Chapter-4

Introduction to IBM PC Assembly Language

4.1 Assembly Language Syntax

Assembly language programs are translated into machine language instruction by an assembler, so they must be written to conform to the assembler's specification. Assembly language code is generally not case sensitive, but we use upper case to differentiate code from the rest of the text.

Statements

Each statements is either an **instruction**, which the assembler translates into machine code, or an **assembler directive**, which instructs the assembler to perform some specific task, such as allocating memory space for a variable or creating a procedure. Both instructions and directives have up to four fields:

name operation operand (s) comment

At least one blank or tab must separate the fields.

An example of an instruction is

```
START:  MOV CX, 5                ; initialize the counter
```

Here, the name field consists of the label START:. The operation is MOV, the operands are CX and 5, and the comment is ; initialize the counter.

An example of an assembler directive is

```
MAIN   PROC
```

MAIN is the name, and operation field contains PROC. This particular directive creates a procedure called MAIN.

4.1.1 Name Field

The name field is used for instruction labels, procedure names, and variable names. The translates names into memory addresses.

Names can be from 1 to 31 characters long, and may consists of letters, digits, and the special characters. It must begin with a character but not with a digit. The assembler does not differentiate between upper and lower case in a name.

4.1.2 Operation Field

For an instruction, the operation field contains a symbolic operation code (opcode). The assembler translates a symbolic opcode into a machine language opcode. For example MOV, ADD, SUB.
In assembler directive, the operation field contains a pseudo-operation code (**pseudo-op**). Pseudo-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.

4.1.3 Operand Field

The operand field specifies the data that are to be acted on by the operation. An instruction may have zero, one, or two operands. For example,

```
NOP                 no operands; does nothing
INC AX              one operand
ADD WORD1, 2        two operands
```

For a two operands instruction the **source operand** can either be a register, or a memory location (variable), or a immediate value (constant); but the **destination operand** is a register, or a memory location (variable), never be an immediate value (constant). On the other hand, at any time both source and destination cannot be memory locations (exceptional cases are MOVSB, MOVSW instructions)

4.1.4 Comment Field

Comments are optional. A comment must be started with a semicolon (;). For example

```
MOV CX, 0           ; CX counts terms, initially 0
```

4.2 Program Data

Numbers

- A binary number must end with letter “B” or “b”.
- A decimal number is ended with an optional “D” or “d”.
- A hex number must begin with a decimal digit and end with the letter “H” or “h”.

Number	Type
11011	decimal
11011B	binary
64223	decimal
-21843D	decimal
1,234	illegal-contains a nondigit character
1B4DH	hex
1B4D	illegal hex number-does not end with “H”
FFFFH	illegal hex number-does not begin with a decimal digit
0FFFFH	hex

Characters

Characters or character strings must be enclosed in a single or double quotes; for example, “A” or ‘hello’. Characters are translated into their ASCII codes by the assembler, so there is no difference between using “A” and 41H in a program.

Table 4.1 Data-Defining Pseudo-ops

Pseudo-op	Stands for
DB	define byte
DW	define word
DD	define doubleword (two consecutive words)
DQ	define quadword (four consecutive words)
DT	define ten bytes (ten consecutive bytes)

4.3 Variables

We use DB and DW to define byte variables, word variables, and arrays of bytes and words.

4.3.1 Byte Variables

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

name DB initial-value

For example,

ALPHA DB 4

For an uninitialized byte type variable a mark (“?”) is used in place of value. For example

BYT DB ?

