# Lab # 03

## Assembly Language Program Structure

An **assembly language** (or **assembler language**) is a low-level programming language for a computer, or other programmable device, in which there is a very strong correspondence between the language and the architecture's machine code instructions. Assembly language is converted into executable machine code by a utility program referred to as an assembler; the conversion process is referred to as assembly, or assembling the code.

### How Does Assembly Language Relate to Machine Language?

*Machine language* is a numeric language specifically understood by a computer's processor (the CPU). All x86 processors understand a common machine language.

*Assembly language* consists of statements written with short mnemonics such as ADD, MOV, SUB, and CALL. Assembly language has a *one-to-one* relationship with machine language: Each assembly language instruction corresponds to a single machine-language instruction.

### How Does C++ and Java Relate to Assembly Language?

High-level languages such as C++ and Java have a one-to-many relationship with assembly language and machine language. A single statement in C++ expands into multiple assembly language or machine instructions. We can show how C++ statements expand into machine code. Most people cannot read raw machine code, so we will use its closest relative, assembly language.

### Is Assembly Language Portable?

A language whose source programs can be compiled and run on a wide variety of computer systems is said to be *portable*. A C++ program, for example, should compile and run on just about any computer, unless it makes specific references to library functions that exist under a single operating system. A major feature of the Java language is that compiled programs run on nearly any computer system.

Assembly language is not portable because it is designed for a specific processor family. There are a number of different assembly languages widely used today, each based on a processor family. Some well-known processor families are Motorola 68x00, x86, SUN Sparc, Vax, and IBM-370. The instructions in assembly language may directly match the computer's architecture or they may be translated during execution by a program inside the processor known as a *microcode interpreter*.

**Assembly Language Syntax:**

**Name: operation operand (s) ;comment**

- **Name field**

Assembler translate name into memory addresses. It can be 31 characters long. The NAME field allows the program to ref of code by name.

Examples of legal names

- COUNTER1
- @character
- SUM_OF_DIGITS • .TEST

Examples of illegal names

- TWO WORDS
- 2abc
- A45.28

- **Operation field**

It contains symbolic operation code (opcode) called **"mnemonics"**(MOV, ADD, e.t.c.). The mnemonic (instruction) and operands together accomplish the tasks for which program was written. The assembler translates mnemonics into machine language opcode.

- **Operand field**

It specifies the data that are to be acted on by the operation. An instruction may have a zero, one or two operands.

Examples

- NOP
- INC AX
- ADD AX, 2

- **Comment field**

A semicolon marks the beginning of a comment. Good programming practice dictates comment on every line Examples

- MOV CX, 0 ; move 0 to CX
- MOV CX, 0 ; CX counts terms, initially 0

## ✓ Program Structure:

The machine language programs consist of code, data and stack. Each part occupies a memory segment. The same organization is reflected in an assembly language program. This time, the code, data and stack structured as program segments. Each program segment is translated into a memory segment by the assembler.

### ✓ Memory Models:

The size of code and data in a program can have 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. They are described in table below. Unless there is a lot of code or data, the appropriate model is SMALL. The .MODEL directive should come before any segment definition.

| Model | Description |
|---|---|
| SMALL | Code in one segment Data in one segment |
| MEDIUM | Code in more than one segment<br>Data in one segment |
| COMPACT | Code in one segment<br>Data in more than one segment |
| LARGE | Code in more than one segment<br>Data in more than one segment<br>No array larger than 64KB |
| HUGE | Code in more than one segment<br>Data in more than one segment<br>Arrays may be larger than 64KB |

- **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*For example, .STACK 100H sets aside 100h bytes for the stack area (a reasonable size for most applications). If size is omitted, 1KB is set aside for the stack- area.

- **Data Segment:**

A program"s data segment contains all the variable definitions. Constant definitions are often made here as well, but they may be placed 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 declaration. For example,

```
.DATA
Word1 Dw 2
Msg Db "This Is A Message"
```

**Code Segment:**

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

Here name is the optional name of the segment (there is no need for a name in a SMALL program, because the assembler will generate an error). Inside a code segment, instructions are organized as procedures. The simplest procedure definition is:

