Chapter 3

Assembly Language Fundamentals

Introduction

You write an assembly program according to a strict set of rules, use an editor or word processor for keying it into the computer as a file, and then use the assembler translator program to read the file and to convert it into machine code.

The two main classes of programming languages are ***high-level*** and ***low-level.*** Programmers writing in a high-level language, such as C or BASIC, use powerful commands, each of which may generate many machine language instructions. Programmers writing in a low-level assembly language, on the other hand, code symbolic instructions, each of which generates one machine instruction.

Despite the fact that coding in a high-level language is more productive, some advantages to coding in assembly language are that it in general:

- Provides more control over handling particular hardware requirements.
- Generates smaller, more compact executable modules
- Results in faster execution

A ***linker*** program for both high- and low-levels completes the process by converting the object code into executable machine language.

Assembly Language Features

The features of this language provide the basic rules and framework for the language.

Program Comments

The use of comments throughout a program can improve its clarity, especially in assembly language, where the purpose of a set of instructions is often unclear.

A comment begins with a semicolon (;), and wherever you code it, the assembler assumes that all characters on the line to its right are comments. A comment may contain any printable character, including a blank.

A comment may appear on a line by itself, like this:

```
        ; Calculate productivity ratio
```

Or on the same line following an instruction, like this:

```
        ADD AX, BX          ; Accumulate total quantity
```

Because a comment appears only on a listing of an assembled source program and generates no machine code, you may include any number of comments without affecting the assembled program’s size or execution.

Reserved Words

Certain names in assembly language are reserved for their own purposes, to be used only under special conditions. Reserved words, by category, include:

- ***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; and
- ***Predefined Symbols,*** such as @Data and @Model, which return information to your program during the assembly.

Using a reserved word for a wrong purpose causes the assembler to generate an error message.

Identifiers

An identifier is a name that you apply to an item in your program that you expect to reference. The two types of identifiers are ***name and label***:

1. ***Name*** refers to the address of a data item, such as COUNTER in

```
COUNTER DB 0
```

NB: The assembler translates names into memory addresses

2. ***Label*** refers to the address of an instruction, procedure, or segment, such as MAIN and B30: in the following statements:

```
MAIN PROC  FAR
B30:  ADD   BL, 25
```

The same rules apply to both names and labels. An identifier can use the following characters:

Category	**Allowable Characters**
Alphabetic letters:	A through Z and a through z
Digits:	0 through 9 (not the first character)
Special characters:	Question mark (?), break, or underscore (_), dollar ($), at (@), dot (.) (not first character)

- Names can be from 1 to 31 characters long

The first character of an identifier must be an alphabetic letter or a special character, except for the dot. Because the assembler uses some special words that begin with the @ symbol, you should avoid using it for your own definitions.

- If period is used, it must be the first character.
- The assembler does not differentiate between upper case and lower case in a name

The names of registers, such as AH, BX, and DS, are reserved for referencing those registers. Consequently, in an instruction such as ADD CX, BX the assembler knows that CX and BX refer to registers. However, in an instruction such as MOV REGSAVE, CX the assembler can

Examples of legal names:

```
COUNTER1
@character
SUM_OF_DIGITS
$1000
```

DONE?
.TEST

Examples of illegal names

Two words	; conatins blank
1digit	;begins with a digit
A.digit	; . not first character

An assembly program consists of a set of statements. The two types of statements are:

1. ***Instructions***, such as MOV and ADD, which the assembler translates to object code; and
2. ***Directives***, which tell the assembler to perform a specific action, such as define a data item.

The format for a statement, where square brackets indicate an optional entry:

[Identifier]	Operation	[operand (s)]	[; comment]

An identifier (if any), operation, and operand (if any) are separated by at least one blank or tab character.

Examples of statements are:

	Identifier	Operation	Operand	Comment
Directive:	COUNT	DB	1	; Name, operation, operand
Instruction:	L30:	MOV	AX, 0	; Label, operation, 2 operands

The identifier, operation, and operand may begin in any column. However, consistently starting at the same column for these entries makes a more readable program. Also, many editor programs provide tab stops every eight positions to facilitate spacing the fields.

The ***operation*** contains a symbolic operation code(opcode). The assembler translates a symbolic opcode into a machine language opcode

The ***operation***, which must be coded, is most commonly used for defining data areas and coding instructions. For a data item, an operation such as DB or DW defines a field, work area, or constant. For an instruction, an operation such as MOV or ADD indicates an action to perform.

