Assembly Language Programming

- Basic programming language available for any processor
- Programs written in English like words by the programmer to represent the binary instructions of a machine
- A symbolic code is given to each instruction called mnemonics
- Unlike high level language, assembly level language lack variables and functions, but have same structure and commands, much like machine language
- Helpful to programmer when speed is required

Assembly Language Syntax

- The main features of ALP are program comments, reserved words, identifiers, statements and directives which provide the basic rules and framework for the language.
- **Program comments:**
  - ➢ The use of comments throughout a program can improve its clarity.
  - ➢ It starts with semicolon (;)
  - ➢ E.g. ADD AX, BX ; Adds AX & BX
- **Reserved words:**
  - ➢ Certain names in assembly language are reserved for their own purpose to be used only under special conditions.
  - ➢ Instructions : Such as MOV and ADD (operations to execute)

Assembly Language Syntax

- **Identifiers:**
  - ➢An identifier (or symbol) is a name that applies to an item in the program that expects to reference
  - ➢ Two types of identifiers are Name and Label
  - ➢ Name refers to the address of a data item such as NUM1 DB 5, COUNT DB 0
  - ➢ Label refers to the address of an instruction
  - ➢ E. g: MAIN PROC FAR

    L1: ADD BL, 73
- **Statements:**
  - ➢ALP consists of a set of statements with two types
  - ➢Instructions, e. g. MOV, ADD
  - ➢ Directives, e. g. define a data item

	Identifiers	operation	operand	comment
Directive:	COUNT	DB	1	; initialize count
Instruction:	L30:	MOV	AX, 0	; assign AX with 0

Assembly Language Syntax

- **Directives:**
  - The directives are the number of statements that enables us to control the way in which the source program assembles and lists
  - These statements called directives act only during the assembly of program and generate no machine-executable code

  - The different types of directives are:
    - **The page and title directives**:
      - The page and title directives help to control the format of a listing of an assembled program. This is their only purpose and they have no effect on subsequent execution of the program.
      - The page directive defines the maximum number of lines to list as a page and the maximum number of characters as a line.
      - PAGE [Length] [Width]
      - Default: Page [50][80]
      - TITLE gives title and place the title on second line of each page of the program
      - TITLE text [comment]

Assembly Language Syntax

- **Directives:**

➢**SEGMENT directive**

It gives the start of a segment for stack, data and code

Seg-name Segment

Seg-name ENDS

➢Segment name must be present, must be unique and must follow assembly language naming conventions.

➢ An ENDS statement indicates the end of the segment

Assembly Language Syntax

- **Directives:**

➢ **Procedure Directives:**

The code segment contains the executable code for a program, which consists of one or more procedures, defined initially with the PROC directives and ended with the ENDP directive

```
Procedure _name PROC [FAR/NEAR]
…………….
…………….
 …………….
Procedure_name ENDP
```

FAR is used for the first executing procedure and rest procedures call will be NEAR

Procedure should be within segment.

Assembly Language Syntax

- **Directives:**

➢ **END Directive**:

- An END directive ends the entire program and appears as the last statement

  END Name

➢ **ASSUME Directive**:

- An .EXE program uses the SS register to address the stack, DS to address the data segment and CS to address the code segment
- Used in conventional full segment directives only
- Assume directive is used to tell the assembler the purpose of each segment in the program
- Assume SS: Stack Seg_name, DS: Data Seg_name CS: code Seg_name

Assembly Language Syntax

- **Directives:**

➢ **END Directive**:

- An END directive ends the entire program and appears as the last statement
- ENDS directive ends a segment and ENDP directive ends a procedure

    END PROC-Name

➢ **ASSUME Directive**:

- An .EXE program uses the SS register to address the stack, DS to address the data segment and CS to address the code segment
- Used in conventional full segment directives only
- Assume directive is used to tell the assembler the purpose of each segment in the program
- Assume SS: Stack name, DS: Data Seg_name CS: code seg_name

Assembly Language Syntax

- **Directives:**

➢**Dn Directive (Defining data types):**

Assembly language has directives to define data. The Dn directive can be any one of the following:

DB	Define byte	1 byte
DW	Define word	2 bytes
DD	Define double	4 bytes
DQ	Define quadword	8 bytes
DT	Define 10 bytes	10 bytes

```
VAL1 DB 25
 ARR DB 21, 23, 27, 53
VAL2 DW 2255
```

