Chapter-3
Programming with 8086 microprocessor

Internal Architecture and Features of 8086 Microprocessor
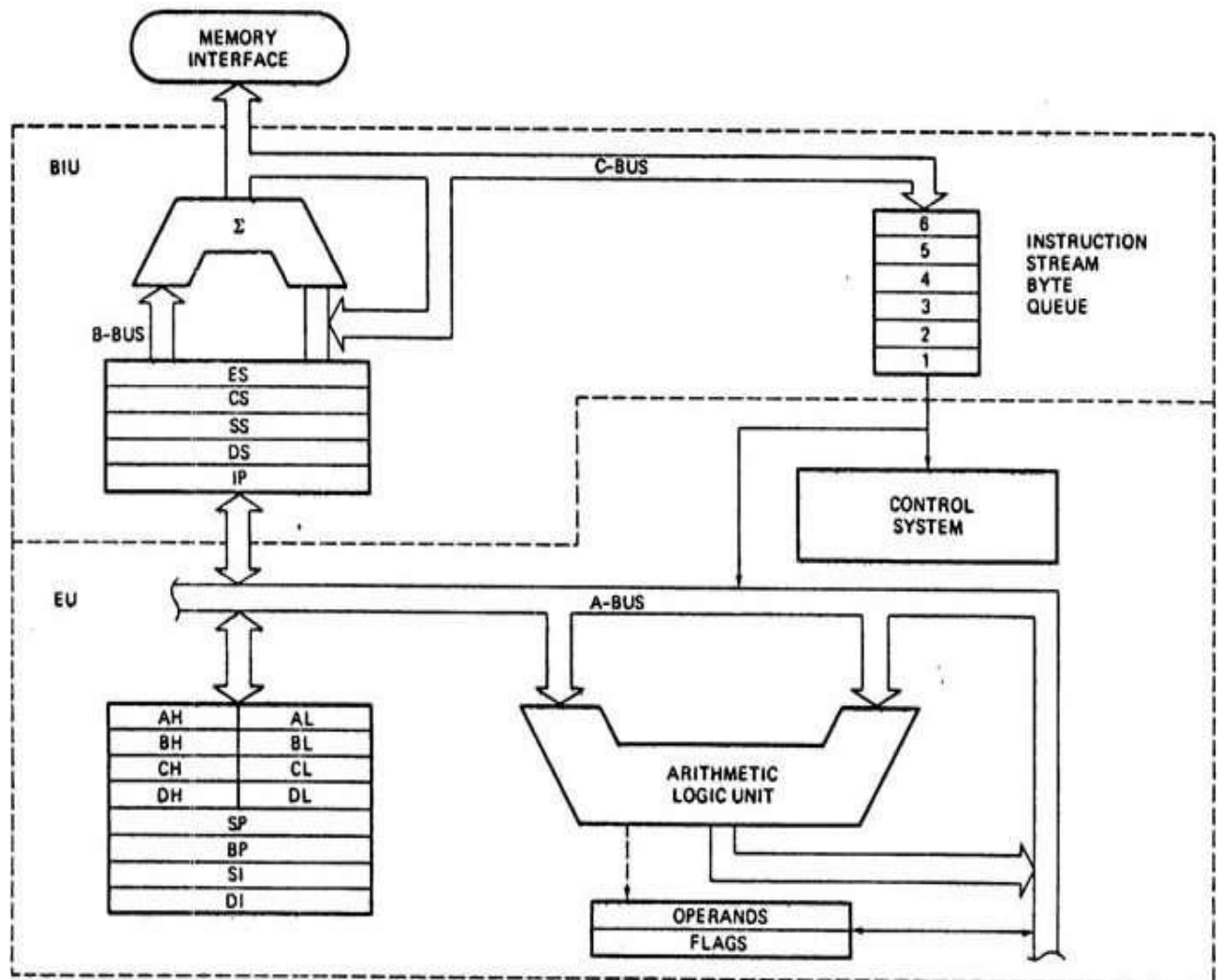

　　Fig: Internal Block Diagram of 8086 Microprocessor

Features of 8086 microprocessor
- Intel 8086 is a widely used 16 bit microprocessor.
- The 8086 can directly address 1MB of memory.
- The internal architecture of the 8086 microprocessor is an example of register based microprocessor and it uses segmented memory.
- It pre-fetches up to 6 instruction bytes from the memory and queues them in order to speed up the instruction execution.

- It has data bus of width 16 bits and address bus of width 20 bits. So it always accesses a 16 bit word to or from memory.
- The 8086 microprocessor is divided internally into two separate units which are Bus interface unit (BIU) and the execution unit (EU).
- The BIU fetches instructions, reads operands and write results.
- The EU executes instructions that have already been fetched by BIU so that instructions fetch overlaps with execution.
- A 16 bit ALU in the EU maintains the MP status and control flags, manipulates general register and instruction operands.

   Bus Interface Unit(BIU) and its Components
   The BIU sends out addresses, fetches instructions from memory reads data from memory or ports and writes data to memory or ports. So it handles all transfers of data and address on the buses for EU. It has main 2 parts instruction queue and segment registers.
- The BIU can store up to 6 bytes of instructions with FIFO (First in First Out) manner in a register set called a queue. When EU is ready for next instruction, it simply reads the instruction from the queue in the BIU. This is done in order to speed up program execution by overlapping instruction fetch with execution. This mechanism is known as pipelining.
- The BIU contains a dedicated address, which is used to produce 20 bit address. Four segment registers in the BIU are used to hold the upper 16 bits of the starting address of four memory segments that the 8086 is working at a particular time. These are code segment, data segment, stack segment and extra segment. The 8086's 1 MB memory is divided into segments up to 64KB each.

- **Code segment register and instruction pointer (IP):** The CS contains the base or start of the current code segment. The IP contains the distance or offset from this address to the next instruction byte to be fetched. Code segment address plus an offset value in the IP indicates the address of an instruction to be fetched for execution.

- **Data Segment**
   Data segment Contains the starting address of a program's data segment. Instructions use this address to locate data. This address plus an offset value in an instruction, causes a reference to a specific byte location in the data segment.

- **Stack segment (SS) and Stack Pointer (SP)**
   Stack segment Contains the starting address of a program's stack segment. This segment address plus an offset value in the stack pointer indicates the current word in the stack being addressed.

- **Extra Segment(ES)**
  It is used by some string (character data) to handle memory addressing. The string instructions always use the ES and destination index (DI) to determine 20 bit physical address.

