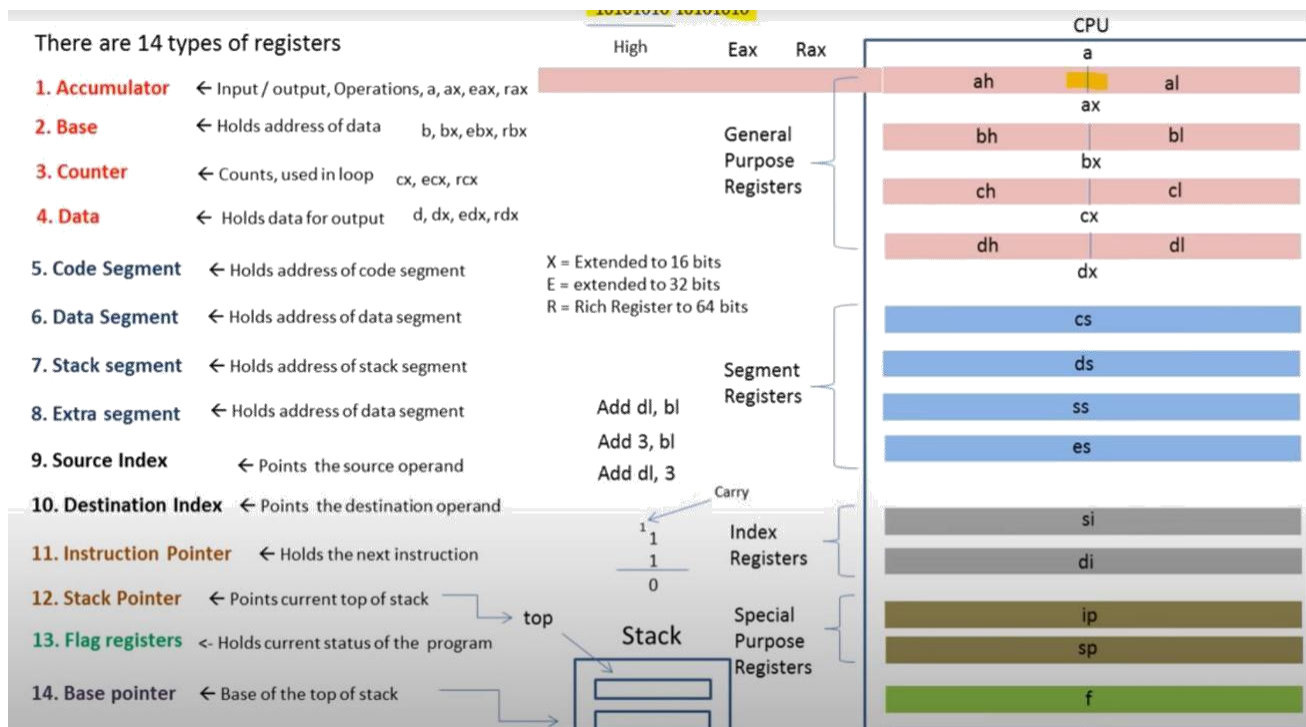
PROCESSOR REGISTERS

There are ten 32-bit and six 16-bit processor registers in IA-32 architecture. The registers are grouped into three categories:

- General registers.
- Control registers.
- Segment registers.

Furthermore, the general registers are further divided into the following groups:

- Data registers.
- Pointer registers.
- Index registers.

DATA REGISTERS

Four 32-bit data registers are used for arithmetic, logical, and other operations. These 32-bit registers can be used in three ways:

