EC 8691 - Microprocessors and Microcontrollers

"Unit - I :: The 8086 Microprocessor."

Introduction to 8086 - Microprocessor architecture - Addressing modes - Instruction set and assembler directives - Assembly language Programming - Modular Programming - Linking and Relocation - Stacks - Procedures - Macros - Interrupts and Interrupt Service routines - Byte and String Manipulation.

Introduction to 8086 Microprocessor

The Intel 8086 is a 16-bit microprocessor, It is ~~packe~~ 40 Pin dual in line package.

Feature of 8086

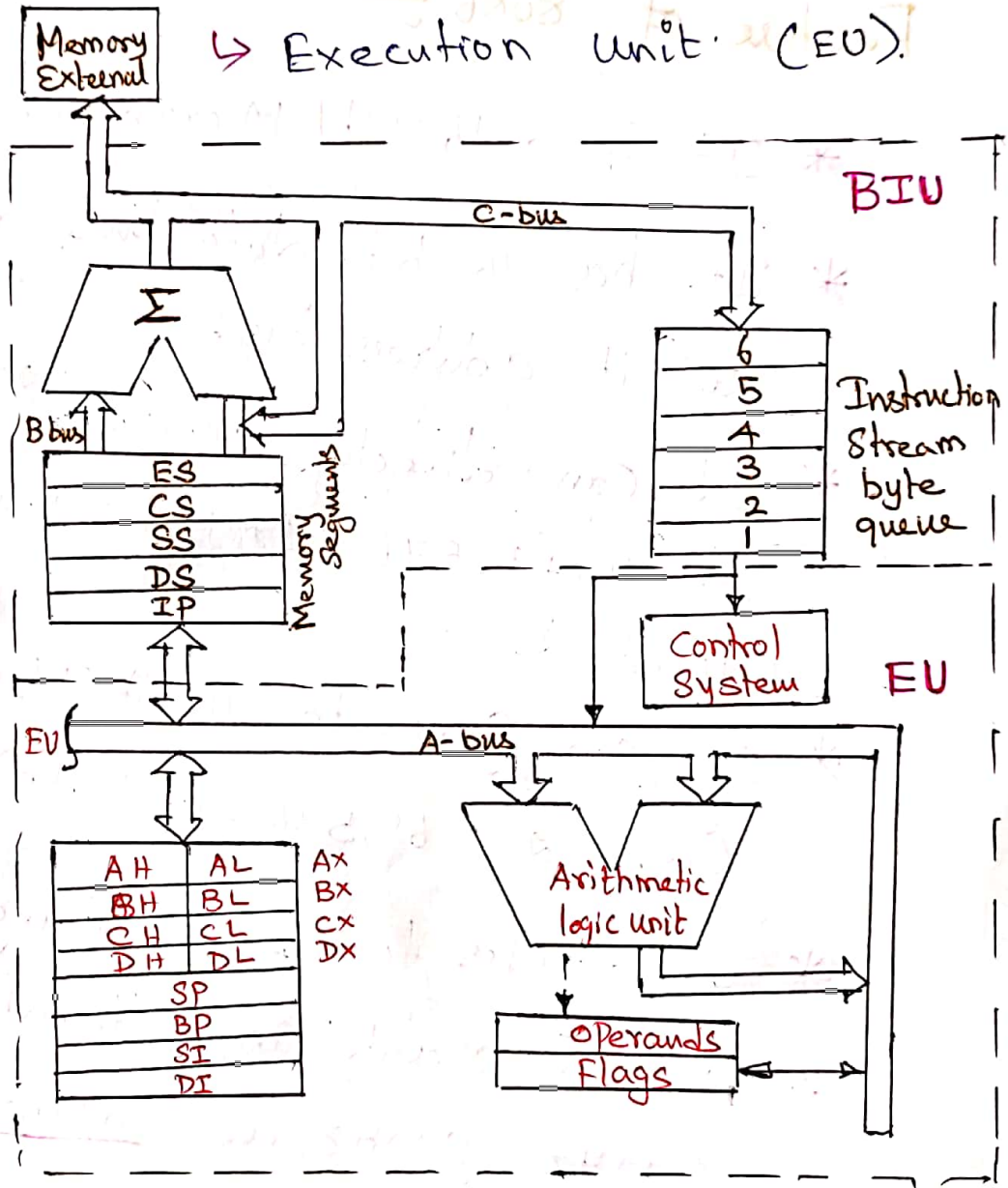
* It is a 16-bit Microprocessor.
* It has 16-bit data bus and 20 bit address bus.
* It can directly access 2^{20} (or) 10,48,576 (1 MByte) memory locations.
* It can generate 16-bit I/O address (ie., 2^{16} = 65,536 I/O Ports)
* It provides fourteen 16-bit registers.
* It can operate in different frequies 5 MHz, 8 MHz and 10 MHz.
* It can operate in two modes
  - ↳ Minimum mode
  - ↳ Maximum mode

Architecture of 8086:

The 8086 internal architecture is shown in figure. It is internally divided into two separate functional units.

↳ Bus interface unit (BIU)

↳ Execution unit (EU).



8086 - Internal block diagram.

Bus Interface Unit [BIU]

It is used to interface to the outside world. It Provides a full 16-bit bi-directional data bus and 20-bit address bus. The bus interface unit is responsible for performing all external bus operations, as listed below.

Functions of Bus Interface Unit

1. It Sends address of the memory or I/O Ports
2. It fetches instruction from memory
3. It reads data from Port/ memory
4. It write data into Port/ memory
5. It Supports instruction Queuing
6. It Provides the address relocation facility.

Instruction Queue.

To Speed up Program execution, the BIU fetches Six instruction bytes ahead and kept ready for execution. is called Queue.

With the help of queue it is Possible to fetch next instruction when Current instruction is in execution.

With the help of Queue instruction fetch time is eliminated as instruction is fetch which execution of Previous instruction takes place.