  Execution Unit (EU)
  The EU decodes and executes the instructions. The EU contains arithmetic and logic (ALU), a control unit, and a number of registers. These features provide for execution of instructions and arithmetic and logical operations. It has nine 16 bit registers which are AX, BX, CX, DX, SP, BP, SI, DI and flag. First four can be used as 8 bit register (AH, Al, BH, BL, CH, DH, DL)

- **AX Register**
  AX register is called 16 bit accumulator and AL is called 8 bit accumulator. The I/O (IN or OUT) instructions always use the AX or AL for inputting/Outputting 16 or 8 bit data from or to I/O port.

- **BX Register**
  BX is known as the base register since it is the only general purpose register that can be used as an index to extend addressing. The BX register is similar to the 8085's H, L register. BX can also be combined with DI or SI as C base register for special addressing.

- **CX register:**
  The CX register is known as the counter register because some instructions such as SHIFT, ROTATE and LOOP use the contents of CX as a Counter.

- **DX register:**
  The DX register is known as data register. Some I/O operations require its use and multiply and divide operations that involve large values assume the use of DX and AX together as a pair. DX comprises the rightmost 16 bits of the 32-bit EDX.

- **Stack Pointer (SP) and Base Pointer (BP):**
  Both are used to access data in the stack segment. The SP is used as an offset from the current stack segment during execution of instructions. The SP's contents are automatically updated (increment/decrement) during execution of a POP and PUSH instructions.
  The BP contains the offset address in the current stack segment. This offset is used by instructions utilizing the based addressing mode.

- **Index register:**
  The two index registers SI (Source index) and DI (Destination Index) are used in indexed addressing. The instructions that process data strings use the SI and DI index register together with DS and ES respectively, in order to distinguish between the source and destination address.

- **Flag register:**
  The 8086 has nine 1 bit flags. Out of 9 six are status and three are control flags. The control bits in the flag register can be set or reset by the programmer.

				O	D	I	T	S	Z		A		P		C

D_{15} D_0

- **O- Overflow flag** This flag is set if an arithmetic overflow occurs, i.e. if the result of a signed operation is large enough to be accommodated in a destination register.
- **D-Direction Flag** This is used by string manipulation instructions. If this flag bit is '0' , the string is processed beginning from the lowest address to the higher address, i.e. auto incrementing mode otherwise the string is processed from the highest address towards the lowest address, i.e. autodecrementing mode.
- **I-Interrupt flag** If this flag is set the maskable interrupts are recognized by the CPU, otherwise they are ignored.
- **T- Trap flag** If this flag is set the processor enters the single step execution mode. In other words, a trap interrupt is generated after execution of each instruction. The processor executes the current instruction and the control is transferred to the Trap interrupt service routine.
- **S - Sign flag:** This flag is set when the result of any computation is negative. For signed computations, the sign flag equals the MSB of the result.
- **Z- Zero** This flag is set when the result of the computation is or comparison performed by the previous instruction is zero. 1 for zero result, 0 fir nonzero result
- **A- Auxiliary Carry** This is set if there is a carry from the lowest nibble, i.e. bit three during the addition or borrow for the lowest nibble i.e. bit three, during subtraction.
- **P- Parity flag** This flag is set to 1 if the lower byte of the result contains even number of 1s otherwise reset.
- **C-Carry flag** This flag is set when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

  SEGMENT AND OFFSET ADDRESS:
- Segments are special areas defined in a program for containing the code, data and stack. A segment begins on a paragraph boundary. A segment register is of 16 bits in size and contains the starting address of a segment.
- A segment begins on a paragraph boundary, which is an address divisible by decimal 16 or hex 10. Consider a DS that begins at location 038EOH. In all cases, the rightmost hex digit is zero, the computer designers decided that it would be unnecessary to store the zero the zero digit in the segment register. Thus 038E0H is stores in register as 038EH.
- The distance in bytes from the segment address to another location within the segment is expressed as an offset or displacement. Suppose the offset of 0032H for above example of data segment. Processor combines the address of the data segment with the offset as:

- **SA: OA** (segment address: offset address)

    038E: 0032 H = 038E * 10 +0032

    = 038EO + 0032

    Physical address = 03912H

Instructions in 8086

1) **Arithmetic Instructions**

a) **ADD reg$_8$ /mem$_8$, reg$_8$/mem$_8$/ Immediate$_8$**

    ADD reg$_{16}$/mem$_{16}$, reg$_{16}$/ mem$_{16}$/ Immediate$_{16}$

    E.g. ADD AH, 15   ; It adds binary number

    ADD AH, NUM1   ADD Al, [BX]

    ADD [BX], CH/CX   ADD AX,[BX]

b) **ADC**: Addition with Carry

    ADC reg/ mem, reg/mem/Immediate data

c) **SUB:** Subtract 8 bit or 16 bit binary numbers

    SUB reg/mem, reg/mem/Immediate

d) **SBB:** Subtract with borrow

    SBB reg/mem, reg/mem/Immediate

e) **MUL** : unsigned multiplication

    MUL reg$_8$/mem$_8$ (8 bit accumulator – AL)

    MUL reg$_{16}$/ mem$_{16}$ (16 bit accumulator-Ax)

    E.g. MUL R$_8$ $(multiplier)$ $\equiv$ R$_8$$\times$AL $\rightarrow$ AX (16 bit result)

    MUL R$_{16}$$(multiplier)$ $\equiv$ R$_{16}$$\times$AL $\rightarrow$ DX:AX (32 bit result)

    IMUL – signed multiplication

    Same operation as MUL but takes sign into account

f) **DIV** reg/mem

    E.g. DIV R$_8$ $\equiv$ AX/R$_8$ $\rightarrow$ (Remainder $\rightarrow$ AH) & (Q$uotient$ $\rightarrow$ AL)

    DIV R$_{16}$ $\equiv$ DX:AX/R$_{16}$$\rightarrow$ (R$emainder$ $\rightarrow$ DX)   & (Q$uotient$ $\rightarrow$ AX)

    IDIV- Signed division

    Same operation as DIV but takes sign into account.

g) **INC/DEC** (Increment/Decrement by 1)

    INC/DEC reg./mem. (8 bit or 16bit)

    E.g. INC AL   DEC BX

    INC NUM1

h) **NEG**- Negate (2's complement)

i) **ASCII-BCD** Conversion
   AAA: ASCII Adjust after addition
   AAS: ASCII Adjust after subtraction
   AAM: Adjust after multiplication
   AAD: Adjust after division
   DAA: Decimal adjust after addition
   DAS: Decimal adjust after subtraction