- As complete 32-bit data registers: EAX, EBX, ECX, EDX.
- Lower halves of the 32-bit registers can be used as four 16-bit data registers: AX, BX, CX and DX.
- Lower and higher halves of the above-mentioned four 16-bit registers can be used as eight 8-bit data registers: AH, AL, BH, BL, CH, CL, DH, and DL
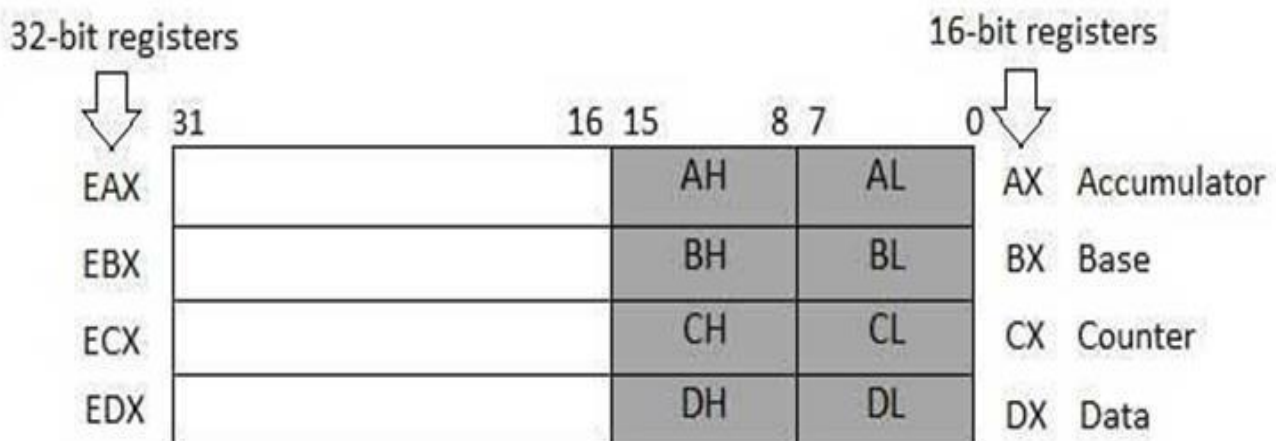
AX (Accumulator): It is used in input/output and most arithmetic instructions. For example, in multiplication operation, one operand is stored in EAX or AX or AL register according to the size of the operand.

BX (Base register): It could be used in indexed addressing.

CX (Counter register): The ECX, CX registers store the loop count in iterative operations.

DX (Data register): It is also used in input/output operations. It is also used with AX register along with DX for multiply and division operations involving large values.
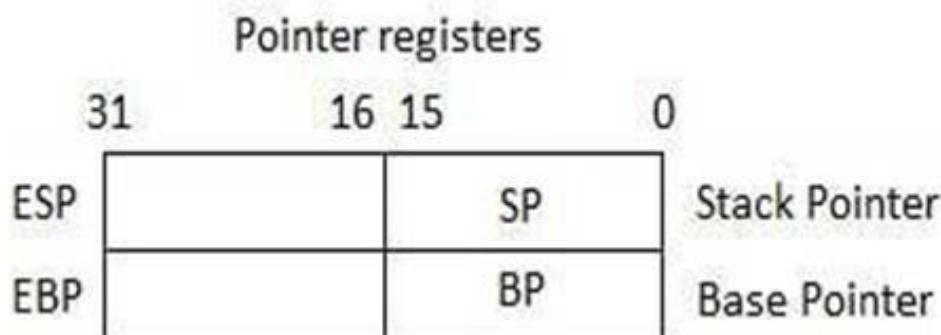
POINTER REGISTERS

The pointer registers are 32-bit EIP, ESP, and EBP registers and corresponding 16-bit right portions IP, SP, and BP. There are three categories of pointer registers:

Instruction Pointer (IP): The 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.

Stack Pointer (SP): The 16-bit SP register provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.

Base Pointer (BP): The 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.
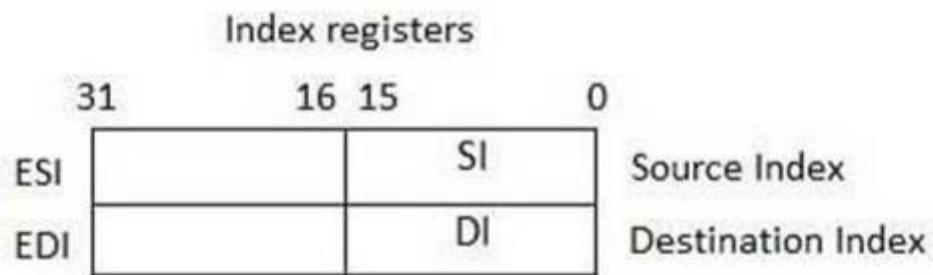

Index Registers

The 32-bit index registers, ESI and EDI, and their 16-bit rightmost portions. SI and DI, are used for indexed addressing and sometimes used in addition and subtraction. There are two sets of index pointers.