- Opcode symbols often describe the operation's function: for example,MOV,ADD,SUB
- In an assembler directive, the operation field contains a pseudo-operation code(Pseudo-op). Psedo-ops are not translated into machine code, rather they simply tell the assembler to do something. For example, the PROC pseudo-op is used to create a procedure.

The ***operand*** (if any) provides information for the operation to act on. For a data item, the operand defines its initial value. For example, in the following definition of a data item named COUNTER, the operation DB means "define byte", and the operand initializes its contents with a zero value:

```
Name        Operation   Operand    Comment
COUNTER     DB          0          ; Define byte with initial 0 value
```

For an instruction, an operand indicates where to perform the action. An instruction's operand may contain one, two, or even no entries. Here are three examples:

```
Operation    Operand    Comment
RET                     ; Return from a procedure
INC          BX         ; Increment BX register by 1
ADD          CX, 25     ; Add 25 to CX register
```

- In a two operand instruction, the first operand is the destination operand. It is the register or memory location where the result is stored. The second operand is the source operand. The source is usually not modified by the instruction.

Directives

Assembly language supports a number of statements that enable you to control the way in which a source program assembles and lists. These statements, called ***directives***, act only during the assembly of a program and generate no machine-executable code.

The most common directives are discussed as follows:

PROC Directive

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

```
Name         operation    operand    comment
Proc-name    PROC          FAR       ; Begin procedure
             …
Proc-name    ENDP                    ; End procedure
```

The ***Proc-name*** must be present, must be unique, and must follow assembly language naming conventions. The operand, FAR, is related to program execution. When you request execution of a program, the program loader uses this procedure as the entry point for the first instruction to execute.

The ENDP directive indicates the end of a procedure and contains the same name as the PROC statement to enable the assembler to relate the end to the start. Because a procedure must be fully contained within a segment, ENDP defines the end of the procedure before ENDS defines the end of the segment.

The code segment may contain any number of procedures used as subroutines, each with its own set of matching PROC and ENDP statements. Each additional PROC is usually coded with (or defaults to) the NEAR operand.

END Directive

An END directive ends the entire program and appears as the last statement. Its format (operation and operand) is:

```
END [procedure-name]
```

The operand may be blank if the program is not to execute; for example, you may want to assemble only data definitions or you may want to link the program with another module. In most programs, the operand contains the name of the first or only PROC designated as FAR, where program execution is to begin.

General Structure
Assembly programs have common structure

.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 assemble 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

Memory model

- The size of code and data a program can have is determined by specifying a memory model using the .MODEL directive.
- The syntax is:
  .MODEL memory_model
- The most frequent memory models are :SMALMAPCT, and LARGE.
  MEMORY MODELS

Model	Description
SMALL	Code in one segment, Data in one segment
MEDIUM	Code in more than one segment Data in one segment
COMPACT	Code in one segment Data in more than one segment
LARGE	Code in more than one segment Data in more than one segment

Data Segment

- A program's data segment contains all the variable definitions.
- Constant definitions are often made here as well, but they may be elsewhere in the program since no memory allocation is involved
- To declare a data segment, we use the directive .DATA, followed by variable and constant declarations.

Defining Types of Data

The assembler provides a set of directives that permits definitions of items by various types and lengths; for example, DB defines byte and DW defines word. A data item may contain an undefined (that is, uninitialized) value, or it may contain an initialized constant, defined either as a character string or as a numeric value.

The format for data definition is:

```
          [name]          Dn              Expression
```

- **Name**: - A program that references a data item does so by means of a name. The name is otherwise optional, as indicated by the square brackets.
- **Directive** (Dn): - The directives that define data items are DB (byte), DW (word), DD (doubleword), DF (farword), DQ (Quadword), and DT (tenbytes), each of which explicitly indicates the length of the defined item.
- **Expression**: - The expression in an operand may specify an uninitialized value or constant value. To indicate an uninitialized item, define the operand with a question mark, such as

```
                    DATAX        DB              ?        ; Uninitialized item
```

Data defining Pseudo-ops

Pseudo-ops	Stands-for
DB	Define Byte
DW	Define word
DD	Define doubleword(two consecutive words)
DQ	Define Quadword(four consecutive words)
DT	Define tenbytes(ten consecutive bytes)

Byte variables

- The assembler directive that defines a byte variable takes the following form:

  Name   **DB**   **initial_value**
- Where the pseudcode DB stands for "Define Byte"

  Eg:

  ALPHA DB 4
- This directive causes the assembler to associate a memory byte with the name ALPHA,and initialize it to 4.
- A question mark("?") can be used in place of an initial value for uninitialized byet

  Eg: **BYT DB ?**

- The decimal range of initial that can be specified is -128 to 127 if signed interpretation is being given, or 0 to 255 for unsigned interpretation. These are the ranges of values that fit in a byte

In this case, when your program begins execution, the initial value of DATAX is unknown to you. You can use the operand to define a constant, such as

```
DATAY     DB          25     ; Initialized item
```

You can freely use this initialized value 25 throughout your program and can even change the value.

