



Introduction to IBM-PC Assembly Language

Covering topics

- Assembly Language Syntax.
- Declaration of variables, simple data movements and arithmetic operations.
- Program organization (code, data and stack).
- I/O operation in Assembly language.
- How to compile and run Assembly language program.

Assembly Language Syntax

- Assembly language programs are translated into machine language instruction by assembler, we will use **emulator**.
- Assembly language code is not case sensitive.
- **Statements:** one per line, which is either an instruction (converted into machine code) or an assembler directive which is a direction to assembler to perform specific operations like allocating memory for variable, or creating a procedure.
- Both directive and instruction has four fields.
 - name operation operand(s) comment
- At least one space must separate the fields.
- Example of an instruction
 - START: MOV CX,5 ;initialize counter
- Example of directive
 - MAIN PROC

Name Field

- Used for instruction labels, procedure names, and variable names.
- Can be from 1-31 characters long, may consists of letters, digits, and special characters. Embedded blanks are not allowed. If a period is used then it must be the starting character, must not begin with a digit(number). Assembler doesn't differentiate between upper and lower case.

Operation Field

- The operation field contains a symbolic operation code(**opcode**), assembler convert opcode into machine language opcode.
- Often describe the operation's functions(**MOV**, **ADD**, **SUB** e.t.c)
- But in an assembler directive, the operation field contains a pseudo-operation code(**pseudo-op**), translated into machine, rather they simply tell the assembler to do something(**PROC** create a procedure).

Operand Field

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

NOP

no operands does nothing

INC AX

one operand; adds 1 to the contents of AX

ADD AX,VAR

two operands ; adds the memory content of VAR to the content in register AX

- In two-operand instruction, the first operand is the destination, which may be memory or register where the result is stored (some instructions may not store the result). The second operand is the source operand.
- In an assembler directive, the operand field usually contains more information about the directive.

Comment Field

- A semicolon marks at the beginning of the field tell the assembler to ignores anything typed after the semicolon.

Program Data

- Processor operates only on digital data. Hence assembler must translate all data representation into binary number. However a programmer may express data as binary, decimal, hexadecimal, and even characters.
- Numbers: All numbers are represented as
 - Binary 0110101B OR 001011b
 - Decimal 2347634D OR 435432d
 - Hexadecimal A23BDH OR 34CAEh
- Characters: Characters must be enclosed in single or double quotes For Example “Assalamoalikum” or ‘abc’. Characters are translated into their ASCII codes by the assembler.

Variables

- Variable are memory locations which has a data type and is assigned a memory address by a program.
- The data-defining pseudo-ops can be used to set aside one or more data items of the given type. Listed as follow:

Pseudo-op	Stands For
DB	define byte
DW	define word
DD	define double word
DQ	define quad word
DT	define ten bytes

- **Byte Variables:** The assembler directive that defines a byte variable is as follow:
 - name DB initial-value
 - VAR DB 3d
 - ❑ A question mark is used to set aside an uninitialized byte.
 - Radius DB ?
 - ❑ In signed interpretation decimal range from -128 to 127, or unsigned 0 to 255.
- **Word Variables:** For defining word variable the assembler directive will be as:
 - name DW initial-value
 - Area DW ?
 - ❑ In signed interpretation decimal range from -32768 to 32767, or unsigned 0 to 65535.

Arrays

- Arrays in assembly language is just a sequence of memory bytes or words. For example defining 3byte array as:
 - BT_ARRAY DB 7h,11h,13h
- The name BT_ARRAY is associated with the first of these bytes, BT_ARRAY+1,BT_ARRAY+2 with second and third simultaneously. If assembler assign 0340h to B_ARRAY, then memory would look like:

Symbol	Address	Contents
B_ARRAY	0340h	7h
B_ARRAY+1	0341h	11h
B_ARRAY+2	0342h	13h

Character Strings

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

```
LETTERS DB  
'ABC'
```

Equivalent

```
LETTERS DB 41H,  
42H,43H
```

Named Constants

- We can use a symbolic name for a constant quantity for the simplicity of the code.
- To assign a name to a constant, we use EQU(equates) pseudo-op as.
 - Name EQU Constant
 - MSG EQU 'Wellcome'
 - Assigns MSG name to 'Wellcome'.