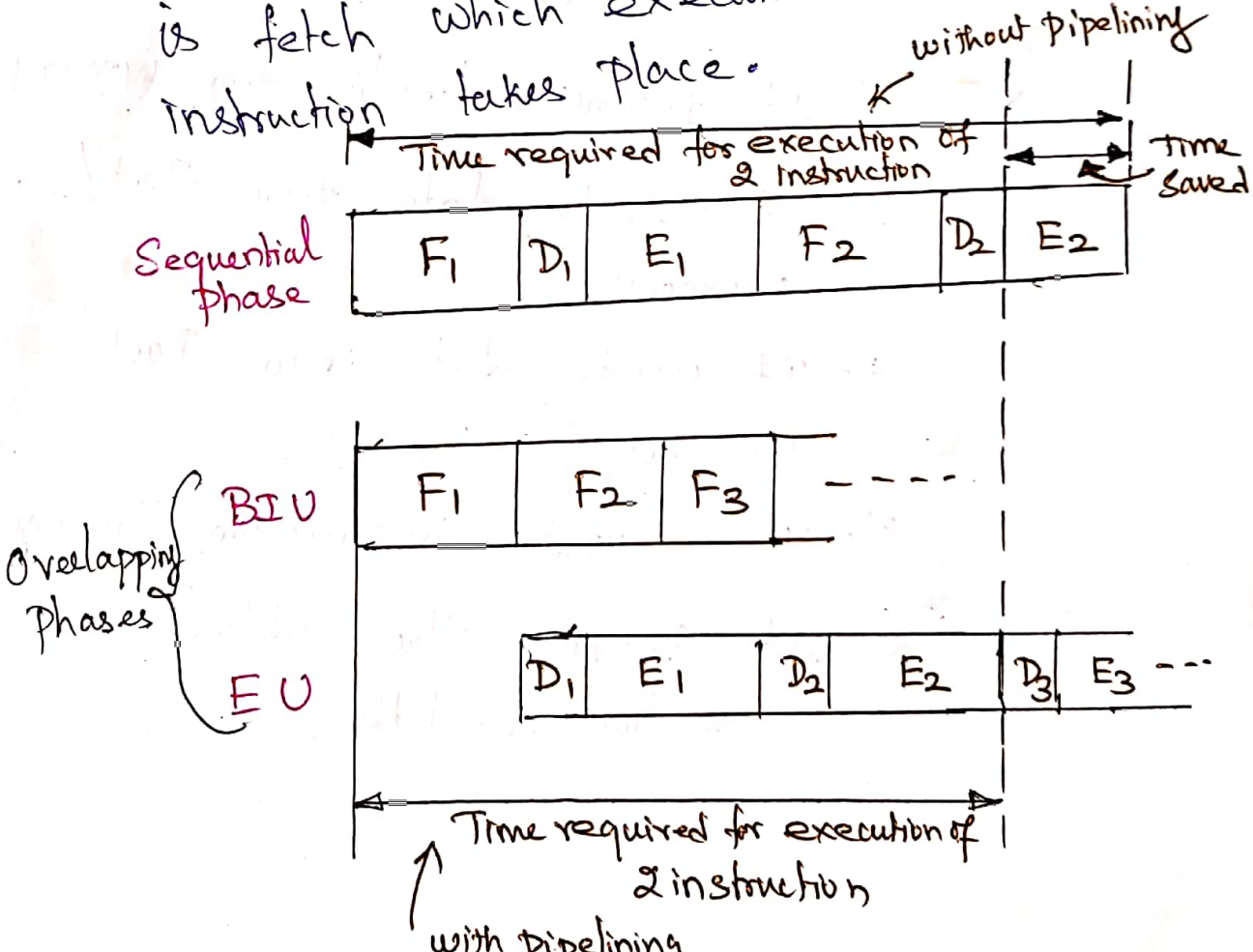


Fig: Pipeling

Memory Segmentation :-

Two types of memory organizations are commonly used.

Linear addressing
Segmented addressing

Linear addressing is method of addressing the entire memory space is available to the Processor in one linear array.

In segmented addressing, on the other hand, the available memory space is divided into "chunks" called segments. Such a memory is known as memory Segments