Word variable

- The assembler directive for defining a word variable has the following form:

  Name DW initial_value
- The decimal range of initial that can be specified is -32768 to 32767 if signed interpretation is being given, or 0 to 65535 for unsigned interpretation. These are the ranges of values that fit in a byte

ARRAYS

- In assembly language, an array is just a sequence of memory bytes or words,
- An expression may contain multiple constants separated by commas and limited only by the length of the line, as follows:

```
DATAZ     DB          21, 22, 23, 24, 25
```

The assembler defines these constants in adjacent bytes, from left to right. A reference to DATAZ is to the first 1-byte constant, 21, and a reference to DATAZ+1 is to the second constant, 22.
For example, the instruction

```
MOV AL, DATAZ+3
```

Loads the value 24 (18H) into the AL register.

Example:

B_ARRAY DB 10H,20H,30H

- The name B_ARRAY is associated with the first of these bytes, B_ARRAY+1 with the second, and B_ARRAY+2 with the third
- If the assembler assigns the offset 0200h to B_ARRAY ,then the memory would look like this:

Symbol	Address	Contents
B_ARRAY	200h	10h
B_ARRAY +1	201h	20h
B_ARRAY +2	202h	30h

- In the same way, an array of words may be defined. For example:

  W_ARRAY   DW   1000,40,29887,329
- Sets up an array of four words with name initial values 1000,40,29887,and 329
- The initial word is associated with the name W_ARRAY , the next one with W_ARRAY +2,the next with W_ARRAY+4, and so on.
- If the array starts at 0300h,it will look like this:

Symbol	Address	Contents
W_ARRAY	0300h	1000d
W_ARRAY +2	0302h	40d
W_ARRAY +4	0304h	29887d
W_ARRAY +6	0306h	329d

- The expression also permits duplication of constants in a statement of the format.

```
[name]          Dn              repeat-count    DUP (Expression) …
```

The following examples illustrate duplication:

```
DW 10 DUP (?)               ; Ten words, uninitialized
DB 5 DUP (2)                ; Five bytes containing 22222
DB 3 DUP (5 DUP (4))        ; Fifteen 4s
```

The third example generates five copies of the digit 4 (44444) and duplicates that value three times, giving fifteen 4s in all.

An expression may define and initialize a ***character string*** or a ***numeric constant.***

Character Strings

Character strings are used for descriptive data such as people's name and product descriptions. The string is defined within single quotes, such as 'PC', or within double quotes, such as "PC". The assembler stores the contents of the quotes as object code in normal ASCII format, without the apostrophes.

DB is the conventional format for defining character data of any length. An example is

```
DB 'Computer City'
```

If a string must contain a single or double quote, you can define it in one of these ways:

```
DB "Crazy Sam's CD Emporium"   ; Double quotes for string, single quotes for
                                ; apostrophe
DB 'Crazy Sam's CD Emporium'   ; Single quotes for string, two single quotes
                               ; for apostrophe
```

Numeric Constants

Numeric constants are used to define arithmetic values and memory addresses. The constant is not defined within quotes, but is followed by an optional radix specifier, such as H in the hexadecimal value 12H. For most of the data definition directives, the assembler stores the generated bytes in object code in reverse sequence, from right to left.

Following are the various numeric formats:

- **Binary**: - uses the binary digits 0 and 1, followed by the radix specifier B. A common use for binary format is to distinguish values for the bit-handling instructions AND, OR, XOR, and TEST.
- **Decimal**: - uses the decimal digits 0 through 9, optionally followed by the radix specifier D, such as 125 or 125D. Although the assembler allows you to define values in decimal format as a coding convenience, it 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. Because the assembler expects that a reference beginning with a letter is a symbolic name, 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. Because the letters D and B act as both radix specifiers and hex digits, they could conceivably cause some confusion.
- **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.