2) Logical/shifting/comparison instructions

a) **Logical**
   AND/OR/XOR reg/mem, reg/mem/immediate
   NOT reg/mem
   E. g. AND AL, AH
       XOR [BX], CL

b) **Rotation**
   ROL- rotate left, ROR-rotate right
   E.g.   ROL AX, 1   ; rotated by 1
           ROL AX, CL   ; if we need to rotate more than one bit
   RCL-rotate left through carry
   RCR-rotate right through carry
   E.g.   RCL AX, 1
           RCL AX, CL   ; Only CL can be used

c) **Shifting**
   SHL -logical shift left
   SHR - logical shift right
   Shifts bit in true direction and fills zero in vacant place
   E.g. SHL reg/mem, 1/CL
   arithmetic shift left
   SAR- arithmetic shift right
   Shifts bit/word in true direction, in former case place zero in vacant place and in later case place previous sign in vacant place.
   E.g. 1 011010 [1 $\Longrightarrow$ 11011010

d) **Comparison**
   CMP –compare
   CMP reg/mem, reg/mem/immediate
   E.g. CMP BH, AL

Operand1		Operand 2	CF	SF	ZF
	>		0	0	0
	=		0	0	1
	<		1	1	0

   TEST: test bits (AND operation)
   TEST reg/mem, reg/mem/immediate

3) **Data Transfer Instructions:**
MOV reg./mem , reg./mem./immediate
LDS: Load data segment register
LEA: load effective address
LES: Load extra segment register
LSS: Load stack segment register
E.g. LEA BX, ARR   = MOV BX, OFFSET ARR
LDS BX, NUM1

                    Segment address →   DS

                    Offset address →   BX

XCHG reg/mem, reg/mem
E.g. XCHG AX, BX
     XCHG AL, BL
     XCHG CL,[BX]
IN AL, DX   ; DX: Port address, AH also in AL
OUT DX, AL/AH

4) **Flag Operation**
CLC: Clear carry flag
CLD: Clear direction flag
CLI: Clear interrupt flag
STC: Set Carry flag
STD: Set direction flag
STI: Set Interrupt flag
CMC: Complement Carry flag
LAHF: Load AH from flags (lower byte)
SAHF: Store AH to flags
PUSHF: Push flags into stack
POPF: Pop flags off stack

5) STACK Operations
PUSH reg$_{16}$
POP reg$_{16}$

6) Looping instruction (CX is automatically used as a counter)
LOOP: loop until complete
LOOPE: Loop while equal
LOOPZ: loop while zero
LOOPNE: loop while not equal
LOOPNZ: loop while not zero

7) Branching instruction
a) Conditional
JA: Jump if Above
JAE: Jump if above/equal
JB: Jump if below
JBE: Jump if below/equal
JC: Jump if carry
JNC: Jump if no carry
JE: Jump if equal
JNE: Jump if no equal
JZ: Jump if zero
JNZ: Jump if no zero
JG: Jump if greater
JNG: Jump if no greater
JL: Jump if less
JNL: Jump if no less
JO: jump if overflow
JS: Jump if sign
JNS: Jump if no sign
JP: jump if plus
JPE: Jump if parity even
JNP: Jump if no parity
JPO: Jump if parity odd

b) Unconditional
CALL: call a procedure   RET: Return
INT: Interrupt   IRET: interrupt return
JMP: Unconditional Jump   RETN/RETF: Return near/Far

8) Type conversion
CBW: Convert byte to word
CWD: Convert word to double word

AX $\rightarrow$ DX: AX

9) **String instructions**
a) MOVS/ MOVSB/MOVSW   ; Move string
  DS: SI source
  DS: DI destination
  CX: String length
b) CMPS/ CMPSB/CMPW   ; Compare string
c) LODS /LODSB/LODW   ; Load string
d) REP   ; Repeat string

Operators in 8086
- Operator can be applied in the operand which uses the immediate data/address.
- Being active during assembling and no machine language code is generated.
- Different types of operators are:
1) **Arithmetic**: + , - , * , /
2) **Logical** : AND, OR, XOR, NOT
3) **SHL and SHR**: Shift during assembly
4) **[ ]**: index
5) **HIGH**: returns higher byte of an expression
6) **LOW**: returns lower byte of an expression.
   E.g. NUM EQU 1374 H
   MOV AL HIGH Num   ; ([AL] $\leftarrow$ 13)
7) **OFFSET**: returns offset address of a variable
8) **SEG**: returns segment address of a variable
9) **PTR**: used with type specifications
       BYTE, WORD, RWORD, DWORD, QWORD
       E.g. INC BYTE PTR [BX]
10) **Segment override**
   MOV AH, ES: [BX]
11) **LENGTH**: returns the size of the referred variable
12) **SIZE:** returns length times type
       E.g.:   BYTE VAR DB?
                   WTABLE DW **10** DUP (?)
                   MOV AX, TYPE BYTEVAR   ; AX = 0001H
                   MOV AX, TYPE WTABLE   ; AX = 0002H
                   MOV CX, LENGTH WTABLE   ; CX = 000AH
                   MOV CX, SIZE WTABLE   ; CX = 0014H

Coding in Assembly language:
Assembly language programming language has taken its place in between the machine language (low level) and the high level language.
- High level language's one statement may generate many machine instructions.
- Low level language consists of either binary or hexadecimal operation. One symbolic statement generates one machine level instructions.

Advantage of ALP
- They generate small and compact execution module.
- They have more control over hardware.
- They generate executable module and run faster.

Disadvantages of ALP:
- Machine dependent.
- Lengthy code
- Error prone (likely to generate errors).

Assembly language features:
The main features of ALP are program comments, reserved words, identifies, statements and directives which provide the basic rules and framework for the language.

Program comments:
- The use of comments throughout a program can improve its clarity.
- It starts with semicolon (;) and terminates with a new line.
- E.g. ADD AX, BX   ; Adds AX & BX

Reserved words:
- Certain names in assembly language are reserved for their own purpose to be used only under special conditions and includes
- Instructions : Such as MOV and ADD (operations to execute)
- Directives: Such as END, SEGMENT (information to assembler)
- Operators: Such as FAR, SIZE
- Predefined symbols: such as @DATA, @ MODEL

Identifiers:
- An identifier (or symbol) is a name that applies to an item in the program that expects to reference.
- Two types of identifiers are Name and Label.
- Name refers to the address of a data item such as NUM1 DB 5, COUNT DB 0
- Label refers to the address of an instruction.
- E. g: MAIN PROC FAR
- L1: ADD BL, 73

Statements:
- ALP consists of a set of statements with two types
- Instructions, e. g. MOV, ADD
- Directives, e. g. define a data item

	Identifiers	operation	operand	comment
Directive:	COUNT	DB	1	; initialize count
Instruction:	L30:	MOV	AX, 0	; assign AX with 0