4.3.2 Word Variables

The assembler directive that defines a word variable has the following form:

name DW initial-value

For example,

```
WRD   DW  -2
```

For an uninitialized word type variable a mark (“?”) is used in place of value. For example,

```
WRD   DW   ?
```

4.3.3 Arrays

In assembly language, an array is just a sequence of memory bytes or words. For 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 address 0200H to B_ARRAY, then the memory would like this:

Symbol	Address	Contents
B_ARRAY	0200H	10H
B_ARRAY+1	0201H	20H
B_ARRAY+2	0202H	30H

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

```
W_ARRAY   DW   1000, 40, 29887, 329
```

If the assembler assigns the offset address 0300H to W_ARRAY, then the memory would like this:

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

Character Strings

An array of ASCII codes can be initialized with a string of characters. For example,

LETERS DB ‘ABC’

is equivalent to

LETERS DB 41H, 42H, 43H

It is possible to combine characters and numbers I one definition; for example

MSG DB ‘HELLO’, 0AH, 0DH, ’$’

4.4 Named Constants

EQU (Equates)

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

name EQU constant

For example, the statement

LF EQU 0AH

assigns the name LF to 0AH. Now the name can be used in place of 0AH anywhere in the program. Thus,

MOV DL, 0AH and MOV DL, LF are equivalent.

Constant can also be a string; for example,

PROMPT EQU 'TYPE YOUR NAME'

Then instead of

PROMPT DB 'TYPE YOUR NAME'

we could say

MSG DB PROMPT

Note: No memory is allocated for EQU names.

4.5. A Few Basic Instructions

There are over a hundred instructions in the instruction set for the 8086. In this section we discuss six of the most useful instructions for transferring data and doing arithmetic.

In the following, WORD1 and WORD2 are word variables, and BYTE1 and BYTE2 are byte variables.

4.5.1 MOV and XCHG Instructions

MOV Instruction

The MOV instruction is used to transfer data between registers, between a register and memory location, or to move a number directly into a register or memory location. The syntax is

MOV destination, source

Here are some example:

Example: MOV AX, WORD1

Example: MOV AX, BX

Again,

MOV AH, 'A'

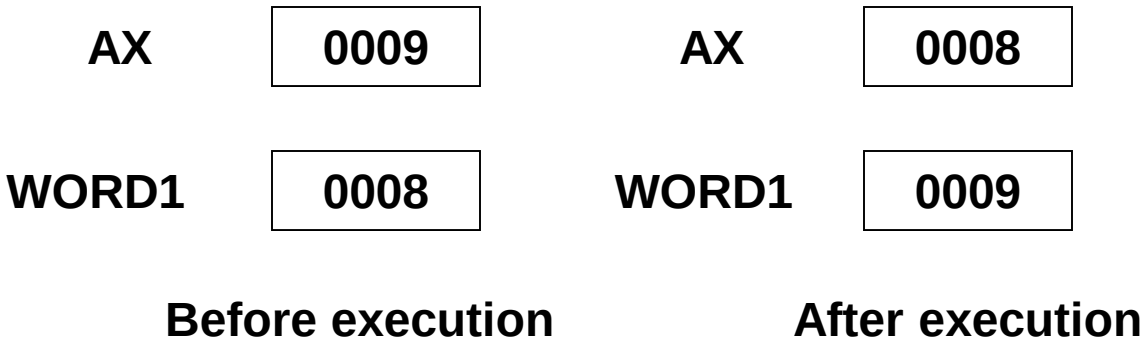
This is a move of the number 41H (the ASCII cod "A") into register AH.

XCHG Instruction

The XCHG (exchange) operations used to exchange the contents of two registers, or a register and a memory location. The syntax is

XCHG destination, source

Example: XCHG AX, WORD1



Restrictions on MOV and XCHG Instruction

There are a few restrictions on the use of MOV and XCHG. Note a MOV or XCHG between memory locations is not allowed. For example,

```
MOV   WORD1, WORD2
```

Is illegal

but we can get around this restrictions by using a register

```
MOV  AX, WORD1
MOV  WORD1, WORD2
MOV  WORD2,  AX
```