Assembly Language Syntax

- **Directives:**

➢ **EQU directive**

- It can be used to assign a name to constants

  E.g. FACTOR EQU 12 #DEFINE PI 3.1416

  MOV BX, FACTOR ; MOV BX, 12

  MOVE EQU MOV; MOVE A,B

- It is short form of equivalent

➢ **DUP Directive**

- It can be used to initialize several locations to zero

  e. g. SUM DW 4 DUP(?) ; SUM[0], SUM[1], SUM[2], SUM[3]

- Reserves four words starting at the offset sum in DS and initializes them to Zero.
- Also used to reserve several locations that need not be initialized. In this case (?) is used with DUP directives.

  E. g. PRICE DB 100 DUP(?) - Reserves 100 bytes of uninitialized data space to an offset PRICE

8086 Program Structure

- **Using Simplified Directives**

```
TITLE program to add two values given in two registers.
DOSSEG
.MODEL small
.STACK 100h
.DATA                                   ; here data segment contains nothing
   sum db ?
.CODE
mov ax, @data
mov ds, ax              ; initializes the data segment
mov cx, 400h            ; CX=400H
mov bx, 100h            ; BX=100H
add cx, bx              ; adds the values of cx and bx and places the result in cx.
mov sum, cx
mov ax, 4c00h
int 21h                 ; returns control to DOS
END
```

8086 Program Structure

- **Using Simplified Directives**
  - **Description**

➢**TITLE:** TITLE is used to print title in the program

➢ **DOSSEG:** It stands for DOS SEGMENT. The DOSSEG defines the standard names for the different memory segments as defined by the DOS. The standard memory segments that are defined by DOS are .data, .code, .stack, and .extra for data, code, stack, and extra segments respectively.

➢**.MODEL**: It is a directive that is used to indicate the memory size that data and code segment can occupy in a program.

MEMORY_MODEL	**Meaning**
Tiny	Code and Data Segment Combined should be less than 64 KB
Small	Code and Data Segment should not exceed 64 KB
Medium	Code Segment can be greater than 64 KB and Data Segment should be less than 64KB
Large	Both Code and Data segment can be greater than 64 KB

8086 Program Structure

- **Using Simplified Directives**
  - **Description**

➢ **.STACK:** It is a physical segment that defines its maximum size which it can occupy in main memory.

➢ **.DATA:** It is a physical data segment which contains the different data and variables required for the program. The data and variables defined in this section are used in code segment

➢ **.CODE:** It is a physical code segment which contains the actual code that we write to solve a specific problem.

8086 Program Structure

- **Using Conventional Directives**

```
TITLE program to add two registers
data SEGMENT                                    ; data segment begins here
   sum db ?
data ENDS                       ; data segment ends here
code SEGMENT                    ; code segment begins here
   ASSUME ds:data, cs:code  ; assumes data as data segment and code as code segment
   mov ax, SEG data
   mov ds, ax                   ; initializes the data segment
   mov ax, 400h
   mov bx, 100h
   add ax, bx                   ; adds the values of ax and bx and places the result in ax.
   mov sum, ax
   mov ax, 4c00h
   int 21h                              ; returns control to DOS
code ENDS                               ; code segment ends here
END
```

Assembling, Linking and Executing

1) Assembling:

- Assembling converts source program into object program if syntactically correct and generates an intermediate **.obj** file or module.
- It calculates the offset address for every data item in data segment and every instruction in code segment.
- A header is created which contains the incomplete address in front of the generated **obj** module during the assembling.
- Assembler complains about the syntax error if any and does not generate the object module.
- Assembler creates **.obj .lst** and **.crf** files and last two are optional files that can be created at run time.