Directives:
The directives are the number of statements that enables us to control the way in which the source program assembles and lists. These statements called directives act only during the assembly of program and generate no machine-executable code. The different types of directives are:

1) **The page and title listing directives:**
   The page and title directives help to control the format of a listing of an assembled program. This is their only purpose and they have no effect on subsequent execution of the program.
   The page directive defines the maximum number of lines to list as a page and the maximum number of characters as a line.
   PAGE [Length] [Width]
   Default: Page [50][80]
   TITLE gives title and place the title on second line of each page of the program.
   TITLE text [comment]

2) **SEGMENT directive**
   It gives the start of a segment for stack, data and code.
   Seg-name   Segment [align][combine]['class']
   Seg-name   ENDS
   - Segment name must be present, must be unique and must follow assembly language naming conventions.
   - An ENDS statement indicates the end of the segment.
   - Align option indicates the boundary on which the segment is to begin; PARA is used to align the segment on paragraph boundary.
   - Combine option indicates whether to combine the segment with other segments when they are linked after assembly. STACK, COMMON, PUBLIC, etc are combine types.
   - Class option is used to group related segments when linking. The class code for code segment, stack for stack segment and data for data segment.

3) **PROC Directives**
   The code segment contains the executable code for a program, which consists of one or more procedures, defined initially with the PROC directives and ended with the ENDP directive.
   PROC - name   PROC [FAR/NEAR]

……………..
……………..
……………..
PROC - name   ENDP
- FAR is used for the first executing procedure and rest procedures call will be NEAR.
- Procedure should be within segment.

4) END Directive
- An END directive ends the entire program and appears as the last statement.
- ENDS directive ends a segment and ENDP directive ends a procedure. END PROC-Name

5) ASSUME Directive
- An .EXE program uses the SS register to address the stack, DS to address the data segment and CS to address the code segment.
- Used in conventional full segment directives only.
- Assume directive is used to tell the assembler the purpose of each segment in the program.
- Assume SS: Stack name, DS: Data Segname CS: codesegname

6) Processor directive
- Most assemblers assume that the source program is to run on a basic 8086 level computer.
- Processor directive is used to notify the assembler that the instructions or features introduced by the other processors are used in the program.
  E.g. **.386** - program for 386 protected mode.

7) Dn Directive (Defining data types)
Assembly language has directives to define data syntax [name] Dn expression
The Dn directive can be any one of the following:

DB	Define byte	1 byte
DW	Define word	2 bytes
DD	Define double	4 bytes
DF	defined farword	6 bytes
DQ	Define quadword	8 bytes
DT	Define 10 bytes	10 bytes

VAL1 DB **25**
ARR DB 21, 23, 27, 53
MOV AL, ARR [2] or
MOV AL, ARR + 2 ; Moves 27 to AL register

8) The EQU directive
- It can be used to assign a name to constants.
- E.g. **FACTOR** EQU **12**

- MOV BX, FACTOR   ; MOV BX, 12
- It is short form of equivalent.
- Do not generate any data storage; instead the assembler uses the defined value to substitute in.

9) DUP Directive
- It can be used to initialize several locations to zero.
   e. g. SUM DW 4 DUP(0)
- Reserves four words starting at the offset sum in DS and initializes them to Zero.
- Also used to reserve several locations that need not be initialized. In this case (?) is used with DUP directives.
      E. g. **PRICE** DB 100 DUP(?)
- Reserves 100 bytes of uninitialized data space to an offset PRICE.

Program written in Conventional full segment directive

```
Page 60,132
TITLE SUM program to add two numbers
;---------------------------------------------------
STACK SEGMENT PARA STACK 'Stack'
DW 32 DUP(0)
STACK ENDS
;---------------------------------------------------
DATA SEG SEGMENT PARA 'Data'
NUM1 DW 3291
NUM 2 DW 582
SUM DW?
DATA SEG ENDS
;---------------------------------------------------
CODE SEG SEGMENT PARA 'Code'
MAIN PROC FAR
    ASSUME SS: STACK, DS:DATASEG, CS:CODESEG
    MOV AX, @DATA
    MOV DS, AX
    MOV AX, NUM1
    ADD AX, NUM2
    MOV AX, 4C00H
    INT 21H
MAIN ENDP
CODESEG ENDS
END MAIN
```

Description for conventional program:

-   STACK contains one entry, DW (define word), that defines 32 words initialized to zero, an adequate size for small programs.
-   DATASEG defines 3 words NUM1, NUM2 initialized with 3291 and 582 and sum uninitialized.
-   CODESEG contains the executable instructions for the program, PROC and ASSUME generate no executable code.
-   The ASSUME directive tells the assembler to perform these tasks.
-   Assign STACK to SS register so that the processor uses the address in SS for addressing STACK.
-   Assign DATASEG to DS register so that the processor uses the address in DS for addressing DATASEG.
-   Assign CODESEG to the CS register so that the processor uses the address in CS for addressing CODESEG.
    When the loading a program for disk into memory for execution, the program loader sets the correct segment addresses in SS and CS.

Program written using simplified segment directives:
    .Model memory model
Memory model can be
TINY, SMALL, MEDIUM, COMPACT, LARGE, HUGE or FLAT
TINY for .com program
FLAT for program up to 4 GB
-   **Assume** is automatically generated
        .STACK [size in bytes]
        Creates stack segment
        .DATA: start of data segment
        .CODE: start of code segment
-   DS register can be initialized as
MOV AX, @DATA
MOV DS, AX

**ALP written in simplified segment directives:**
    Page 60, 132
    TITLE Sum program to add two numbers.
    .MODEL SMALL
    .STACK 64
    .DATA
        NUM1 DW 3241
        NUM 2 DW 572

```
        SUM DW   ?
        .CODE
        MAIN PROC FAR
            MOV AX, @ DATA          ; set address of data segment in DS
            MOV DS, AX
            MOV AX, NUM1
            ADD AX, NUM2
            MOV SUM, AX
            MOV AX, 4C00H           ; End processing
            INT 21H
        MAIN ENDP                ; End of procedure
        END MAIN                    ; End of program
```

DOS Debug(TASM)
1) Save the code text in **.ASM** format and save it to the same folder where masm and link files are stored.
2) Open dos mode and reach within that folder.
3) \> tasm filename.asm   ⟹ makes.obj
4) \> tlink filename   ⟹ makes .exe
5) \> filename.exe   ⟹ run the code
6) \> td filename.exe   ⟹ debug the code [use F7 and F8]

Assembling, Linking and Executing
1) Assembling:
- Assembling converts source program into object program if syntactically correct and generates an intermediate **.obj** file or module.
- It calculates the offset address for every data item in data segment and every instruction in code segment.
- A header is created which contains the incomplete address in front of the generated **obj** module during the assembling.
- Assembler complains about the syntax error if any and does not generate the object module.
- Assembler creates **.obj .lst** and **.crf** files and last two are optional files that can be created at run time.
- For short programs, assembling can be done manually where the programmer translates each mnemonic into the machine language using lookup table.
- Assembler reads each assembly instruction of a program as ASCII character and translates them into respective machine code.

