CHAPTER FOUR
ASSEMBLY LANGUAGE

Compiled By: Seble N

Why Assembly Language?

- Advantages of coding in Assembly Language
  - results in faster execution
  - generates smaller, more compact executable modules
  - provides more control over handling particular hardware requirements

Assembly Language: Current Usage

- Code that must interact directly with the hardware, for example in device drivers
- Computer viruses, bootloaders or other items very close to the hardware
- Situations where no high-level language exists, on a new or specialized processor
- Programs that need precise timing such as real-time programs like simulations, flight navigation systems, and medical equipment
- When a stand-alone executable of compact size is required that must execute without run-time components or libraries associated with a high-level language

Features of Assembly Languages

- Program comments
- Reserved Words
- Identifiers
- Statements
- Directives
- Data types

Program Comments

- In assembly program comments are defined using ';' character
  - Example
    - ADD AX, BX ; Add AX with BX
- Comments won't be translated to machine codes

Reserved Words

- Are words that have special meaning to the assembler
- Reserved words can be categorized into:
  - ***Instructions***
    - such as MOV, and ADD, which are operations that the computer can execute
  - ***Directives***
    - such as END or SEGMENT, which you use to provide information to the assembler
  - ***Operators***
    - such as FAR and SIZE, which you use in expressions
  - ***Predefined Symbols***
    - such as @Data and @Model, which return information to your program during the assembly

Identifiers

- The two types of identifiers are ***name and label***:
  - ***Name***: refers to the address of a data item, such as COUNTER in
    - COUNTER DB 0
  - ***Label***: refers to the address of an instruction, procedure, or segment, such as L1: in
    - L1: ADD BL, 25
- An identifier can use the following characters:
  - Alphabetic letters, Digits and Special characters: Question mark (?), underscore (_), dollar ($), at (@), dot (.)
    - The first character of an identifier must be an alphabetic letter or a special character, except for the dot
    - The names of registers, such as AH, BX, and DS, are reserved for referencing those registers.
- Assembly Language is Not CASE Sensitive!!!

Statements

- The two types of statements are:
  - ***Instructions***, such as MOV and ADD, which the assembler translates to object code
  - ***Directives***, which tell the assembler to perform a specific action, such as define a data item

- The format for a statement
  - [Identifier]Operation[operand (s)][; comment]

- Example
  - Instruction: L1: MOV AX, 0 ;store 0 in Ax

Directives

- Enable you to control the way in which a source program assembles and lists
- Act only during the assembly of a program and generate no machine-executable code
- Are part of the assembler's syntax, but are not related to the microprocessor instruction set
- The most common directives are:
  - **The PAGE and TITLE Listing Directives**
  - **SEGMENT Directive**
  - **PROC Directive**
  - **END Directive**
  - **ASSUME Directive**

PAGE Directive

- PAGE directive designates the maximum number of lines to list on a page and the maximum number of characters on a line
- Its format is
  - PAGE [Length] [, Width]
    - Ex. page 60,132
- Under a typical assembler, the number of lines per page may range from 10 through 255, and the number of characters per line may range from 60 through 132
- Omission of a PAGE statement causes the assembler to default to PAGE 50, 80

