Microprocessors & Microcontrollers (ECE3004)

Module – 02 (Part – 02)

8086 Microprocessor

Dr. Susant Kumar Panigrahi
Assistant Professor
School of Electrical & Electronics Engineering

8086 μP: Introduction to Programming

Instruction Format

- ✓ Instruction contains one or more number of fields:

OpCode + Operand

Operation Code Field: It indicates the type of operation to be performed by CPU.

Operand Field: The CPU executes the instruction using the information that resides in these field

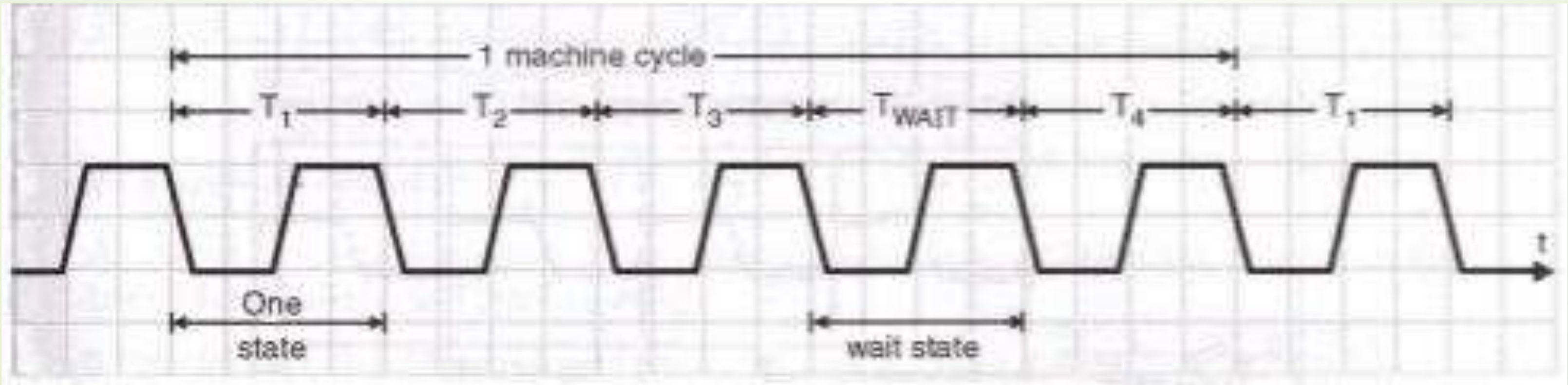
- **Ex:**

```
✓ MOV AL, BL          ; Instruction only consists of Opcode
✓ MOV AL, 20H         ; Instruction consists both opcode and operand
                      ; MOV AL : Opcode
                      ; 20H : Operand
```

```
✓ ADD     AX, BX
✓ ADD     AL, FFH
✓ XCHG    BX, AX
```

Clock Cycle, Machine Cycle and Instruction Cycle

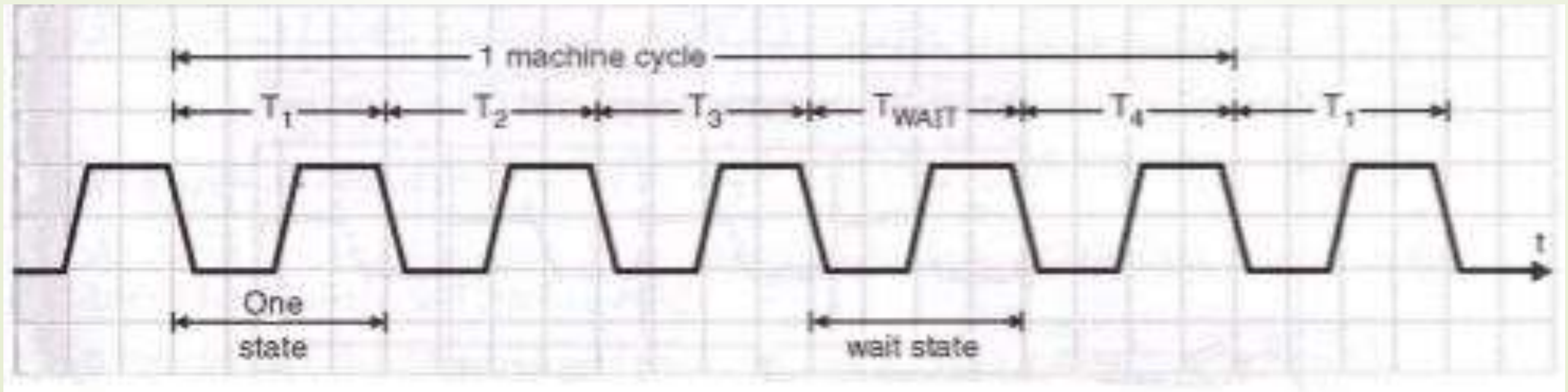
- **Clock Cycle:**
  - ✓ The speed of a computer processor, or CPU, is determined by the **clock cycle**, which is the amount of time between two pulses of an oscillator.
  - ✓ Generally speaking, the higher number of pulses per second, the faster the computer processor will be able to process information.
  - ✓ The clock cycle is the smallest unit of time, that μP can detect. For example 8086 μP produces 8MHz of clock frequency, i.e. 8,000,000 pulses per second.



Clock, T-state, machine cycle etc.

Clock Cycle, Machine Cycle and Instruction Cycle

- **Machine Cycle:**
  - ✓ It is the time required by the μP to complete the operation of accessing memory or I/O devices is called **machine cycle**.
  - ✓ A machine cycle consists of several T-states (or clock cycles). For an example 8086 microprocessor requires at least four T-states or clock cycles to produce one machine cycle.
  - ✓ If 8086 is connected to a slower device, it requires four T-states with an additional wait state, T_w in-between T_3 and T_4 states. Thus the total cycles that are required are: T_1, T_2, T_3, T_w and T_4.

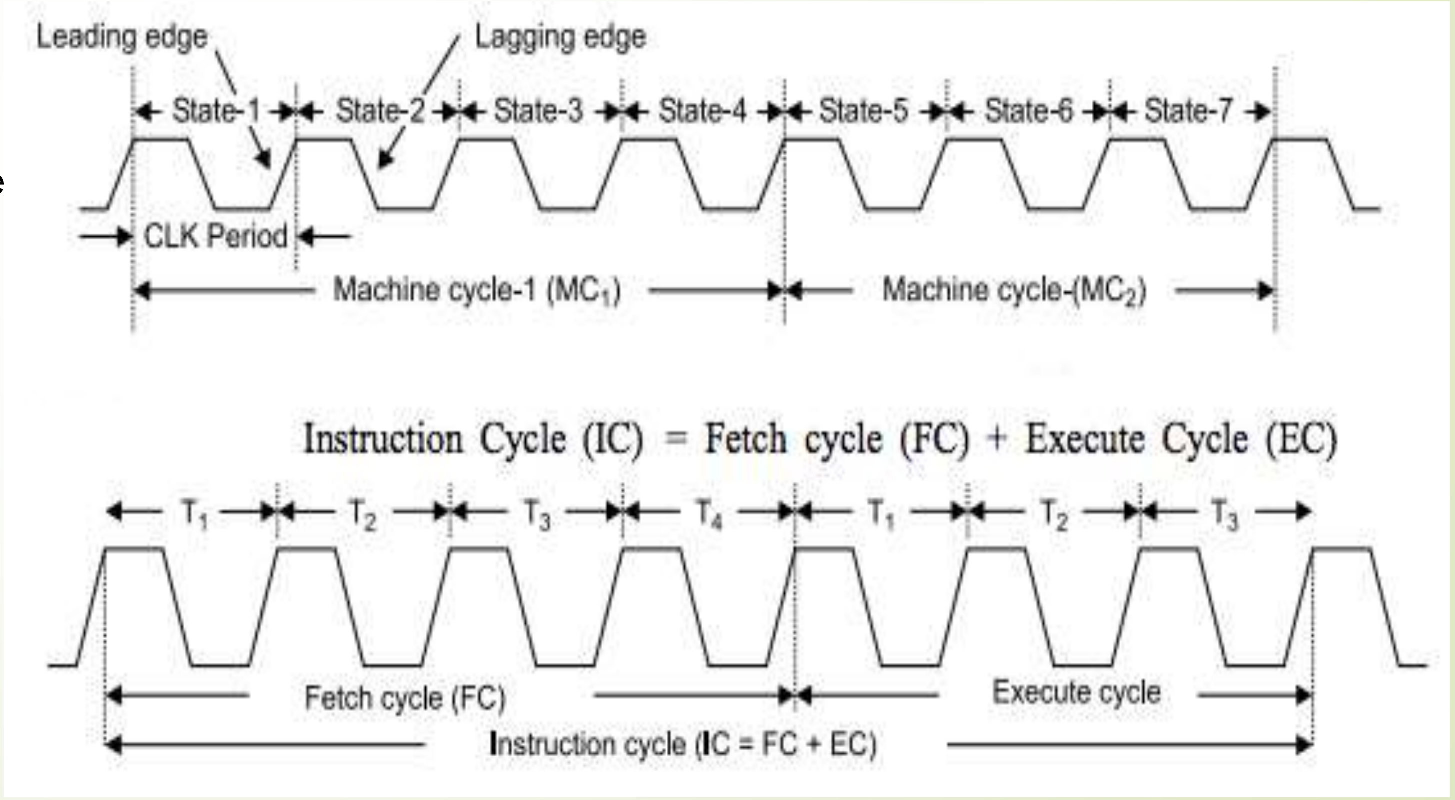


Clock, T-state, machine cycle etc.