1 M bytes memory is divided into number of segments each of 64K bytes. in size.

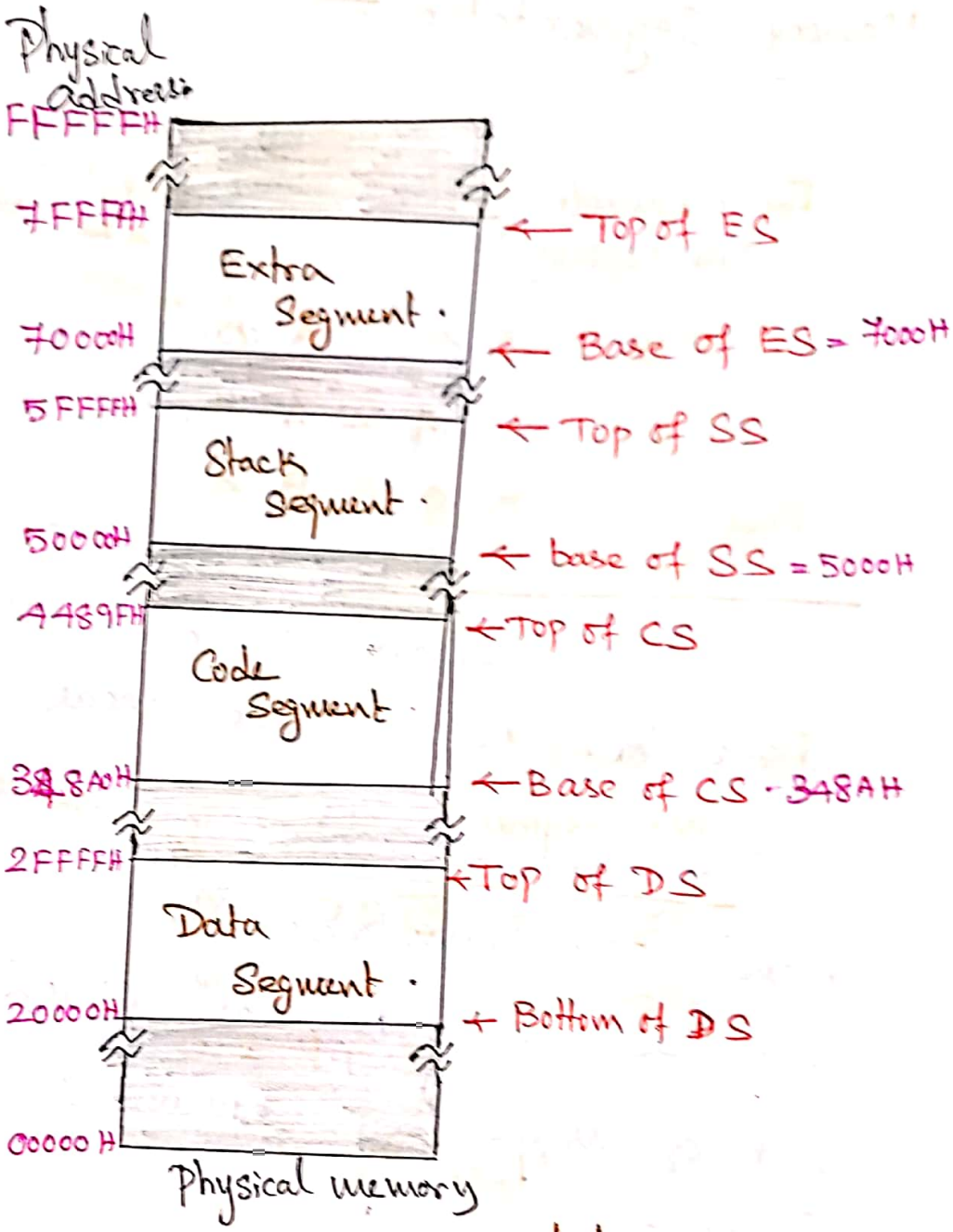


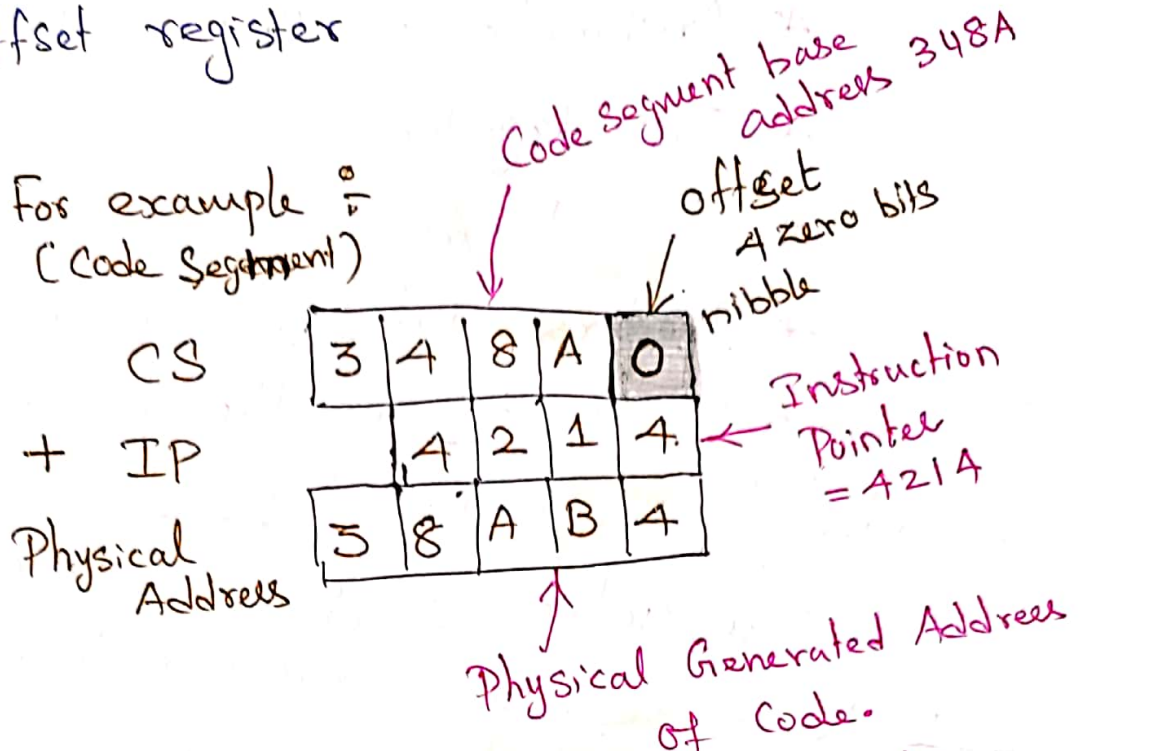
Fig: Memory Segmentation.

Generation of 20-bit address

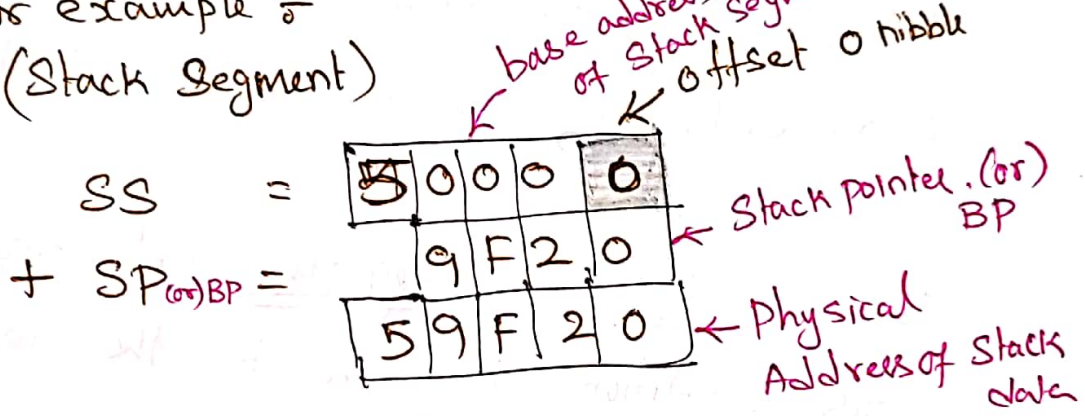
To access a specific memory location from any four Segment we need 20 bit Physical address. The 8086 generates this address using the Contents of Segment register and

Offset register

For example :-
(Code Segment)



For example :-
(Stack Segment)



Execution Unit [EU]

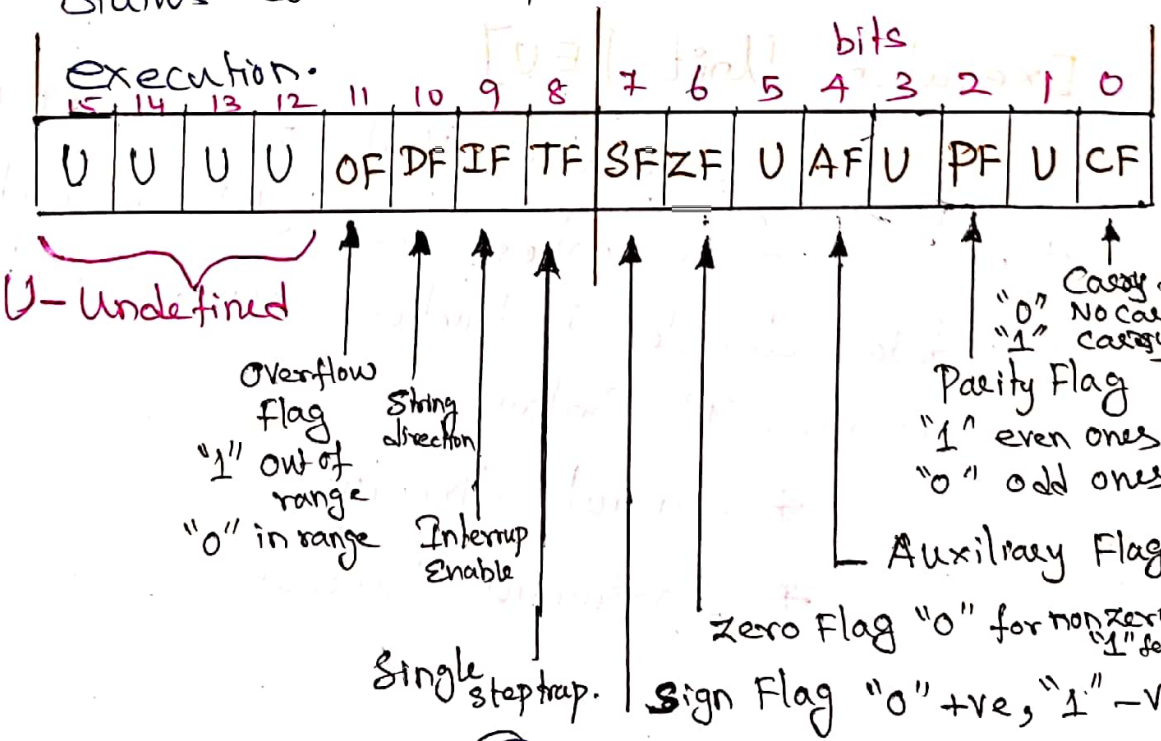
The execution Unit of 8086 tells the BIU from where to fetch instructions (or) data, decodes instructions and executes instructions. It Contains.

* Control Circuity
* Instruction Decoder

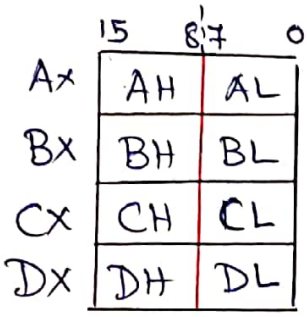
* Arithmetic Logic Unit (ALU)
* Flag Register
* General Purpose Register.
* Pointers and Index Register.

The Control Circuitry in the EU directs the internal operations. A decoder in the EU Perform action based on instructions ALU Perform 16 bit operations like add, Subtract, AND, OR, XOR, increment, decrement, Complement and Shift binary numbers:

The 16-bit Flag Register indicates the Status Condition Produced by the instruction execution.



The General Purpose Registers has four 16-bit registers AX, BX CX and DX It can be split into two 8 bits registers.



The letter L and specify the lower and higher bytes of a particular register.
The letter x indicates 16-bit register

The Four Segment Registers are Code Segment (CS), the data Segment (DS), the Stack segment (SS), the Extra Segment (ES) used to hold the base address of Segment to generate Physical Address.

CS	DS	ES	SS

The Pointers and Index Registers used to generate 20 bit physical address. The Pointer registers are Instruction Pointer (IP), Base Pointer (BP) and Stack pointer (SP).

