2.1 INTRODUCTION TO ASSEMBLY LANGUAGE

Assembly language unlocks the secret of your computer's hardware and software. It teaches you about the way the computer's hardware and operating system work together and how, the application programs communicate with the operating system. Assembly language, unlike high level languages, is machine dependent. Each microprocessor has its own set of instructions, that it can support. Here we will discuss, only the IBM-PC assembly language. It consists of the Intel 8086/8088 instruction set. The instructions for the Intel 8088 may be used without modification on all its enhancements - 80186,80286,80386,80486 and Pentium.

2.2 Why learn Assembly Language?

You must learn assembly language for various reasons:

1. It helps you understand the computer architecture and operating system.
2. Certain programs, requiring close interaction with computer hardware, are sometimes difficult or impossible to do in high level languages. Example: a telecommunication program for the IBM-PC.
3. High level languages, out of necessity, impose rules about what is allowed in a program. For example, Pascal does not allow, a character value to be assigned to an integer variable. Assembly language, in contrast, has very few restrictions or rules; nearly every thing is left to the discretion of the programmer. The price for such freedom is the need to handle many details that would otherwise be taken care by the programming language itself.
4. One of the most important advantages of assembly language, is that the programs written in assembly language are at least 30% dense than the same program written in high level language. The reason for this is, that as of today the compilers are still not so intelligent to take advantage of some of the complex instructions of the assembly language. Example: if you write a high level program to compare two strings, it will translate the code, using simple instruction like MOV, CMP, JMP etc. While the same thing can be written in assembly, by using REPE and CMPSB. Obviously the code is much smaller.

We can summaries the above reasons by:

- ✧ Accessibility to system hardware.
- ✧ Space and time efficiency.

2.3 Assembly Language Syntax and Program Structure

2.3.1 Introduction

A processor can directly execute a machine language program. Though it is possible to program directly in machine language, assembly language uses mnemonics to make programming easier. An assembly language program uses mnemonics to represent symbolic instructions and the raw data that represent variables and constants.

A machine language program consists of: a list of numbers representing the bytes of machine instructions to be executed and data constants to be used by the program.

2.3.2 Assembly Language Syntax

An assembly language program consists of statements. The syntax of an assembly language program statement obeys the following rules:
- Only one statement is written per line.
- Each statement is either an instruction or an assembler directive.
- Each instruction has an opcode and possibly one or more operands.
- An opcode is known as a mnemonic.
- Each mnemonic represents a single machine instruction.
- Operands provide the data to work with.

2.3.2.1 Assembler Directives

Pseudo instructions or assembler directives are instructions that are directed to the assembler. Assembler directives affect the generated machine code, but are not translated directly into machine code. Directives can be used to declare variables, constants, segments, macros, and procedures as well as supporting conditional assembly.
In general, a directive contains pseudo-operation code, tells the assembler to do a specific thing, and is not translated into machine code.

- ***Segment directives***

Segments are declared using directives. The following directives are used to specify the following segments:

.stack
.data
.code

Stack Segment
- Used to set aside storage for the stack.
- Stack addresses are computed as offsets into this segment.
- Use: .stack followed by a value that indicates the size of the stack.

Data Segment
- Used to set aside storage for variables.
- Constants are defined within this segment in the program source.
- Variable addresses are computed as offsets from the start of this segment.
- Use: .data followed by declarations of variables or definitions of constants.

Code Segment

The code segment contains executable instructions macros and calls to procedures.
Use: .code followed by a sequence of program statements.

- ***Memory Models***

The memory model specifies the memory size assigned to each of the different parts or segments of a program. There exist different memory models for the 8086 processor.

The .*MODEL* Directive

The memory model directive specifies the size of the memory the program needs. Based on this directive, the assembler assigns the required amount of memory to data and code. Each one of the segments (stack, data and code), in a program, is called a ***logical segment***.

Depending on the model used, segments may be in one or in different physical segments. This directive is placed at the very beginning of the program.

The general structure for this directive is:

.MODEL memory_model

Where memory_model can be:

- TINY
- SMALL
- COMPACT
- MEDIUM
- LARGE
- HUGE

SMALL Model

In the SMALL model all code is placed in one physical segment and all data in another physical segment.

In this model, all procedures and variables are addressed as NEAR by pointing to their offsets only.

2.3.2.2 Instructions

Definition:

An instruction in assembly language is a symbolic representation of a single machine instruction. In its simplest form, an instruction consists of a mnemonic and a list of operands.

A mnemonic is a short alphabetic code that assists the CPU in remembering an instruction. This mnemonic can be followed by a list of operands. Each instruction in assembly language is coded into one or more bytes. The first byte is generally an OpCode, i.e. a numeric code representing a particular instruction. Additional bytes may affect the action of the instruction or provide information about the data needed by the instruction.

Instruction Semantics:

The following rules have to be strictly followed in order to write correct code.

1 - Both operands have to be of the same size:

Instruction	**Correct**	**Reason**
MOV AX, BL	No	Operands of different sizes
MOV AL, BL	Yes	Operands of same sizes
MOV AH, BL	Yes	Operands of same sizes
MOV BL, CX	No	Operands of different sizes

2 - Both operands cannot be memory operands simultaneously:

Instruction	Correct	Reason
MOV i , j	No	Both operands are memory variables
MOV AL, i	Yes	Move memory variable to register
MOV j, CL	Yes	Move register to memory variable

3 - First operand, or destination, cannot be an immediate value:

Instruction	Correct	Reason
ADD 2, AX	No	Move register to constant
ADD AX, 2	yes	Move constant to register

2.4 Writing a Program

How to write an assembly language program?