Clock Cycle, Machine Cycle and Instruction Cycle

Instruction Cycle:

- ✓ The time a μP needs to fetch and execute one entire instruction is known as instruction cycle.
- ✓ There are typically four stages of an instruction cycle that VPU carries out:
  - ➢ Fetch the instruction
  - ➢ Decode the instruction
  - ➢ Read the effective addre
  - ➢ Execute the instruction



A Simple Program

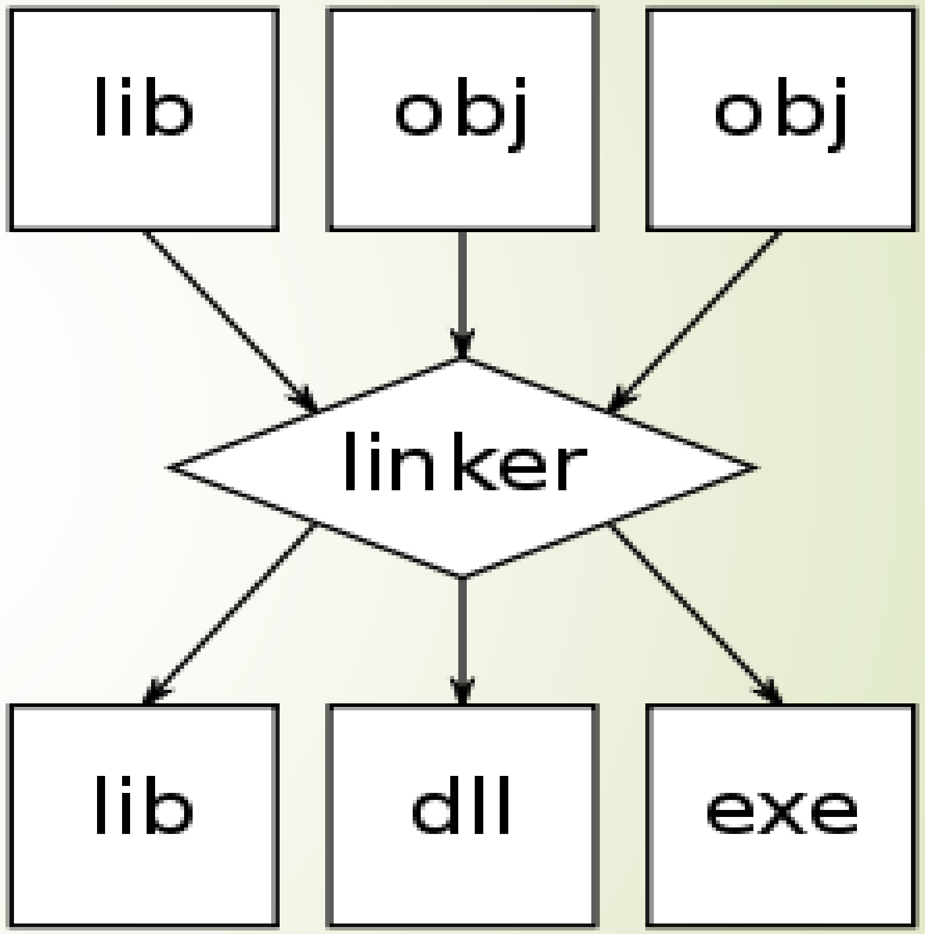
- *Here is a short program that demonstrates the use of MOV instruction:*

```
ORG 100h              ; this directive required for a simple 1 segment .com program.
MOV AX, 0B800h        ; set AX to hexadecimal value of B800h.
MOV DS, AX            ; copy value of AX to DS.
MOV CL, 'A'           ; set CL to ASCII code of 'A', it is 41h.
MOV CH, 1101_1111b ; set CH to binary value.
MOV BX, 15Eh           ; set BX to 15Eh.
MOV [BX], CX           ; copy contents of CX to memory at B800:015E
RET                   ; returns to operating system.
```

8086 Microprocessor - Linking and relocation

- **Linker** is a computer utility program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another 'object' file.

- **Relocation** is the process of assigning load addresses for position-dependent code and data of a program and adjusting the code and data to reflect the assigned addresses.



Assembler Directives

Assembler:

- ✓ It is a program that accepts an assembly language program as input and converts it into an object module and prepares for loading the program into memory for execution.
- ✓ Loader (linker) further converts the object module prepared by the assembler into executable form, by linking it with other object modules and library modules.
- ✓ The final executable map of the assembly language program is prepared by the loader at the time of loading into the primary memory for actual execution.
- ✓ The assembler prepares the relocation and linkages information (subroutine, ISR) for loader.
- ✓ Thus the basic task of an assembler is to generate the object module and prepare the loading and linking information.

Assembler Directives

Assembler: (Procedure for assembling a program)

- ✓ The assembler reads the instruction statement by statement, sequentially.
- ✓ Along with instruction an assembler needs some hints from the programmer:
  - ✓ The required storage for particular constant or variable (size of a constant/variable).
  - ✓ Logical names of the segment.
  - ✓ Types of different routines and modules.
  - ✓ End of the segment.
  - ✓ End of the program.
- ✓ These kind of hints are given by the predefined alphabetic strings called "**Assembler Directives**".

Assembler Directives

Directives:

✓ The directives instruct the assembler to correctly interpret the program to code it appropriately. On the other hand operator (also a part of assembler) helps the assembler to correctly assign a particular constant with a label or initialize particular memory location or labels with constants.

Prog.: Write an Assembly Language Program (ALP) to add two 26-bit numbers.

Approach: you should write the codes to satisfy all syntaxes mentioned by the assembler. Follow the rules of assembler not the processor.

✓ Thus in order to execute the addition of two numbers we must write some assembler directives to execute the program.

General Syntax

```
Data    Segment



Data    Ends


Code    Segment
Start:


Code    Ends
        End       Start
```

"Segment": Directive given with name "Data"

"Ends": Directive indicates end of Segment

"Segment": Directive given with name "Code"

"Start": Directive indicates start of prog.

"End Start": Directive indicates end of prog.