Basic Instruction

- There are hundred of instructions in the instruction set of 8086. Here we discuss 6 of the most useful instructions for transferring data and performing arithmetic operations.
- **MOV and XCHG:**
- **MOV** used to transfer data between registers, between register and memory, or to move a number directly into a register or memory location.
 - MOV destination, source
 - e.g. MOV AX, PIE
 - this cause the AX contents to be replaced by the content of memory location PIE. The contents of PIE is unchanged.
- **XCHG** operation is used to exchange the contents of two registers, or a register and a memory location.
 - XCHG destination, source
 - e.g. XCHG AH,BL
 - this instruction will swaps the contents of AH and BL.
- **Restrictions on MOV and XCHG:** MOV or XCHG between memory locations is not allowed.

MOV

Source Operand	Destination Operands			
	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 Operands	
	General Register	Memory Location
General Register	Yes	Yes
Memory location	Yes	No

ADD and SUB instructions

- **ADD and SUB** are used to add or subtract the contents of:

- Two registers
- A register and a memory location
- Add number to a register or a memory location.

ADD destination, source

SUB destination, source

e.g. SUB AX, 55d

Legal Combination of Operands for ADD and SUB		
Source Operand	Destination Operands	
	General Register	Memory Location
General Register	Yes	Yes
Memory location	Yes	No
Constants	Yes	Yes

INC, DEC, and NEG instructions

- **INC and DEC** are used to add or subtract 1 to or from the content of:

- Registers
- A memory location
- Add number to a register or a memory location.

INC destination

DEC destination

e.g. DEC COUNTER, INC CX

- **NEG** used to negate the contents of the destination. NEG does this by replacing the contents by its two's complement.

NEG destination

e.g. NEG AX

- **Type agreement of Operands:** In two-operand instruction must be of the same type, both must be byte or both must be words.

Program Structure

- Machine language program consist of code, data, and stack. Each part occupies a memory segment.
- The same organization is reflected in assembly language program. Here the data, code and stack are structured as program segment.
- Each program segment is translated into a memory segment by the assembler.

Memory Models

- The size of code and data a program can have, is determined by specifying a memory model using the model directive.
 .MODEL memory_model
- Most frequently used memory models are SMALL, MEDIUM, COMPACT, and LARGE which are described in below table:

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 64kb
HUGE	Code in more than one segment Data in more than one segment Arrays may be larger than 64kb

Data Segments

- Data segment contains all the variables definition. Constant definitions are also made here as well, but may place elsewhere in the program since no memory allocation is involved.
- Use the directive `.DATA` to declare data segment, followed by the variable and constant declarations. As:

```
.DATA  
WORDVAR1      DW      5  
WORDVAR2      DW      7  
MASK          EQU     10110001B  
MSG           DB      "WELLCOME"
```

Stack Segment

- Stack segment is used to set aside a block of memory for stack.
Declaration syntax is:
 - `.STACK size`
 - size is optional number that specifies the stack area size in bytes.
 - `.STACK 100H`
 - set aside 100h bytes for the stack area. If no size is specified 1kb area will be aside.

Code Segments

- Code segment contains program's instructions.
- Syntax is:
 - `.CODE` `name`
 - where `name` is optional name for code segment(no need of name for SMALL programs, if used assembler will generate an error).
- Inside code segment, instructions are organized as procedures and looks like.
 - `name` `PROC`
 - `;body of the procedure`
 - `name` `ENDP`
 - where `name` is the name of the procedure. `PROC` and `ENDP` are pseudo-ops that delineate the procedure.

Now that we have studied all the program segments, now we are able to write a general form of a .SMALL model program.

```
.MODEL    SMALL
.STACK    100h
.DATA
VAR1      DB    20d
VAR2      DB    45d
VARRESULT DW    ?
.CODE
MAIN      PROC
MOV       AX,@DATA
MOV       DS,AX
MOV       AL,VAR1
ADD       AL,VAR2
MOV       VARRESULT,AX
MOV       AH,4Ch
INT       21h
MAIN      ENDP
END MAIN
```

Input and output instruction

- CPU communicates with peripherals thorough I/O register called I/O ports.
- IN and OUT access these ports directly. These operations are essential for fast I/O operations, however most application program don't use I/O because:
 - Ports address vary among computer models.
 - Its much easier to program I/O with the service routines provided by manufacturer.
- Two categories of I/O service routines.
 - BIOS routines
 - DOS routines
- To invoke a DOS or BIOS routine, INT instruction is used.
INT interrupt_number
where interrupt_number specifies a routine. e.g. **INT 16h** invoke a BIOS routine that perform keyboard input.