The Index Registers are Source Index (SI) and Destination Index (DI) Registers

Addressing Modes :-

Addressing modes ~~is~~ different ~~mode~~ methods of addressing data to Processor. They are

* Immediate Addressing Mode
* Register Addressing Mode
* Direct Addressing Mode
* Register Indirect Addressing Mode
* Base Plus Index Addressing Mode
* Register Relative Addressing Mode
* Base Relative plus Index Addressing mode
* I/O Ports Addressing mode:

Immediate Addressing Mode :-

In immediate addressing mode, 8 bit (or) 16 bit data can be specified as a part of instruction.

Example :-

	Comment
MOV BL, 26H	BL ← 26
MOV CX, 4508H	CX ← 4508

Register Addressing Mode :-

In Register Addressing Mode, instruction specify the Register where the data is present.

Example :-

	Comment.
MOV BX, CX	BX ← CX
MOV CL, AL	CL ← AL

Direct Addressing Mode :-

In direct Addressing mode, a 16-bit memory address (offset) is specified directly in instruction

Example :-

MOV AX, [5000H] Content of [DS * 10H + 5000H] is copied to AX

Register Indirect Addressing Mode :-

In this addressing mode, the Effective Address [EA] is specified in either a pointer register (or) an index Register. Example :-

MOV DL, [BP] Content of [[SS] x 10H + [BP]] is copied to DL

Base plus Index Addressing Mode :-

In this addressing mode, uses one base register (BP (or) BX) and one index register (DI (or) SI) to indirectly address memory.

Example :-

MOV CX, [BX+DI]

Cx ← [[BX] x10H + [DI] x10H]

Register Relative Addressing :-

Register relative addressing is Similar to base-plus-index addressing. Here, the data in a segment of memory are addressed by adding the displacement to the Content of a base (or) an index register (BP, BX, DI (or) SI)

Example :-

MOV Cx, [Bx + 0003H]

The Content of address BX+0003 is moved to cx Register.

Base Relative Plus Index Addressing :-

The base relative plus index addressing mode is similar to the base plus index addressing mode, but it adds a displacement.

Example :-

MOV AL, [BX + SI + 10H]

The content of [BX + SI + 10H] is moved (copied) to AL Register

I/O Ports Addressing Mode :-

The are two types of Port addressing mode

Direct port addressing
In Direct port addressing

Direct port mode:

Here port number is an 8-bit immediate operand

Example :- OUT 05H, AL

Indirect Port mode :-

Here port number is taken from Dx

Example :- IN AL, DX

8086 Micro Processor Instruction Sets :-

The instruction set of the 8086 is divided into eight groups as follows

* Data transfer Instruction
* Arithmetic & Logical Instruction
* String Instruction
* Program Control transfer Instruction
* Iteration Instruction.
* Processor Control Instruction. (machine control)
* External Hardware Synchronization instruction
* Interrupt Instruction.

Data Transfer Instruction :-

A Instruction used to transfer the data from Register to Register, Register to memory, Register to Port are called data Transfer Instruction.

Example :-

```
MOV AX, BX   ; Comment Bx Content is copied to Ax

PUSH CX      ; Decrements SP (stack Point by 2 and copy Cx to Stack.

LES CX, [3483H] ; Copy Contents of memory at displacement of 3483 in DS to CX

ie CL ⇐ [DS+3483]
   CH ⇐ [DSx10+3484]

XCHG BX, CX  ; Content of Bx & Cx is Exchanged
```

Arithmetic & Logical Instruction :-

A Instructions which peeform Arithmetic operations like addition, Subtraction, Decrement, Increment, multiplication, Division and logical operations like "AND", "OR", "XOR", "NOT". "ROTATE" is called Arithmetic and Logical Instructions.

Example :-

```
ADD AL, 0F0H    ; Add immediate number F0H to Content of AL.
SBB DX, BX      ; DX ← DX - BX - CY
                                  ↑ Borrow.
DEC AL          ; Decrement AL by 1.
INC BX          ; Increment BX by 1
MUL BL          ; A* * BL → AX.
DIV CX          ; AX / CX → Quotient in Ax
                            remainder in Dx.
AND BL, AL      ; BL = BL "AND with" AL
OR CX, 00F0H    ; Cx = CX "OR with" 00F0
XOR BL, AL      ; BL = BL ⊕ AL
NOT AL          ; AL = ĀL
SAL CX, 1       ; Shift word in Cx
  ↑                 by 1 bit Poisition
Indicate. Left.     left
CMP CX, BX      ; Compare word in Bx
                  with word in Cx.
```

String Compare Instruction:-

The String Comparison instructions allow the programmer to test a section of memory against a constant (or) against another section of memory.

The 8086 Provides two instructions for String Comparisons: CMPS (compare string) and SCAS (string Scan).

Example :-

```
MOV SI, STR1      ; SI ← Pointer STR1
MOV DI, STR2      ; DI ← Pointer STR2
CLD               ; Auto increment SI & DI
CMPS STR1, STR2   Compare STR1 & STR2
MOV CX,100        Cx = 100 (counter)
MOV SI, STR1      SI ← STR1
MOV DI, STR2      DI ← STR2
REPE CMPSB        Repeat compare until end of string
```

Program Control transfer Instruction :-

A Instruction always cause the microprocessor to fetch its next instruction from the location Specified (or) indicated

by instruction rather than from the next address location.

Example :-

```
JMP  5000H  ; Unconditional jump
JZ   3000H  ;
JNC  4000H  ; } Conditional jump
JL   3000H  ;
CALL Cx     ; Call a subprogram.
```

Iteration Control instruction :-

These instructions are used to execute a series of instructions some number of times. The number is specified in the Cx register. The Cx register is automatically decremented by one, each time after execution of Loop instruction.

Example :-

```
Loop STR;
```

will jump to a ~~des~~ STR until Cx = 0

Processor Control instruction (or) Machine Control instruction.

Instructions which Control the machine itself is called machine Control instruction.

Example :-

```
STC ;   Set the carry flag
CLC ;   Clear carry flag
```

External Hardware Synchronization instruction :-

This instruction execution, will cause 8086 enters into Stop ~~fetching~~ fetching (or) idle Condition and Stay in this Condition until a signal is given Externally

Example :-

```
HLT ;   Stop fetching unbill a signal in INTR
WAIT ;  idle Condition untill a signal in TEST Pin
```

Interrupt Instruction :-

This instructions interrupt the current Program and Call a Interrupt Subroutine (Interrupt Service routine).

Example :-

```
INT type  ; Interrupt will call a 4 times of type no.
INTO      ; Interrupt when overflow flage
IRET      ; Interrup Return
```

Assembler Directives and Operators :-

There are some instructions in the assembly language Program which are not a Part of Processor instruction set. The instructions are instructions to the assembler, linker and loader. These are referred to as Pseudo-operations (or) as assembler directives.

The Commonly used assembler directive are :-

ALIGN :-

The align directive forces the assembler to align the next segment at an address divisible by specified divisor.

Example :-

ALIGN 2 ; directive is used to start the data segment on a word boundary

ALIGN 4 ; double word boundary

ASSUME :-

The 8086, at any time, can directly address four physical segments which include a code segment, a data segment, a stack segment and an extra segment. That is, the ASSUME directive assigns a logical assembler what addresses will be in the segment registers at execution time.

Example :-

```
ASSUME  CS ;    Code Segment
ASSUME  SS ;    Stack Segment.
```

- CODE :-

This directive Provides Shortcut in definition of the Code segment.