Assembler Directives

Directives: [Previous Prog. Example]

- ✓ The assembler will create two "segments" and name them:
  - Data
  - Code
- ✓ Inside Data segment we will define variables: (as in case of C)
- ✓ Syntax:

Data Segment

A	*DB*	*25H*
B	*DW*	*4523H*
C	*DW*	*?*

Data Ends

int	i=25;
Int	b;
double	C;

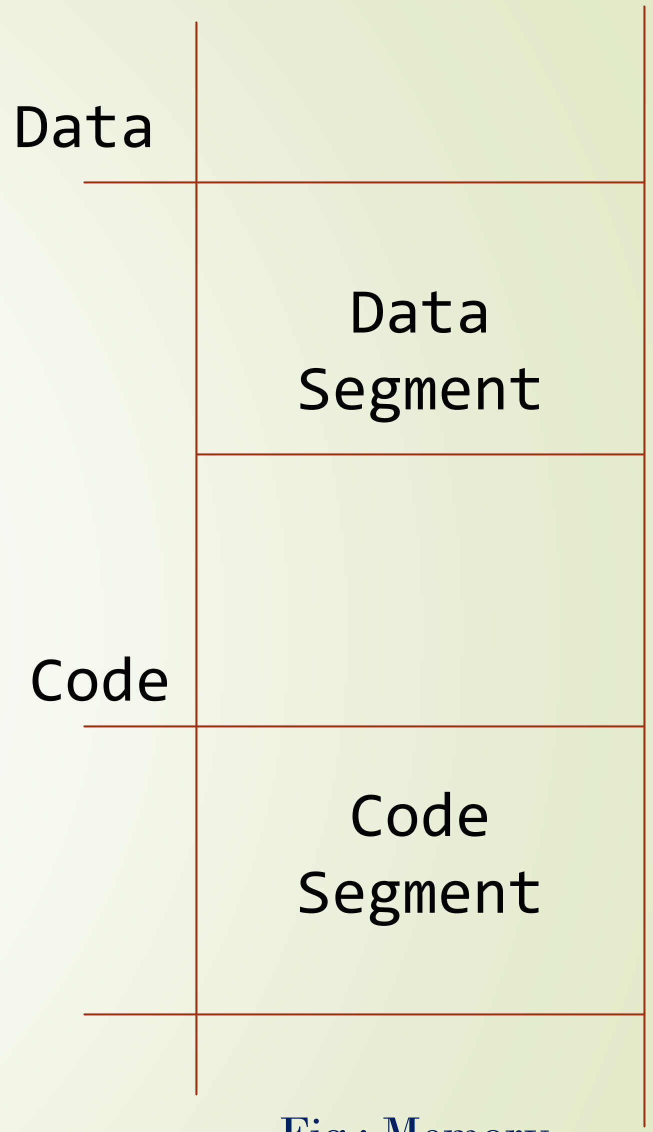


Fig.: Memory

Assembler Directives

- **Variables:** **[Directives]**
  - ✓ DB: Define Byte (8-bits)
  - ✓ DW: Define Word (16-Bits)
  - ✓ DD: Define Double Word (32 Bits)
  - ✓ DQ: Define Quad Byte (64 Bits)
  - ✓ DT: Define Ten Bytes (80 Bits)

Ex:

```
  A           DB          45H
```

The assembler will create a variable with name A and will give it a value 45H.

Since it is defined inside data segment it will create a memory location with address of A and stores the value 45H.

Ex:

```
 C           DW          B58EH
```

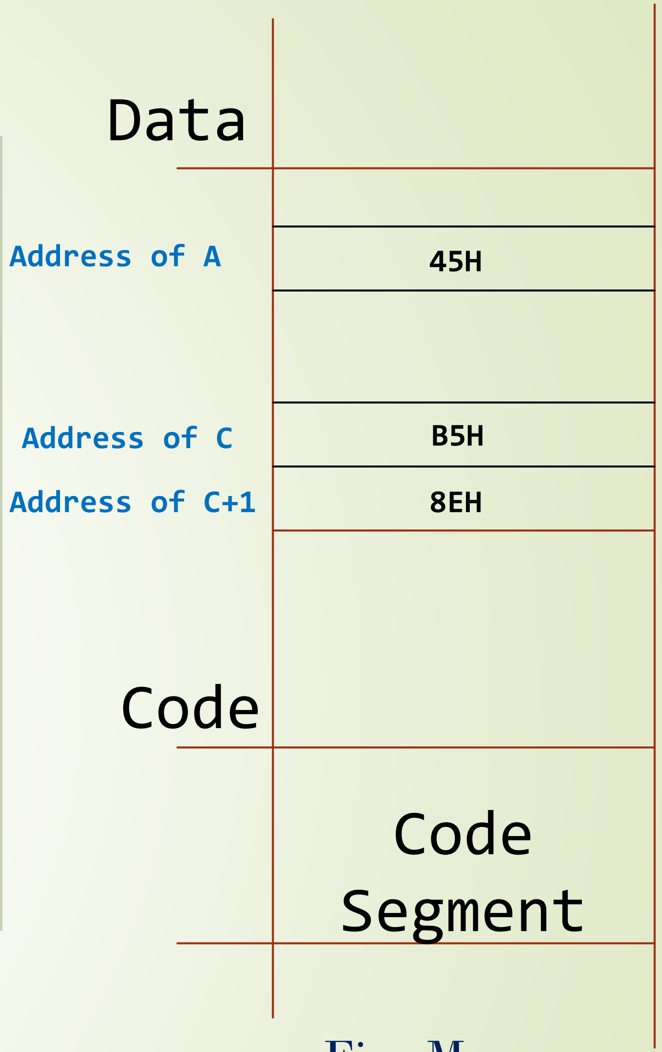


Fig.: Memory

Q.: The processor 8086 is a 16-bit and can perform 16-bit data operations at a time (only in case of division and mul it needs 32 bit bit). Then, why there is provision for 64-bit or 80-bit operation?

Assembler Directives

- **Variables:** **[Variable Directives Conti…]**

MOV CL, A

CL will get the value of A: CL ← 45H

✓ How "MOV CL, A" works internally:

- ✓ The assembler converts the instruction from assembly language to machine level language:

  MOV CL, A

- ✓ MOV CL: Opcode and A: Variable defined by assembler directive
- ✓ As 'A' is a variable and it may change in subsequent operations. So assembler will not change the value of 'A' every time, it will check the address in the segment and gets the value from it.

- **NOTE**:
  - ✓ For predefined instructions (Like: MOV CL, ADD AX, BX and so on…) the assembler replaces the instruction with its corresponding opcode.
  - ✓ However, for variable it puts address of the variables that is located inside the memory.
  - ✓ More precisely, the assembler replaces the variable with its 'Offset Address'.

Assembler Directives

- **Assume: [Directives]**

✓ ASSUME tells the assembler what names have been chosen for Code, Data Extra and Stack segments. Informs the assembler that the register CS is to be initialized with the address allotted by the loader to the label CODE and DS is similarly initialized with the address of label DATA.

Example:

ASSUME CS: Name of code segment

ASSUME DS: Name of the data segment

ASSUME CS: Code1, DS: Data1

ASSUME ES: Extra, SS: Stack

✓ Assume states the assembler that the segment given by the name such as: Data or Data1, Code, Stack or Extra are belongs to DS, CS, SS and ES segments, respectively.

✓ Remember: We need to initialize the corresponding address i.e. the segment or base address of the segment using segment registers.

✓ To Do So: We should start writing the instruction from the label 'START :'. Now START: is the fixed label, the name can't be changed and the segment registers need to be initialized here.

A Complete Program

- *Here is a short program that demonstrates the use directives: ALP to add two 16-bit number.*

```
Data    Segment
        A       DW      5489H
        B       DW      2418H
        Sum     DW      ?
        Carry   DB      00H
        Data    Ends

Code    Segment
        Assume CS: Code, DS: Data
        Start:  MOV     AX,     Data
                MOV     DS,     AX
                MOV     AX,     A
                ADD     AX,     B
                JNC     SKIP
                INC     Carry

        SKIP:   MOV     Sum,    AX

                Code    Ends
                        End     Start
```

- *Write an assembly language program to find the length of the given string.*

A Complete Program

```
CODE    SEGMENT
        ASSUME CS : CODE, DS : DATA
Start:  MOV     AX, DATA
        MOV     DS,  AX

        MOV     AL,  '$'
        MOV     CX, 00H
        MOV     SI,   OFFSET  STR1
BACK :  CMP     AL,  [SI]
        JE      GO
        INC          CL
        INC          SI
        JMP          BACK
GO :    MOV     LENGTH,     CL
        HLT
        CODE    ENDS
```

```
DATA      SEGMENT
          STR1       DB 'STUDENT BOX OFFICE$'
          LENGTH     DB          ?
DATA      ENDS

          End        Start
```

A Complete Program: Practice

- *Write an Assembly Language Program (ALP) to perform addition of two ASCII bytes.*
- *Write an ALP to perform division of 16-Bit unsigned numbers by an 8-bit unsigned number.*
- *ALP o perform division of two signed numbers.*
- *Write an assembly language program to reverse the given string*
- *Write an assembly level program to print a given string.*
- *Write an ALP to add two 8-bit BCD numbers.*

Assembler Directives

- **MACROS**

✓ Macros provide a means by which a block of text (*code*, *data* etc.) can be represented by a name (called a *macro name*).

✓ When the assembler encounters that name later in your program, the block of text associated with the macro name is substituted. The process is referred to as *macro expansion*.