Table 4.2 : Legal combinations of operands for MOV and XCHG

MOV

Source Operand	Destination operand			
	General register	Segment register	Memory location	Constant
General register	yes	yes	yes	no
Segment register	yes	no	yes	no
Memory location	yes	yes	no	no
Constant	yes	no	yes	no

XCHG

Source Operand	Destination operand	
	General register	Memory location
General register	yes	yes
Memory location	yes	no

4.5.2 ADD, SUB, INC, and DEC Instruction

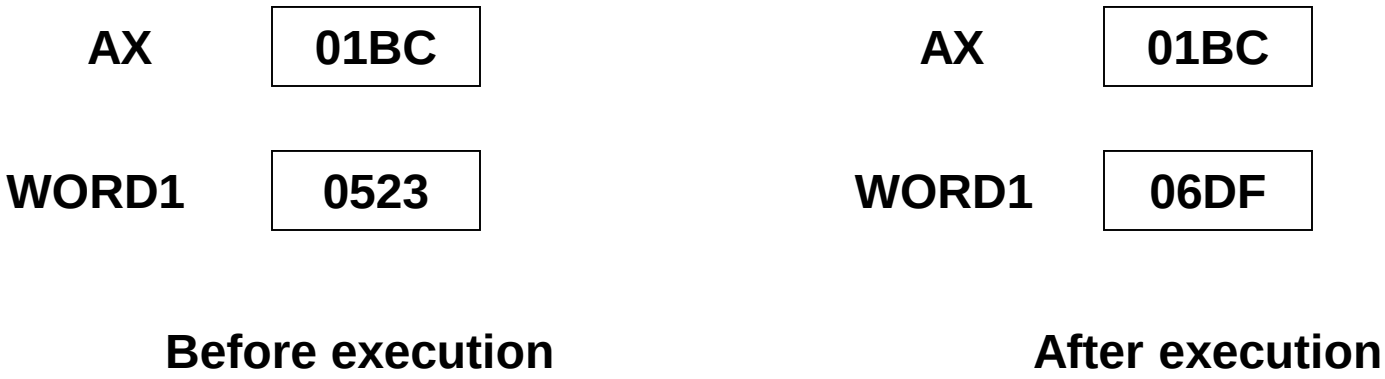
ADD and SUB Instructions

The ADD and SUB instructions are used to add or subtract the contents of two registers, a register and memory location, or to add (subtract) a number to (form) a register or a memory location. The syntax is

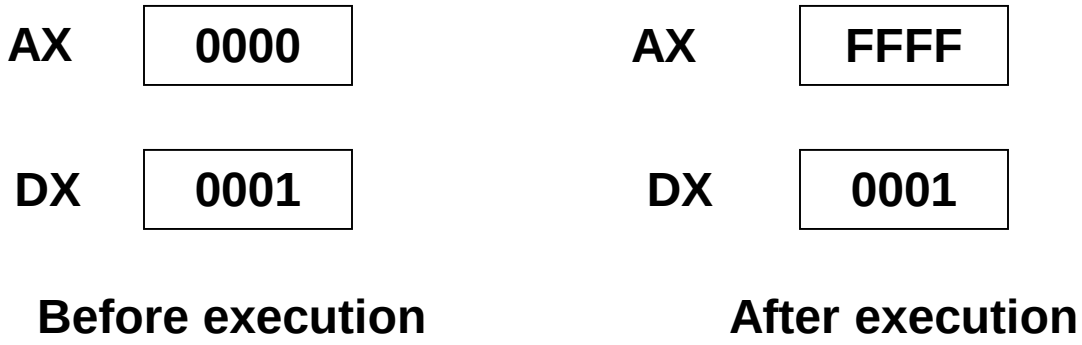
ADD destination, source

SUB destination, source

Example: ADD WORD1, AX



Example: SUB AX, DX



Direct addition or subtraction between memory location is illegal. For example,

ADD BYTE1, BYTE2

is not legal.

