Intel 8086 MICROPROCESSOR ARCHITECTURE

Features

- ***It is a 16-bit µp.***
- ***8086 has a 20 bit address bus can access up to 2^{20} memory locations (1 MB).***
- ***It can support up to 64K I/O ports.***
- ***It provides 14, 16 -bit registers.***
- ***Word size is 16 bits and double word size is 4 bytes.***
- ***It has multiplexed address and data bus AD0- AD15 and A16 – A19.***

- ***8086 is designed to operate in two modes, Minimum and Maximum.***
- ***It can prefetches up to 6 instruction bytes from memory and queues them in order to speed up instruction execution.***
- ***It requires +5V power supply.***
- ***A 40 pin dual in line package.***
- ***Address ranges from 00000H to FFFFFH***

Intel 8086 Internal Architecture

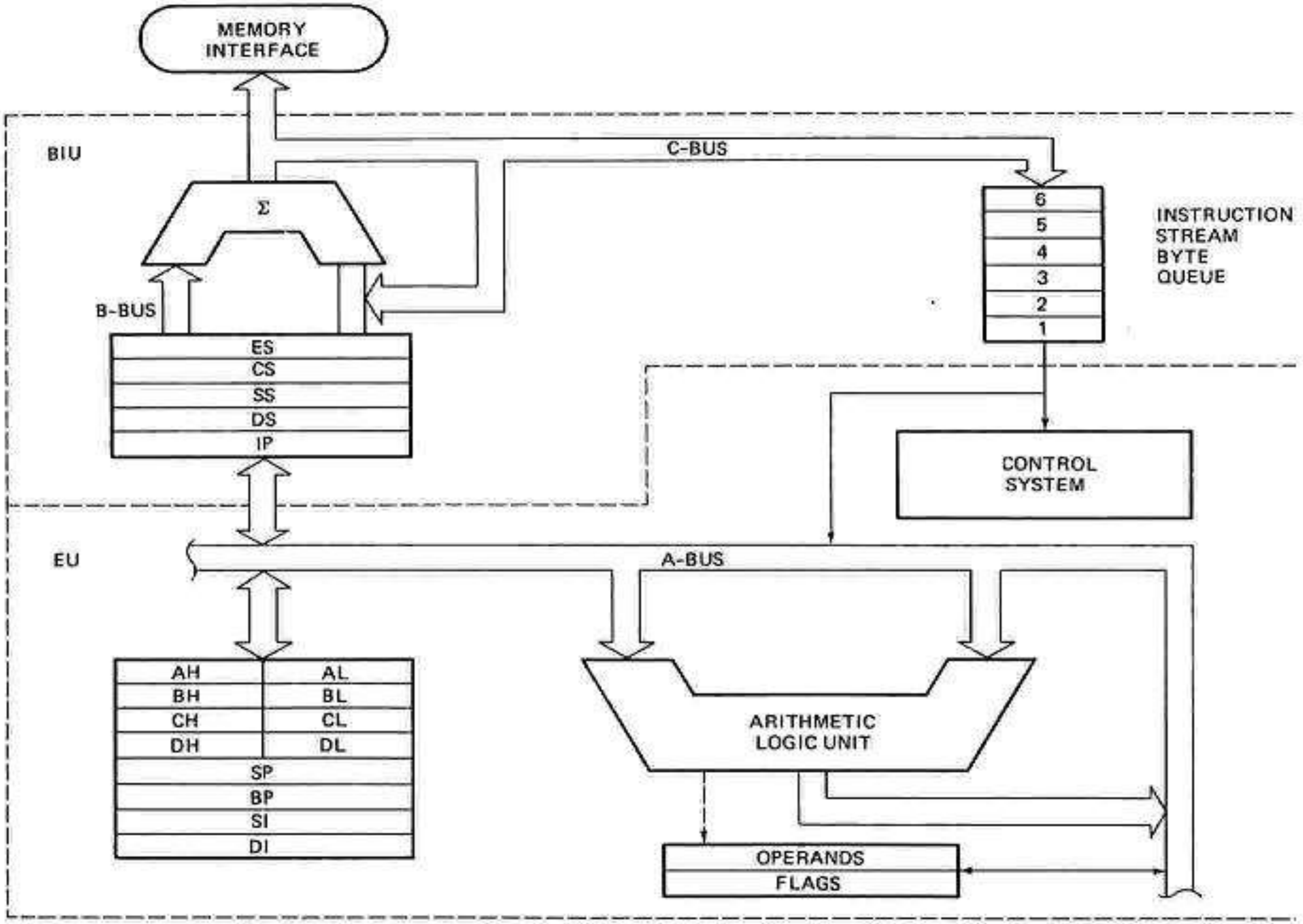


FIGURE 8086 internal block diagram. (*Intel Corp.*)

Internal architecture of 8086

- 8086 has two blocks BIU and EU.
- The BIU handles all transactions of data and addresses on the buses for EU.
- The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating the addresses of the memory operands. The instruction bytes are transferred to the instruction queue.
- EU executes instructions from the instruction system byte queue.

- BIU contains
  Instruction queue,
  Segment registers,
  Instruction pointer,
  Address adder.

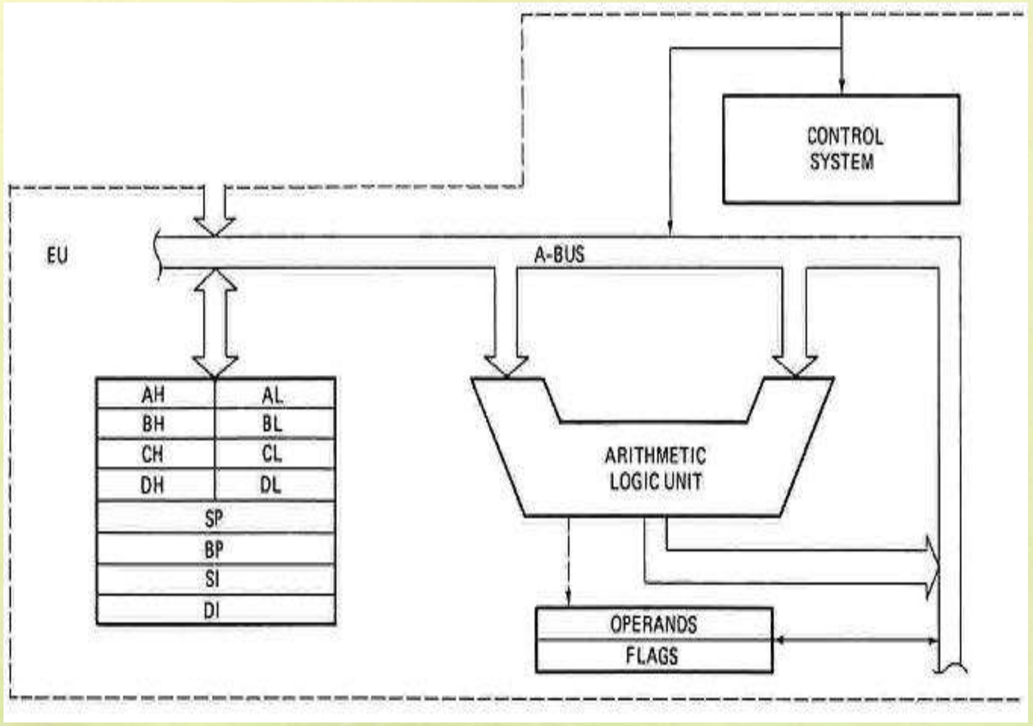
- EU contains
  Control circuitry,
  Instruction decoder,
  ALU,
  Pointer and Index register,
  Flag register.

EXECUTION UNIT

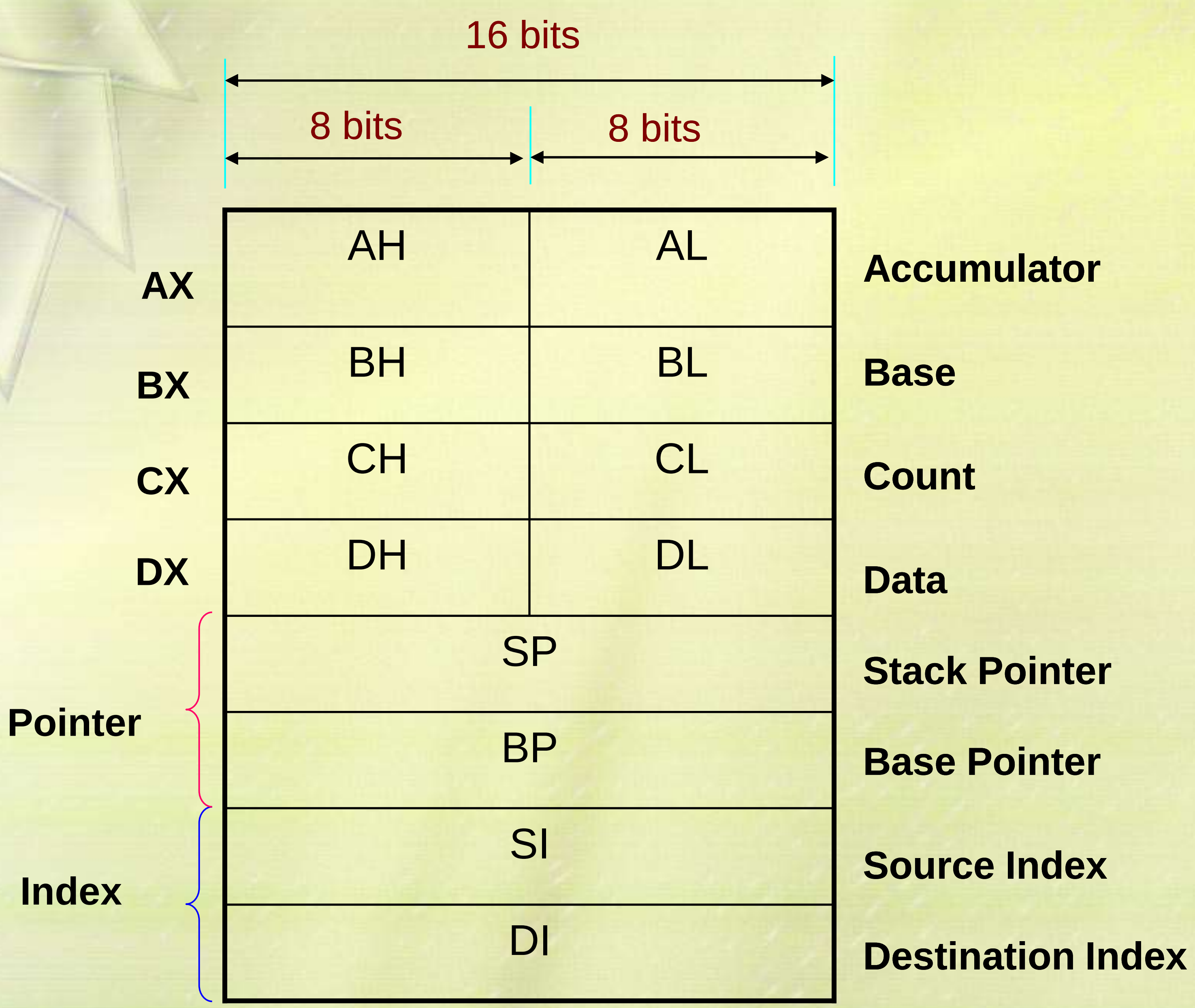
- Decodes instructions fetched by the BIU
- Generate control signals,
- Executes instructions.

The main parts are:

- Control Circuitry
- Instruction decoder
- ALU



EXECUTION UNIT – General Purpose Registers



EXECUTION UNIT – General Purpose Registers

Register	Purpose
AX	Word multiply, word divide, word I /O
AL	Byte multiply, byte divide, byte I/O, decimal arithmetic
AH	Byte multiply, byte divide
BX	Store address information
CX	String operation, loops
CL	Variable shift and rotate
DX	Word multiply, word divide, indirect I/O (Used to hold I/O address during I/O instructions. If the result is more than 16-bits, the lower order 16-bits are stored in accumulator and higher order 16-bits are stored in DX register)

Pointer And Index Registers

- used to keep offset addresses.
- Used in various forms of memory addressing.
- In the case of SP and BP the default reference to form a physical address is the Stack Segment (SS-will be discussed under the BIU)
- The index registers (SI & DI) and the BX generally default to the Data segment register (DS).