✓ In assembly language, macros can be defined with MACRO and ENDM directives. The macro text begins with the MACRO directive and ends with the ENDM directive. The macro definition syntax is:

```
macro_name        MACRO        parameter1 , parameter2, .. .
                  macro body
                  ENDM
```

✓ In the MACRO directive, the parameters are optional. macro_name is the name of the macro that, when used later in the program, causes a macro expansion. To invoke or call a macro, use the macro_name and supply the necessary parameter values. The format is

```
macro_name    argument1 , argument2, .. .
```

- *Here is a short program that demonstrates the use MACRO: ALP to add two 16-bit number.*

A Complete MACRO Program

```
ADDITION        MACRO           NO1, NO2, RESULT
                MOV             AX,       NO1
                MOV             BX,       NO2
                ADD             AX,       BX
                MOV             RESULT, AX
                ENDM

ASSUME CS:CODE,         DS:DATA

                DATA SEGMENT
                NUM1            DW        1000H
                NUM2            DW        2000H
                RES             DW        ?
                DATA            ENDS

CODE SEGMENT

        START:
                MOV     AX,     DATA ; initialize data segment
                MOV     DS,     AX
                ADDITION NUM1, NUM2, RES ; Macro is invoked here.
                MOV     AH,     4CH
                INT     21H
                CODE            ENDS
                                END       START
```

Assembler Directives

- **Procedure**
  - ✓ A procedure is a logically self-contained unit of code designed to perform a particular task. These are sometimes referred to as *subprograms* and play an important role in modular program development.
  - ✓ Procedures also receive a list of arguments just like functions. However, procedures, after performing their computation based on the values of the arguments, may return zero or more results back to the calling procedure.
  - ✓ Assemblers provide two directives to define procedures in the assembly language: PROC and ENDP. The PROC directive (stands for PROCedure) signals the beginning of a procedure, and ENDP (END Procedure) indicates the end of a procedure.
  - ✓ In addition, the PROC directive may optionally include NEAR or FAR to indicate whether the procedure is a NEAR procedure or a FAR procedure. The general format is: (When NEAR or FAR is not included in the PROC directive, the procedure definition defaults to NEAR procedure.)

```
proc-name       PROC    NEAR
           or
proc-name       PROC
```

A typical procedure definition is

```
proc-name       PROC
                <procedure body>
proc-name       ENDP
```

- *Here is a short program that demonstrates the use PROCEDURE: ALP to add and subtract numbers.*

Procedure Example program:

```
ASSUME CS:CODE, DS:DATA, SS:STACK_SEG

DATA SEGMENT
        NUM1     DB       50H
        NUM2     DB       20H
        ADD_RES DB        ?
        SUB_RES DB        ?
        DATA              ENDS

STACK_SEG SEGMENT

DW      40      DUP(0) ; stack of 40 words, all initialized to zero
        TOS LABEL WORD
STACK_SEG       ENDS

CODE SEGMENT

START:  MOV      AX,      DATA ; initialize data segment
        MOV      DS,      AX
        MOV AX, STACK_SEG ; initialize stack segment
        MOV      SS,      AX
        MOV SP, OFFSET TOS ; initialize stack pointer to TOS
        CALL     ADDITION
CALL      SUBTRACTION
MOV       AH,       4CH
          INT       21H

ADDITION            PROC      NEAR
MOV       AL,                 NUM1
MOV       BL,                 NUM2
ADD       AL,                 BL
MOV       ADD_RES,            AL
          RET
ADDITION            ENDP

SUBTRACTION         PROC
MOV        AL,                NUM1
MOV       BL,                 NUM2
SUB       AL,                 BL
MOV       SUB_RES,            AL
          RET
SUBTRACTION         ENDP

CODE                ENDS
                    END       START
```

Macros and Procedures:

Macros are similar to procedures in some respects, yet are quite different in many other respects.

Procedure:

- ✓ Only one copy exists in memory. Thus memory consumed is less. "Called" when required;
- ✓ Execution time overhead is present because of the call and return instructions.

Macro:

- ✓ When a macro is "invoked", the corresponding text is "inserted" in to the source. Thus multiple copies exist in the memory leading to greater space requirements.
- ✓ However, there is no execution overhead because there are no additional call and return instructions. The code is in-place.

Assembler Directive

- ✓ DB (DEFINE BYTE)
- ✓ DD (DEFINE DOUBLE WORD)
- ✓ DQ (DEFINE QUADWORD)
- ✓ DT (DEFINE TEN BYTES)
- ✓ DW (DEFINE WORD)
- ✓ Segment
- ✓ ENDS (END SEGMENT)
- ✓ END (END PROCEDURE)
- ✓ ALIGN
- ✓ ASSUME
- ✓ EQU (EQUATE)
- ✓ ORG (ORIGIN)
- ✓ PROC (PROCEDURE)
- ✓ ENDP (END PROCEDURE)
- ✓ EXTRN
- ✓ PUBLIC
- ✓ LENGTH
- ✓ OFFSET
- ✓ PTR (POINTER)
- ✓ EVEN (ALIGN ON EVEN MEMORY ADDRESS)
- ✓ NAME
- ✓ LABEL
- ✓ STACK_SEG SEGMENT STACK
- ✓ SHORT
- ✓ TYPE
- ✓ GLOBAL (DECLARE SYMBOLS AS PUBLIC OR EXTRN)
- ✓ INCLUDE (INCLUDE SOURCE CODE FROM FILE)

Few Programs for Practice

https://www.youtube.com/watch?v=IjiNDnR_vn8
https://www.youtube.com/watch?v=BjI9Ypz09RA
More on Strings…..

- *Two memory location r1 and r2 store 07h and 3fh respectively. exchange the values in these memory location without using the exchange instruction*

```
DATA    SEGMENT
        R1      DB      07H
        R2      DB      3FH
DATA ENDS

CODE    SEGMENT
 ASSUME CS: CODE, DS:DATA

 START:  MOV     AX, DATA    ; Initialize the data segment register
         MOV     DS, AX
         MOV     AL,R1           ; R1 is move the data to Al
         MOV     BL,R2           ; R2 is move the data to BL
         MOV     R1,BL           ; BL is transfer the data to R1
         MOV     R2,AL           ; AL is transfer the data to R2
CODE ENDS
```

Two memory location r1 and r2 store 07h and 3fh respectively. Swap the values in these memory location with using the exchange instruction

```
DATA    SEGMENT
        R1      DB      07H
        R2      DB      3FH
DATA ENDS

CODE SEGMENT
 ASSUME CS: CODE, DS:DATA
 START:  MOV    AX,  DATA    ;Initialize the data segment register
         MOV    DS, AX              ; AX is move the data to DS
         MOV    AL,R1              ;R1 is move the data to AL
         XCGH   AL,R2    ;R2 and AL is exchange the data using XCGH
         MOV    R1,AL              ; AL is transfer the data to R1
CODE ENDS
         END    Start
```

Two memory location r1 and r2 store 07h and 3fh respectively. exchange the values in these memory location with using the indirect addressing modes

```
DATA        SEGMENT
            R1     DB     07H
            R2     DB    3FH
DATA ENDS

CODE SEGMENT
            ASSUME CS: CODE, DS:DATA
 START:     MOV        AX, DATA       ;Initialize the data segment register
            MOV        DS, AX
            LEA        SI, R1          ;Offset address of R1 in SI
            LEA        DI, R2           ;Offset address of R1 in DI
            MOV        AL, [SI]        ;[SI] R1 is transfer the data to AL
            MOV        BL, [DI]         ;[DI] R2 is transfer the data to BL
            MOV        [SI], BL        ;Bl is transfer the data to [SI] R1
            MOV        [DI], AL          ;Al is transfer the data to [DI] R2
CODE ENDS
            END        START
```

compare the two value are A and B, show the three condition are: equal to, a is greater b and a is lesser than b. write the program and display the messages through the screen

```
ORG 100h
.MODEL SMALL
.DATA
        NUM_1 DB 23H
        NUM_2 DB 93H
        MSG   DB "Equal Numbers$"
        MSG1  DB "Number 1 is less than Number 2$"
        MSG2  DB "Number 1 is greater than Number 2$"

.CODE
MOV AX, @DATA
MOV DS, AX

MOV AH,NUM_1
MOV CH,NUM_2
CMP AH, CH
JE L1  ;If AH and CH are equal
JB L2  ;If AH is less than CH
JA L3  ;If AH is greater than CH

L1:
MOV DX,OFFSET MSG
MOV AH,09H
INT 21H
RET

L2:
MOV DX,OFFSET MSG1
MOV AH,09H
INT 21H
RET

L3:
MOV DX,OFFSET MSG2
MOV AH,09H
INT 21H
RET
```