A solution of the above problem is

MOV AL, BYTE2

ADD BYTE1, AL

Table 5.2 : Legal combinations of operands for ADD and SUB

Source Operand	Destination operand	
	General register	Memory location
General register	yes	yes
Memory location	yes	no
Constant	yes	yes

INC and DEC Instructions

INC (increment) is used to add 1 to contents of a register or memory location and DEC (decrement) subtracts 1 from a register or memory location. The syntax is

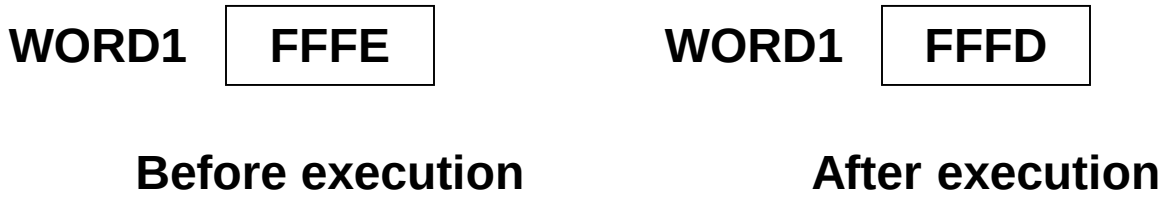
INC destination

DEC destination

Example: INC WORD1

Before execution **After execution**

Example: DEC WORD1



4.5.3 NEG Instruction

NEG is used to negate the contents of the destination. NEG does this by replacing the contents by its two's complement. The syntax is

NEG destination

The destination may be a register or memory location.

Example: NEG BX

Type Agreement of Operands

The operands of the preceding two-operand instruction must be of the same type; that is, both bytes or words. Thus an instruction such as

MOV AX, BYTE1

is not allowed. However, the assembler will accept both of the following instruction:

MOV AH, 'A'

and

MOV AX, 'A'

4.7. Program Structure

The code, data, and stack are structured as program segments. Each program segment is translated into a memory segment by the assembler.

4.7.1 Memory Models

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 frequently used memory models are SMALL, MEDIUM, COMPACT, and LARGE. The .MODEL directive should come before any segment definition.

Table 4.4. 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 no array larger than 64k bytes
HUGE	code in more than one segment data in more than one segment arrays larger than 64k bytes

4.7.2 Data Segment

A program's **data segment** contains all the variable definitions. To declare a data segment, we use directive .DATA, followed by variable and constant declarations. For example,

```
.DATA
WORD        DW      2
MSG         DB      'THIS IS A MESSAGE'
MASK        EQU     10010010B
```

4.7.3 Stack Segment

The purpose of the **stack segment** declaration is to set aside a block of memory (the stack area) to store the stack. The declaration syntax is

```
.STACK      size
```

where size is in bytes.

For example,

.STACK 100H

4.7.4 Code Segment

The **code segment** contains a program's instructions. The declaration syntax is

.CODE name

where name is the optional and for a SMALL memory segment it is not given.

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

```
name   PROC
; body of the procedure
name ENDP
```

For example,

```
.CODE

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

4.7.4 Putting It Together