SP: Stack pointer

- Used with SS to access the stack segment

BP: Base Pointer

- Primarily used to access data on the stack
- Can be used to access data in other segments

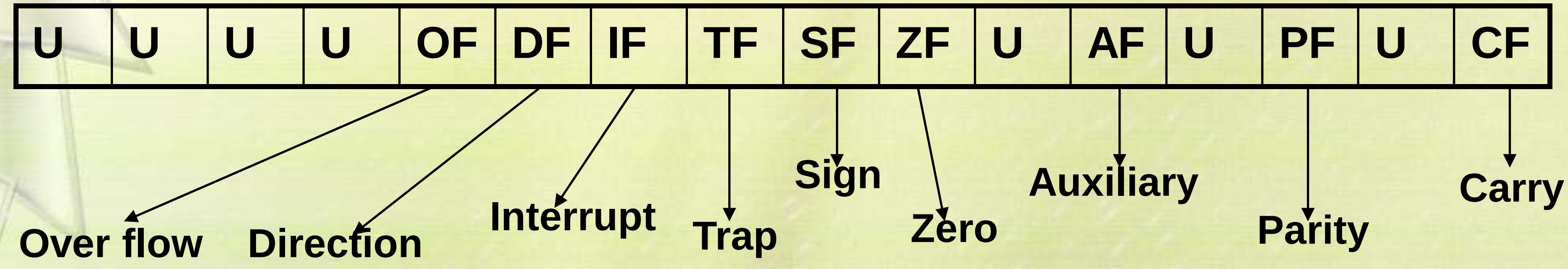
- SI: Source Index register
  - is required for some string operations
  - When string operations are performed, the SI register points to memory locations in the data segment which is addressed by the DS register. Thus, SI is associated with the DS in string operations.

- DI: Destination Index register
  - is also required for some string operations.
  - When string operations are performed, the DI register points to memory locations in the data segment which is addressed by the ES register. Thus, DI is associated with the ES in string operations.

- The SI and the DI registers may also be used to access data stored in arrays

EXECUTION UNIT – Flag Register

- A flag is a flip flop which indicates some conditions produced by the execution of an instruction or controls certain operations of the EU .
- In 8086 The EU contains
  - □ a 16 bit flag register
  - □9 of the 16 are active flags and remaining 7 are undefined.
    - □ 6 flags indicates some conditions- status flags
    - □3 flags –control Flags



U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

U - Unused

EXECUTION UNIT – Flag Register

Flag	Purpose
Carry (CF)	Holds the carry after addition or the borrow after subtraction. Also indicates some error conditions, as dictated by some programs and procedures .
Parity (PF)	PF=0;odd parity, PF=1;even parity.
Auxiliary (AF)	Holds the carry (half – carry) after addition or borrow after subtraction between bit positions 3 and 4 of the result (for example, in BCD addition or subtraction.)
Zero (ZF)	Shows the result of the arithmetic or logic operation. Z=1; result is zero. Z=0; The result is 0
Sign (SF)	Holds the sign of the result after an arithmetic/logic instruction execution. S=1; negative, S=0

Flag	Purpose
Trap (TF)	A control flag. Enables the trapping through an on-chip debugging feature.
Interrupt (IF)	A control flag. Controls the operation of the INTR (interrupt request) I=0; INTR pin disabled. I=1; INTR pin enabled.
Direction (DF)	A control flag. It selects either the increment or decrement mode for DI and /or SI registers during the string instructions.
Overflow (OF)	Overflow occurs when signed numbers are added or subtracted. An overflow indicates the result has exceeded the capacity of the Machine

Execution unit – Flag Register

- Six of the flags are status indicators reflecting properties of the last arithmetic or logical instruction.
- For example, if register AL = 7Fh and the instruction ADD AL,1 is executed then the following happen

AL = 80h

CF = 0; there is no carry out of bit 7

PF = 0; 80h has an odd number of ones

AF = 1; there is a carry out of bit 3 into bit 4

ZF = 0; the result is not zero

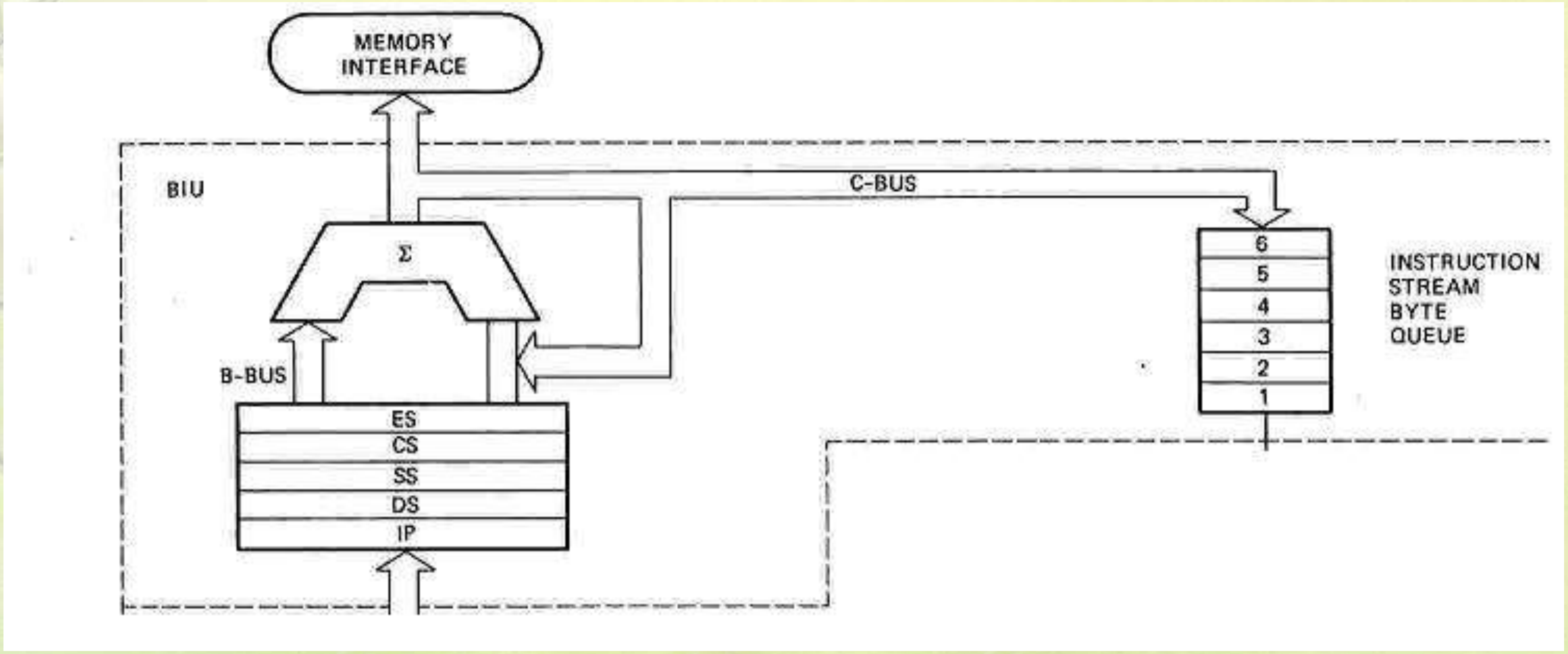
SF = 1; bit seven is one

OF = 1; the sign bit has changed

BUS INTERFACE UNIT (BIU)

Contains

- **6-byte Instruction Queue (Q)**
- **The Segment Registers (CS, DS, ES, SS).**
- **The Instruction Pointer (IP).**
- **The Address Summing block (Σ)**