- INT 21H is used to invoke large number of DOS functions by placing a function number in AH and then invoking INT 21H.

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

- INT 21h functions expects input values to be in certain register and return output values in other registers which are discussed as:

Function: 1	Single Key Input
Input AH=1	
Output AL=ASCII code if character key pressed else 0	

Function: 2 Display a character or execute a control function

Input AH=2

DL=ASCII code of the display character or control character

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

```
.MODEL                SMALL
.STACK                100h
.DATA
VAR1                  DB      23d
VARRESULT             DW      ?
.CODE
MAIN                  PROC
MOV                   AX,@DATA
MOV                   DS,AX
MOV                   AH,2
MOV                   DL,'?'
INT                   21h
MOV                   DL,0DH
INT                   21h
MOV                   DL,0AH
INT                   21h
MOV                   AH,1
INT                   21h
ADD                   AL,VAR1
MOV                   AH,2
MOV                   DL,AL
INT                   21h
MOV                   AH,4Ch
INT                   21h
MAIN                  ENDP
END                   MAIN
```

Creating and running a program

- Steps in creating and running a program.
- Create a source program using any text editor and save it with extension .asm
- Use an assembler (MASM) to create machine language object file.
- Use LINK program to link one or more object to create a run file.
- Use DEBUG program to trace the program step by step.

Assembling a program through MASAM

- MASAM prints the default names enclosed in square brackets for files it can create, then waits for us to supply names for the files.
- The name NUL means that no file will be created.
- **Source Listing File (.LST)** : A line-number text file that displays assembly code and corresponding machine code side by side and gives other information about the program. Helpful for debugging purpose.
- **Cross-Reference File(.CRF)**: List all names appear in the program and the line number on which they occur. Useful in locating variables and labels in a large program.

Linking a program

- The .OBJ file created in step 2 is a machine language file, but will not be executed, because it has no proper run file format.
- It isn't known where a program will be loaded in memory, so some machine code addresses may not have been filled in.
- Some names used in the program may not have been defined in the program. E.g. It may be necessary to create several files for a large program and a procedure in one file may refer to a name defined in another file.
- The LINK program takes one or more object files, fills in any missing addresses, and combines the object files into a single executable file(.EXE) . This file can be loaded into memory and can be run.

Displaying a string

- We had used 1, and 2 with INT 21h to display a single character, we will examine another 9 function with INT 21h.

FUNCTION 9: Displaying a string

Input: DX=offset address of the string.

The string must end with a '\$' character

- To explore this function, we will create a program that prints a message "Welcome" on the screen.

Load Effective Instruction(LEA)

- Function 9 of the INT 21h expects the offset address of the character string to be in DX, to get it there we need the LEA instruction.

LEA destination, source

- where destination is a general register and source is a memory location.



```
LEA    DX,MSG
```

Program Segment Prefix(PSP)

- When program is loaded into memory DOS prefaces it with a 256 byte PSP, which contains more information about the program. DOS places its data in both DS and ES before executing the program.
- As a result DS does not contain the segment number of the data segment. So a program containing data segment must begin with following instructions:

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

- @DATA is the name defined by the .DATA directive, assembler translates it into a segment number.
- With DS initialize we can print the “Wellcome” message by placing its address in DX and executing INT 21h.



```
.MODEL    SMALL  
.STACK    100h  
.DATA  
MSG                DB        "Wellcome$"  
.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
```

```

.MODEL      SMALL
.STACK      100h
.DATA
LF          EQU      0AH
CR          EQU      0DH
MSG1        DB      "Enter a lowercase character:$"
MSG2        DB      0DH,0AH,"In upper case it is:"
CHAR        DB      ?, '$'
.CODE
MAIN PROC
MOV         AX,@DATA
MOV         DS,AX
LEA         DX,MSG1
MOV         AH,9
INT         21H
MOV         AH,1
INT         21H
SUB         AL,20h
MOV         CHAR,AL
LEA         DX,MSG2
MOV         AH,9
INT         21H
MOV         AH,4CH
INT         21H
MAIN ENDP
END MAIN

```