**Lab 02 : Assembly Language Fundamentals**

**Objectives:**

* **Fundamentals of Assembly language**
* **Defining Data**
* **Intrinsic Data Types**
* **Data Definition Statement**
* **Data Initializations**
* **Multiple Initializations**
* **String Initialization**

**Steps Involved in Creating and Running a Program:**

**Assembler:**

It converts the assembly language to machine language (Object Code)

May contain unresolved references (ie. file contains some or all of complete program)

**Linker:**

A program that combines object files to create an single “executable” file.

Major functional difference is that all references are resolved. (ie. Program contains all parts

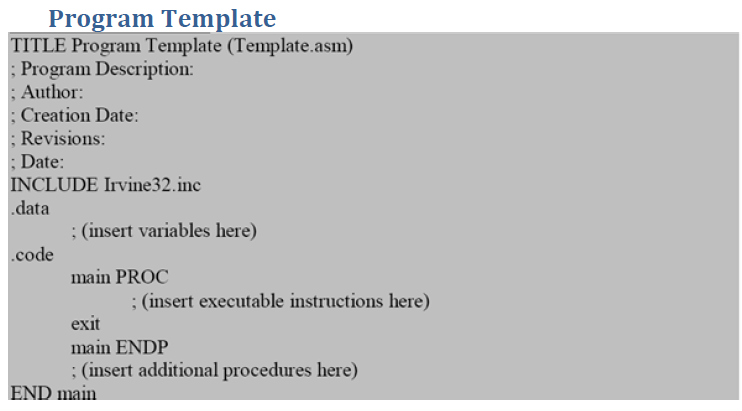
needed to run) A program that loads executable files into memory, and may initialize some registers (e.g. IP ) and starts it going.

**Debugger:**

A program that loads but controls the execution of the program · To start/stop execution, to

view and modify state variables.

**Basic Elements:**



**Integar Constants:**

**I**t is made up of optional leading sign, one or more digits and an optional suffix character.

[ {+ | -} ] *digits radix*

Examples:

26 for decimal

26d for decimal

10111110b for binary

42o for octal

1Ah for Hexadecimal

0A3h for Hexadecimal

**Real Number Constants :**

*[sign] integar . [integar] [exponent]*

where exponent is E [{+|-}] integar

**Character Constants**

It is a single character enclosed in either single or double quotes.

Example: ‘A’ or “d”

**String Constants :**

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

**Identifiers:**

It is a programmer chosen name for a variable, constant,procedure or code label.

**DATA TYPES:**

➢ BYTE 8-bit unsigned integer. B stands for byte

➢ SBYTE 8-bit signed integer. S stands for signed

➢ WORD 16-bit unsigned integer (can also be a Near pointer in real-address mode)

➢ SWORD 16-bit signed integer

➢ DWORD 32-bit unsigned integer (can also be a Near pointer in protected mode).

D stands for double

➢ QWORD 64-bit integer. Q stands for quad

**Examples:**

➢ value1 BYTE 'A' ; character constant

➢ value2 BYTE 0 ; smallest unsigned byte

➢ value3 BYTE 255 ; largest unsigned byte

➢ value4 SBYTE −128 ; smallest signed byte

➢ value5 SBYTE +127 ; largest signed byte

➢ greeting1 BYTE "Good afternoon”

➢ greeting2 BYTE 'Good night'

➢ list BYTE 10,20,30,40 ; Multiple initializers

A question mark (?) initializer leaves the variable uninitialized, implying it will be assigned a

value at runtime:

value6 BYTE ?

**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:

**1.** Label (optional)

**2.** Instruction mnemonic (required)

**3.** Operand(s) (usually required)

**4.** 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.

**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.

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

* **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

**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, and
* Segment registers.

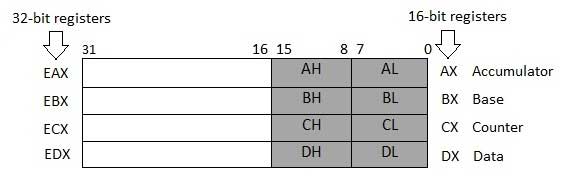
Furthermore, the general registers are further divided into the following groups:

* 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.

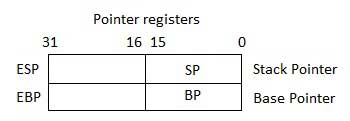
**Pointer Registers:**

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.

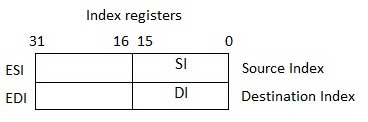


## Index Registers

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.



**Declaring Variables**

The define assembler directive is used for allocation of storage space. It can be used to reserve as well s initialize one or more bytes.  
The syntax for storage allocation statement for initialized data is

**[variable-name] define-directive initial-value [,initial-value]**

Where, *variable-name* is the identifier for each storage space. The assembler associates an offset value for each variable name defined in the data segment.  
Following are some examples of using define directives:

**choice DB ‘Y’  
number DW 12345  
neg\_number DW -12345  
big\_number DQ 123456789  
real\_number1 DD 1.234  
real\_number2 DQ 123.456**

The following points are to be considered:

* Each byte of character is stored as its ASCII value in hexadecimal.
* Each decimal value is automatically converted to its 16-bit binary equivalent and stored as a hexadecimal number.
* Negative numbers are converted to its 2's complement representation.
* Short and long floating-point numbers are represented using 32 or 64 bits, respectively.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*