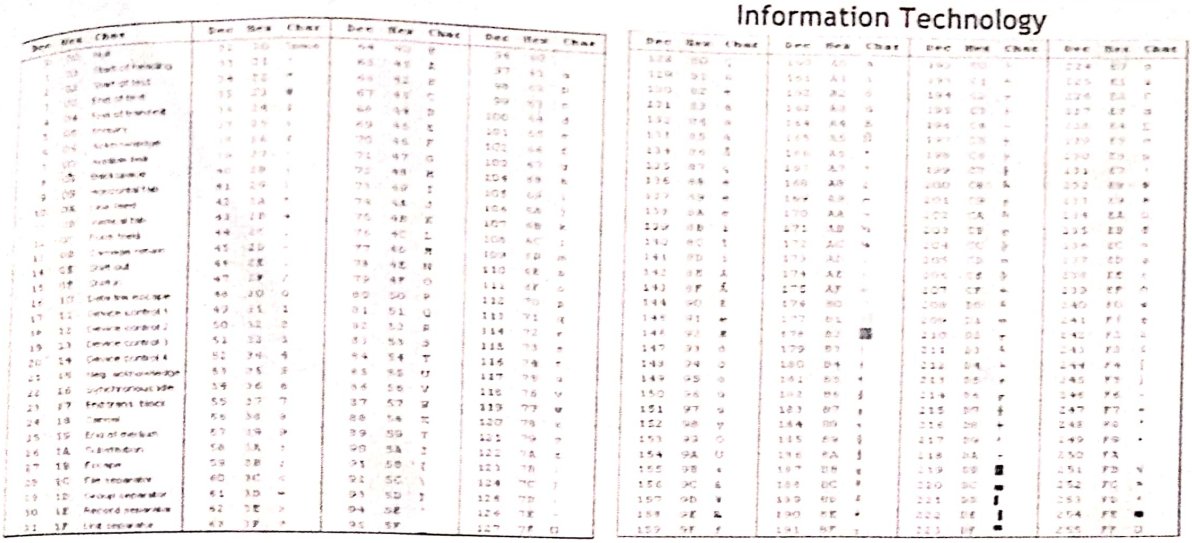
Name PROC ;body of the procedure
Name ENDP ;where name is the name of the procedure;

PROC and ENDP are pseudo-ops that delineate the procedure. Here is an example of a code segment definition:

```
.CODE
MAIN            PROC
 ; main procedure instructions
MAIN ENDP
; other procedures go here
End Main
```

```
.MODEL SMALL
.STACK 100H
.DATA
; data definitions go here
.CODE
MAIN PROC
; instructions go here
MAIN ENDP
 ;other procedures go here
END MAIN
;The last line in the program should be the END directive, followed by name of the
main procedure
```

**ASCII Character Chart** ASCII, American Standard Code for Information Interchange, is a scheme used for assigning numeric values to punctuation marks, spaces, numbers and other characters. ASCII uses 7 bits to represent characters. The values 000 0000 through 111 1111 or 00 through 7F are used giving ASCII the ability to represent 128 different characters. An extended version of ASCII assigns characters from 80 through FF.



**Disk operating system (DOS) routines**

INT 21H is used to invoke a large number of DOS function. The type of called function is specified by pulling a number in AH register. For example

AH=1 input with echo

AH=2 single-character output

AH=9 character string output

AH=8 single-key input without echo

AH=0Ah character string input

**Single-Key Input with Echo:** Output: AL= ASCII code if character key is pressed, otherwise 0.

```
MOV AH,1
INT 21 ; read character will store in AL register
```

**Single – Key Input without Echo:**

```
MOV AH,8
INT 21 ; read character will store in AL register
```

**Single-Character Output:** DL= ASCII code of character to be displayed.

```
MOV AH,2
MOV DL,"?"
INT 21h ; displaying character ?
```