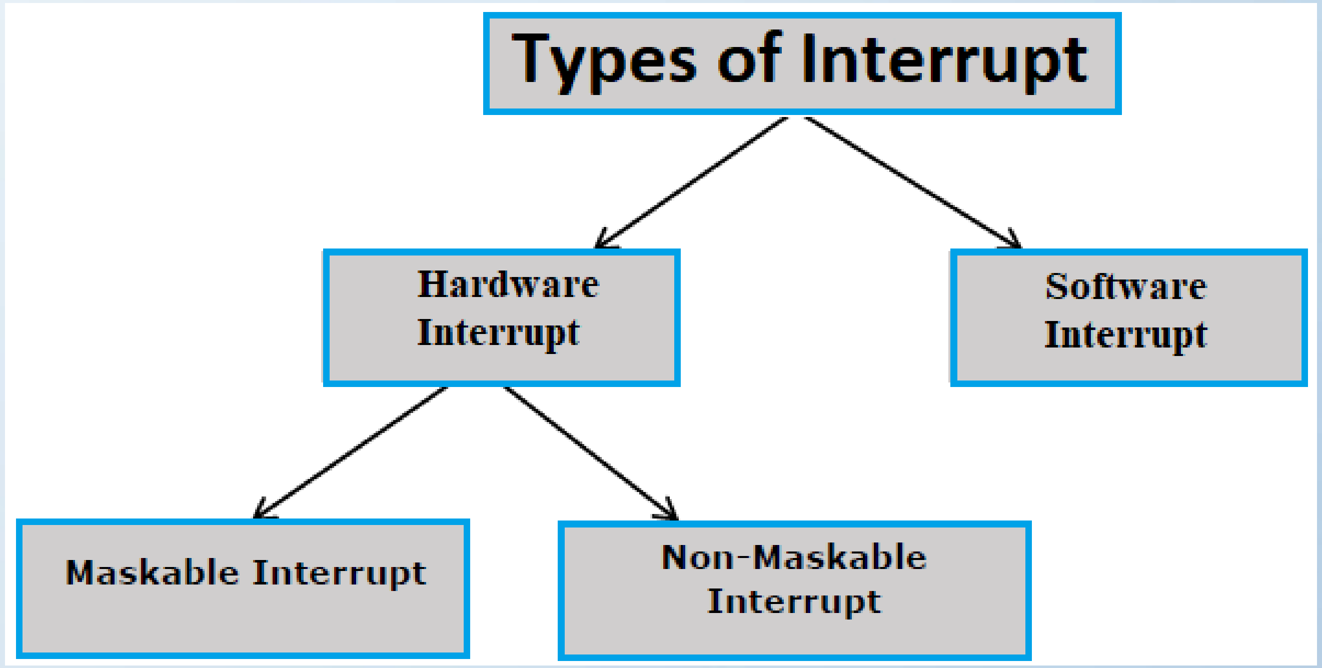
A string "MY NAME IS" is stored in memory as MES. The name in 14 characters is to be stored along with MES, to make it a 25 character string to look like: MY NAME IS "your name".

```
DATA        SEGMENT
            MES         DB          "MY NAME IS"            ; 11 character string
            NAME1       DB           "PRASHANT RISHI"       ; 14 characters
            MES1        DB          25 DUP (0) ;declare an array of 25 bytes :each byte initialized to 0
DATA ENDS

CODE SEGMENT
            ASSUME CS: CODE, DS: DATA, ES: DATA ; Initialize Data Segment and Extra Segment registers

START:      MOV AX, DATA
            MOV DS, AX
            MOV ES, AX
            LEA SI, MES                 ; Point SI to Source String
            LEA DI, MES1                ; Point DI to Destination String
            MOV CX, 11                  ; move 11 to CX Counter Register
            CLD; Clear DF to auto increment SI and DI
REP MOVSB ; Move NAME1 String to MES1. DI already points to address inMES1 where NAME1 ; must start
            LEA          SI, NAME1
            MOV         CX, 14
CLD
            REP MOVSB
CODE ENDS
```

8086 μP: Interrupts and Interrupt Service Routines (ISR)

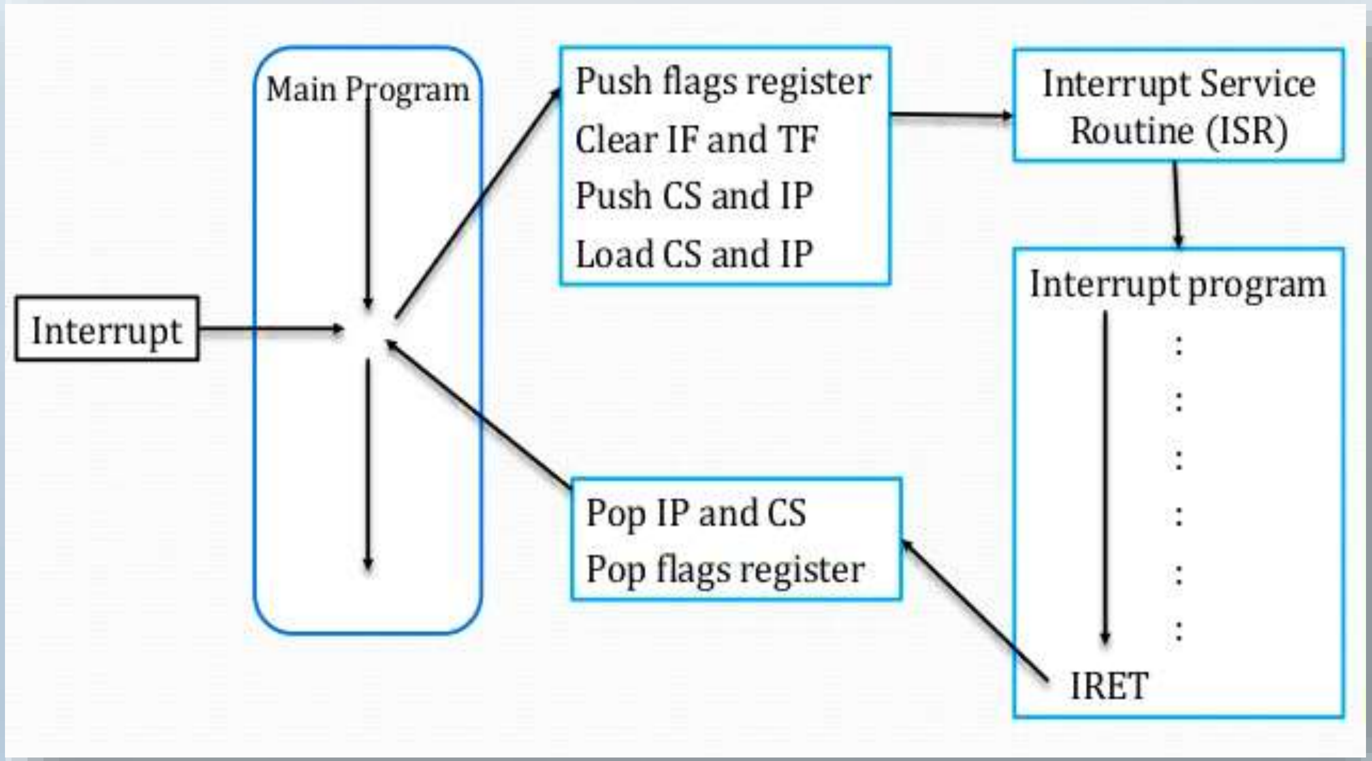
8086 μP Interrupts

- **Interrupt** is the method of creating a temporary halt during program execution and allows peripheral devices to access the microprocessor.
- The microprocessor responds to that interrupt with an ISR (Interrupt Service Routine), which is a short program to instruct the microprocessor on how to handle the interrupt.
- 2 types:
  - ✓ Software Interrupt (Triggered by a software instruction)
  - ✓ Hardware Interrupt (Triggered by an external hardware module)
    - ✓ Maskable
    - ✓ Non-Maskable



How Interrupt Works?