TITLE directive

- TITLE directive to cause a title for a program to print on line 2 of each page of the program listing
- Its format is
  - TITLE filename[comment]
    - Example
      - TITLE ASMSORT Assembly Program to sort CD titles

```
                                page 60,132
                        TITLE   A04DEFIN (EXE)  Define data directives
                                .MODEL  SMALL
                                .DATA
                        ;       DB - Define Bytes:
                        ;       -----------------
0C00 00                 BYTE1   DB    ?               ;Uninitialized
0001 30                 BYTE2   DB    48              ;Decimal constant
0002 30                 BYTE3   DB    30H             ;Hex constant
0003 7A                 BYTE4   DB    01111010B       ;Binary constant
0004 000A[ 00 ]         BYTE5   DB    10 DUP(0)       ;Ten zeros
000E 50 43 20 45 6D 70  BYTE6   DB    'PC Emporium'
     6F 72 69 75 6D                                   ;Character string
0019 31 32 33 34 35     BYTE7   DB    '12345'         ;Numbers as chars
001E 01 4A 61 6E 02 46  BYTE8   DB    01,'Jan',02,'Feb',03,'Mar'
     65 62 03 4D 61 72                                ;Table of months

                        ;       DW - Define Words:
                        ;       -----------------
002A FFF0               WORD1   DW    0FFF0H          ;Hex constant
002C 007A               WORD2   DW    01111010B       ;Binary constant
002E 001E R             WORD3   DW    BYTE8           ;Address constant
0030 0002 0004 0006 0007 WORD4  DW    2,4,6,7,9       ;Table of 5 constants
     0009
003A 0008[ 0000 ]       WORD5   DW    8 DUP(0)        ;Six zeros

                        ;       DD - Define Doublewords:
                        ;       ------------------------
004A 00000000           DWORD1  DD    ?               ;Uninitialized
004E 0000A25A           DWORD2  DD    41562           ;Decimal value
0052 00000018 00000030  DWORD3  DD    24, 48          ;Two constants
005A 00000001           DWORD4  DD    BYTE3 - BYTE2   ;Difference
                                                      ;  betw addresses
                        ;       DQ - Define Quadwords:
                        ;       ----------------------
005E 0000000000000000   QWORD1  DQ    0               ;Zero constant
0066 395E000000000000   QWORD2  DQ    05E39H          ;Hex constant
006E 5AA2000000000000   QWORD3  DQ    41562           ;Decimal constant
                                END
```

Segment Directives

- Are the directives used to define segments
- Format

```
Segment-name        SEGMENT        [align] [combine] ['class']
      …
Segment-name        ENDS
```

- Segment-name: must be unique
- SEGMENT : defines start of the segment
- ENDS: indicates the end of a segment

Segment Directives Cont'd

- The SEGMENT statement may contain three types of options:
  - Alignment
    - indicates the boundary on which the segment is to begin
    - The typical requirement is PARA as well as the default
    - Other align types include byte, word, dword & page
  - Combine
    - indicates whether to combine the segment with other segments when they are linked after assembly
    - Combine types are STACK, COMMON, PUBLIC, and AT expression
  - Class
    - enclosed in apostrophes, is used to group related segments when linking
    - The classes 'code' for the code segment, 'data' for the data segment, and 'stack' for the stack segment

ASSUME Directive

- used to tell the assembler the purpose of each segment in the program
- Format
  - ASSUME SS:stacksegname, DS:datasegname, CS:codesegname, …
    - is coded in the code segment

PROC Directive

- Used to define procedures

```
proc-name    PROC    [operand]
        …
proc-name    ENDP
```

  - PROC : indicates the start of a procedure
  - ENDP: indicates the end of a procedure
  - operand: could be 'FAR' or 'NEAR'
    - If declared FAR, the program loader uses this procedure as the entry point for the first instruction to execute
    - A code segment can have multiple procedures, but only one of the routines can be declared as 'FAR' others will be set 'NEAR' by default

END Directives

- ENDS directive
  - ends a segment
- ENDP directive
  - ends a procedure
- END directive
  - ends the entire program

Sample Assembly Program Structure

```
                page        60, 132
TITLE           Sample.asm   Segments of an .EXE program
; ------------------------------------------------------------------------------
STACK           SEGMENT  PARA    STACK    'Stack'
                ....
STACK           ENDS
;------------------------------------------------------------------------------
DATASEG     SEGMENT  PARA   'Data'
                ...
DATASEG     ENDS
;------------------------------------------------------------------------------
CODESEG     SEGMENT  PARA   'Code'
                MAIN        PROC   FAR
                ASSUME   SS:stack, DS:dataseg, CS:codeseg
                ...
                MAIN        ENDP                    ; End of procedure
CODESEG     ENDS                                    ; End of segment
                END         MAIN                    ; End of program
```