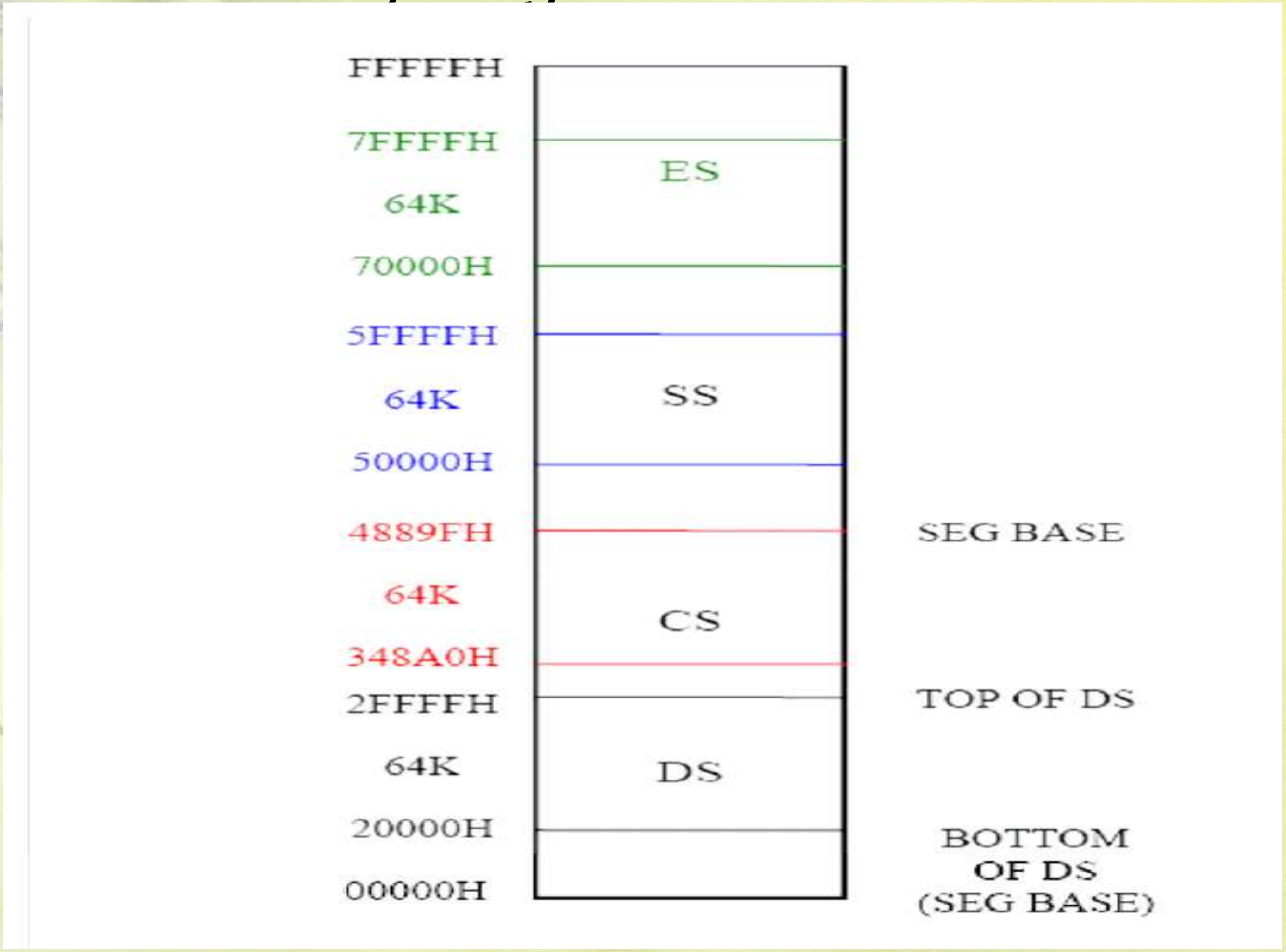


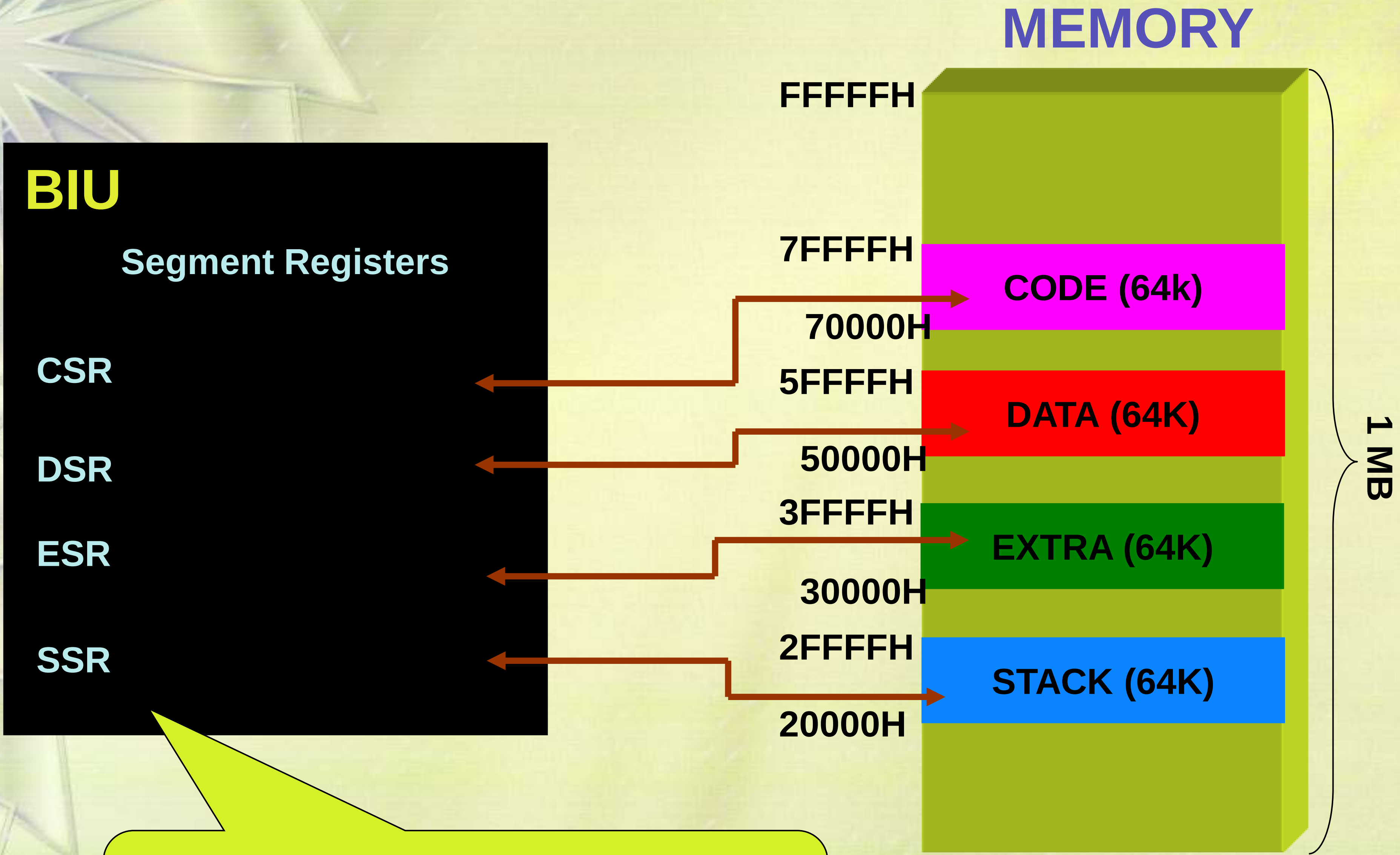
THE QUEUE (Q)

- The BIU uses a mechanism known as an **instruction stream queue** to implement a ***pipeline architecture.***

- This queue permits pre-fetch of up to **6 bytes** of instruction code. Whenever the queue of the BIU is not full, it has room for at least two more bytes and at the same time the EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by pre-fetching the next sequential instruction.

Memory Segmentation





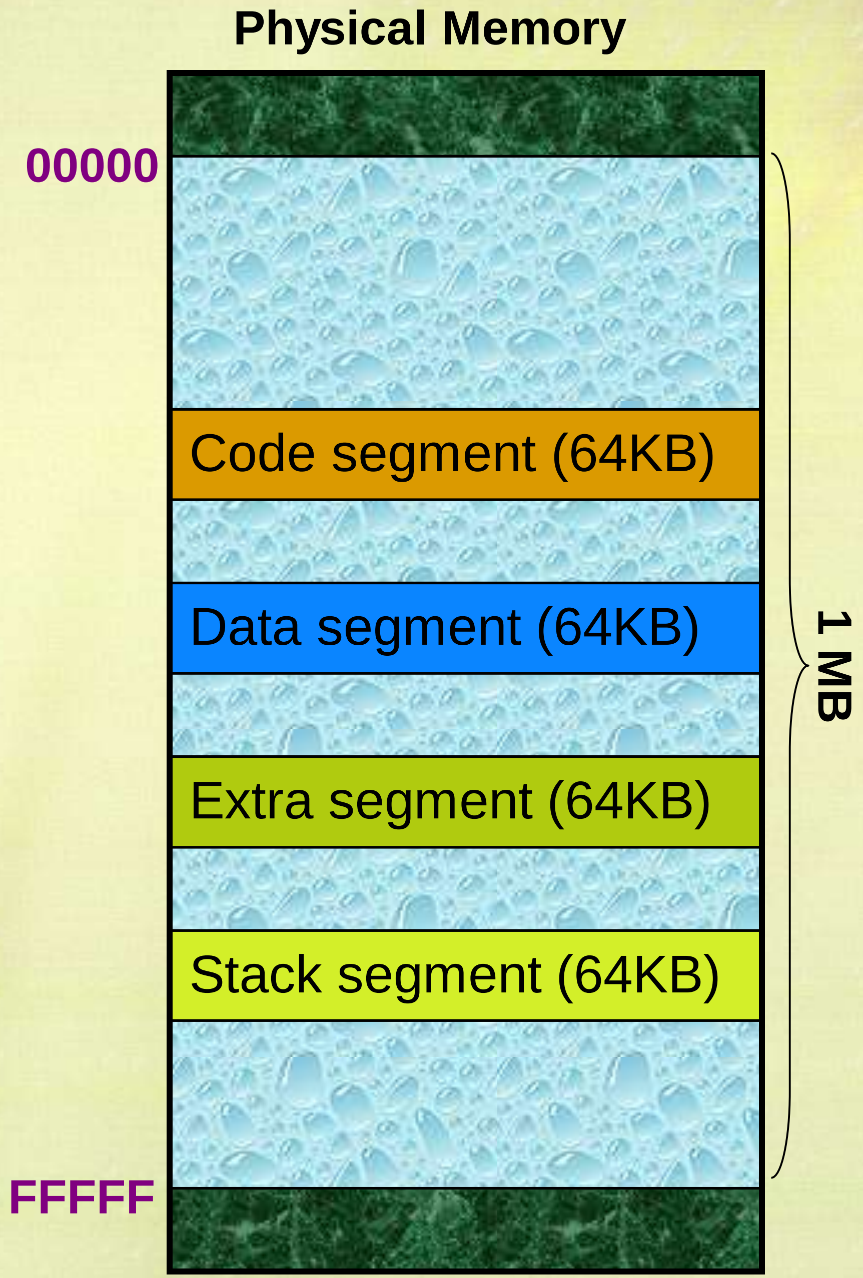
Each segment register store the upper 16 bit of the starting address of the segments

Segmented Memory

- The memory in an 8086/88 based system is organized as segmented memory.

- The CPU 8086 is able to address 1Mbyte of memory.

- The Complete physically available memory may be divided into a number of logical segments.



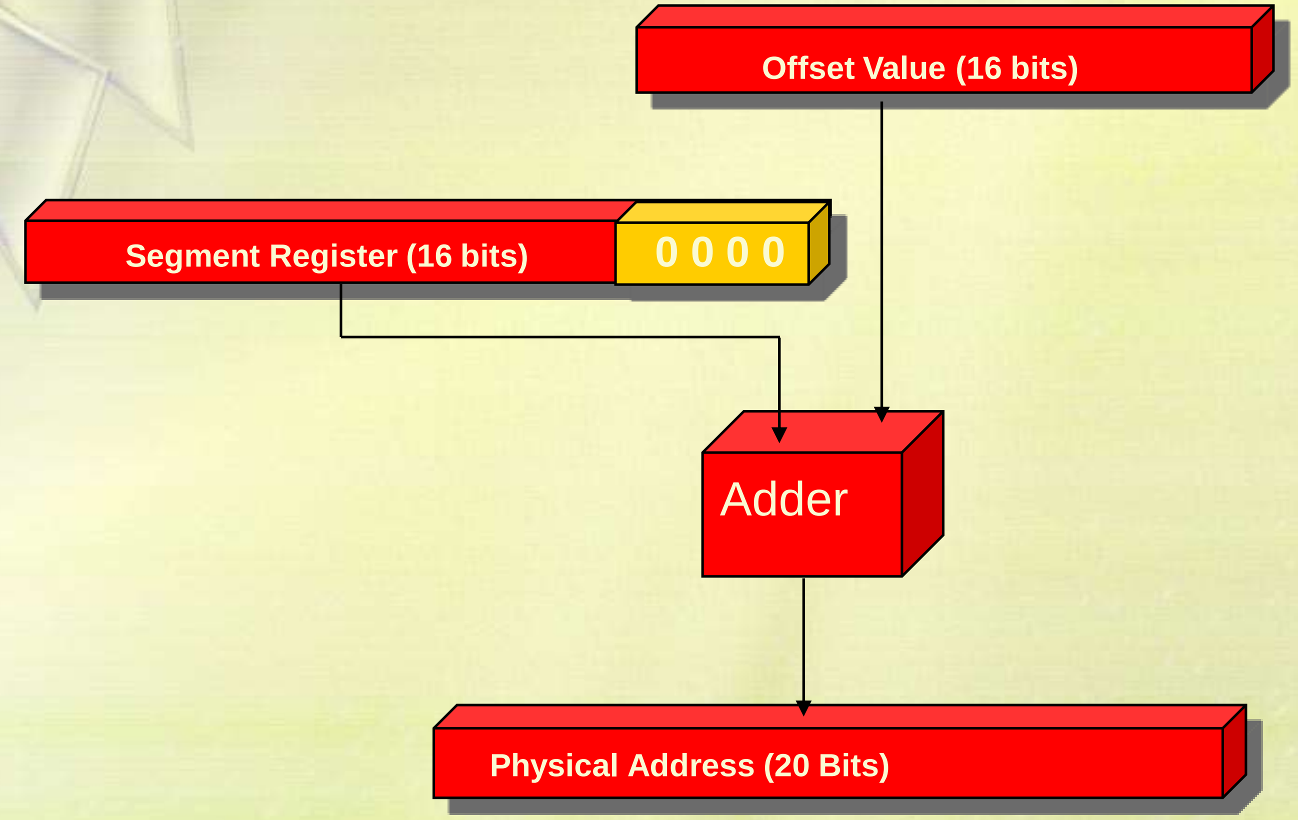
- The size of each segment is 64 KB
- A segment may be located any where in the memory
- Each of these segments can be used for a specific function.
  - Code segment is used for storing the instructions.
  - The stack segment is used as a stack and it is used to store the return addresses.
  - The data and extra segments are used for storing data byte.