- ✓ There are 256 interrupts from $INT\ 0$ to $INT\ 255$ each with specific address.
- ✓ While executing the program, if μP encounters an interrupt; it will go the Interrupt Service Routine (ISR) by changing the address of instruction pointer (IP).
- ✓ Since ISR address is fixed based on $INT\ n$ (interrupt number), μP knows what address to be put in IP. However, it doesn't know where to come back or the return address.



- ✓ Thus the μP does the following actions:

i. Complete the current instruction, that is in progress.

ii. Pushes the return address into the stack (i.e. the address of the next instruction cycle).

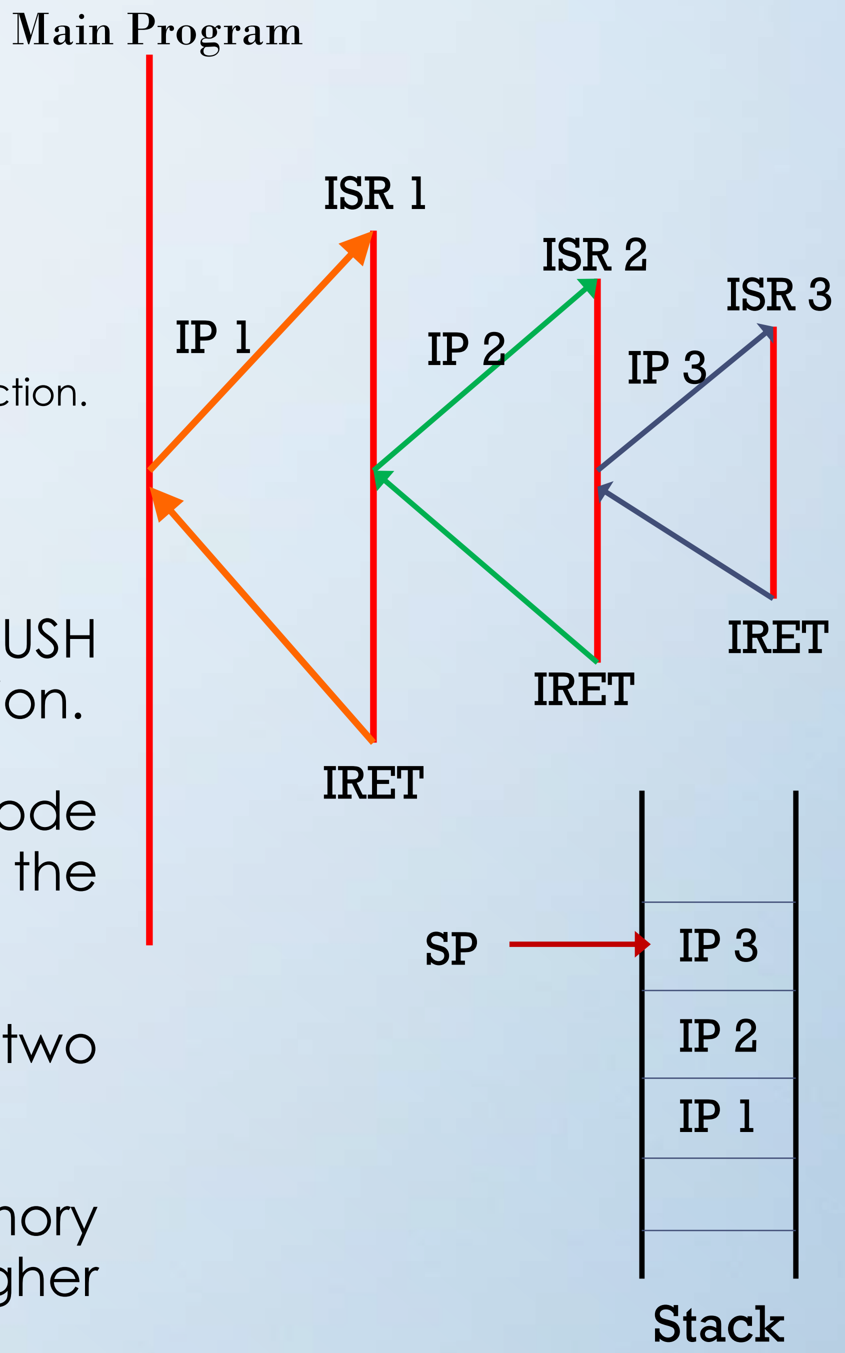
iii. After completing the ISR μP pops the address stored in the stack and return to the next instruction.

How Interrupt Works?

```
; To Server ISR and to locate ISR address.
PUSH        IP1
PUSH        IP2
PUSH        IP3
; To return back to the main program's next instruction.
POP         IP3
POP         IP2
POP         IP1
```

NOTE:

- ✓ Don't mess the stack. Remember the number od PUSH operations must be equal to the number of POP operation.
- ✓ The main program and ISR are not in the same code segment. Thus not only offset address (IP), but also the segment address (CS) will change.
- ✓ Both CS and IP are 16-bit registers thus require two consecutive locations in stack to store them.
- ✓ The lower byte will be stored in the lower address memory location and the higher byte will be stored in the higher address location.

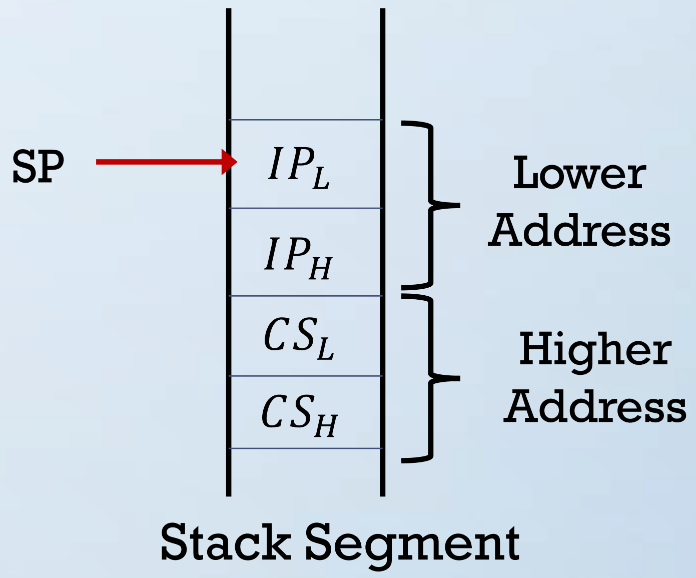


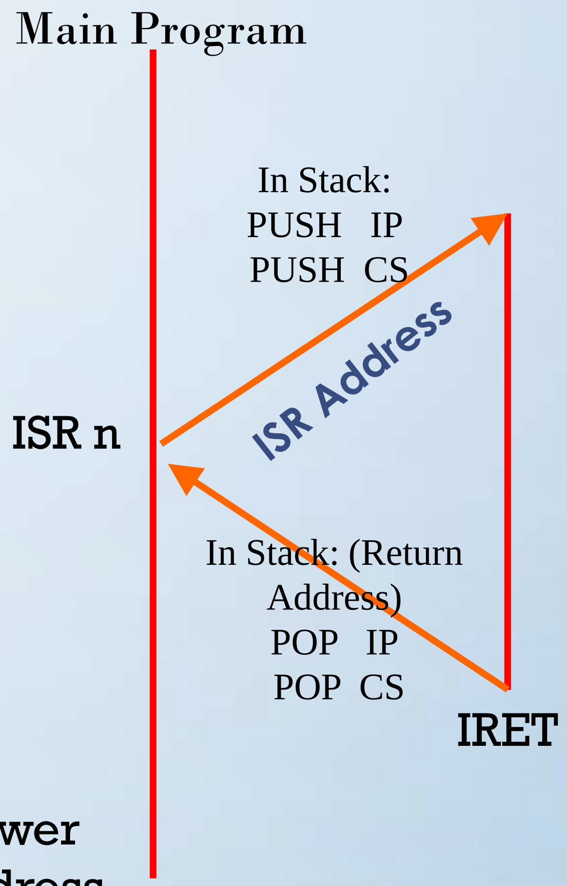
How Interrupt Works?

NOTE:

- ✓ Between CS and IP;
  - ✓ IP will be stored in the lower address location of stack
  - ✓ CS will be stored in the higher address location of stack.