- DATA :-

This directive Provides Shortcut in definition of the data segment.

DB, DW, DD, DQ and DT :-

These directive are used to define different types of variable (or) to set aside one (or) more Storage locations of corresponding data type in memory.

Example :-

```
AMT DB 10H, 20H, 30H, 40H ;  Declare array of 4 bytes named AMT
```

DB - Define Byte

DW - Define Word

DD - Define Double word

DQ - Define Quad Word

DT - Define Ten bytes.

DUP :-

The DUP directive can be used to initialize several locations and to assign values to the location.

Example :-

DUP TAB DW 10 DUP(0); Reserve an array of 10 words of memory and initialize al 10 words with 0 with array named TAB.

END :-

The END directive is put after the last statement of Program to tell the assembler that this is the end of the program module.

EQU :-

The EQU directive is used to redefine a data name (or) variable with another data name, variable or immediate value.

EVEN :-

Even tells the assembler to advance its location Counter if necessary so that the next defined data item (or) label is aligned on an even storage boundary.

EXTRN :-

The EXTRN directive is used to inform assembler that the names (or) lable following the directive are in some other assembly module.

GROUP :-

A Program may Contain. Several Segments of the Same type (Code, data, or Stack). The Purpose of the Group is to Collect them all under one name, So that they reside within

one Segment, usually a data Segment.

Format : Name GROUP Seg-name, ---- Segname

LABEL :

It is an operator which tells the assembler to determine the number of elements in same named data item such as a string (or) array.

Example :

MOV BX, LENGTH STRING1.

MACRO :

The macros in the Programs can be defined by MACRO directive.

ENDM :

ENDM directive is used along with the MACRO directive. ENDM defines the end of the Macro.

.MODEL :

It is available in MASM Version 5.0 and above. This directive Provides short cuts in defining Segments.

NAME :-

It is used at the Start of a Source Program to give Specific names to each assembly module.

OFFSET :-

It is an operator which tells the assembler to determine the offset (or) displacement of a named data item (variable) from the start of the segment which contains it.

Example :-

```
MOV AX, OFFSET MES1.
```

• STACK :-

This directive Provides Shortcut in definition of the Stack Segment

Format :-

```
.STACK 100
```

Variables, Suffix and Operators :-

Variable :-

A Variable is an identifier that is associated with the first byte of data item.

Example :-

Array DB 10, 20, 30, 40, 50

Suffix :-

In assembly language Programming base of the number of indicated by a suffix as follows :

* B - Binary
* D - Decimal
* O - octal
* H - Hexadecimal

operators :-

Arithmetic operators :-

"+", "-", "*" and "/"

Logical operators :-

"AND", "OR", "NOT" and "XOR"

Assembly Language Program :-

To make Programming easier, usually Programmers write Programs in assembly language. They then translate the assembly language Program

to machine language.

Assembler Instruction Format :-

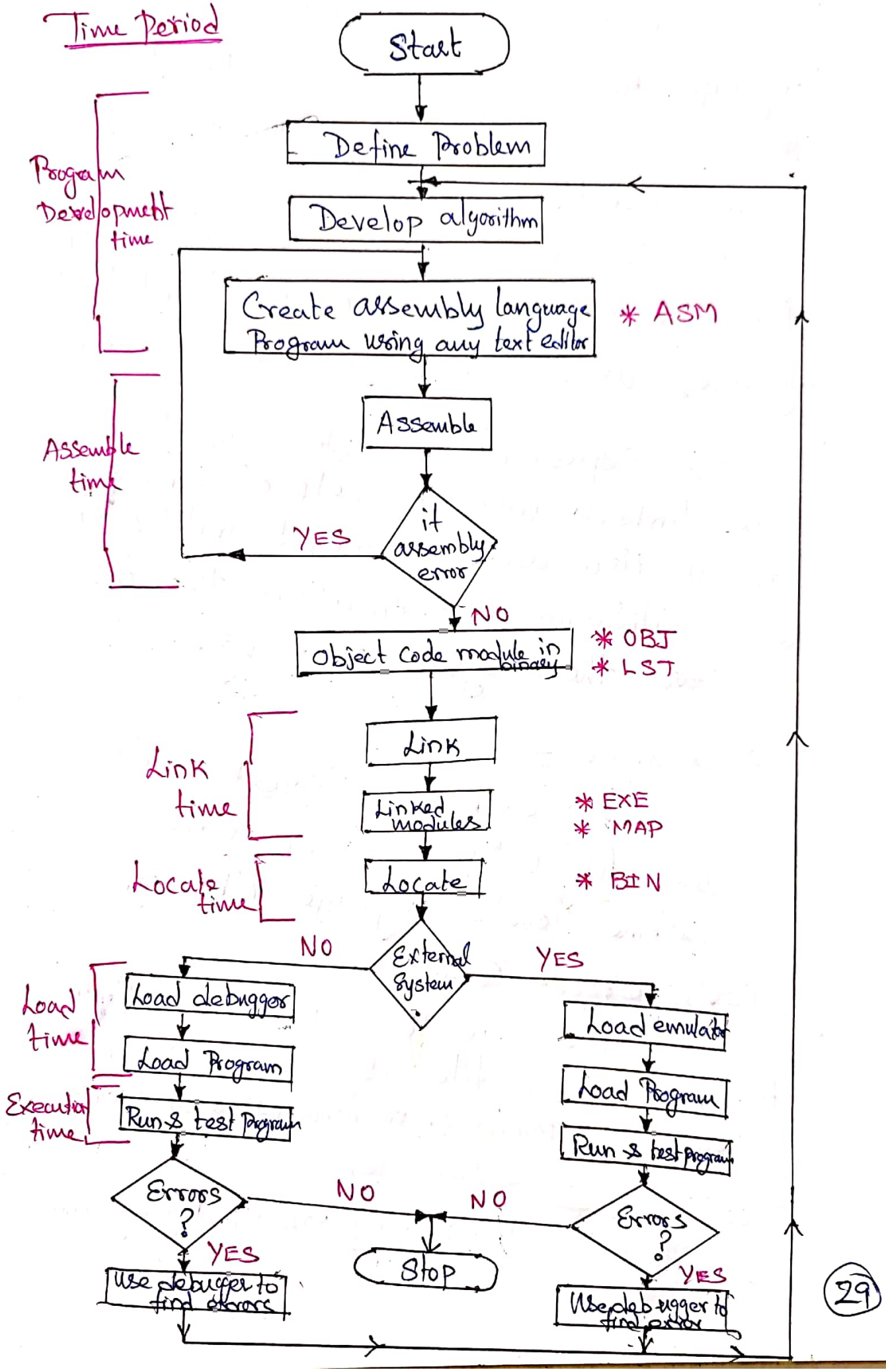
The assembly text is usually divided into fields,

Label : Mnemonic Operand1, Operand2; Comments

Assembly Process :-

Generating machine Code from assembly language Program manually is very time Consuming and Complex Process. Thus assembly language tools such as assembler, linker, loader and debugger are used to develop and execute assembly language Programs.

The steps involved in developing and executing assembly language Programs is shown in figure.



The first step is to write an assembly language Program which is written with an ordinary text editor such as word star, edit and so on.

The assembler translates assembly language Program to machine level lan -guage, usually known as object code.

The Separate assembled modules are linked with the help of Link and it also adds required initidization (or) finalization code to allow the OS to start the Program running.

Assembler :

Assembler Converts a Source file (Mnemonics) into machine level language (binary (or) object Code).