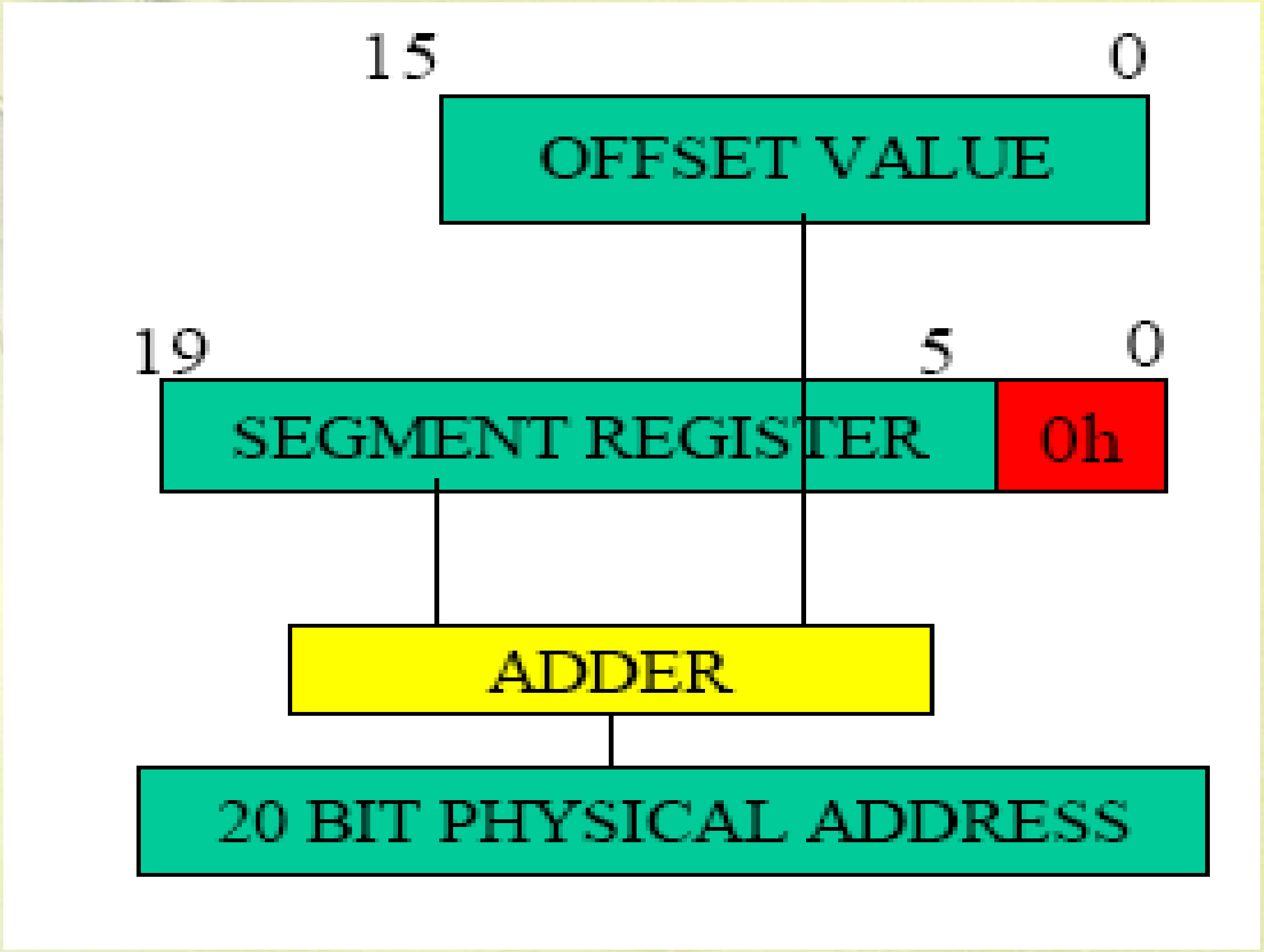
- The 4 segments are Code, Data, Extra and Stack segments.
- A Segment is a 64kbyte block of memory.
- The 16 bit contents of the segment registers in the BIU actually point to the starting location of a particular segment.
- Segments may be overlapped or non-overlapped

Segment registers

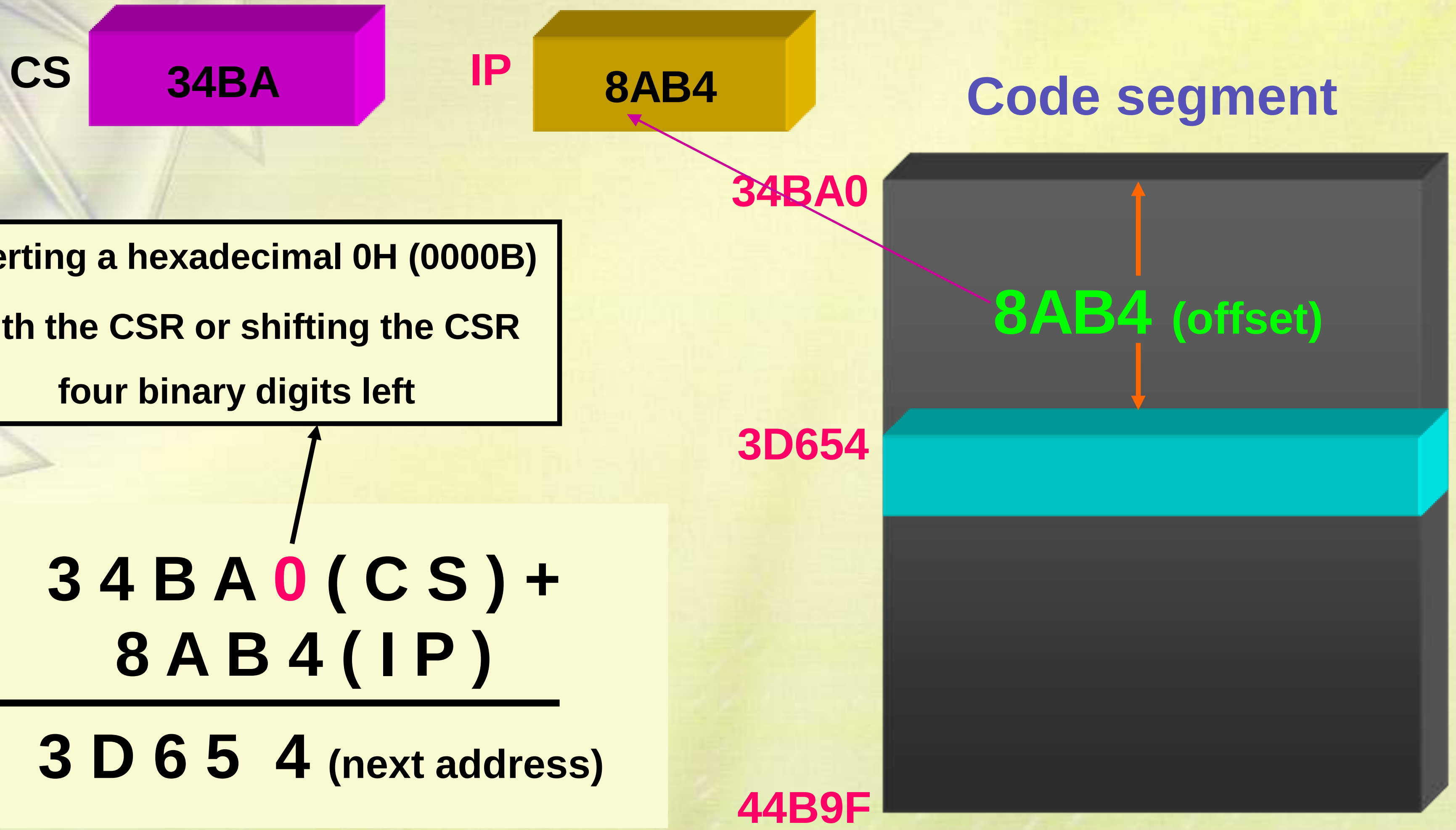
- In 8086/88 the processors have 4 segments registers
- Code Segment register (CS), Data Segment register (DS), Extra Segment register (ES) and Stack Segment (SS) register.
- All are 16 bit registers.
- Each of the Segment registers store the upper 16 bit address of the starting address of the corresponding segments.

Memory Address Generation




15
0
OFFSET VALUE
19
5
0
SEGMENT REGISTER
0h
ADDER
20 BIT PHYSICAL ADDRESS

- **The following examples shows the CS:IP scheme of address formation:**



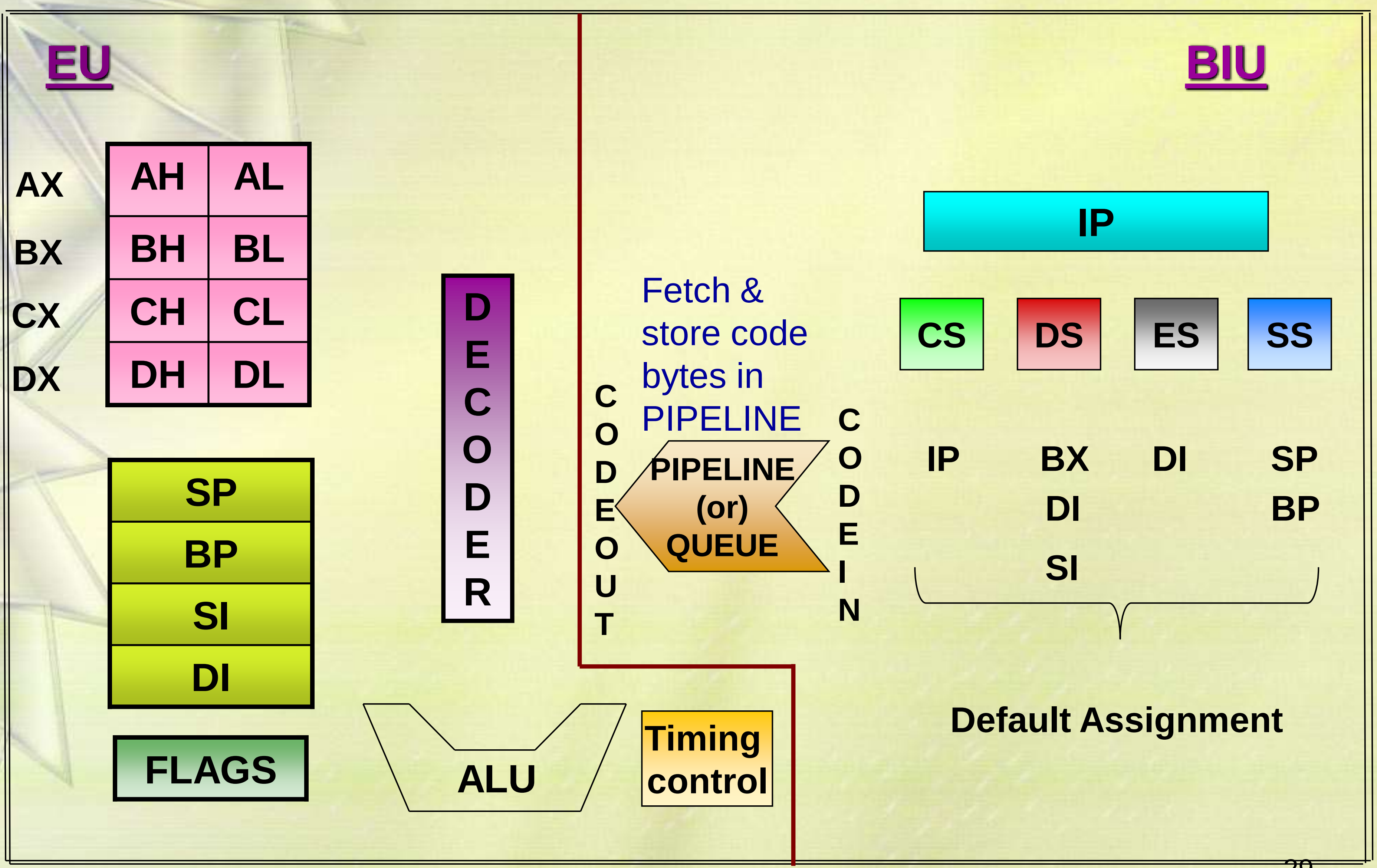
Inserting a hexadecimal 0H (0000B) with the CSR or shifting the CSR four binary digits left

```
3 4 B A 0 ( C S ) +
  8 A B 4 ( I P )
-------------------
3 D 6 5  4 (next address)
```