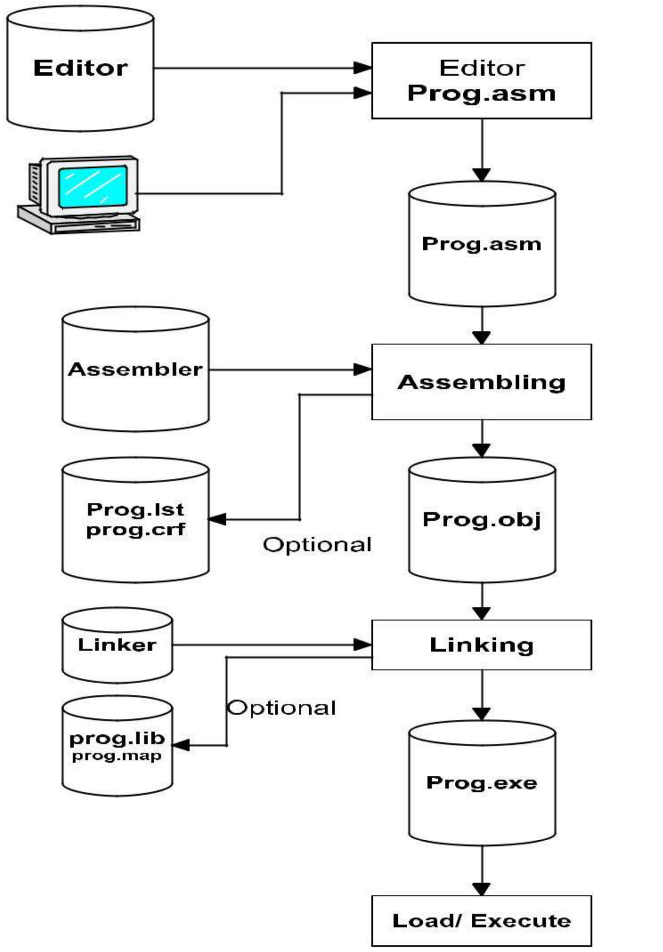

Fig: Steps in assembling, linking & Executing

Assembler Types:

a) One pass assembler:

- This assembler scans the assembly language program once and converts to object code at the same time.
- Works fine with backward referencing
- Problem with forward referencing

```
Backward referencing Forward Referencing
L1:…………..                      JNZ L2
   ……………                       ……………
   JNZ L1                        L2:…………..
```

b) Two pass assembler

- This type of assembler scans the assembly language twice.
- First pass generates symbol table of names and labels used in the program and calculates their relative address.
- This table can be seen at the end of the list file and here user need not define anything.
- Second pass uses the table constructed in first pass and completes the object code creation.
- This assembler is more efficient and easier than earlier.

2) Linking:

- This involves the converting of **.OBJ** module into **.EXE**(executable) module i.e. executable machine code.
- It completes the address left by the assembler.
- It combines separately assembled object files.
- Linking creates **.EXE**, **.LIB**, **.MAP** files among which last two are optional files.

3) Loading and Executing:

- It Loads the program in memory for execution.
- It resolves remaining address.
- This process creates the program segment prefix (PSP) before loading.
- It executes to generate the result.

Sample program $\xrightarrow{\text{assembling}}$ object Program $\xrightarrow{\text{linking}}$ executable program

Writing .COM programs:

- It fits for memory resident programs.
- Code size limited to 64K.
- **.com** combines PSP, CS, DS in the same segment
- SP is kept at the end of the segment (FFFF), if 64k is not enough, DOS Places stack at the end of the memory.
- The advantage of **.com** program is that they are smaller than **.exe** program.
- A program written as **.com** requires ORG 100H immediately following the code segment's SEGMENT statement. The statement sets the offset address to the beginning of execution following the PSP.

```
.MODEL TINY
.CODE
ORG 100H                        ; start at end of PSP
BEGIN:JMP MAIN                  ;Jump Past data
A1 DW215
B1DW125
C1DW ?
MAIN PROC
MOV AX, A1
ADD AX, B1
MOV C1, AX
MOV AX, 4C00H
INT 21H
MAIN ENDP
END BEGIN
```

Macro Assembler:

- A macro is an instruction sequence that appears repeatedly in a program assigned with a specific name.
- The macro assembler replaces a macro name with the appropriate instruction sequence each time it encounters a macro name.
- When same instruction sequence is to be executed repeatedly, macro assemblers allow the macro name to be typed instead of all instructions provided the macro is defined.
- Macro are useful for the following purposes:

oTo simplify and reduce the amount of repetitive coding.
oTo reduce errors caused by repetitive coding.
OTo make an assembly language program more readable.
O Macro executes faster because there is no need to call and return.
O Basic format of macro definition:

```
Macro nameMACRO [Parameter list]          E.g.Addition  MACRO
………………………
………………………                                 IN AX, PORT
[Instructions]                            ADD AX, BX
………………………                                 OUT PORT, AX
………………………                                 ENDM
ENDM
```