PAGE NO. 115
DATE

Chapter-4) The art of assembly language programming

As compared to the high level programming, assembly language programming is slightly cumbersome. The programs written in assembly language are compact and efficient.

A program has to be converted to machine code for execution, so it is performed by translator called assembler.

Assembly language programming requires good knowledge of machine architecture, operating system and programming principles. Assembly language is case insensetive, so program can be loaded either in upper case lower case and upper case. The program development tools such as editor, assemblers, linker and debugger are required for the programming.

Program development steps:

1. Defining the problem
2. Algorithm
3. Flowchart
4. Initialization checklist
5. Choosing instructions
6. Converting algorithms into assembly language program.

1) Defining the problem:-

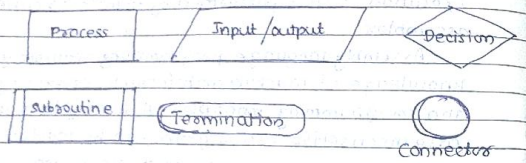
The first step in writing the program is to think very carefully about the problem that you want the program to solve. At this point, you need not to write down the programs, but you must know what you would like to do.

2. Algorithm:-

The formula or sequence or tasks need to performed by your program can be specified as a step in general english often called as algorithm.

[3] Flow chart :

The flowchart is a graphically representation of the program operation or task. The specific operation or task is represented by graphical symbol such as circle, rectangle, diagonal, square and parallelogram etc.



[4] Initialization checklist :

In the program there are many variables, constants and various parts of system such as segment registers, flags, stack, programmable ports etc. which must be initialized properly.

The best way to approach the initialization task is to make the check list of the entire variables, constants, all the registers, flags and programmable ports in the program.

At this point you will come to know which parts on the checklist will have to be initialized.

[5] Choosing instructions :

Next step is to choose proper instructions that performs your problem's operations or tasks. This is an important step, so you must know entire instruction set of microprocessor, the operation performed and flag affected after execution by the instructions.

[6] Converting algorithms into assembly language programs :

Once you have selected the instructions for the operation to be performed, then arrange these instructions in sequence as per algorithm, so the desired output must be obtained after the execution by instruction.

In assembly language program, a first step is to set up and declare the data structure that the algorithm will be working with, then write down the instruction required for initialization at the start of the code section.

Next determine the instructions required to implement the major actions in the algorithm and declare how data must be positioned for these instructions.

Assembly language program development tools :

[1] Editor
[2] Assembler
[3] Linker
[4] Debugger

[1] Editor :- An editor is a program which helps you to construct your assembly language program in right format so that the assembler will translate it correctly to machine language. So, we can type our program using editor. This form of our program is known as source program. The Dos based editor such as EDIT, Wordstar, and Norton Editor etc. can be used to type your program.

[2] Assembler :- An assembler is a program that translate assembly language program to the correct binary code from each instruction i.e machine code and generate file called as object file with extension .obj Some examples of assembler are TASM Borland's Turbo Assembler and MASM Microsoft Micro Assembler.

[3] Linker: A linker is a program, which combines, if required more than one separately assembled module into one executable program. Such as two or more programs and also generates .exe module and initializes it with special instructions to facilitate its subsequent loading and execution.
Some examples of linkers are TLINK (Borland's Turbo Linker and LINK Microsoft's linker etc.

[4] Debugger: Debugger is a program that allows the execution of program in single step mode under the control of the user. The process of locating and correcting the errors using a debugger is called debugging.
Some examples of debugger are Dos Debug command, Borland's Turbo Debugger TD, Microsoft's Debugger known as Code View CV, etc.

Program Development Process (PDP):- The PDP is a process which consists of analysis, design, development, implementation, translation, testing, debugging and maintainence of program. It is an interactive and iterative process that involves the following steps:

[1] Source file creation:- The file containing program statement is known as source file. The source file is created and edited using text editor and must have an extension .ASM

[2] Object code generation:- The language translator is used to translate source program to re-locatable object file. The assembler is used to translate assembly language source code to relocatable object code.

[3] Executable file creation:- Linker is used to create an executable file.

[4] Program running:- The executable file can be run by entering the name of executable file on the prompt and then by pressing ENTER key on keyboard.

[5] Program testing:- The result or output generated by the program has to be tested for their validity. The result must satisfy the program specification. If any errors occur during the result, then the program should be debug.

[6] Program debugging:- The errors in the program can be located using debugger. The executable file of the program to be debugged must be created with program debug options.