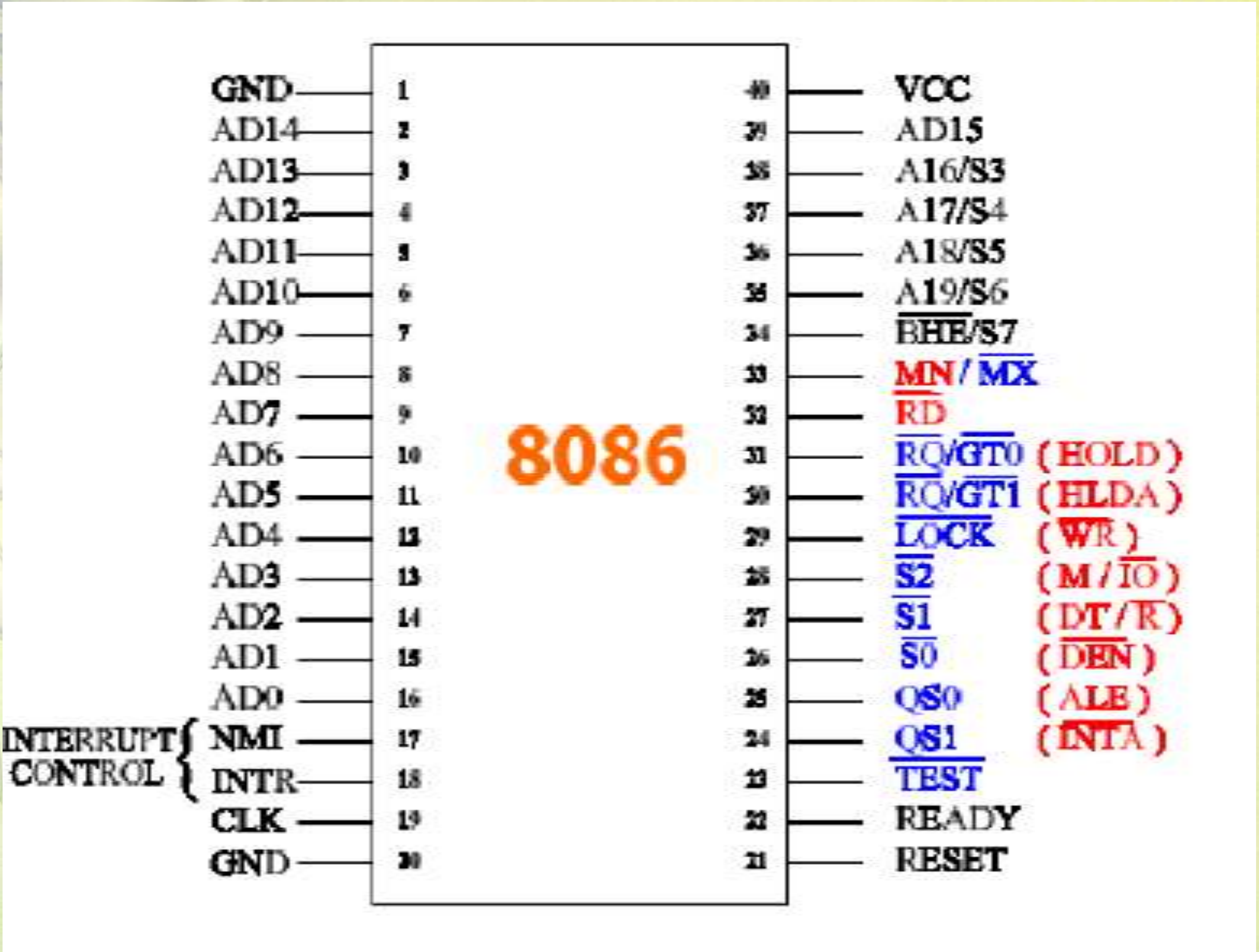
Segment and Address register combination

- CS:IP
- SS:SP SS:BP
- DS:BX DS:SI
- DS:DI (for other than string operations)
- ES:DI (for string operations)

Summary of Registers & Pipeline of 8086 µP



Pin Diagram of 8086



Q. Explain the addressing modes of 8086.

Solⁿ Addressing Modes of 8086:-

1. Immediate Addressing Mode
2. Direct Addressing Mode
3. Register Addressing Mode
4. Register Indirect Addressing Mode.
5. Indexed Addressing Mode.
6. Register Relative Addressing Mode
7. Based Indexed Addressing Mode
8. Relative Based Indexed Addressing Mode

1. Immediate Addressing Mode:

* Here Immediate data is a part of instruction
* Immediate data may be 8 Bit or 16 Bit in size

Eg.
```
MOV AX, 0005H
MOV AL, 56 H
```

2. Direct Addressing Mode:

* Memory Address (offset) is directly specified in the instruction.

Eg.
```
MOV AX, [5000H]
```

3. Register Addressing Mode:

* Here source & destination can be one of 8086 resisters.

Eg
```
MOV AX, BX ;   16 Bit data transfer
```

4. <u>Register Indirect Addressing Mode :-</u>

* The offset Address of data is in either Bx or SI or DI register.
* The default segment is either DS or ES

<u>Eg:</u> MOV Ax, [Bx]

5 <u>Indexed Addressing Mode :</u>

* Here Data is available is at an offset address stored on SI or DI.

<u>Eg</u> MOV Ax, [SI]

6 <u>Register Relative Addressing Mode</u>

* Here physical address is given as 10H * DS + 50 + Bx

<u>Eg</u> MOV Ax, 50 H [Bx]

7. <u>Base Indexed Addressing Mode</u>

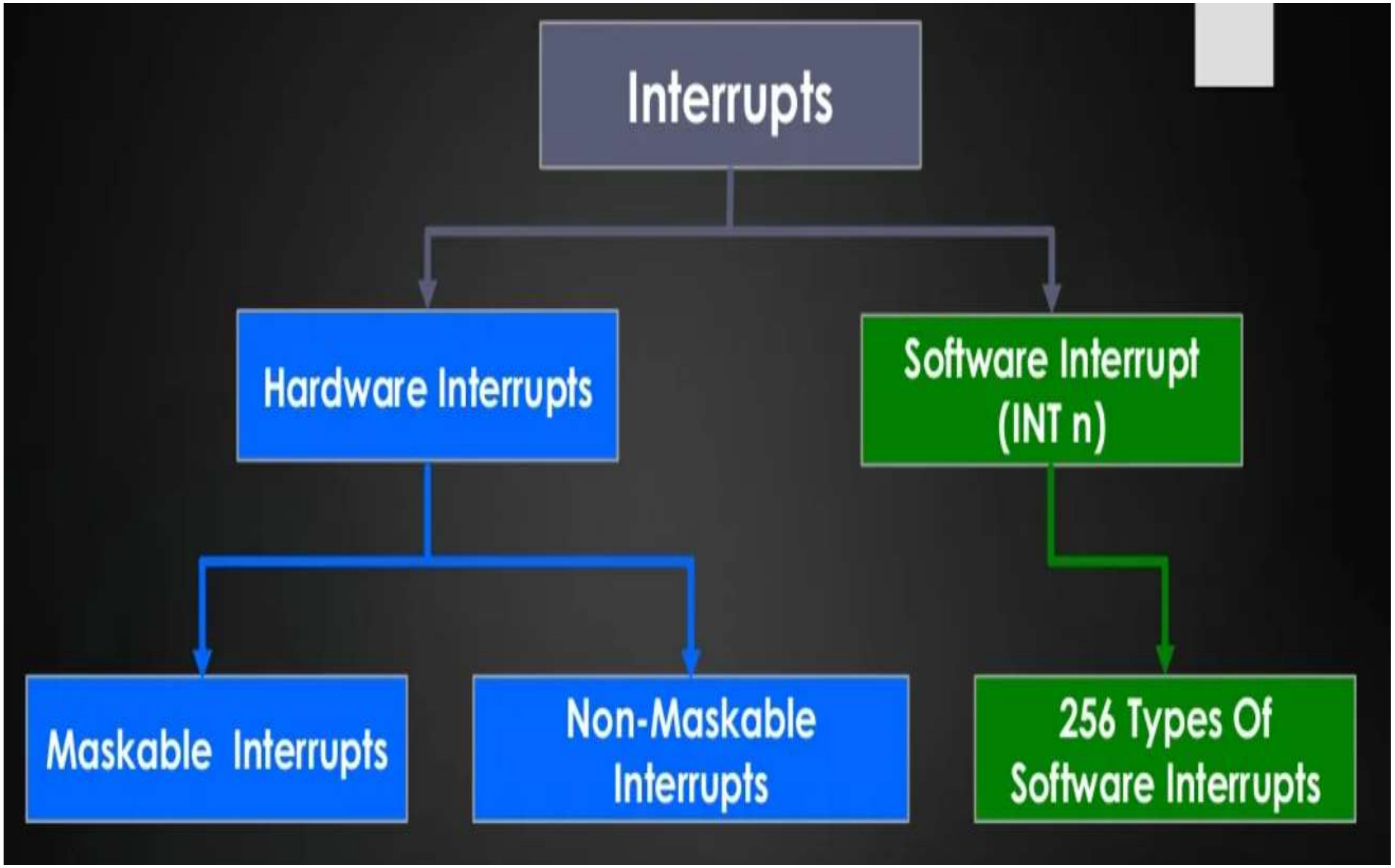
* Here Bx is base register and SI is index register.
* Physical Address is calculated as 10H * DS + [Bx] + [SI]

<u>Eg</u> MOV Ax, [SI][Bx]

8 <u>Relative Base Indexed Addressing Mode</u>

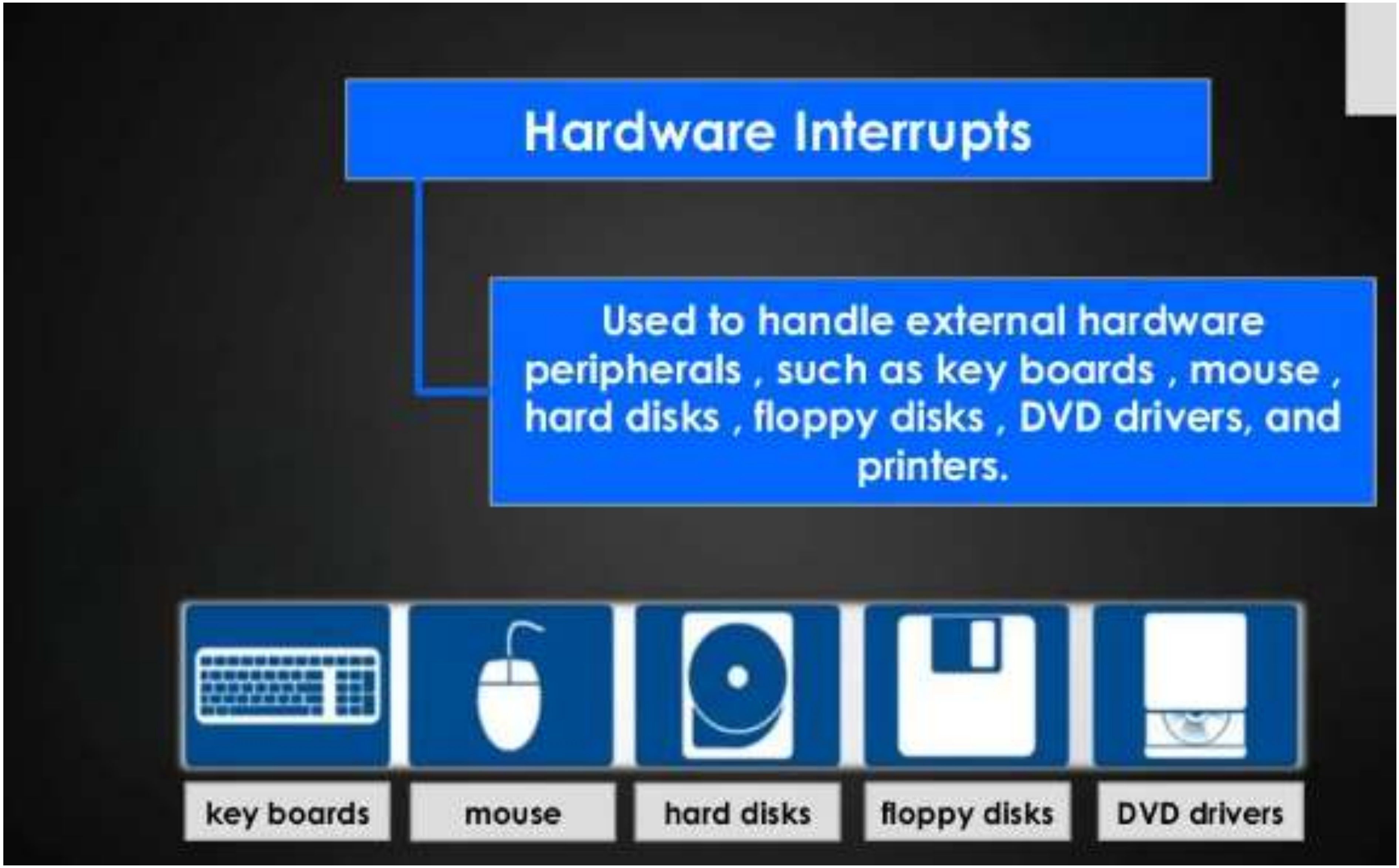
<u>Eg</u> MOV Ax, 50H [Bx] [SI]