Assembler Types:
There are two types of assemblers:
a) One pass assembler:
- This assembler scans the assembly language program once and converts to object code at the same time.

- This assembler has the program of defining forward references only.
- The jump instruction uses an address that appears later in the program during scan, for that case the programmer defines such addresses after the program is assembled.

b) Two pass assembler
- This type of assembler scans the assembly language twice.
- First pass generates symbol table of names and labels used in the program and calculates their relative address.
- This table can be seen at the end of the list file and here user need not define anything.
- Second pass uses the table constructed in first pass and completes the object code creation.
- This assembler is more efficient and easier than earlier.
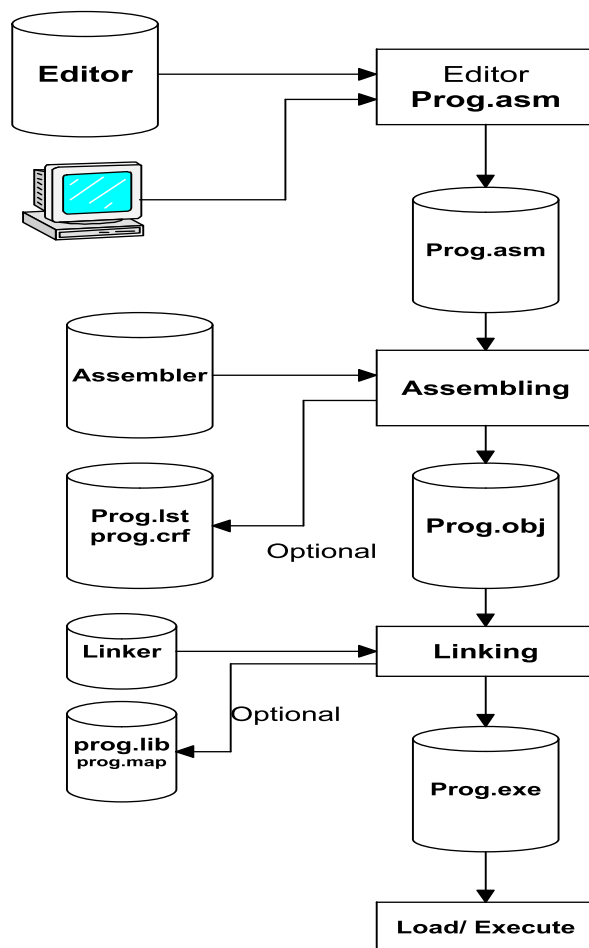
Fig: Steps in assembling, linking & Executing

2) Linking:
- This involves the converting of **.OBJ** module into **.EXE**(executable) module i.e. executable machine code.
- It completes the address left by the assembler.

- It combines separately assembled object files.
- Linking creates **.EXE**, **.LIB**, **.MAP** files among which last two are optional files.

3) Loading and Executing:
- It Loads the program in memory for execution.
- It resolves remaining address.
- This process creates the program segment prefix (PSP) before loading.
- It executes to generate the result.

Sample program <u>assembling</u> object Program   <u>linking</u>   executable program

Writing .COM programs:
- It fits for memory resident programs.
- Code size limited to 64K.
- **.com** combines PSP, CS, DS in the same segment
- SP is kept at the end of the segment (FFFF), if 64k is not enough, DOS Places stack at the end of the memory.
- The advantage of **.com** program is that they are smaller than **.exe** program.
- A program written as **.com** requires ORG 100H immediately following the code segment's SEGMENT statement. The statement sets the offset address to the beginning of execution following the PSP.

```
        .MODEL TINY
        .CODE
        ORG 100H                ; start at end of PSP
        BEGIN:JMP MAIN          ;Jump Past data
            VAL1 DW 5491
            VAL2 DW 372
            SUM DW ?
        MAIN:  PROC NEAR
            MOV Ax, VALL
            ADD AX, VAL2
            MOV SUM, AX
            MOV AX, 4C00H
            INT 21H
        MAIN ENDP
        END BEGIN
```

Macro Assembler:
- A macro is an instruction sequence that appears repeatedly in a program assigned with a specific name.
- The macro assembler replaces a macro name with the appropriate instruction sequence each time it encounters a macro name.

- When same instruction sequence is to be executed repeatedly, macro assemblers allow the macro name to be typed instead of all instructions provided the macro is defined.
- Macro are useful for the following purposes:
  o To simplify and reduce the amount of repetitive coding.
  o To reduce errors caused by repetitive coding.
  o To make an assembly language program more readable.
  o Macro executes faster because there is no need to call and return.
  o Basic format of macro definition:

  ```
  Macro name          MACRO [Parameter list]      ; Define macro
                      ……………………….
                      ……………………….
                      [Instructions]              ; Macro body
                      ……………………….
                      ……………………….
                      ENDM                        ; End of macro
  ```

  E.g.   **Addition**   MACRO
  ```
                      IN AX, PORT
                      ADD AX, BX
                      OUT PORT, AX
                      ENDM
  ```

Passing argument to MACRO:
- To make a macro more flexible, we can define parameters as dummy argument

```
Addition MACRO VALL1, VAL2
        MOV   AX, VAL1
        ADD AX, VAL2
        MOV SUM, AX
        ENDM

.MODEL SMALL
.STACK 64
.DATA
        VAL1   DW     3241
        VAL2   DW     571
        SUM    DW     ?
.CODE
MAIN PROC FAR
        MOV AX, @ DATA
        MOV DS, AX
```

　　　　　　　　Addition VAL1, VAL 2
　　　　　　　　MOV AX, 4C00H
　　　　　　　　INT 21H
　　　　　　MAIN ENDP
　　　　　　END MAIN

Addressing modes in 8086:
Addressing modes describe types of operands and the way in which they are accessed for executing an instruction. An operand address provides source of data for an instruction to process an instruction to process. An instruction may have from zero to two operands. For two operands first is destination and second is source operand. The basic modes of addressing are register, immediate and memory which are described below.