Simplified Segment Directives

- Some assemblers provide shortcuts in defining segments
- For example, in TASM to use the shortcuts you have to initialize the memory model before defining any segment
- The general format to define a memory model

  .MODEL memory-model

Simplified Segment Directives …

MODEL	Number of Code Segments	Number of Data Segments
Tiny	1 segment < 64Kb	
Small	1<64k	1<64k
Medium	Any number, any size	1<64k
Compact	1<64k	Any number, any size
Large	Any number, any size	Any number, any size

- The .MODEL directive automatically generates the required ASSUME statement for all models
- .STACK, .DATA, .CODE are used to define the different segments

Basic Structure of Assembly Programs

```
.MODEL      SMALL       ; Defining the model you are going to use throughout your program
.STACK      100           ; Defining your stack segment and its corresponding size
.DATA                     ; A directive used to define your data segment

   ; your data definition goes here

.CODE                     ; A directive used to define the code segment
     MAIN PROC            ; Defining the main function/procedure where the execution starts

   ; your code goes here


    MOV AX, 4C00H         ; An interrupt instruction which tells the assembler this is end of
    INT 21H               ; processing

     MAIN ENDP            ; A directive used to end the main procedure
END MAIN                  ; A directive which tells the assembler that this is end of the program
```

NB:-

- In every program you have to set your model before any instruction
- You can change the stack size based on your program
- You can leave the '.DATA' directive if you don't have any data definitions
- Any instruction written under the end of processing instruction is not going to be executed

Data Definition

- A data item may contain an undefined value, or a constant, or a character string, or a numeric value
- The format for data definition is:
  - [name] Dn expression
  - Name: identifier
  - Dn: Directives can be:
    - DB: byte DQ:quadword
    - DW: word DT:tenbytes
    - DD: doubleword
    - each of which explicitly indicates the length of the defined item
  - Expression:
    - can be unnitialized: ?
    - can be assigned a constant: such as 25, 21,'a',"Hello"

Defining Numeric Data

- X DB 25
- X DB ? ; uninitialized
- X DB 25, 03, 45 ;defined in adjacent bytes
  - X refers to the first 1-byte constant, 25, and a reference to X+1 is to the second constant, 03 and so on

Defining Numeric Data …

- The expression also permits duplication of constants
  - Definition Format
    - [name] Dn repeat-count DUP (Expression)
- Examples
  - Y DB 5 DUP (2) ; Five bytes containing 2
  - X DW 10 DUP (?) ;Ten words, uninitialized
  - Z DB 3 DUP (5 DUP (4)) ;generates five copies of the digit 4 (44444) and duplicates that value three times, giving fifteen 4s in all

Defining Character Data

- A string can be defined using either single or double quotes
- DB is the conventional format for defining character data of any length
- Example
  - ch DB ‘B’
  - Name DB ‘Abebe’

Equate Directives

- EQU directives are used for redefining symbolic names with other names and numeric values with names
- These directives do not generate any data storage
  - **Equal-sign Directive**
    - PI = 3.1416
  - **EQU Directive**
    - X EQU 12
    - Y DB X DUP(?)

Numeric Literals

- **Binary**:
  - uses the binary digits 0 and 1, followed by the radix specifier B. Example 01011010B
- **Decimal**:
  - uses the decimal digits 0 through 9, optionally followed by the radix specifier D, such as 125 or 125D.
  - the assembler converts your decimal values to binary object code and represents them in hexadecimal. For example, a definition of decimal 125 hex 7D.
- **Hexadecimal**:
  - uses the hex digits 0 through F, followed by the radix specifier H.
  - the first digit of a hex constant must be 0 to 9
  - Examples are 3DH and 0DE8H, which the assembler stores as 3D and (with bytes in reverse sequence) E80D, respectively.
- **Real**:
  - The assembler converts a given real value (a decimal or hex constant followed by the radix specifier R) into floating-point format for use with a numeric coprocessor.