8086 Interrupts


Interrupts
Hardware Interrupts
Software Interrupt (INT n)
Maskable Interrupts
Non-Maskable Interrupts
256 Types Of Software Interrupts

Hardware Interrupts

The interrupts initiated by external hardware by sending an appropriate signal to the interrupt pin of the processor is called hardware interrupt. The 8086 processor has two interrupt pins INTR and NMI. The interrupts initiated by applying appropriate signal to these pins are called hardware interrupts of 8086.



Hardware Interrupts

Used to handle external hardware peripherals , such as key boards , mouse , hard disks , floppy disks , DVD drivers, and printers.

key boards

mouse

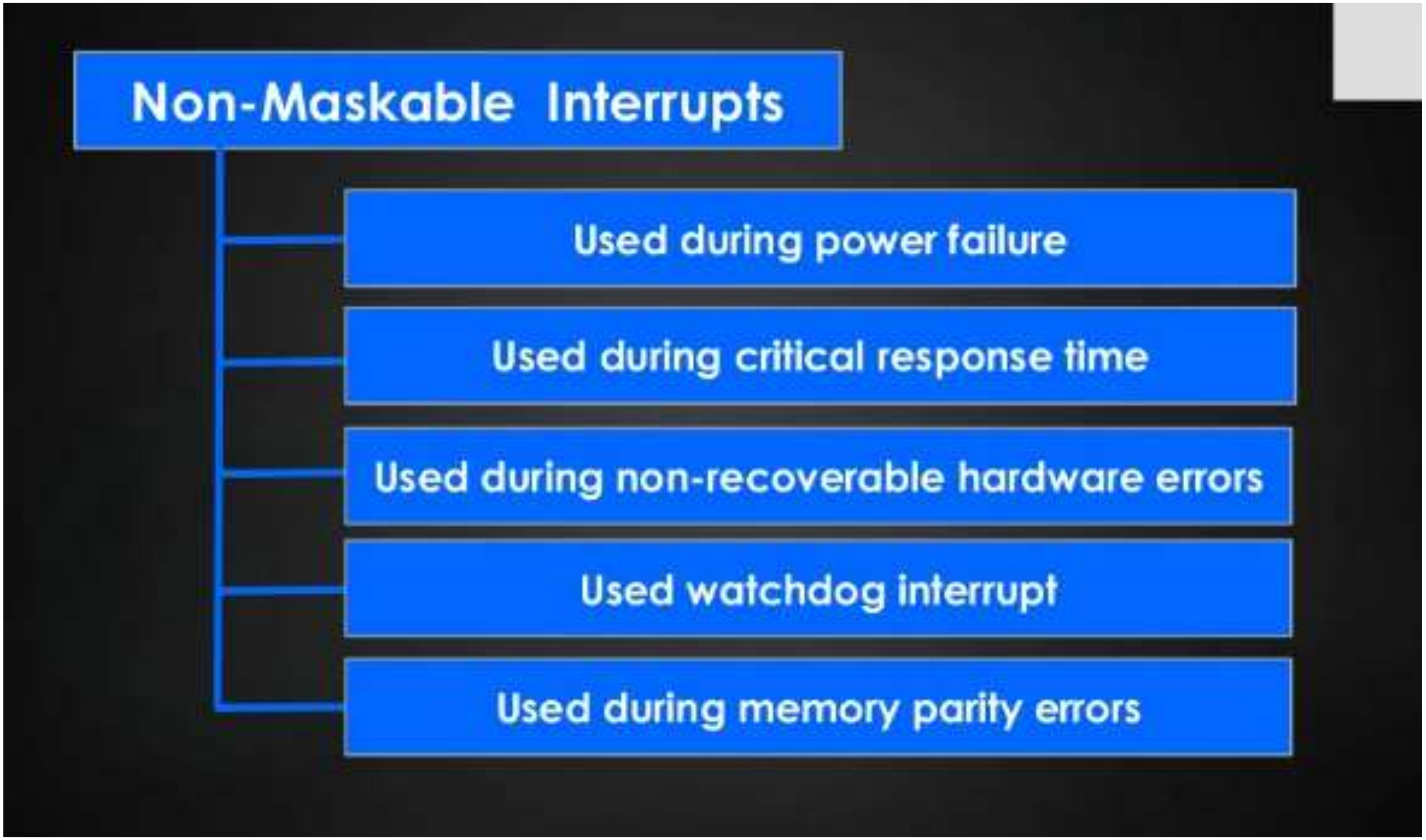
hard disks

floppy disks

DVD drivers

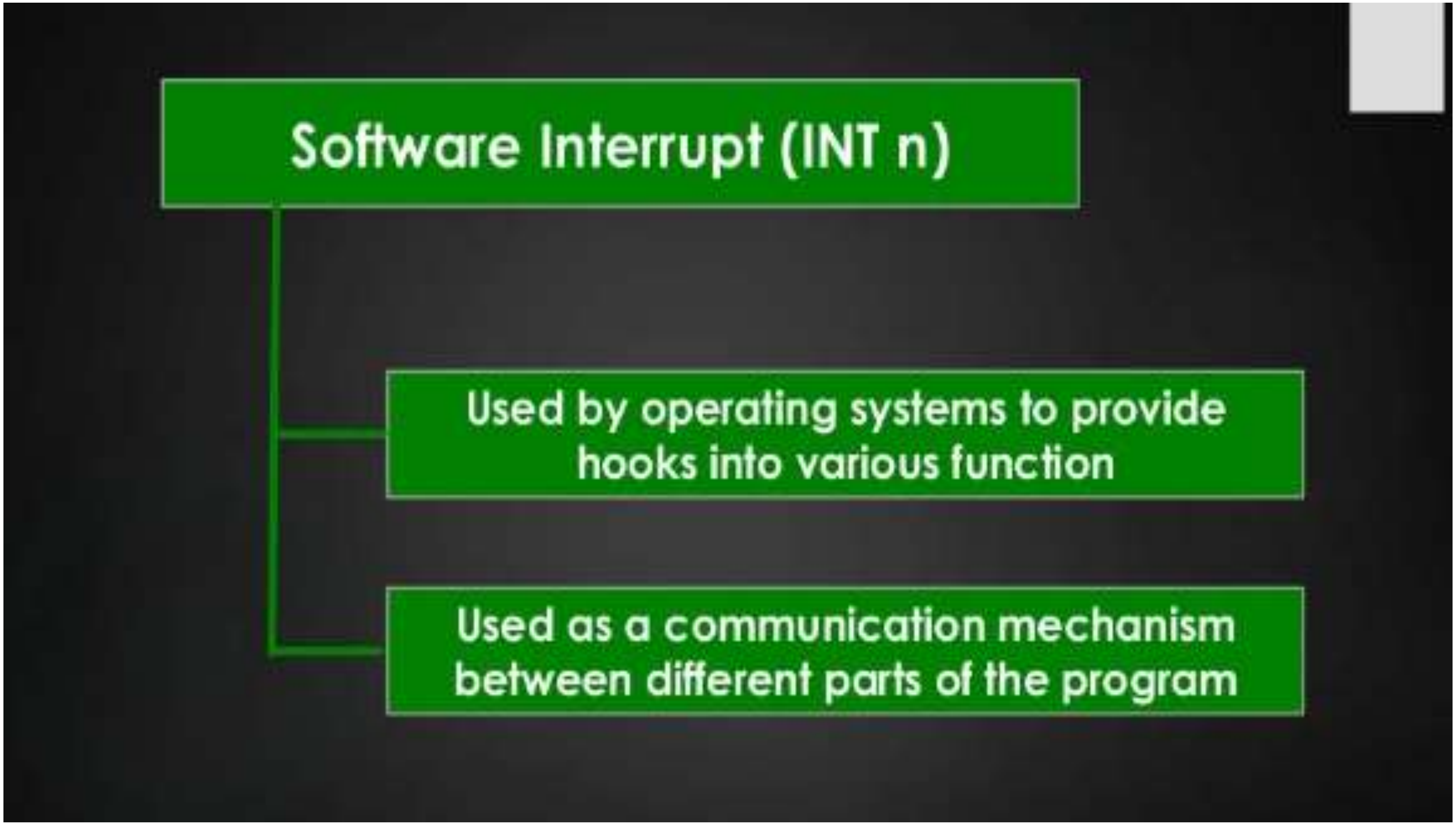
Maskable & Non-Maskable Interrupts

The processor has the facility for accepting or rejecting hardware interrupts. Programming the processor to reject an interrupt is referred to as masking or disabling and programming the processor to accept an interrupt is referred to as unmasking or enabling. In 8086 the interrupt flag (IF) can be set to one to unmask or enable all hardware interrupts and IF is cleared to zero to mask or disable a hardware interrupts except NMI. The interrupts whose request can be either accepted or rejected by the processor are called maskable interrupts.



Non-Maskable Interrupts

- Used during power failure
- Used during critical response time
- Used during non-recoverable hardware errors
- Used watchdog interrupt
- Used during memory parity errors



Software Interrupt (INT n)

- Used by operating systems to provide hooks into various function
- Used as a communication mechanism between different parts of the program

8086 INTERRUPT TYPES
256 INTERRUPTS OF 8086 ARE DIVIDED IN TO 3 GROUPS

1. **TYPE 0 TO TYPE 4 INTERRUPTS-**
   These Are Used For Fixed Operations And Hence Are Called Dedicated Interrupts

2. **TYPE 5 TO TYPE 31 INTERRUPTS**
   Not Used By 8086,reserved For Higher Processors Like 80286
   80386 Etc

3. **TYPE 32 TO 255 INTERRUPTS**
   Available For User, called User Defined Interrupts These Can Be H/W Interrupts And Activated Through Intr Line Or Can Be S/W Interrupts.

- Type – 0 Divide Error Interrupt

  Quotient Is Large Cant Be Fit In Al/Ax Or Divide By Zero

- Type –1 Single Step Interrupt

  Used For Executing The Program In Single Step Mode By Setting Trap Flag

- Type – 2 Non Maskable Interrupt

  This Interrupt Is Used For Execution Of NMI Pin.

- Type – 3 Break Point Interrupt

  Used For Providing Break Points In The Program

- Type – 4 Over Flow Interrupt

  Used To Handle Any Overflow Error.