Assembler directives and operator:
Directives:- Assembly language program supports a number of reserve words i.e keywords that enables you to control the way which a program assembles and lists. These are called as assembly directives. They are divided into various categories:

[1] Data definition and storage allocation directives:
(a) DB: Define byte → The directive DB is used to define a byte type variable. It can be used to single or multiple byte variables. The range of values that can be stored in a byte is 0 to 255 for unsigned numbers and -128 to 127 for signed numbers

General form:
Name_of_variable DB Initialization_Value

Ex:
NUM DB ? ; Allocate 1 memory location
ARRAY DB 12, 22, 82, 42, 52 ; Allocate 5 memory location

120

(b) DW: Define word ▶ The directive DW is used to define a word type i.e 2 bytes variable. It can be used to define single or multiple word variables. The range of values that can be stored in a word is 0 to 65535 for unsigned numbers and -32768 to 32767 for signed numbers.

General form:

Name_of_variable DW Initialization_value

Ex:

NUM DW ? ; Allocate two memory locations
TABLE DW 1,2,3,5,6,4 ; Allocate 10 memory locations.

(c) DD: Define double word ▶ The directive DD is used to define a double word type i.e 4 byte type variable.

Range for unsigned: 0 to 2^{32-1}
Range for signed: -2^{32-1} to $+2^{32-1}-1$
Range for floating point numbers: 10^{-38} to 10^{38}

General form:

Name_of_variable DD Initialization_values

Ex:

NUM DD ? ; Allocate 4 memory locations
TABLE DD 1,2,3,5,9 ; Allocate 20 memory locations

(d) DQ: Define Quad Word ▶ The directive quadword DQ is used to define a quad i.e 8 byte type variable

Range for unsigned: 0 to 2^{64-1}
Range for signed: -2^{64-1} to $+2^{64-1}-1$
Range for floating point numbers: 10^{-308} to 10^{308}

General form:

Name_of_variable DQ Initialization_values

Ex:

NUM DQ ? ; Allocate 8 memory locations
LIST DQ 10 DUP(0) ; Allocate 80 memory locations

121

(e) DT: Define Ten Byte ▶ The directive DT is used to define a ten byte type variable.

Range for unsigned: 0 to 2^{80-1}
Range for signed: -2^{80-1} to $+2^{80-1}-1$
Range for floating point numbers: 10^{-4932} to 10^{4932}

General form:

Name_of_variable DT Initialization_value

Examples:

NUM DT ? ; Allocate 10 memory locations
TABLE DT 1,2,3,5,9 ; Allocate 50 memory locations

(f) STRUCT: Structure declaration ▶ The directive STRUCT is used to declare the data type which a collection of primary data types. (DB, DW, DD). The structure declaration allows the user to define a variable which has more than one data type.

General form:

STRUCTURE_NAME STRUCT
.....
; Sequence of DN, DW, DD directives for declaring ; field
.....
STRUCTURE_NAME ENDS

Structure variable declaration:- The structure declaration does not allocate memory space, but merely defines pattern. Storage space is allocated only when structure variable is defined. The general form of a structure definition is as follows:

Variable Structure_Name <initialization>

Accessing structure variable field :- The structure variable field can be accessed by using a dot operator (.) known as structure access operator.

General form :
Structure_variable . Field_Name

Example :
```
EMPLOYEE  STRUCT
EMP_NUM   DW  ?
EMP_NAME  DB  25 DUP (0)
EMP_DEPT  DB  30 DUP (0)
EMP_AGE   DB  ?
EMPLOYEE  ENDS
```

(g) RECORD ▶ The directive RECORD is used to define a bit pattern within a byte or a word. It is similar to the bit-wise access in C language. The RECORD defination helps in encoding and decoding of bit for which some meaning is assigned.

General form :
Record_Name RECORD Field_Specification1 .. Field Specification n

Ex: In Serial communication, initializations of the port parameters are coded bit-wise as follows :
```
Bit 7, 6, 5     ; Baud rate
Bit 4, 3        ; Parity
Bit 2           ; Stop bits
Bit 1, 0        ; Word length
```
Using RECORD directive, the above byte can be coded as follows :
```
SERIAL_COM RECORD BAUD:3, PARITY:2, STOP:1,
WORDLENGTH:2
```

(h) EQU : Equate to ▶ The EQU directive is used to declare the symbols to which some constant value is assigned. Such symbols are called as macro symbols, so macro assembler will replace every occurrances of the symbol in a program by its value.

General form :
Symbol_Name EQU Expression

Example :
```
NUM EQU 100
INCREMENT EQU INC
```

(i) ORG : Originate ▶ The directives ORG assigns the location counter with the value specified in directive. It helps in placing the machine codes in the specified location while translating the instructions into machine codes by assembler.

General form :
ORG [$+] Numeric_value

Example :
```
ORG 100H
ORG $
ORG $+100
```

(j) ALIGN : Alignment of memory addresses ▶ The directive ALIGN is used to force the assembler to align the next data item or instruction according to given value.