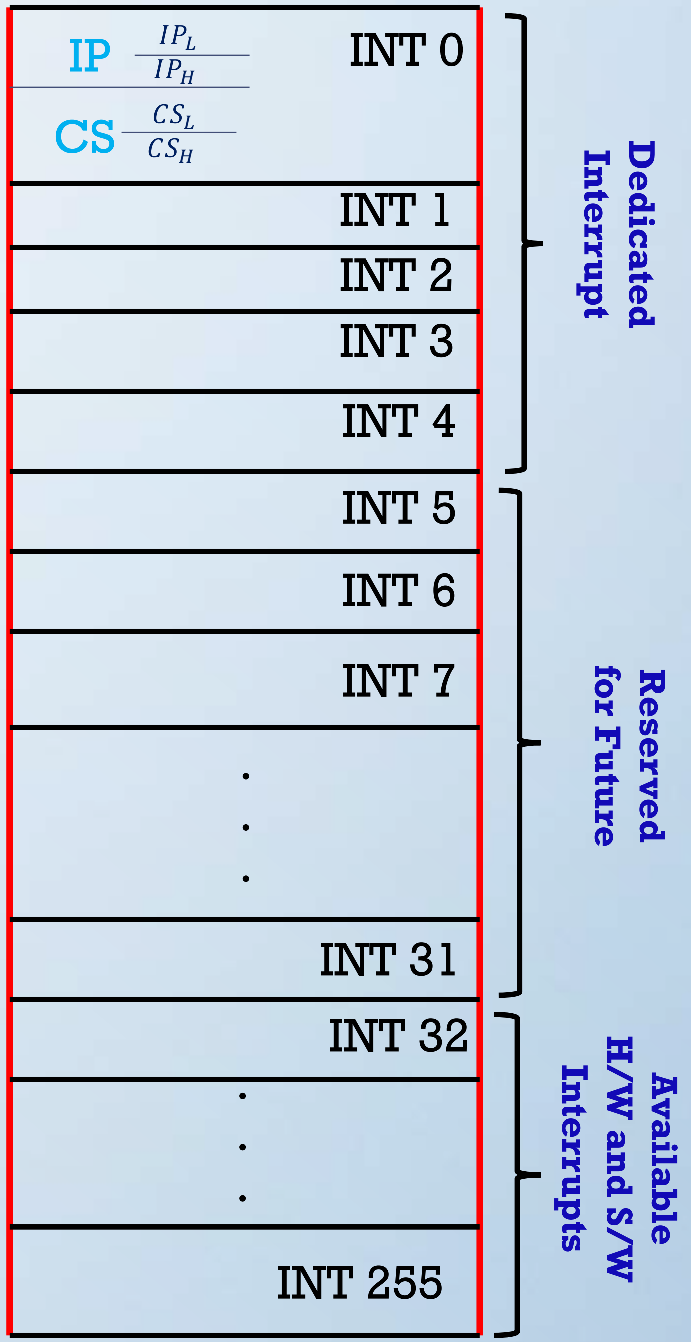
```
; To Server ISR and to locate ISR address.
PUSH        CS
PUSH        IP
; To return back to the main program's next instruction.
POP         IP
POP         CS
```





Interrupt Vector Table (IVT)

- ✓ There are 256 interrupts in 8086 μP. Each interrupt has different address to serve the ISR. The *interrupt vector table (IVT) stores the address of ISR for each interrupt.*
- ✓ 256 Interrupts: 256 Address in IVT (formed by CS and IP)
- ✓ CS and IP each of 16-bits; thus requires 4-bytes of memory locations to store.
- ✓ In total, the size of IVT = $256\ Interrupts \times 4\ address = 1KB$.
- ✓ Moreover, the IVT is in memory and its address is fixed.
- ✓ INT 0 to INT 4: Dedicated Interrupt
- ✓ INT 5 to INT 31: Reserved for future
- ✓ INT 32 to INT 255: Available H/W or S/W Interrupt.



IVT 8086

Interrupt Vector Table (IVT)

- ✓ **EX**: Consider interrupt 1 (INT 1) has occurred and the ISR address is given as follows:

ISR Address = 456789 H

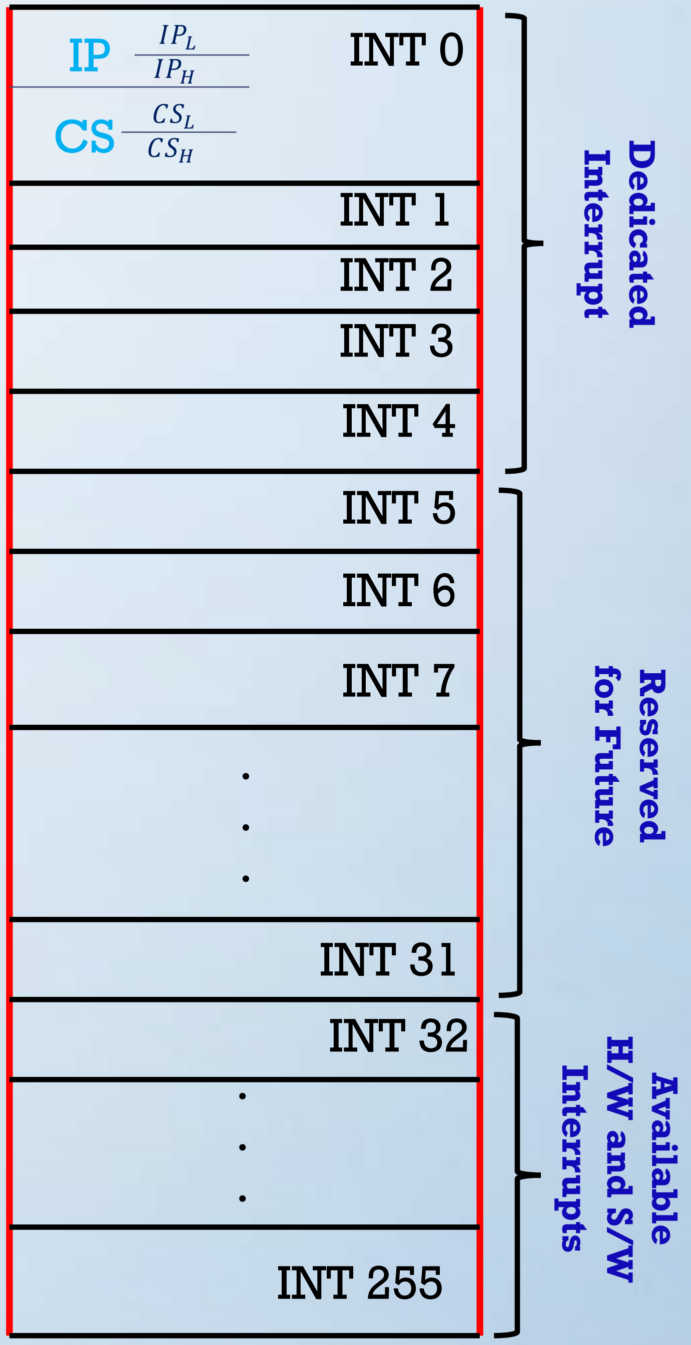
Then find out the location of INT 1 and the content in those location.

Approach:

INT 1 Location	*In general for n^{th} interrupt*
$1 \times 4 + 0$	$n \times 4 + 0$
$1 \times 4 + 1$	$n \times 4 + 1$
$1 \times 4 + 2$	$n \times 4 + 2$
$1 \times 4 + 3$	$n \times 4 + 3$

ISR Address is CS = 5000 H and IP = 6789 H

Location	Content
00004 H	89 (IP_L)
00005 H	67 (IP_H)
00006 H	00 (CS_L)
00007 H	50 (CS_H)



IVT 8086

8086 Microprocessor – Software Interrupts

- Some instructions are inserted at the desired position into the program to create interrupts. These interrupt instructions can be used to test the working of various interrupt handlers. It includes –
- **INT- Interrupt instruction with type number**

  It is 2-byte instruction. First byte provides the op-code and the second byte provides the interrupt type number. There are 256 interrupt types under this group.

  When an interrupt is activated, following actions take place –

  1. Completes the current instruction that is in progress.
  2. Pushes the Flag register values on to the stack.
  3. Pushes the CS (code segment) value and IP (instruction pointer) value of the return address on to the stack.
  4. IP is loaded from the contents of the word location **(int type*4) H**.
  5. CS is loaded from the contents of the next word location **(int type*4)+2 H**.
  6. Set Interrupt flag and trap flag.