8086 Instructions

- The 8086 microprocessor supports 8 types of instructions-
- Data Transfer Instructions
- Arithmetic Instructions
- Bit Manipulation Instructions
- String Instructions
- Program Execution Transfer Instructions (Branch & Loop Instructions)
- Processor Control Instructions
- Iteration Control Instructions
- Interrupt Instructions

Data Transfer Instructions

- **MOV** – Used to copy the byte or word from the provided source to the provided destination.
- **PPUSH** – Used to put a word at the top of the stack.
- **POP** – Used to get a word from the top of the stack to the provided location.
- **PUSHA** – Used to put all the registers into the stack.
- **POPA** – Used to get words from the stack to all registers.
- **XCHG** – Used to exchange the data from two locations.
- **XLAT** – Used to translate a byte in AL using a table in the memory.
- **IN** – Used to read a byte or word from the provided port to the accumulator.
- **OUT** – Used to send out a byte or word from the accumulator to the provided port.
- **LEA** – Used to load the address of operand into the provided register.
- **LDS** – Used to load DS register and other provided register from the memory
- **LES** – Used to load ES register and other provided register from the memory.
- **LAHF** – Used to load AH with the low byte of the flag register.
- **SAHF** – Used to store AH register to low byte of the flag register.
- **PUSHF** – Used to copy the flag register at the top of the stack.
- **POPF** – Used to copy a word at the top of the stack to the flag register.

Arithmetic Instructions

- **ADD** – Used to add the provided byte to byte/word to word.
- **ADC** – Used to add with carry.
- **INC** – Used to increment the provided byte/word by 1.
- **AAA** – Used to adjust ASCII after addition.
- **DAA** – Used to adjust the decimal after the addition/subtraction operation.
- **SUB** – Used to subtract the byte from byte/word from word.
- **SBB** – Used to perform subtraction with borrow.
- **DEC** – Used to decrement the provided byte/word by 1.
- **NPG** – Used to negate each bit of the provided byte/word and add 1/2's complement.
- **CMP** – Used to compare 2 provided byte/word.
- **AAS** – Used to adjust ASCII codes after subtraction.
- **DAS** – Used to adjust decimal after subtraction.
- **MUL** – Used to multiply unsigned byte by byte/word by word.
- **IMUL** – Used to multiply signed byte by byte/word by word.
- **AAM** – Used to adjust ASCII codes after multiplication.
- **DIV** – Used to divide the unsigned word by byte or unsigned double word by word.
- **IDIV** – Used to divide the signed word by byte or signed double word by word.
- **AAD** – Used to adjust ASCII codes after division.

Bit Manipulation Instructions

- **NOT** – Used to invert each bit of a byte or word.
- **AND** – Used for adding each bit in a byte/word with the corresponding bit in another byte/word.
- **OR** – Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.
- **XOR** – Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.
- **TEST** – Used to add operands to update flags, without affecting operands.
- **SHL/SAL** – Used to shift bits of a byte/word towards left and put zero(S) in LSBs.
- **SHR** – Used to shift bits of a byte/word towards the right and put zero(S) in MSBs.
- **SAR** – Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.

Bit Manipulation Instructions

- **SAR** – Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.
- **ROL** – Used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag [CF].
- **ROR** – Used to rotate bits of byte/word towards the right, i.e. LSB to MSB and to Carry Flag [CF].
- **RCR** – Used to rotate bits of byte/word towards the right, i.e. LSB to CF and CF to MSB.
- **RCL** – Used to rotate bits of byte/word towards the left, i.e. MSB to CF and CF to LSB.

String Instructions

- **REP** – Used to repeat the given instruction till CX ≠ 0.
- **REPE/REPZ** – Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- **REPNE/REPNZ** – Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- **MOVS/MOVSB/MOVSW** – Used to move the byte/word from one string to another.
- **COMS/COMPSB/COMPSW** – Used to compare two string bytes/words.
- **INS/INSB/INSW** – Used as an input string/byte/word from the I/O port to the provided memory location.
- **OUTS/OUTSB/OUTSW** – Used as an output string/byte/word from the provided memory location to the I/O port.
- **SCAS/SCASB/SCASW** – Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.
- **LODS/LODSB/LODSW** – Used to store the string byte into AL or string word into AX.

Program Execution Transfer Instructions (Branch and Loop Instructions)

- **JA/JNBE** – Used to jump if above/not below/equal instruction satisfies.
- **JAE/JNB** – Used to jump if above/not below instruction satisfies.
- **JBE/JNA** – Used to jump if below/equal/ not above instruction satisfies.
- **JC** – Used to jump if carry flag CF = 1
- **JE/JZ** – Used to jump if equal/zero flag ZF = 1
- **JG/JNLE** – Used to jump if greater/not less than/equal instruction satisfies.
- **JGE/JNL** – Used to jump if greater than/equal/not less than instruction satisfies.
- **JL/JNGE** – Used to jump if less than/not greater than/equal instruction satisfies.
- **JLE/JNG** – Used to jump if less than/equal/if not greater than instruction satisfies.
- **JNC** – Used to jump if no carry flag (CF = 0)
- **JNE/JNZ** – Used to jump if not equal/zero flag ZF = 0

Program Execution Transfer Instructions (Branch and Loop Instructions)

- **JNO** – Used to jump if no overflow flag OF = 0
- **JNP/JPO** – Used to jump if not parity/parity odd PF = 0
- **JNS** – Used to jump if not sign SF = 0
- **JO** – Used to jump if overflow flag OF = 1
- **JP/JPE** – Used to jump if parity/parity even PF = 1
- **JS** – Used to jump if sign flag SF = 1

Processor Control Instructions

- **STC** – Used to set carry flag CF to 1
- **CLC** – Used to clear/reset carry flag CF to 0
- **CMC** – Used to put complement at the state of carry flag CF.
- **STD** – Used to set the direction flag DF to 1
- **CLD** – Used to clear/reset the direction flag DF to 0
- **STI** – Used to set the interrupt enable flag to 1, i.e., enable INTR input.
- **CLI** – Used to clear the interrupt enable flag to 0, i.e., disable INTR input.

Interrupt Instructions

- **INT** – Used to interrupt the program during execution and calling service specified.
- **INTO** – Used to interrupt the program during execution if OF = 1
- **IRET** – Used to return from interrupt service to the main program

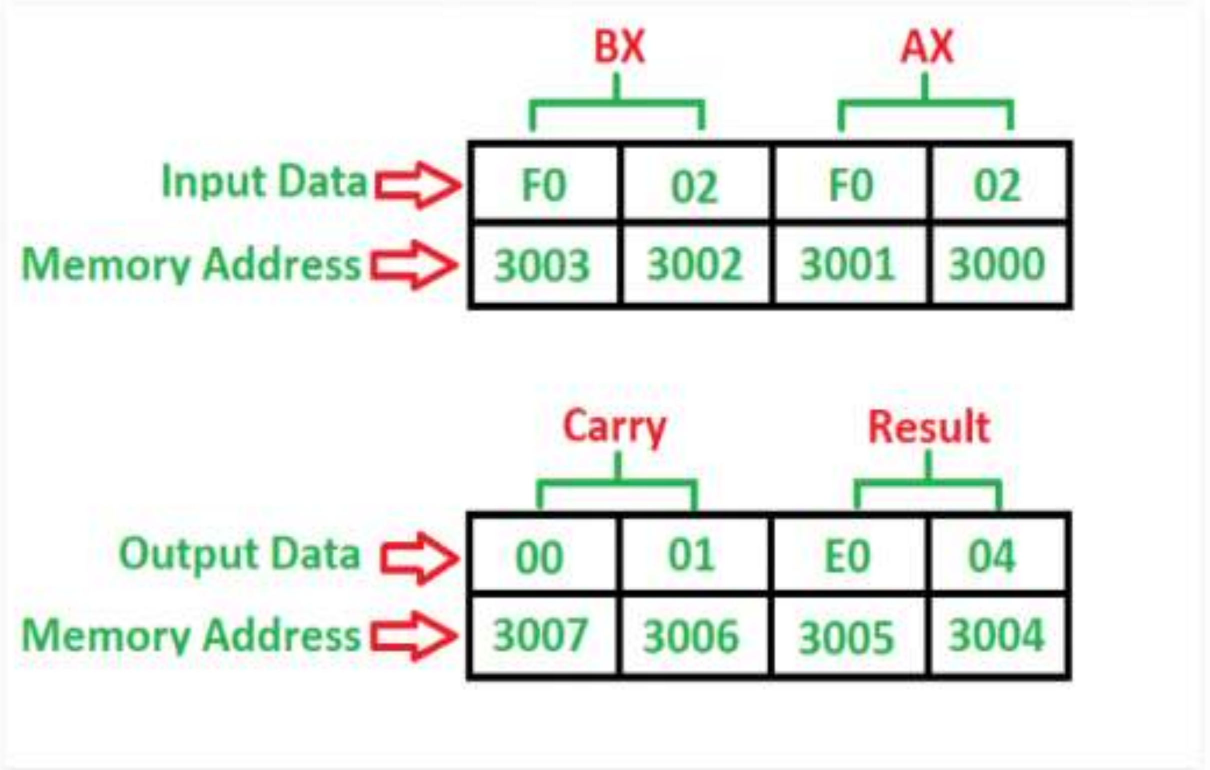
Iteration Control Instructions

- **LOOP** − Used to loop a group of instructions until the condition satisfies, i.e., CX = 0
- **LOOPE/LOOPZ** − Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0
- **LOOPNE/LOOPNZ** − Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0
- **JCXZ** − Used to jump to the provided address if CX = 0

8086 program to add two 16-bit

Problem – Write a program to add two 16-bit numbers where starting address is **2000** and the numbers are at **3000** and **3002** memory address and store result into **3004** and **3006** memory address.

Example –



Algorithm –

1. Load 0000H into CX register (for carry)
2. Load the data into AX(accumulator) from memory 3000
3. Load the data into BX register from memory 3002
4. Add BX with Accumulator AX
5. Jump if no carry
6. Increment CX by 1
7. Move data from AX(accumulator) to memory 3004
8. Move data from CX register to memory 3006
9. Stop

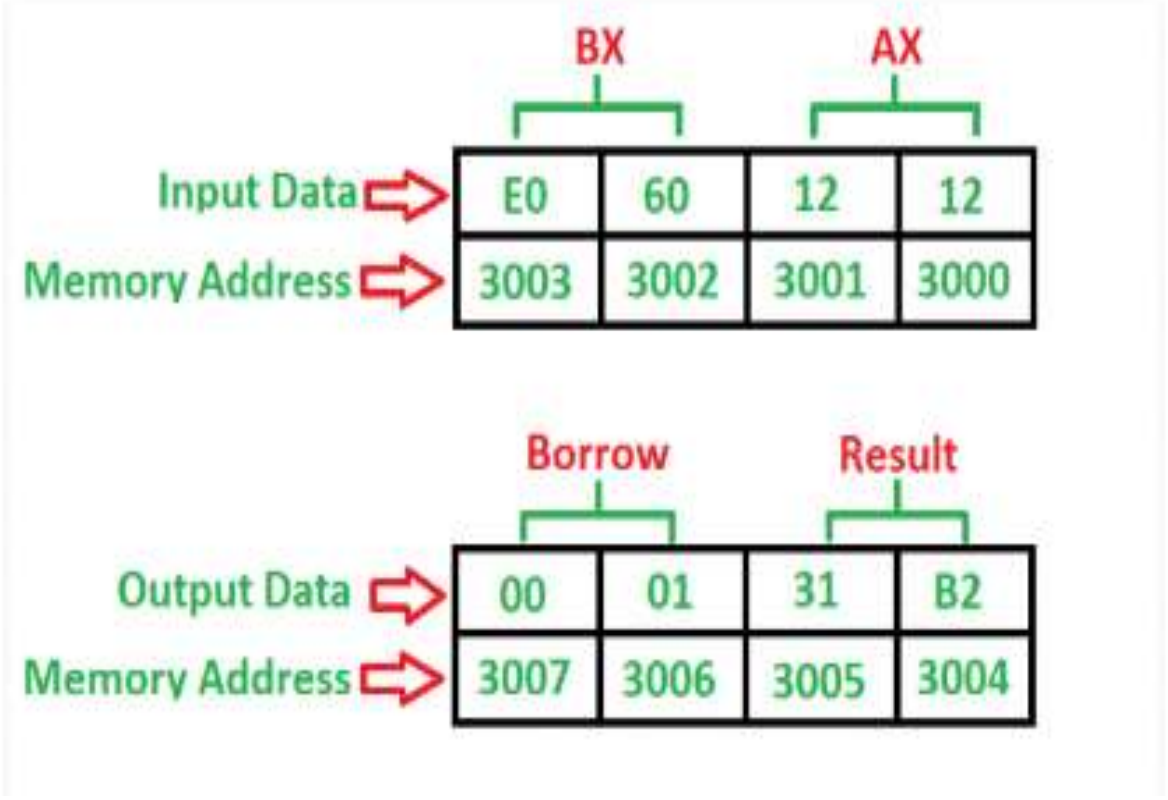
Program –

Memory	Mnemonics	Operands	Comment
2000	MOV	CX, 0000	[CX] <- 0000
2003	MOV	AX, [3000]	[AX] <- [3000]
2007	MOV	BX, [3002]	[BX] <- [3002]
200B	ADD	AX, BX	[AX] <- [AX] + [BX]
200D	JNC	2010	Jump if no carry
200F	INC	CX	[CX] <- [CX] + 1
2010	MOV	[3004], AX	[3004] <- [AX]
2014	MOV	[3006], CX	[3006] <- [CX]
2018	HLT		Stop