General form :
ALIGN Numeric_value

The number must be a power of 2 such as 2, 4, 8 or 16

Example:
ALIGN 4
In above statement, the assembler advances its location counter to next address that is evenly divisible by 4. If location counter is already at the required address, then it is not advanced. The assembler fills unused bytes with 0's for data and NOP for instructions.

(k) EVEN: Align as even memory location ▶ The directive EVEN is used to inform the assembler to increment the location counter to the next even memory address. If the location counter is already pointing to even memory address, it should not be incremented

General form:
EVEN

Ex:- DATA SEGMENT
NAME DB 'VIJAY$'
EVEN
AGE DB 30
DATA ENDS

(l) LABEL ▶ The directive LABEL enables you to redefine the attribute of data variable or instruction label.

General form:
Variable_name LABEL type_specifier

Examples:
TEMP LABEL BYTE
NUM LABEL WORD

(m) DUP: Duplicate memory location ▶ The DUP directive can be used to generate multiple bytes or words with known as well as uninitialized values.

Example:
TABLE DW 100 DUP (0)
STARS DB 50 DUP ('*')
ARRAY3 DB 30 DUP(?)

Program Organisation Directives:

(a) ASSUME ▶ The directive ASSUME informs the assembler the name of the logical segment that should be used for specified segment. When program is loaded, the processor segment registers should point to the respective logical segment.

General form:
ASSUME
Seg_Reg: Seg_Name, ..., Seg_Reg: Seg_Name
Where, ASSUME is assembler directive

Example:
ASSUME CS:CODE, DS:DATA, SS:STACK, ES:EXTRA

(b) SEGMENT ▶ The directive segment is used to indicate the beginning of logical segment. The directive SEGMENT follows the name of segment.

General form:
segment_name SEGMENT [WORD/PUBLIC]

The use of type specifies WORD indicates that the segment has to be located at the next available address, otherwise the segment will be located at the next available paragraph (16-byte size) which might waste upto 15 bytes of memory. The type specifier public indicates that the given segments, which have same name.

Ex:- DATA SEGMENT
...Program data definition
DATA ENDS
CODE SEGMENT
... Program code
CODE ENDS

(c) ENDS : End of the segment ▶ The directive ENDS informs the assembler the end of the segment. The directive ENDS and SEGMENT must enclosed the segment data or code of the program.

General form:

Segment_Name ENDS

Examples:

```
DATA SEGMENT
    .....
    Program data definition
    .....
DATA ENDS
CODE SEGMENT
    .....
    Program code here
    .....
CODE ENDS
```

(d) END : End of the program ▶ The directive END is used to inform assembler the end of the program

General form:

END [start_Address]

(e) • CODE : simplified code segment directive ▶ This simplified segment directive defines the code segment. All executable code must be placed in this segment.

General form:

• CODE [name]

(f) • DATA : Simplified DATA segment directive ▶ This simplified directive defines the data segment for initialized near data. All data definition and declaration must be placed in this segment.

General form:

• DATA

(g) • STACK : simplified stack segment directive ▶ This simplified segment directive defined the stack segment and default size of stack is 1024 bytes which you may override.

General form:

• STACK 100

(h) • MODEL : Memory model declaration for segments ▶ This simplified segment directive creates default segments

General form:

• MODEL memory_small

Memory models are:

* TINY : since MASM 6.0, used for .COM program
* SMALL : All data in one segment and all code in one segment
* MEDIUM : All data in one segment, but code in more than one segment
* COMPACT : Data in more than one segment, but code in one segment.
* LARGE : Both data and code in more than one segment, but no array may exceed 64 Kbytes
* HUGE : Both data and code in more than one segment, an array may exceed 64 Kbytes

Value Returning Attribute Directives :

(a) LENGTH ▶ The directive LENGTH informs the assembler about the number of the elements in a data items such as array. If the array is defined with DB, then it returns number of bytes allocated to the variable. If an array is defined with DW, then it returns the number of word allocated to the array variable.

General form:

LENGTH Variable_Name

Examples:

MOV CX, LENGTH NAME
MOV DX, LENGTH ARRAY

(b) SIZE ▶ The directive SIZE is same as LENGTH except that it returns the number of bytes allocated to the data item instead of the number of elements in it.

General form:

SIZE variable_name

Examples:

MOV AX, SIZE NAME
MOV DX, SIZE TOTAL

(c) OFFSET ▶ The directive OFFSET informs the assembler to determine the displacement of the specified variable with respect to the base of the segment. It is usually used to load an offset of a variable into the register. Using this offset value, a variable can be referenced using indexed addressing modes.

General form:

OFFSET Variable_Name

Examples:

MOV SI, OFFSET ARRAY
MOV DX, OFFSET MSG
MOV BX, OFFSET NUM

(d) SEG : Segment ▶ The directive SEG is used to determine the segment in which the specified data items is defined.