8086 Microprocessor – Software Interrupts

Execution of the INT instruction 'INT n' (n = 0 to 255) may be used to cause the 8086 to branch to any of the 256 interrupt types mentioned in the interrupt vector table. For example, INT 40 will branch to ISR specified by Vector/Type 40 (memory location 160 = A0H) in the interrupt vector table. One of the possible uses of this generalized software interrupt caused by 'INT n' instruction is to test the different Interrupt Servicing Routines. For example, 'INT 0' causes the execution of ISR of the Type/Vector 0, i.e. Divide-by Zero interrupt. Similarly, NMI ISR may be tested by the INT 2 instruction.

Type 0 : INT 0 (Divide by zero)

- The 8086 will automatically do a type 0 interrupt if the result of a DIV operation or an IDIV operation is too large to fit in the destination register.
- For a type 0 interrupt, the 8086 pushes the flag register on the stack, resets IF and TF and pushes the return addresses on the stack.

8086 Microprocessor – Software Interrupts

Type 1 : INT 1 (Single Stepping)

- The 8086 will enter into single step mode and will stop after it executes each instruction and wait for further direction from user.
- The use of single step execution feature is found in some of the monitor & debugger programs. It will execute one instruction and stop, then the contents of registers and memory locations can be examined.
- When TF (Trap Flag) is set, the 8086 executes the Single Step Mode Interrupt after the execution of each instruction. The Single Step Mode Interrupt occurs at Type/Vector 1 in the interrupt vector table. The 8086 takes the following steps when a Single Step Interrupt occurs.
  1. The flag register contents are pushed onto the stack. The stack pointer is decremented by 2.
  2. The flag register contents are cleared. This disables the maskable interrupt INTR and single step logic during the execution of ISR.
  3. The CS (Code Segment) register contents are pushed onto the stack. The stack pointer is decremented by 2.
  4. The IP (Instruction Pointer) register contents are pushed onto the stack. The stack pointer is decremented by 2.
  5. The new CS register and IP register contents are taken from Type/Vector 1 of the interrupt vector table.

8086 Microprocessor – Hardware Interrupts

Type 2 : NMI

- It is a single non-maskable interrupt pin (NMI) having higher priority than the maskable interrupt request pin (INTR).

INTR (hardware interrupt)

- The INTR interrupt is activated by an I/O port. The 8086 INTR input allows some external signal to interrupt execution of a program
- Unlike the NMI input, however, INTR can be masked so that it cannot cause an interrupt. If the interrupt flag is cleared, then the INTR input is disabled.

8086 Microprocessor – Software Interrupts

Type 3: INT 3 (Break Point Interrupt Instruction)

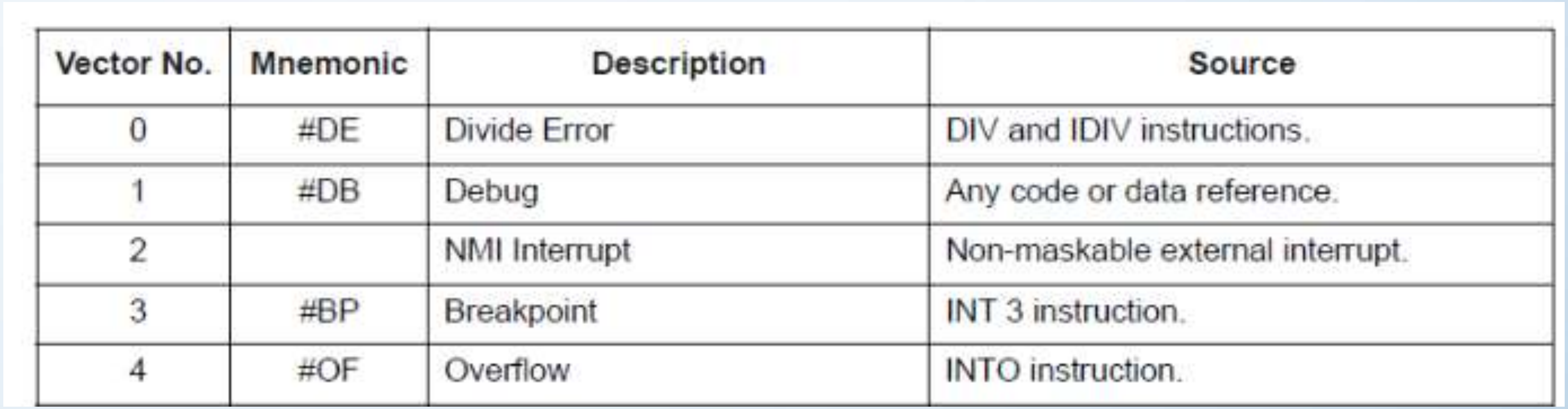
- The main use of the type 3 interrupt is to implement a breakpoint function in a system.
- Whenever a breakpoint is inserted, the system executes the instructions up to the breakpoint and then goes to the breakpoint procedure.
- It is a 1-byte instruction having op-code is CCH. These instructions are inserted into the program so that when the processor reaches there, then it stops the normal execution of program and follows the break-point procedure.
  1. Its execution includes the following steps –
  2. Flag register value is pushed on to the stack.
  3. CS value of the return address and IP value of the return address are pushed on to the stack.
  4. IP is loaded from the contents of the word location 3×4 = 0000CH
  5. CS is loaded from the contents of the next word location.
  6. Interrupt Flag and Trap Flag are reset to 0

8086 Microprocessor – Software Interrupts

Type 4: INT 4 (Interrupt on overflow instruction)

- The 8086 overflow flag will be set if the signed result of an arithmetic operation on two signed numbers is too large to be represented in the destination register or memory location.
- It is a 1-byte instruction and their mnemonic INTO. The op-code for this instruction is CEH. As the name suggests it is a conditional interrupt instruction, i.e. it is active only when the overflow flag is set to 1 and branches to the interrupt handler whose interrupt type number is 4. If the overflow flag is reset then, the execution continues to the next
- Its execution includes the following steps –
  1. Flag register values are pushed on to the stack.
  2. CS value of the return address and IP value of the return address are pushed on to the stack.
  3. IP is loaded from the contents of word location 4×4 = 00010H
  4. CS is loaded from the contents of the next word location.
  5. Interrupt flag and Trap flag are reset to 0
- Example: Addition of 8 bit signed number 01101100 and the 8 bit signed number 010111101

8086 Microprocessor – interrupts



Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.

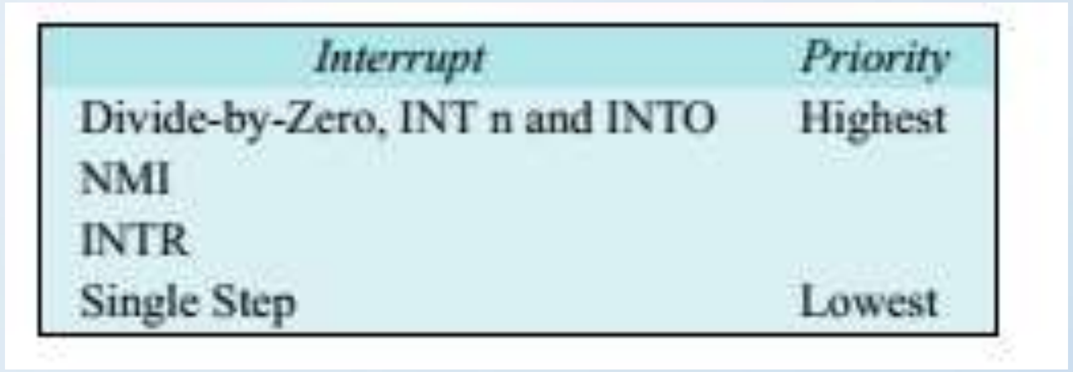
8086 Interrupt Priority

Interrupt Priority

- If two or more interrupts occur at the same time then the highest priority interrupt will be serviced first, and then the next highest priority interrupt will be serviced.
- The interrupt that has a lower address, has a higher priority. **(INT 0>INT 1>INT 2..)**

Example:

- If suppose that the INTR input is enabled, the 8086 receives an INTR signal during the execution of a divide instruction, and the divide operation produces a divide by zero interrupt **(INT 0)**.
- Since the internal interrupts-such as divide error, INT 0 and INT 4 have higher priority than INTR the 8086 will do a divide error interrupt response first.



Interrupt	*Priority*
Divide-by-Zero, INT n and INTO	Highest
NMI	
INTR	
Single Step	Lowest