8086 program to subtract two 16-bit

Problem – Write a program to subtract two 16-bit numbers where starting address is **2000** and the numbers are at **3000** and **3002** memory address and store result into **3004** and **3006** memory address.

Example –



Algorithm -

1. Load 0000H into CX register (for borrow)
2. Load the data into AX(accumulator) from memory 3000
3. Load the data into BX register from memory 3002
4. Subtract BX with Accumulator AX
5. Jump if no borrow
6. Increment CX by 1
7. Move data from AX(accumulator) to memory 3004
8. Move data from CX register to memory 3006
9. Stop

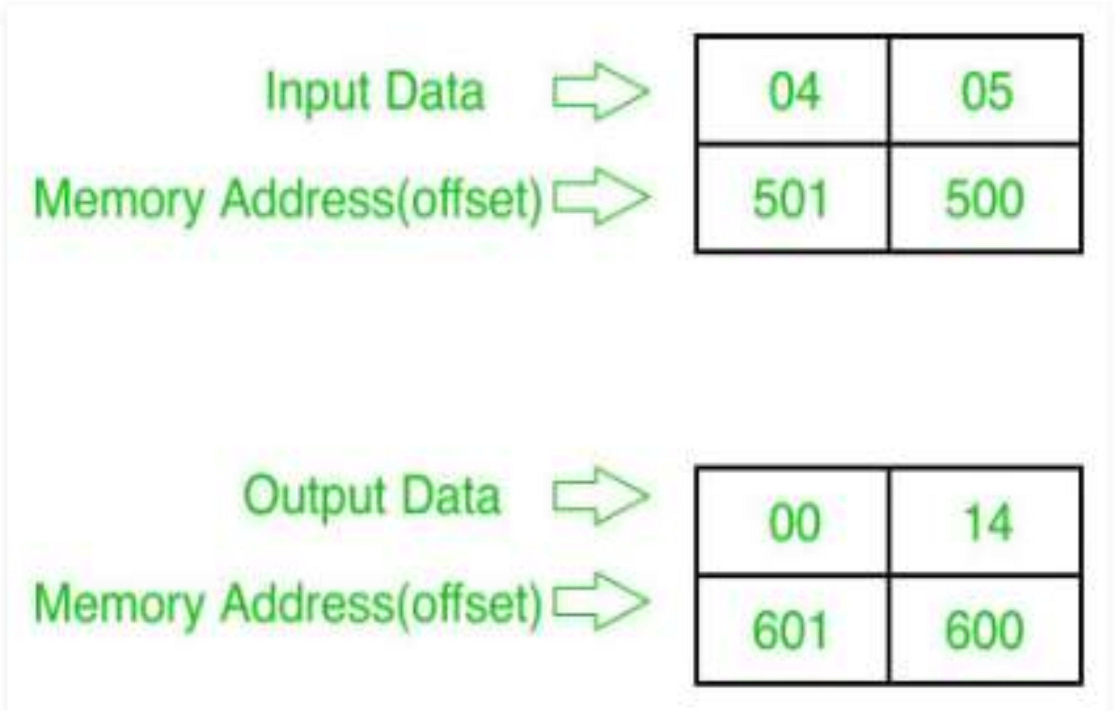
Program –

Memory	Mnemonics	Operands	Comment
2000	MOV	CX, 0000	[CX] <- 0000
2003	MOV	AX, [3000]	[AX] <- [3000]
2007	MOV	BX, [3002]	[BX] <- [3002]
200B	SUB	AX, BX	[AX] <- [AX] – [BX]
200D	JNC	2010	Jump if no borrow
200F	INC	CX	[CX] <- [CX] + 1
2010	MOV	[3004], AX	[3004] <- [AX]
2014	MOV	[3006], CX	[3006] <- [CX]
2018	HLT		Stop

8086 program to multiply two 8-bit

Problem – Write a program in 8086 microprocessor to multiply two 8-bit numbers, where numbers are stored from offset 500 and store the result into offset 600.

Examples – Inputs and output are given in Hexadecimal representation.



Algorithm –

1. Load data from offset 500 to register AL (first number)
2. Load data from offset 501 to register BL (second number)
3. Multiply them (AX=AL*BL)
4. Store the result (content of register AX) to offset 600
5. Stop

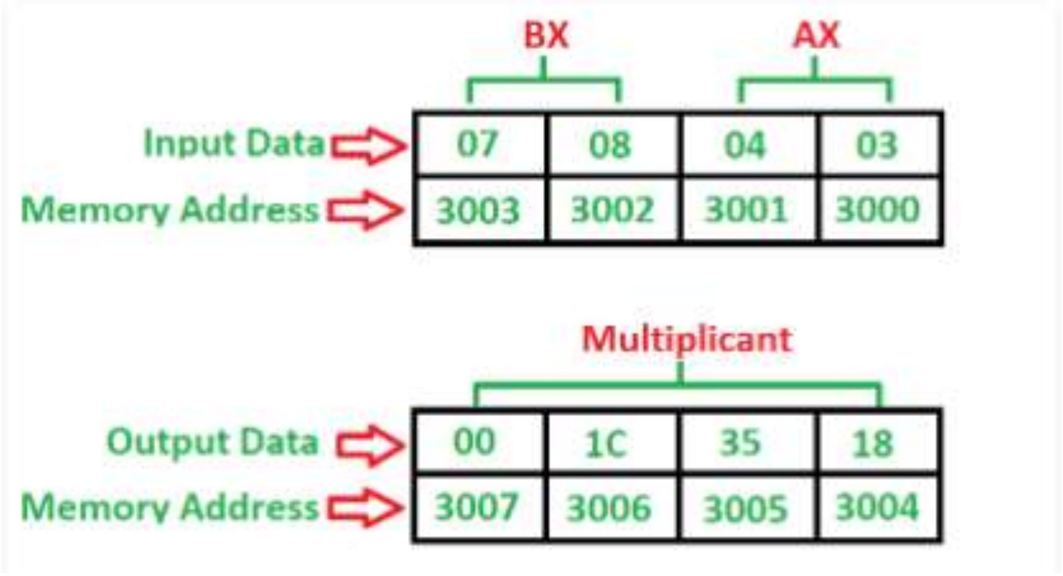
Program –

MEMORY ADDRESS	MNEMONICS	COMMENT
400	MOV SI, 500	SI=500
403	MOV DI, 600	DI=600
406	MOV AL, [SI]	AL<-[SI]
408	INC SI	SI=SI+1
409	MOV BL, [SI]	BL<-[SI]
40B	MUL BL	AX=AL*BL
40D	MOV [DI], AX	AX->[DI]
40F	HLT	END

8086 program to multiply two 16-bit

Problem – Write a program to multiply two 16-bit numbers where starting address is **2000** and the numbers are at **3000** and **3002** memory address and store result into **3004** and **3006** memory address.

Example –



Algorithm –

1. First load the data into AX(accumulator) from memory 3000
2. Load the data into BX register from memory 3002
3. Multiply BX with Accumulator AX
4. Move data from AX(accumulator) to memory
5. Move data from DX to AX
6. Move data from AX(accumulator) to memory
7. Stop

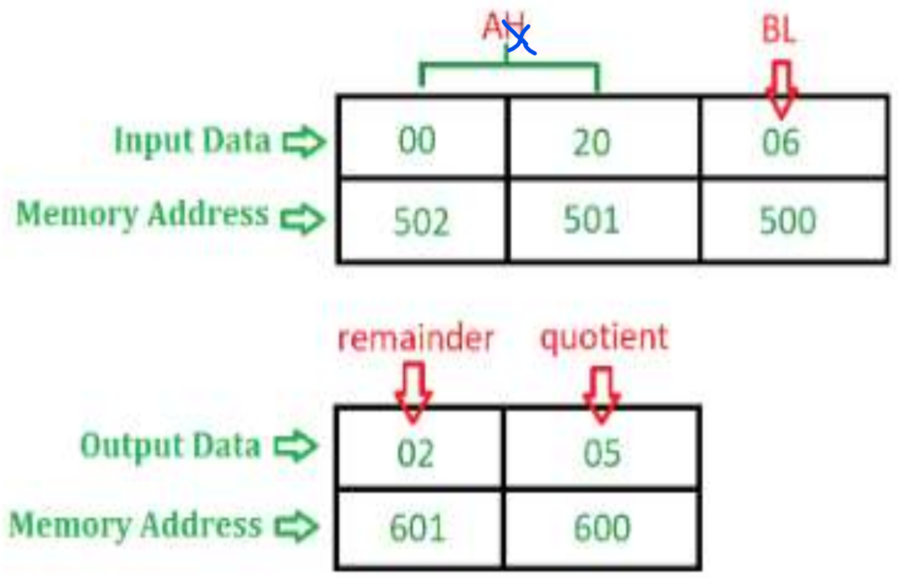
Program –

Memory	Mnemonics	Operands	Comment
2000	MOV	AX, [3000]	[AX] <- [3000]
2004	MOV	BX, [3002]	[BX] <- [3002]
2008	MUL	BX	[AX] <- [AX] * [BX]
200A	MOV	[3004], AX	[3004] <- AX
200E	MOV	AX, DX	[AX] <- [DX]
2010	MOV	[3006], AX	[3006] <- AX
2014	HLT		Stop

8086 program to divide a 16 bit number by an 8 bit number

Problem – Write an assembly language program in 8086 microprocessor to divide a 16 bit number by an 8 bit number.

Example –



Algorithm –

1. Assign value 500 in SI and 600 in DI
2. Move the contents of [SI] in BL and increment SI by 1
3. Move the contents of [SI] and [SI + 1] in AX
4. Use **DIV** instruction to divide AX by BL
5. Move the contents of AX in [DI].
6. Halt the program.

Assumption – Initial value of each segment register is 00000.

Calculation of physical memory address –

Memory Address = Segment Register * 10(H) + offset,

where Segment Register and Offset is decided on the basis of following table.

Program –

MEMORY ADDRESS	MNEMONICS	COMMENT
0400	MOV SI, 500	SI <- 500
0403	MOV DI, 600	DI <- 600
0406	MOV BL, [SI]	BL <- [SI]
0408	INC SI	SI <- SI + 1
0409	MOV AX, [SI]	AX <- [SI]
040B	DIV BL	AX <- AX / BL
040D	MOV [DI], AX	[DI] <- AX
040F	HLT	End of program