Named Constants

Equate Directives (EQU-equates)

The assembler provides Equal-sign, and EQU directives for redefining symbolic names with other names and numeric values with names. These directives do not generate any data storage; that is, a program cannot, say, add to an EQU item when it executes. Instead, the assembler uses the defined value to substitute in other statements.
The advantage of equate directives is that many statements may use the assigned value. If the value has to be changed, you need change only the equate statement. The result is a program that is more readable and easier to maintain;
The **Equal-sign Directive**: - enables you to assign the value of an expression to a name, and may do so any number of times in a program. The following examples illustrate its use:

```
VALUE_OF_PI = 3.1416
RIGHT_COL = 79
SCREEN_POSITIONS = 80 * 25
```

Examples of the use of the preceding directives are:

```
IMUL AX, VALUE_OF_PI            ; Multiply AX by 3.1416
CMP BL, RIGHT_COL               ; Compare BL to 79
MOV CX, SCREEN_POSITIONS        ; Move 2000 to CX
```

When using this directive for defining a doubleword value, first use the .386 directive to notify the assembler:

```
.386
DBLWORD1 = 42A3B05CH
```

The **EQU Directive: -**

- To assign a name to a constant, we can use the **EQU** (equates) pseudo-op. The syntax is:
  name EQU   constant

Example:

LF EQU 0AH

Assigns the name LF to 0AH, the ASCII code of the line feed character. The name LF may now be used in place of 0AH anywhere in the program.
Consider the following EQU statement coded in the data segment:

```
FACTOR EQU 12
```

The name, in this case FACTOR, may be any name acceptable to the assembler. Now whenever the word FACTOR appears in an instruction or another directive, the assembler substitutes the value 12. For example, the assembler converts the directive

```
        TABLEX DB FACTOR DUP(?)
```

To its equivalent value

```
        TABLEX DB 12 DUP(?)
```

TABLEX may be defined by EQU only once so that it cannot be redefined by another EQU. An instruction may also contain an equated operand, as in the following:

```
        RIGHT_COL EQU 79
                  …
                  MOV CX, RIGHT_COL      ; Move 79 to CX
```

You can also equate symbolic names, as in the following code:

```
        ANNL_TEMP DW 0
                  …
        AT        EQU ANNL_TEMP
        MPY       EQU MUL
```

The first EQU equates the nickname AT to the defined item ANNL_TEMP. For any instruction that contains the operand AT, the assembler replaces it with the address of ANNL_TEMP. The second EQU enables a program to use the word MPY in place of the regular symbolic instruction MUL.

Note: No memory is allocated for EQU names.

Stack Segment

- The purpose of stack segment is to set aside a block of memory (the stack area) to store the stack.
- The stack area should be big enough to contain the stack at its maximum size.

The declaration syntax is:

```
    .stack  size
```

- Where size is optional number that specifies the stack area size in bytes.
  For example:
  .stack 100h ; sets aside 100h bytes for the stack area. If size is omitted, 1KB is set.

Code Segment

The code segment contains a program’s instructions. The declaration syntax is:

.code   name

Where name is optional name of the code segment

- Inside code segment, instructions are organized as procedures.
- The simplest procedure definition is :

```
Name proc

     ;body of the procedure

Name ENDP
```

Example:

```
MAIN   PROC

    ; main procedure instructions
MAIN ENDP
; other  procedures go here
```

Putting it together

```
.MODEL SMALL
.STACK 100H
.DATA
    ; data definition go here
.CODE

    MAIN PROC

     ; instructions go here
  MAIN ENDP
    ;other procedures go here
  END MAIN
```

Note: The last line in program should be the END directive, followed by name of the main procedure.

Assembling, Linking, and Running Programs

Assembly programs go through **three main stages** before execution:
A source program written in assembly language cannot be executed directly on its target computer. It must be translated, or *assembled* into executable code. In fact, an assembler is very similar to a *compiler*, the type of program you would use to translate a C++ or Java program into executable code.
The assembler produces a file containing machine language called an *object file*. This file isn't quite ready to execute. It must be passed to another program called a *linker*, which in turn produces an *executable file*. This file is ready to execute from the operating system's command prompt.

The Assemble-Link-Execute Cycle

The process of editing, assembling, linking, and executing assembly language programs. Following is a detailed description of each step.
Step 1: A programmer uses a **text editor** to create an ASCII text file named the *source file*.
Step 2: The **assembler** reads the source file and produces an *object file,* a machine-language translation of the program. Optionally, it produces a *listing file*. If any errors occur, the programmer must return to Step 1 and fix the program.
Step 3: The **linker** reads the object file and checks to see if the program contains any calls to procedures in a link library. The **linker** copies any required procedures from the link library, combines them with the object file, and produces the *executable file*.
Step 4: The operating system **loader** utility reads the executable file into memory and branches the CPU to the program's starting address, and the program begins to execute.
See the topic "Getting Started" on the author's Web site (www.asmirvine.com) for detailed instructions on assembling, linking, and running assembly language programs using Microsoft Visual Studio.