These are the steps that should be followed for writing an assembly language program:

1- Define the problem.
2- Write the algorithm.
3- Translate into assembly mnemonics.
4- Test and Debug the program in case of errors.

The translation phase consists of the following steps:
- Define type of data the program will deal with.
- Write appropriate instructions to implement the algorithm.

2.5 ASSEMBLY LANGUAGE PROGRAM DEVELOPMENT TOOLS

Now that you have some idea, about how to go about writing assembly language programs, you might want to write your own programs, and try them out on the machine. To do that, there are some developmental tools required. Let us study them now. The discussion is from the point of view of the end user, and not the system programmer.

2.5.1 Editor

An editor is a program which, when run on a system, lets you type in text, and store in a file. This text could also be your assembly language program. There are a number of editors available on PC. The editor helps you type the program in required format. This form of the program is called as the source program. The editor gives you all the flexibility, to insert lines, delete lines, insert words, characters, delete words, characters etc. In short all the features that you can think of while writing text, and more. After the program is typed, it can be stored in some secondary storage, like hard disk, floppy diskette etc, for permanent storage.

2.5.2 Assembler

An assembler program is used to translate assembly language mnemonics to the binary code for each instruction. After the complete program has been written, with the help of an editor, it is then assembled with the help of an assembler.

An assembler works in two phases, i.e., it reads your source code two times. In the first pass, the assembler, collects all the symbols defined in the program, along with their offsets, in symbol table. On a second pass through the source program, it produces a binary code for each instruction of the program, and give all the symbols an offset with respect to the segment, from the symbol table.

The assembler generates two files: the object file and the list file. The object file contains the binary code for each instruction in the program. It is created only when your program has been successfully assembled, with no errors. The errors that are detected by the assembler, are called the syntax errors. These are like:

```
MOVE AX,BX ; undeclared identifier MOVE.
MOV  AX,BL ; illegal operands
```

These are just two of the syntax errors that you can get when your program contains such kind of mistakes. (Exact description of the errors defer from assembler to assembler). In the first statement, it reads the word MOVE, it tries to match with its mnemonics set, as there is no mnemonic with this spelling, it assumes it to be an identifier, and looks for its entry in the symbol table. It does not even find it there, therefore, gives an error 'undeclared identifier'. In the second error, the two operands are of different kind. 8086 expects, both the identifier to be of the same kind, byte or word. But in the above case, one is a byte variable, while the other is a word variable. An assembler does not detect logical errors in your programs, that is your responsibility. List file is optional, and contains, the source code, the binary equivalent of each instruction, and the offsets of the symbols in the program. This file is for documentation purposes. Some of the assemblers available on PC are, MASM (Microsoft Assembler), TASM (TURBO) etc.

2.5.3 Linker

For modularity of your program, it is better to break your programs, into several subroutines. It is even better, to put the common routine, like reading a hexadecimal number, writing a hexadecimal number etc., which could he used by a lot of your other programs also, into a separate file. These files are assembled separately. After each ,has been successfully assembled, they can be linked together to form a large file, which constitutes your complete program. The file containing the common routines, can be linked to your other programs also. The program that links your programs is called the linker. The linker produces a link file which contains the binary codes for all compound modules. The linker also produces a link map which contains the address information about the linked files. The linker, however, does not assign absolute addresses to your program. It only assigns continuous relative addresses to all the modules linked, starting from zero. This form of program is said to be relocatable, because it can be put anywhere in memory to be run. This form of code can be even be carried to other machines, of the same kind, or compatible to the present machine, to be run successfully. The linker available on your PC is LINK ,TURBO has a built in linker.

2.5.4 Loader

Loader is a program, which assigns absolute addresses to the program. These addresses are generated, by adding to all the offsets, the address from where the program is loaded into the memory. Loader comes into action, when you execute your program. This program is brought from the secondary memory, like disk, or floppy diskette, into the main memory at a specific address. Let us assume the program was loaded at address 1000h, then 1000h is added to all the offsets to get the absolute address. Once the program has been loaded, it is now ready to run.

2.5.5 Debugger

If your program requires no external hardware or requires hardware directly accessible from your system, then you can use a debugger to debug your program. Debugger allows you to load your program into just like a loader, and, troubleshoot your program. While debugging, you can run your program in single step, set breakpoints, view the contents of registers or memory locations. You can even change the contents of the register or memory location, and run your program with new value. This helps you to isolate the problems in your programs. The problems can be corrected with the help of an editor, and the whole procedure of assembling, linking and executing your program can be repeated. Debugger helps you detect the logical errors, that could not be detected by the assembler.

The following steps showed the process of translating an assembly program into executable file:

1. The *assembler* produces an object file from the assembly language source.
2. The object file contains machine language code with some external and relocatable addresses that will be resolved by the linker. Their values are undetermined at that stage.
3. The *linker* extract object modules (compiled procedures) from a library and links them with the object file to produce the executable file.
4. The addresses in the executable file are all resolved but they are still virtual addresses.