1) **Register Addressing:**
   For this mode, a register may contain source operand, destination operand or both.
   E.g.　　MOV AH, BL
   　　　　MOV DX, CX

2) **Immediate Addressing**
   In this type of addressing, immediate data is a part of instruction, and appears in the form of successive byte or bytes. This mode contains a constant value or an expression.
   E.g.　　MOV AH, 35H
   　　　　MOV BX, 7A25H

3) **Direct memory addressing:**
   In this type of addressing mode, a 16-bit memory address (offset) is directly specified in the instruction as a part of it. One of the operand is the direct memory and other operand is the register.
   E.g. ADD AX, [5000H]
   Note: Here data resides in a memory location in the data segment, whose effective address may be computed using 5000H as the Offset address and content of DS as segment address. The effective address, here, is **10H*DS + 5000H**.

4) **Direct offset addressing**
   In this addressing, a variation of direct addressing uses arithmetic operators to modify an address.
   E.g.　　 ARR DB 15, 17, 18, 21
   　　　　 MOV AL, ARR [2]　　 ; MOV AL, 18
   　　　　 ADD BH, ARR+3　　　 ; ADD BH, 21

5) **Indirect memory addressing:**
   Indirect addressing takes advantage of computer's capability for segment: offset addressing. The registers used for this purpose are base register (BX and BP) and index register (DI and SI)
   E.g.　　MOV [BX], AL

ADD CX, [SI]

6) **Base displacement addressing:**

This addressing mode also uses base registers (BX and BP) and index register (SI and DI), but combined with a displacement (a number or offset value) to form an effective address.

E.g.   MOV BX, OFFSET ARR

≡ LEA BX, ARR

MOV AL, [BX +2]

ADD TBL [BX], CL

TBL [BX] ⟹ [BX + TBL] e.g. [BX + 4]

7) **Base index addressing:**

This addressing mode combines a base registers (BX or BP) with an index register (SI or DI) to form an effective address.

E.g.   MOV AX, [BX +SI]

ADD [BX+DI], CL

8) **Base index with displacement addressing**

This addressing mode, a variation on base- index combines a base register, an index register, and a displacement to form an effective address.

E.g.   MOV AL, [Bx+SI+2]

ADD TBL [BX +SI], CH

9) **String addressing:**

This mode uses index registers, where SI is used to point to the first byte or word of the source string and DI is used to point to the first byte or word of the destination string, when string instruction is executed. The SI or DI is automatically incremented or decremented to point to the next byte or word depending on the direction flag (DF).

E.g. MOVS, MOVSB, MOVSW

Examples:

```
TITLE Program to add ten numbers
        .MODEL SMALL
        .STACK 64
        .DATA
ARR DB 73, 91, 12, 15, 79, 94, 55, 89
SUM DW ?
        .CODE
MAIN PROC FAR
    MOV AX, @DATA
    MOV DS, AX
```

```
                MOV CX, 10
                MOV AX, 0
                LEA BX, ARR
        L2: ADD Al, [BX]
                JNC L1
                INC AH
        L1: INC BX
                LOOP L2
                MOV SUM, AX
                MOV AX, 4C00H
                INT 21H
        MAIN ENDP
        END MAIN
```

DOS FUNCTIONS AND INTERRUPTS
(KEYBOARD AND VIDEO PROCESSING)

The Intel CPU recognizes two types of interrupts namely hardware interrupt when a peripheral devices needs attention from the CPU and software interrupt that is call to a subroutine located in the operating system. The common software interrupts used here are INT 10H for video services and INT 21H for DOS services.

INT 21H:

It is called the DOS function call for keyboard operations follow the function number. The service functions are listed below:

 # 00H- It terminates the current program.
- Generally not used, function 4CH is used instead.

 # 01H- Read a character with echo
- Wait for a character if buffer is empty
- Character read is returned in AL in ASCII value

 # 02H- Display single character
- Sends the characters in DL to display
- MOV AH, 02H
- MOV DL, 'A'   ; move Dl, 65
- INT 21H

 # 03H and 04H – Auxiliary input/output
- INT 14H is preferred.

 # 05H – Printer service
- Sends the character in DL to printer

 # 06H- Direct keyboard and display
- Displays the character in DL.

 # 07H- waits for a character from standard input

- does not echo.

08H- keyboard input without echo
- Same as function 01H but not echoed.

09H- string display
- Displays string until '$' is reached.
- DX should have the address of the string to be displayed.

0AH – Read string

OBH- Check keyboard status
- Returns FF in AL if input character is available in keyboard buffer.
- Returns 00 if not.

0CH- Clear keyboard buffer and invoke input functions such as 01, 06, 07, 08 or 0A.
- AL will contain the input function.

INT 21H Detailed for Useful Functions

 # 01H

        MOV, AH 01H; request keyboard input INT 21H
- Returns character in AL. IF AL= nonzero value, operation echoes on the screen. If Al= zero means that user has pressed an extended function key such as F1 OR home.

 # 02H

        MOV AH, 02H; request display character
        MOV DL, CHAR; character to display
        INT 21H

- Display character in D2 at current cursor position. The tab, carriage return and line feed characters act normally and the operation automatically advances the cursor.

 # 09H

        MOV Ah, 09H; request display
        LEA DX, CUST_MSG; local address of prompt
        INNT 21H
        CUST_MSG DB "Hello world", '$'
- Displays string in the data area, immediately followed by a dollar sign ($ or 24H), which uses to end the display.

 # OAH

        MOV AH, 0AH   ; request keyboard input
        LEA DX, PARA_ LIST   ; load address of parameter list
        INT 21H

        Parameter list for keyboard input area :
        PARA_LIST LABEL BYTE; start of parameter list
        MAX_LEN DB 20; max. no. of input character

ACT _ LEN DB ? ; actual no of input characters
KB-DATA DB 20 DUP ('); characters entered from keyboard

- LABEL directive tells the assembler to align on a byte boundary and gives location the name PARA _LIST.
- PARA_LIST & MAX_LEN refer same memory location, MAX_LEN defines the maximum no of defined characters.
- ACT_LEN provides a space for the operation to insert the actual no of characters entered.
- KB_DATA   reserves spaces (here 20) for the characters.