```
.MODEL          SMALL
.STACK          100H

.DATA
; data definitions go here

.CODE

MAIN   PROC
; instructions go here
MAIN   ENDP

; other procedures go here
END     MAIN
```

4.8 Input Output Instructions

CPU communicates with the peripherals through I/O registers called I/O ports. There are two instructions, IN and OUT, that access the ports directly. These instructions are used when fast I/O is essential; for example, in a game program. However , most applications programs do not use IN and OUT because (1) port addresses vary among computer models, and (2) it's much easier to program I/O with the service routines provided by the manufacturer.

There are two categories of I/O service routines: (1) the Basic Input/output System (BIOS) routines and (2) the DOS routines. The BIOS routines are stored in ROM and interact directly with the I/O ports. We use them to carry out basic screen operations such as moving cursor and scrolling the screen. The DOS routines can carry out more complex tasks; for example, printing a character string; actually they use the BIOS routines to perform direct I/O operations.

The INT Instruction

To invoke a DOS routine, the **INT** (interrupt) instruction is used. It has the format

INT interrupt_number

where interrupt_number is a number that specifies a routine. For example, INT 16H invoke a BIOS routine that perform keyboard input.

4.8.1 INT 21H

INT 21H may be used to invoke a large number of DOS functions; a particular function is requested by placing a function number in the AH register and invoking INT 21H.

Function number	Routine
1	single-key input
2	single-character output
9	character string output

Function 1:

Single-key Input

Input: AH = 1

Output: AL = ASCII code if character key is pressed

= 0 if non-character key is pressed

To invoke a routine, execute these instructions:

```
MOV AH,1
INT 21H
```

If a character key is pressed, AL gets its ASCII code; the character is also displayed on the screen. If any other key is pressed, such as an arrow key, F1-F10, and so on, AL will contain 0.

Function 2:

Display as character or execute a control function

Input:	AH = 2
	DL = ASCII code of the display character or control character
Output:	AL = ASCII code of the display character or control character

To display a character with this function, we put its ASCII code in DL. For example:

```
MOV AH,2
MOV DL, '?'
INT 21H
```

If DL contains the ASCII code of a control character, the INT 21H causes control function to be performed. The principal control characters are:

ASCII code (Hex)	Symbol	Function
7	BEL	beep (sounds a tone)
8	BS	backspace
9	HT	tab
A	LF	line feed (new line)
D	CR	carriage return (start of current line)

On execution, AL gets the ASCII code of the control character.

4.9 A First Program

Our first program will read a character from the keyboard and display it at the beginning of the next line.

```
TITLE    PGM4_1:  ECHO PROGRAM

.MODEL SMALL
.STACK 100H
.CODE
MAIN PROC

MOV AH,2
MOV DL, '?'
INT 21H

MOV AH,1
INT 21H
MOV BL,AL
MOV AH,2
MOV DL,0DH
INT 21H
MOV DL,0AH
INT 21H

MOV DL,BL
INT 21H

MOV AH,4CH
INT 21H

MAIN ENDP
END MAIN
```

Because no variables were used, the data segment was omitted.

4.11 Displaying a String

The INT 21H function that can be used to display a character string:

INT 21H, Function 9:

Display a String

Input: DX = offset address of string.
The string must end with a '$' character.

The '$' marks the end of the string and is not displayed. If the string contains the ASCII code of a control character, the control function is performed.

Here is an example of a string:

```
MSG  DB  'HELLO!$'
```

The LEA Instruction

LEA stands for “Load Effective Address”.

INT 21H, function 9, expects the offset address of the character to be in DX.

The syntax of LEA instruction:

LEA destination, source

where destination is a general register and source is a memory location. For example,

LEA DX, MSG

puts the offset address of the variable MSG into DX.

Program Segment Prefix

When a program is loaded in memory, DOS prefaces it with a 256 byte **program segment prefix (PSP)**. The PSP contains information about the program. So that programs may access this area, DOS places its segment number in both DS and ES before executing the program. The result is that DS does not contain the segment number of the data segment. To correct this, a program containing data segment begins with these two instructions:

```
MOV AX, @DATA
MOV DS, AX
```

@DATA is the name of the data segment defined by .DATA. The assembler translates the name @DATA into a segment number.

Our second program will print a sting of characters on monitor.

```
TITLE    PGM4_2:  PRINT STRING PROGRAM

.MODEL SMALL
.STACK 100H
.DATA
 MSG  DB  'HELLO$'
.CODE
MAIN PROC

MOV AX, @DATA
MOV DS, AX
LEA DX, MSG
MOV AH,9
INT 21H
MOV AH,4CH
INT 21H

MAIN ENDP
END MAIN
```