We Can use a suitable Editor to type .asm file. By using MASM assembless (Microsoft macro assembler) (or) TASM (Turbo assembler)

Command :

C:\MASM\BIN\>MASM myProg.asm

Linker :-

A linker is a Program used to join together several object files into one large object file.

Command :-

```
C:\MASM\BIN\> LINK myprog.Obj
```

Modular Programming :-

Programming task is divided into Subtasks, Separate modules (Program) are written for performing Subtasks. Each module is tested Separately. The Common routines required in modules are written in Separate module and they are "called" from individual modules as Per Sequence. Programming done in this fashion is called "Modular Programming"

Advantage of Modular Programming:

* Coding is Simple, easy and short
* It is easier to design, test and debug.
* It is easier to understand
* It is easy to modify the code.

Modular Program used three Components

* Structure
* Procedures (or) Subroutines
* Macros

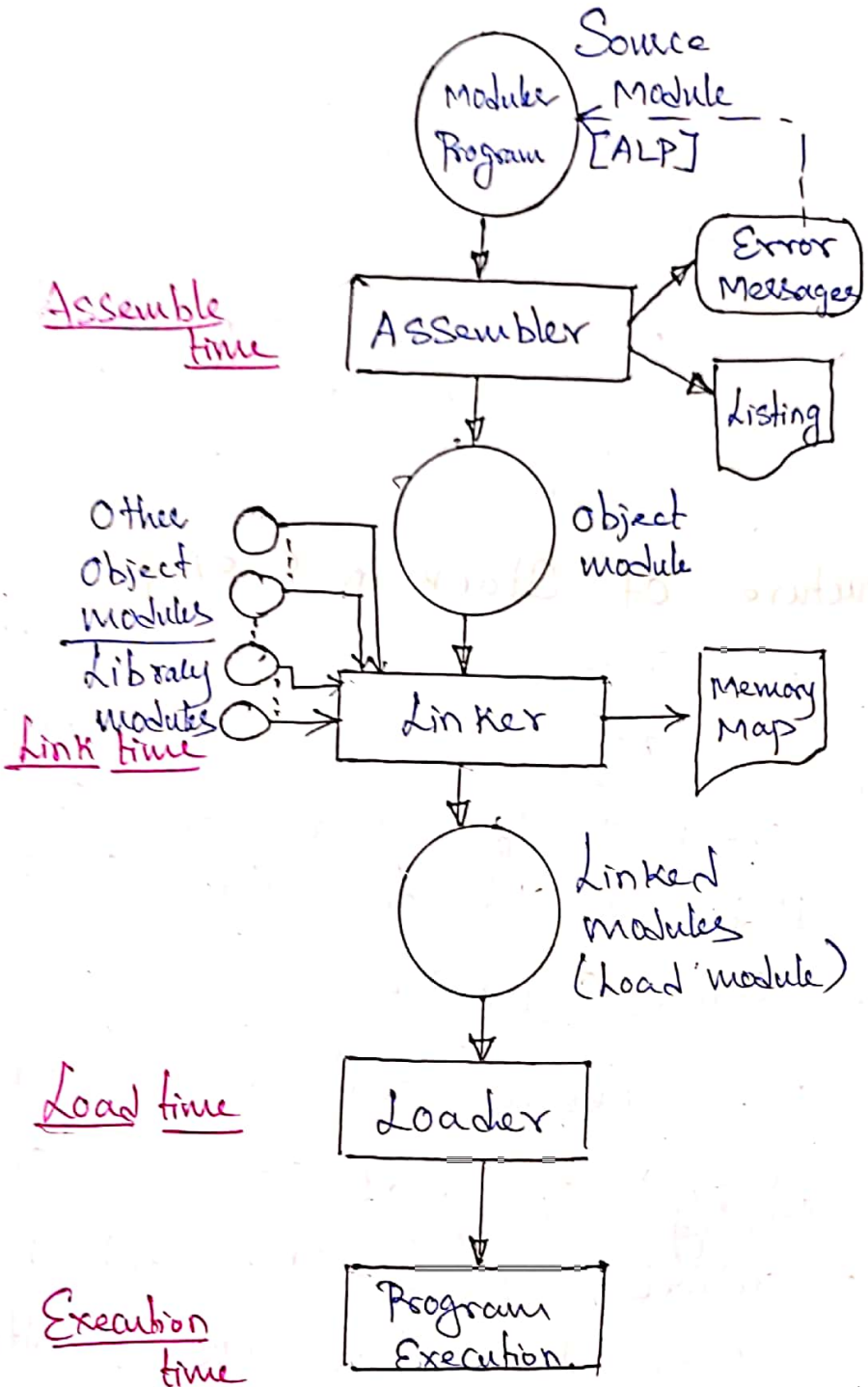
Linking and Relocation :-

Generating machine code from assembly language Program is done with the help of assembler, linker, loader and debugger.

The Steps involved in developing and executing assembly language Program is shown below.

The first Step in the development Process is to write an assembly language Program.

The assembler translates assembly language Statements to machine level language (binary equivalents) known as object code.



The object code modules contains information about where the module to be loaded in memory. Linker is to create link between the modules. the Program loader copies the Program into Computer's main memory and Execution begins.

Stack :-

The Stack is a portion of read/write memory set aside by the user for the purpose of storing information temporarily. PUSH instruction used to write data on Stack and POP used to read data from Stack.

Structure of Stack in 8086 :-

The 8086 has a special 16-bit register, SP to work as a Stack Pointer. The Stack pointer SP register Contains the 16-bit offset from the Start of the Stack Segment.

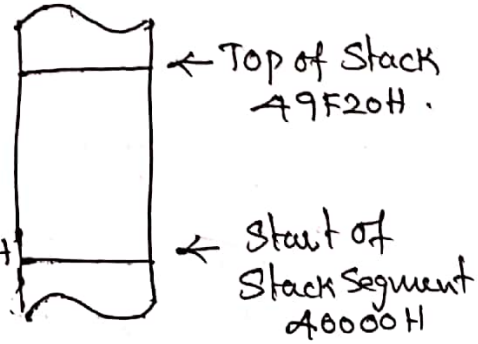
The Physical address is generated by adding Content of SP and Segment base address in SS. The Content of Stack Segment register are Shifted four bits left and the Content of Stack pointer SP are added.

SS = 4000 $\xrightarrow[4 \text{ bits}]{\text{Shifted}}$ 40000H.