Source Index (SI): It is used as source index for string operations.

Destination Index (DI): It is used as destination index for string operations.

BASIC ELEMENT OF ASSEMBLY LANGUAGE

INTEGER CONSTANTS

Integer constants are made up of an optional leading sign, one or more digits and an optional suffix character.

Format:

[{+ | -}] digits radix

Examples:

26	for decimal
26d	for decimal
10111110b	for binary
42o	for octal
1Ah	for Hexadecimal
0A3h	for Hexadecimal

CHARACTER CONSTANTS

Character constants are made up of a single character enclosed in either single or double quotes.

Example:

'A' "d"

STRING CONSTANTS

A string of characters enclosed in either single or double quotes.

Example:

"Hello World"

IDENTIFIERS

An identifier is a programmer-defined name of a variable, procedure or code label.

Format:

They may contain between 1 and 247 characters. They are not case sensitive. The first character must be a letter (A..Z, a..z), underscore (_), @ , ?, or $. Subsequent characters may also be digits.

An identifier cannot be the same as an assembler reserved word. For example: reserved words are instruction mnemonics, directives, attributes, operators, predefined symbols.

Examples:

myVar
_abc
hello2

MEMORY SEGMENTS

A segmented memory model divides the system memory into groups of independent segments referenced by pointers located in the segment registers. Each segment defines the area of our program that contains data variables, code and stack, respectively.

Data segment: It is the memory region, where data elements are stored for the program. This section cannot be expanded after the data elements are declared, and it remains static throughout the program.

Code segment: This section defines an area in memory that stores the instruction codes. This is also a fixed area.

Stack segment: This segment contains data values passed to procedures within the program.

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. They can assign names to memory segments. In MASM, directives are case insensitive. For example, it recognizes .data, .DATA and .Data as equivalent.

Let us see what different directives we can use to define segments of our program:

The **.DATA** directive identifies the area of a program containing variables:

Syntax:

.data

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

Syntax:

.code

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

Syntax:

.stack 100h

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)

The basic syntax of an Assembly Language instruction is as:

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

where elements in square brackets are optional. We will now see what each of these elements.

Label: A *label* is an identifier that acts as a place marker for instructions and data. A label placed just before an instruction implies the instruction's address. Similarly, a label placed just before a variable implies the variable's address.

Mnemonics: An instruction mnemonic is a short word that identifies an instruction to perform an operation. Following are examples of instruction mnemonics:

mov:	Moves (assigns) one value to another.
add:	Adds two values
sub:	Subtracts one value from another
mul:	Multiplies two values
jmp:	Jumps to a new location
call:	Calls a procedure

Operands: Assembly language instructions can have between zero and three operands, each of which can be a register, memory operand, constant expression, or input-output port.

Example	Operand Type
96	Constant (*immediate value*)
2 + 4	Constant expression
eax	Register
count	Memory

Example:

The MOV instruction has two operands:

mov count , ebx   ; move EBX to count

In a two-operand instruction, the first operand is called the destination. The second operand is the source. In general, the contents of the destination operand are modified by the instruction.

Comments: Comments are an important way for the writer of a program to communicate information about the program's design to a person reading the source code.

Comments can be specified in two ways:

• **Single-line comments**: beginning with a semicolon character (;). All characters following the semicolon on the same line are ignored by the assembler.

• **Block comments**: beginning with the COMMENT directive and a user-specified symbol. For example,

COMMENT !

This line is a comment.

This line is also a comment.

!

DATA TYPES

MASM defines **intrinsic data types** , each of which describes a set of values that can be assigned to variables and expressions of the given type.

BYTE	8-bit unsigned integer
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer
SWORD	16-bit signed integer

DWORD   32-bit unsigned. D stands for double
QWORD   64-bit integer. Q stands for quad
TBYTE   80-bit integer. T stands for ten

SECTION 4: EXERCISE

Implement all of these equations in assembly language.

- 47 + 39 + 60 + 85 + 64+54o-0Ah

- 30- 9 + 186 – 150

- 101110 + 50Ah + 6710d + 1010001 + F

- 10001101 – D83h + 385+10 + 1111101 – E+F

- edx = -eax + 1 + ebx + edx – ecx + 0Ah-65o+73d

- eax = 5ADh – eax + 65o + 65d – 11110111 + 150