**Input a String:**

```
MOV AH, 0A
INT 21
```

**Display a String:** DX= offset address of a string. String must end with a „$" character.

```
LEA DX, n1
MOV AH,9
```

INT 21

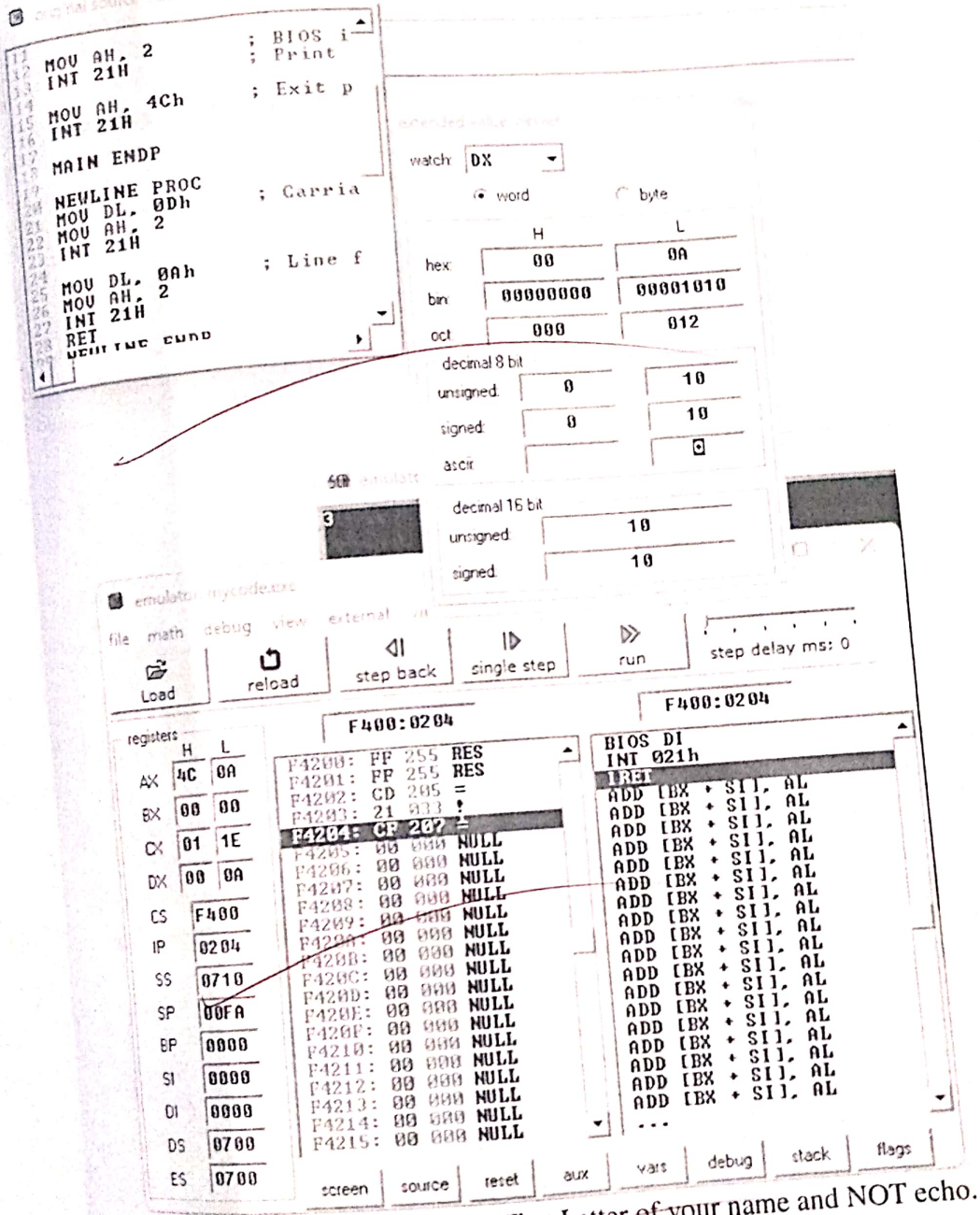
## EXERCISE:

Q1 Write down the CODE of following scenario

- Input a one-digit number from keyboard
- Save that number to DL register
- Display the number you have entered in new line

```
.MODEL SMALL
.STACK 100h
.CODE
MAIN PROC
   MOV AH, 1
   INT 21H

   MOV DL, AL

   CALL NEWLINE

   MOV AH, 2
   INT 21H

   MOV AH, 4Ch
   INT 21H

 MAIN ENDP

 NEWLINE PROC
    MOV DL, 0Dh
    MOV AH, 2
    INT 21H

    MOV DL, 0Ah
    MOV AH, 2
    INT 21H
    RET
NEWLINE ENDP

END MAIN
```

OUTPUT




Q2 Write down the CODE to input First Letter of your name and NOT echo.

```
.MODEL SMALL
.STACK 100h
.CODE
MAIN PROC
   MOV AH, 0
   INT 16H
```