Example:

```
            TITLE to display a string
            .MODEL SMALL
            .STACK 64
            .DATA
                    STR DB 'programming is fun', '$'
            .CODE
            MAIN PROC FAR
                    MOV AX, @DATA
                    MOV DS, AX
                    MOV AH, 09H            ;display string
                    LEA DX, STR
                    INT 21H
                    MOV AX, 4C00H
                    INT 21H
            MAIN ENDP
            END MAIN
```

INT 10H
It is called video display control. It controls the screen format, color, text style, making windows, scrolling etc. The control functions are:

 # 00H – set video mode
```
            MOV AH, 00H            ; set mode
            MOV AL, 03H            ; standard color text
            INT 10H               ; call interrupt service
```

 # 01H- set cursor size
```
            MOV AH, 01H
            MOV CH, 00H            ; Start scan line
            MOV CL, 14H            ; End scan line
            INT 10H               ; (Default size 13:14)
```

 # 02H – Set cursor position:
```
        MOV AH, 02H
        MOV BH, 00H          ; page no
        MOV DH, 12H          ; row/y (12)
        MOV DL, 30H          ; column/x (30)
        INT 10H
```

 # 03H – return cursor status
```
        MOV AH, 03H
        MOV BH, 00H;
        INT 10H
        Returns: CH- starting scan line, CL-end scan line, DH- row, DL-column
```

 # 04H- light pen function

 # 05H- select active page
```
        MOV AH, 05H
        MOV AL,page-no.      ; page number
        INT 10H
```

 # 06H- scroll up screen
```
MOV AX, 060FH        ; request scroll up one line (text)
MOV BH, 61H          ; brown background, blue foreground
MOV CX, 0000H        ; from 00:00 through
MOV DX, 184F H       ; to 24:79 (full screen)
INT 10H
AL= number of rows (00 for full screen)
BH= Attribute or pixel value
CX= starting row: column
DX= ending row: column
```

 # 07H-Scroll down screen
Same as 06H except for down scroll

 # 08H (Read character and Attribute at cursor)
```
MOV AH, 08H
MOV BH, 00H          ; page number 0(normal)
INT 10H
AL= character
BH= Attribute
```

 # 09H -display character and attribute at cursor
```
MOV AH, 09H
MOV AL, 01H          ; ASCII for happy face display
```

```
MOV BH, 00H            ; page number
MOV BL, 16H            ; Blue background, brown foreground
MOV CX, 60             ; No of repeated character
INT 10H
```

 # 0AH-display character at cursor

```
MOV AH, 0AH
MOV Al, Char
MOV BH, page _no
MOV BL, value
MOV CX, repetition
INT 10H
```

 # 0BH- Set color palette
- ✓ Sets the color palette in graphics mode
- ✓ Value in BH (00 or 01) determines purpose of BL
- ✓ BH= 00H, select background color, BL contains 00 to 0FH (16 colors)
- ✓ BH = 01H , select palette, Bl, contains palette

```
MOV AH, 0BH
MOV AH, 0BH
MOV BH, 00H; background            MOV BH, 01H  ; select palette
MOV BL, 04H; red                   MOV BL, 00H   ; black
INT 21H                            INT 21H
```

#0CH- write pixel Dot
- Display a selected color

```
AL=color of the pixel      CX= column
BH=page number             DX= row

MOV AH, 0CH
MOV Al, 03
MOV BH,0
MOV CX, 200
MOV DX, 50
INT 10H
It sets pixel at column 200, row 50
```

#0DH- Read pixel dot
- Reads a dot to determine its color value which returns in AL

```
MOV AH, 0DH
MOV BH, 0     ; page no
MOV CX, 80    ; column
MOV DX, 110  ; row
INT 10H
```

#OEH- Display in teletype mode
- Use the monitor as a terminal for simple display
  MOV AH, 0EH
  MOV AL, char
  MOV BL, color; foreground color
  INT 10H

#OF H- Get current video mode
Returns values from the BIOS video .
AL= current video mode   **MOV AH, 0FH**
AH= no of screen columns   **INT 10H**
BH = active video page

```
TITLE To Convert letters into lower case
 .MODEL SMALL
 .STACK 99H
 .CODE
 MAIN PROC
          MOV AX, @ DATA
          MOV DS, AX
          MOV SI, OFFSER STR
     M:   MOV DL, [SI]
          MOV CL, DL
          CMP DL, ' $'
          JE N
          CMP DL, 60H
          JL L
     K:   MOV DL, CL
          MOV AH, 02H
          INT 21H
          INC SI
          JMP M
     L:   MOV DL, CL
          ADD DL, 20H
          MOV AH, 02H
          INT 21H
          INC SI
          JMP M
     N:   MOV AX, 4C00H
          INT 21H
          MAIN ENDP
     .DATA
          STR DB 'I am MR Rahul ", '$'
```

```
                    END MAIN

        TITLE to reverse the string
            .MODEL SMALL
            .STACK 100H
            .DATA
                STR1 DB " My name is Rahul" , '$'
                STR2 db 50 dup ('$')
             .CODE
        MAIN PROC FAR
                MOV BL,00H
                MOV AX, @ DATA
                MOV DS, AX
                MOV SI, OFFSER STR1
                MOV DI, OFFSET STR2
        L2:     MOV DL, [SI]
                CMP DI, '$'
                JE L1
                INC SI
                INC BL
                JMP L2
        L1:     MOV CL, BL
                MOV CH, 00H
                DEC SI
        L3:     MOV AL, [SI]
                MOV [DI], AL
                DEC SI
                INC DI
                LOOP L3
                MOV AH,09H
                MOV DX, OFFSET STR2
                INT 21H
                MOV AX, 4C00H
                INT 21H
        MAIN ENDP
        END MAIN

        TITLE to input characters until 'q' and display
            .MODEL SMALL
            .STACK 100H
            .DATA
                STR db 50 DUP ('$')
             .CODE
        MAIN PROC FAR
```

```
             MOV AX, @ DATA
             MOV DS, AX
             MOV SI, OFFSET STR
L2:     MOV AH, 01H
             INT 21H
             CMP AL, 'q'
             JE L1
             MOV [SI] , AL
             INC SI
             JMP L2
L1:     MOV AH, 09H
             MOV DX, OFFSET STR
             INT 21H
             MOV AX, 4C00H
             INT 21H
MAIN ENDP
END MAIN
```

Calling procedure/subroutine

```
Procname PROC FAR
……………………………..
……………………………..
Procname ENDP
```

Here the code segment consists only one procedure. The FAR operand in this case informs the assembler and linker that the defined procedure name is the entry point for program execution, whereas the ENDP directive defines the end of the procedure. A code segment however, may contain any number of procedures, each distinguished by its own PROC and ENDP directives.

A called procedure is a section of code that performs a clearly defined task known as subroutine which provides following benefits.

- Reduces the amount of code because a common procedure can be called from any number of places in the code segment.
- Encourage better program organization.
- Facilitates debugging of a program because defects can be more clearly isolated.
- Helps in the ongoing maintenance of programs because procedures are readily identified for modification.