SS×10 ⇒ 40000 H

SP ⇒ 9F20 H

49F20 ← physical address



PUSH and POP operations :-

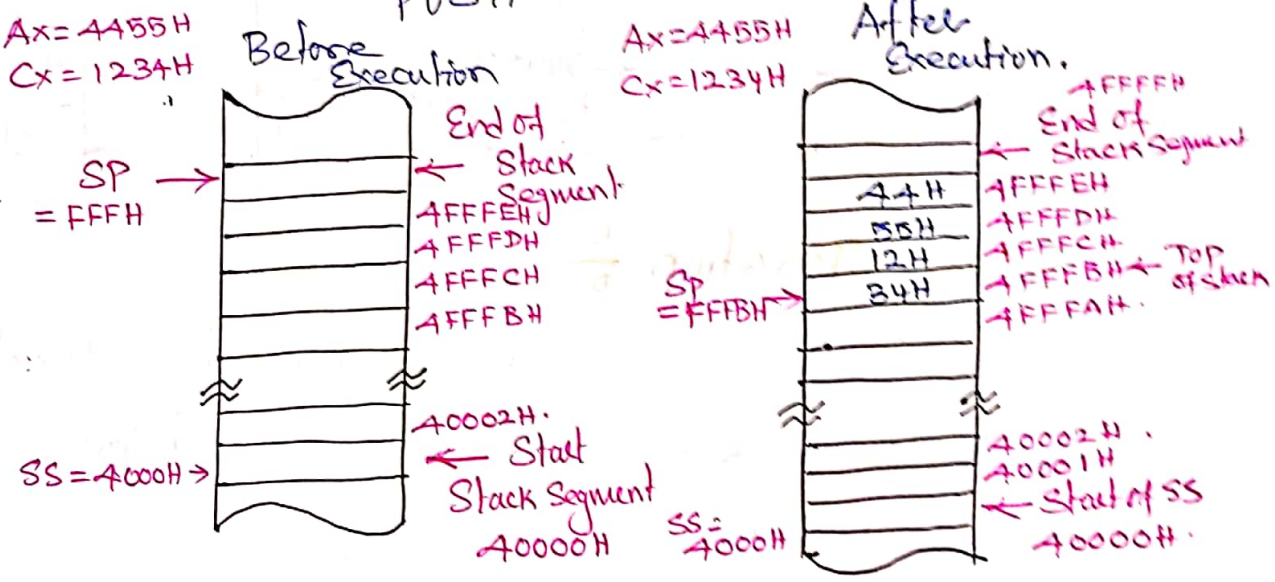
PUSH operation :-

The PUSH instruction decrement Stack Pointer by two and Copies a word from Some Source to the location in the Stack where the Stack Pointer Points.

for Example.

PUSH AX

PUSH BX



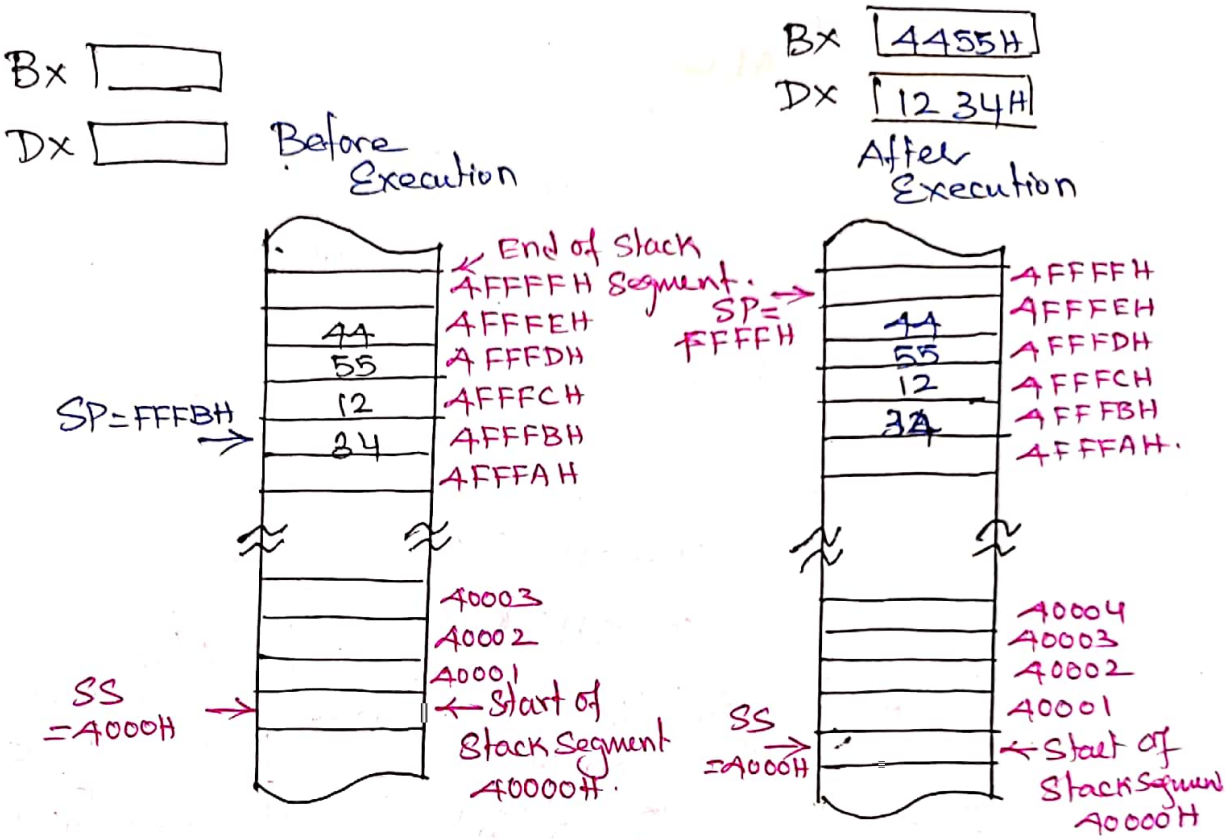
POP Operation :-

The POP instruction Copies a word from the Stack location pointed by the Stack Pointer to the destination. After the word is Copied, the stack pointer is automatically incremented by 2.

for Example

POP DX

POP BX



CALL operation 6/0

The CALL instruction is used to transfer execution to a Sub program (or) Procedure.

There are two basic types of CALLs, near and far.

A near CALL is a call to a Subprogram which is in the Same Code Segment. when CALL instruction executes it decrements the stack pointer by two and Copies the offset of the next instruction address.

It loads IP with the offset of the first instruction of the Subprogram address in Same Segment.

A far CALL is a call to Subprogram which is in a different Segment from that which Contains the CALL instruction. when far CALL instruction executes, it decrements the Stack Pointer by two and Copies the Contents of the CS Register to the Stack. It then decrement the Stack Pointer by two again and Copies the offset of the instruction after the CALL to the Stack. Finally it loads CS with the segment base of the Segment which Contains the Subprogram and IP with the offset of the first instruction of the Subprogram. In that Segment.

RET Operation

The RET instruction will return execution from a Subprogram to the next instruction after the CALL Instruction

if near RET, then the return will be done by replacing the instruction pointer with a word from the top of the Stack.

if far RET, then instruction Pointer will be replaced by the word at the top of the Stack. The Stack Pointer will then incremented by two. The Code segment register is then replaced with a word from the new Stack top address.

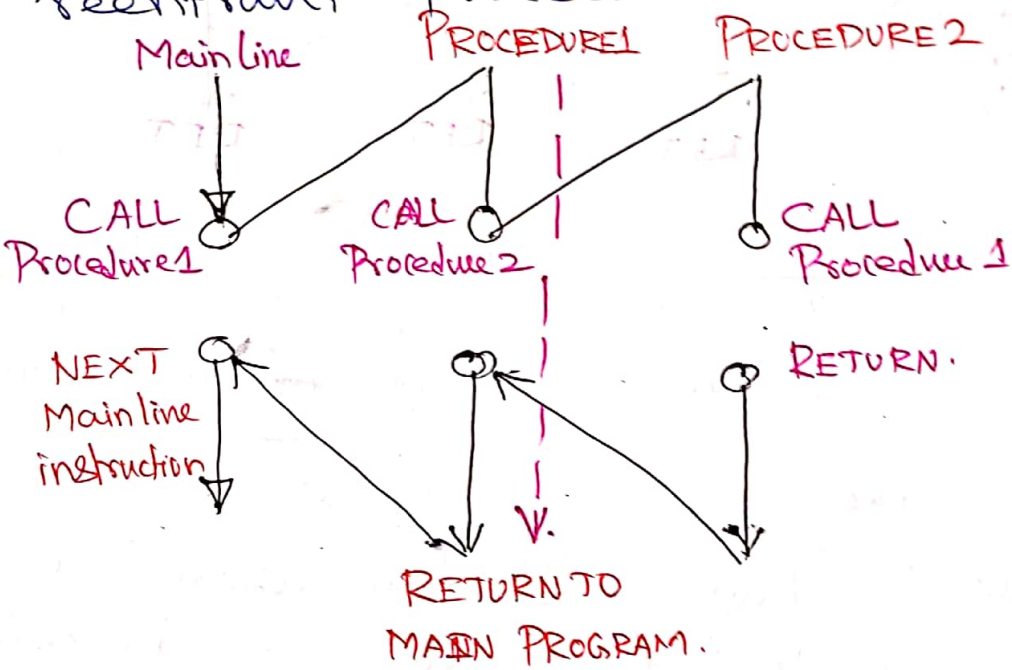
Procedures

whenever we need to use a group of instructions several times throughout a program there are two ways we can avoid having to write the group of instructions again and again.