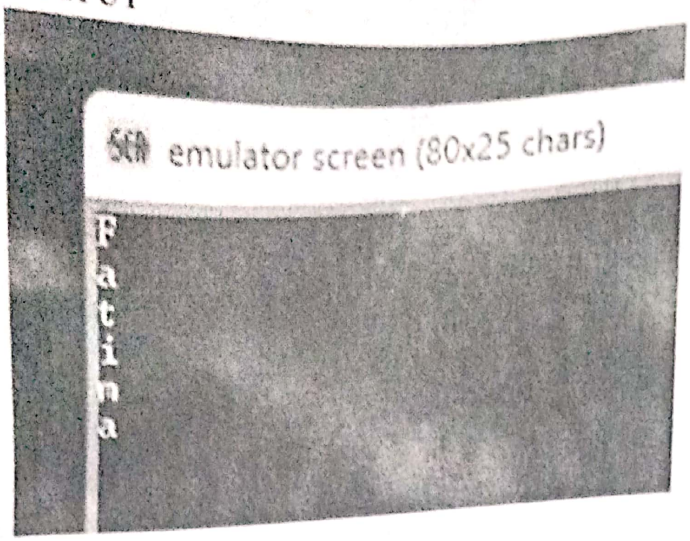
```
    MOV DL, AL

    CALL NEWLINE

    MOV AH, 2
    INT 21H

    MOV AH, 4Ch
    INT 21H

MAIN ENDP

NEWLINE PROC
    MOV DL, 0Dh
    MOV AH, 2
    INT 21H

    MOV DL, 0Ah
    MOV AH, 2
    INT 21H
    RET
NEWLINE ENDP

END MAINQ3
```

Write down the CODE to Display your name through ASCII Code (One character @ a time) each new character should display in new line.

```
.MODEL SMALL
.STACK 100h
.CODE
MAIN PROC

    MOV DL, 46h   ; 'F'
    MOV AH, 2
    INT 21H
    CALL NEWLINE

    MOV DL, 61h   ; 'a'
    MOV AH, 2
    INT 21H
    CALL NEWLINE

    MOV DL, 74h   ; 't'
```

```
    MOV AH, 2
    INT 21H
    CALL NEWLINE

    MOV DL, 69h   ; 'i'
    MOV AH, 2
    INT 21H
    CALL NEWLINE

    MOV DL, 6Dh   ; 'm'
    MOV AH, 2
    INT 21H
    CALL NEWLINE

    MOV DL, 61h   ; 'a'
    MOV AH, 2
    INT 21H
    CALL NEWLINE

    MOV AH, 4Ch
    INT 21H

MAIN ENDP

NEWLINE PROC
    MOV DL, 0Dh
    MOV AH, 2
    INT 21H

    MOV DL, 0Ah
    MOV AH, 2
    INT 21H
    RET
NEWLINE ENDP

END MAIN
```

OUTPUT

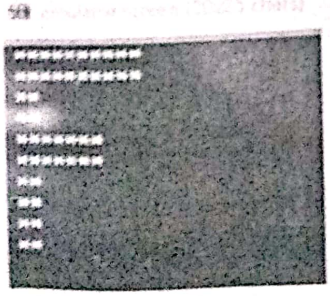



Q4 Design first letter of your name through suitable ASCII codes.

```
.MODEL SMALL
.STACK 100h

.DATA
line1  db '**********', 0Ah, 0Dh, '$'
line2  db '**********', 0Ah, 0Dh, '$'
line3  db '**       ', 0Ah, 0Dh, '$'
line4  db '**       ', 0Ah, 0Dh, '$'
line5  db '********   ', 0Ah, 0Dh, '$'
line6  db '********   ', 0Ah, 0Dh, '$'
line7  db '**       ', 0Ah, 0Dh, '$'
line8  db '**       ', 0Ah, 0Dh, '$'
line9  db '**       ', 0Ah, 0Dh, '$'
line10 db '**       ', 0Ah, 0Dh, '$'

.CODE
MAIN PROC
  MOV AX, @DATA
  MOV DS, AX

  ; Print each line
  MOV DX, OFFSET line1
  MOV AH, 9
  INT 21H

 MOV DX, OFFSET line2
 MOV AH, 9
 INT 21H

MOV DX, OFFSET line3
MOV AH, 9
```

```
INT 21H

MOV DX, OFFSET line4
MOV AH, 9
INT 21H

MOV DX, OFFSET line5
MOV AH, 9
INT 21H

MOV DX, OFFSET line6
MOV AH, 9
INT 21H

MOV DX, OFFSET line7
MOV AH, 9
INT 21H

MOV DX, OFFSET line8
MOV AH, 9
INT 21H

MOV DX, OFFSET line9
MOV AH, 9
INT 21H

MOV DX, OFFSET line10
MOV AH, 9
INT 21H

MOV AH, 4Ch
INT 21H

MAIN ENDP
END MAIN
```

OUTPUT



3/2/2(