General form:

SEG Variable_Name

Example:

MOV DS, SEG MSG
MOV ES, SEG LIST

(e) TYPE ▶ The directive TYPE is used to determine the type of data item. It determines the number of byte allocated to the data TYPE. Assembler allocates one byte for DB, two byte for DW and four byte for DD type variable definition.

General form:

TYPE Variable_Name

Examples:

ADD BX, TYPE NUM
SUB DX, TYPE PI

Procedure Definition directives:

(a) PROC : Procedure ▶ The directive PROC indicates beginning of a procedure and follows with the name of the procedure. The term FAR and NEAR follows the PROC directive indicating the type of procedure. If it is not specified then by default assembler assumes NEAR procedure.

General form:

Procedure_Name PROC [NEAR/FAR]

Example:

```
ADD PROC NEAR
    ____
    ____
    Procedure code
    ____
    ____
ADD ENDP
```

(b) ENDP : End of procedure ▶ The directive ENDP informs the assembler the end of the procedure. The directive ENDP and PROC must enclose the procedure code.

General form:

Procedure_Name ENDP

Examples:

FACTORIAL ENDP
HEXTOASC ENDP

Macro definition directives ▶

(a) MACRO ▶ The directive MACRO informs the assembler the beginning of a macro. It consists of name of a macro followed by keyword MACRO and macro arguments if any.

General form:

Macro_Name MACRO [Argument1, ... ArgumentN]

Example:

```
DISP MACRO MSG
PUSH AX
PUSH DX
MOV AH,09H
LEA DX, MSG
INT 21H
POP DX
POP AX
ENDM
```

(b) ENDM : End of MACRO ▶ The directive ENDM informs the assembler the end of macro.

General form:

ENDM

Data control directives:

(a) PUBLIC ▶ The directive PUBLIC informs the assembler that the specified variable or segment can be accessed from other program modules.

General form:

PUBLIC variable1, variable2, ... variableN

Example:

PUBLIC MSG, NAME, NUM
PUBLIC ARRAY

132

(b) EXTERN ▶ The directive EXTERN informs the assembler that the data items or label following the directive will be used in a program module, which is defined in the other program modules.

General form:
For variable reference
EXTERN
variable_name1 : reference_type, ... variable_nameN : reference_type

For procedure reference
EXTERN procedure_name : [FAR/NEAR]

Examples:
EXTERN MSG : BYTE, NAME : WORD, NUM : BYTE
EXTERN DISPLAY : NEAR
EXTERN DISPLAY : FAR

(c) PTR ▶ The directive PTR is used to indicate the type of the memory access i.e BYTE / WORD / DWORD
For instance, if the assembler encounters the instruction like INC [SI], it will not be able to decide whether to code for byte increment or word increment.
It can be solved using PTR directive with the instruction as INC BYTE PTR [SI] for byte increment or INC WORD PTR [SI] for word increment

Examples:
INC BYTE PTR [DI]
ADD AL, BYTE PTR NUM
DEC WORD PTR [BX]

133

Branch displacement directives:

(a) SHORT ▶ The directive SHORT informs the assembler that the one byte displacement is required to code to jump instruction. Normally, two bytes are reserved to store the target address in the jump instruction. The target must be in range of -128 to +127 bytes from the address of the jump instruction.

Examples:
JMP SHORT NEXT

(b) LABEL ▶ The directive LABEL assigns a name to the current value in the location counter. The LABEL directive must be followed by the definition of datatype to which label is associated.

General form:
LABEL label_name label_type

Examples:
LABEL A_REF WORD
A DW 1000
The Label A_REF can be used as reference to the variable A
NEXT_NUM LABEL FAR
NEXT_NUM:
...
...
LOOP NEXT_NUM

File Inclusion Directive : → PTO

INCLUDE ► The directive INCLUDE informs the assembler to include the statement defined in the include file. The name of the include file follows the statement INCLUDE

General form: INCLUDE <File path specification with file name>

Example:
INCLUDE C:\TASM\MACRO.LIB
INCLUDE C:\TASM\BIN\MYPROC.LIB

Target Machine Code Generation Control Directive:
The assembler will recognise only 8086 instructions by default. It is because the program that confirm themselves to the 8086 instruction set that can run on any IBM PC, irrespective of the microprocessor i.e 8086, 80186, 80286, etc.

So special directive can be used at the beginning of the program to inform to the assembler to generate a code for the specific processes and these are listed below:

General form:
- .186 – Generate machine code for 80186 processor only
- .286 – Generate machine code for 80286 processor only
- .386 – Generate machine code for 80386 processor only
- .486 – Generate machine code for 80486 processor only
- .586 – Generate machine code for 80586 processor only i.e pentium

Difference between Assembler directive & instruction

Assembler directive	Instructions
① They give directions to assembler	① They perform operation specified by it using processor
② Assembler directives does not execute	② Instructions are executed by processor.