A CALL to a procedure within the same code segment is NEAR CALL<. A FAR CALL calls a procedure labeled FAR, possibly in another code segment.

```
        DISPLAY PROC NEAR
                MOV AH, 09H
                MOV DX, OFFSET STR
                INT 21H
                RET
        DISPLAY ENDP
```

- ✓ To display number contained in [BX]

```
        DISPLAY PROC NEAR
                MOV DI, [BX]
                ADD DI, 30
                MOV AH, 02H
                INT 21H
                RET
        DISPLAY ENDP
```

INT 10H Video service:

<Video –modes>

Text mode	Row×column	Color	No.of Pages	Resolution	colors
00	25×40	Color	8	360×400	16 colors
01	25×40	Color	8	360×400	16 colors
02	25×80	Color	4	720×400	16 colors
03(by default)	25×80	color	4	720×400	16 colors
07	25×80	Mono-hrome	0	720×400	16 colors

Graphic mode	Color	Pages	Resolution	No of colors
04	Color	8	320×200	4
05	Color	8	320×200	4
06	Color	8	640×200	2
0D	Color	8	320×200	16
0E	Color	4	640×200	16
0F	Mono chrome	2	640×350	1
10	Color	2	640×350	16
11	Color	1	640×480	2
12	Color	1	640×480	16
13	Color	1	320×200	256

Attribute

	Background	Foreground
Attribute:	BL R G B	I R G B
Bit number:	7 6 5 4	3 2 1 0

I – Intensity, BL - Blink

Color	Hex Value
Black	0
Blue	1
Green	2
Cyan	3
Red	4
Magnet	5
Brown	6
White	7
Gray	8
Light Blue	9
Light Green	A
Light cyan	B
Light red	C
Light magenta	D
Yellow	E
Bright white	F

TITLE sorting the numbers – descending order

```
        DOSSEG
                .MODEL SMALL
                .STACK 100H
                .CODE
        MAIN PROC FAR
                        MOV AX, @ DATA
                        MOV DS, AX
                        MOV DX, 4H
            DOPASS:   MOV CX, 4H
                        MOV SI, 00H
            CHECK:     MOV AL, ARR [SI]
                        CMP ARP [SI+1] , AL
                        JC NOSWAP
                        MOV BL, ARR [SI + 1]
                        MOV ARR[SI +1] , AL
                        MOV ARR [SI], BL
            NOSWAP:   INC SI
                        LOOP CHECK
                        DEC DX
                        JNZ DOPASS
                        MOV AX, 4C00H
                        INT 21H
                        MAIN ENDP
            .DATA
                        ARR DB 8,2,9,4,7
            END    MAIN
```

Note: Display if numbers are with 1 digit

```
            MOV CX, 05H
            MOV SI, 00H
L:      MOV DL, ARR[SI]
            ADD DL, 30H
            MOV AH, 02H
            INT 21H
            MOV DL,' '
            MOV AH, 02H
            INT 21H
            INC SI
            LOOP L
            MOV AX, 4C00H
            INT 21H
        MAIN ENDP
```

TITLE addition of 100 natural even numbers

```
        .MODEL SMALL
        .STACK 100H
        .DATA
                TEN DW 10
        .CODE
MAIN        PROC FAR
                MOV AX, @ DATA
                MOV DS, AX
                MOV CX, 63H
                MOV AX, 02H
                MOV DX, 04H
    L1:     ADD AX, DX
                ADD DX, 02H
                LOOP L1
    L2:     MOV DX, 0000H
                DIV TEN         ; DX: AX /10
                INC CX
                ADD DX, 30H   ; remainder
                PUSH DX
                CMP AX, 00H   ; quotient
                JE L3
                JMP L2
    L3:     POP DX
                MOV AH, 02H
                INT 21H
                LOOP L3
                MOV AX, 4C00H
                INT 21H
        MAIN ENDP
        END MAIN
```

TITLE to display string at (10,40) with green background and red foreground

```
        dosseg
        .Model small
        .Stack 100H
        .Code
    MAIN PROC FAR
        MOV AX, @ DATA
        MOV DX, AX
        MOV SI, OFFSET VAR1
    L2:     MOV AH, 02H              ; Set cursor position
        MOV DH, ROW
        MOV DL, COL
```

```
                INT 10H
                MOV AL, [SI]
                CMP AL, '$'
                JE L1
                MOV AH, 09H
                MOV DH, ROW
                MOV DL, COL
                MOV BL, 24H          ;background & foreground
                MOV BH, 00h          ; page
                MOV CX, 01H          ; no. of repeated characters
                INT 10H
                INC SI
                INC COL
                JMP L2
        L1:     MOV AX, 4C00H
                INT 21H
        MAIN ENDP
                .DATA
                ROW DB 10
                COL DB 40
                VAR1 DB "video model", '$'
        END MAIN



            TITLE TO GENERATE MULTIPLICATION TABLE
                .MODEL SMALL
                .STACK 32
                .DATA
                        NUM1 DB 5
                        NUM2 DB 1
                        TAB DB 10 DUP (?)
                .CODE
            MAIN PROC FAR
                MOV AX, @ DATA
                MOV DS, AX
                MOV BX, 0
                MOV CX, 10
          L1:   MOV AL, NUM1
                MUL NUM2
                MOV TAB [BX], AL
                INC BX
                INC NUM2
                LOOP L1
                MOV AX, 4C00H
```

```
                    INT 21H
            MAIN ENDP
            END MAIN


        TITLE to add 10 sixteen bit Numbers in memory table
            .MODEL SMALL
            .STACK 32
            .DATA
            NUM DW DUP (2)
            NUM DW DUP (3)
            SUMH DW 0
            SUML DW 0
            .CODE
                    MAIN PROC FAR
                    MOV AX, @ DATA
                    MOV DS, AX
                    MOV CX, 10
                    MOV AX, 0
                    MOV BX, 0
            L1:     ADD AX, NUM [BX]
                    MOV SUML, AX
                    JNC L2
                    INC SUMH
            L2:     ADD BX, 2
                    LOOP L1
                    MOV AX, 4C00H
                    INT 21H
            MAIN ENDP
            END MAIN
```

Note: To access the data of the memory i.e. table.
We use e.g. NUM[BX]
Increasing the BX register by 2

SUBROUTINE TO CLEAR THE SCREEN

```
SCR_CLEAR PROC NEAR
        MOV AX, 0600H       ; Request scroll
        MOV BH, 61H  ; blue on brown for attribute on pixel(generally (07H) white on  black
        MOV CX, 0000        ; Full screen
        MOV DX, 184FH
        INT 10H
        RET
SCR_CLEAR ENDP
```

⟶ AH-06h: Scroll upward of lines in a specified area of the screen.
   AL- 00H caused entire screen to scroll up, effectively clearing it. Setting a nonzero
   value in AL causes the number of lines to scroll up.