One way is to write the group of instructions as a separate procedure. we can then just CALL the Procedure whenever we need to execute that group of instruction.

Reentrant Procedure :-

In some situations it may happen that Procedure 1 is called from main Program, Procedure 2 is called from Procedure 1 is again called from Procedure 2. In this situation Program execution flow reenters in the Procedure1. This type of Procedure are called reentrant Procedures.

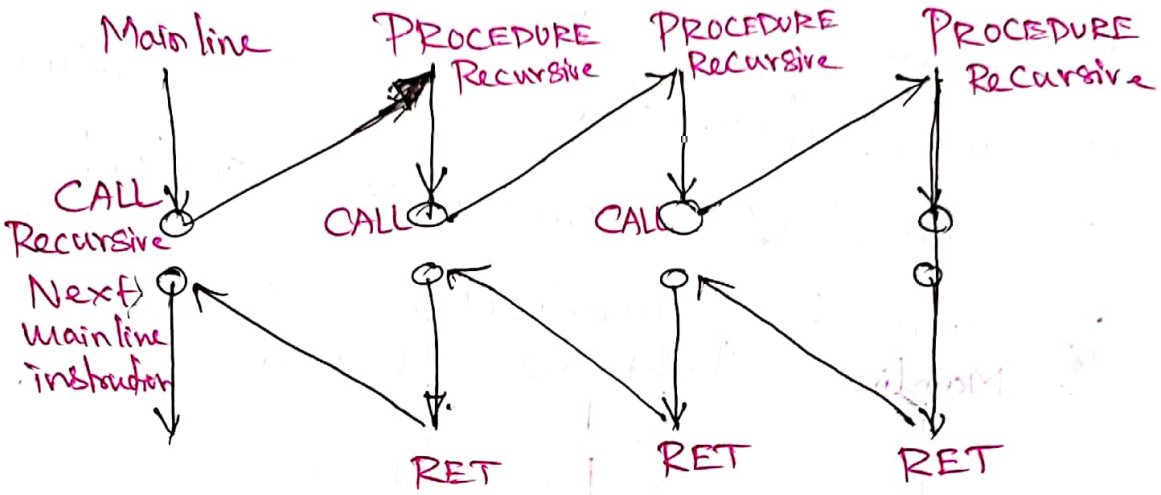


Recursive Procedure :-

A recursive Procedure is a Procedure which calls itselfs.

Recursive Procedure are used to work with Complex data Structure called trees.

if the Procedures is called with N (recursion depth) = 3. Then the n is decremented by one after each Procedure CALL and the Procedure is called until n = 0.



Macro's

Macro is a group of instructions. The macro assembler generates the code in the Program each time where the macro is 'Called'. Macros can be defined by MACRO and ENDM assembler directives.

Example :- Macro definition for initialization of Segment registers.

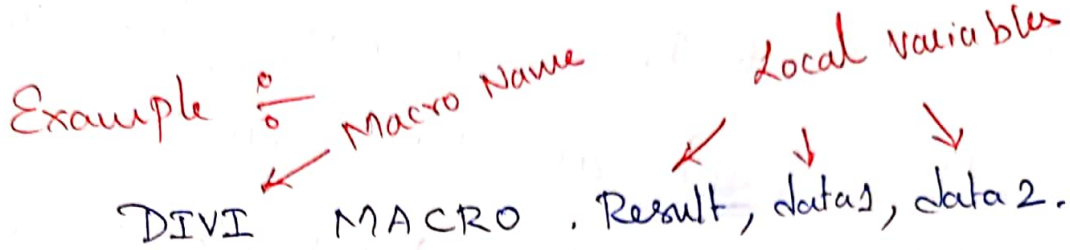
```
INIT  MACRO          ← Define Macro
      MOV AX, @data  }
      MOV DS, AX     } Body of macro definition
      MOV ES, AX     }
      ENDM           ← End macro.
```

(40)

Local Variables and Labels in a Macro :-

Body of the Macro can use local variables as a Passing Paraments.

Example :-



This variables can used and available only inside the Macro.

Local Labels used in jump instruction can only used inside the Macro..

8086 Interrupts :-

The word interrupt is to break the sequence of operation. While the CPU executing a Program, an interrupt breaks the normal sequence of execution of instructions and diverts its execution to some other program called Interrupt Service Routine (ISR).

After Executing ISR, the control is transfered back again to the Main Program which was executed at the time of interruption.

8086 Processor has two interrupt Pins such as NMI and INTR. The NMI is a non-maskable interrupt, which means NMI cannot be masked (or) disabled. The INTR may be masked by using the Interrupt flag (IF).

Types of Interrupts :-

* External Interrupts (or) Internal Interrupts
* Hardware Interrupts (or) Software Interrupts
* Maskable (or) Non-Maskable Interrupts.

External Interrupts :-

External interrupt is an interrupt which is generated outside the Processor, for example, Printer interrupt.

External Signal is applied on 8086 INTR and NMI Pin to Create External interrupt.

Software Interrupt :-

Interrupt which is created using Program (software) is called Software Interrupt.

Example :-

INT 0

Maskable Interrupt :-

Interrupt which can be disabled (or) masked by using Program with the help of Interrupt flag (IF) is called Maskable Interrupt. (Lower Priority)

Example :-

INTR.

Non Maskable Interrupt :-

Interrupt which cannot able to mask (or) disable (or) hide is called Non Maskab Interrupt. usually This interrupts have (higher Priority)

Example :-

NMI

Internal Interrupts :-

Internal, which is created internally by the 8086 Processor is called Internal Interrupts.

Hardware Interrupts :-

The interrupts initiated by external hardware by sending an appropriate signal on in interrupt INTR (or) NMI is called Hardware Interrupts

256 Interrupts of 8086 are divided into 3 Groups as

1) TYPE 0 to TYPE 4 interrupts.

These are used for fixed operations and hence called as dedicated interrupts.

2) TYPE 5 to TYPE 31 Interrupts

Not used by 8086

3) TYPE 32 to TYPE 255 Interrupts.

Available for user so called User defined Interrupts.

↳ TYPE 0 ⇒ Divide Error Interrupt

TYPE 1 ⇒ Single step Interrupt.

TYPE 2 ⇒ Non Maskable Interrupt.

TYPE 3 ⇒ Break Point Interrupt.

TYPE 4 ⇒ Overflow Interrupt.

BYTE and STRING Manipulation :-

STRING Instructions :-

MOVSB – Move Byte String

MOVSW – Move word string

CMPSB – Compare Byte String

CMPSW – Compare word string

SCASB – Scan Byte string

SCASW – Scan Word String

LODSB – Load Byte String

LODSW – Load Word String

STOSB – Store Byte String

STOSW. – Store word string