Microprocessors & Microcontrollers (ECE3004)

Module – 02 (Part – 01)

8086 Microprocessor

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

VIT BHOPAL

Module-2 Syllabus

8086 Microprocessor:

- Introduction to 8086 – Signals and pins - 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.

Comparison with 8085

- **World Length:**
  8085 is 8 bit microprocessor whereas 8086 is 16 bit microprocessor.
- **Address Bus:**
  8085 has 16 bit address bus and 8086 has 20 bit address bus.
- **Memory:**
  8085 can access up to $2^{16} = 64$ KB of memory whereas 8086 can access up to $2^{20} = 1$ MB of memory.
- **Instruction Queue:**
  8085 doesn't have an instruction queue whereas 8086 has instruction queue.
- **Pipelining:**
  8085 does not support pipelined architecture whereas 8086 supports pipelined architecture

Comparison with 8085

- **Multiprocessing Support:**
  8085 does not support multiprocessing support whereas 8086 supports.
- **I/O:**
  8085 can address $2^8 = 256$ I/O's and 8086 can access $2^{16} = 65{,}536$ I/O's
- **Arithmetic Support:**
  8085 only supports integer and decimal whereas 8086 supports integer, decimal and ASCII arithmetic.
- **Flag:**

8085 - 5 flags (Sign Flag, Zero Flag, Auxiliary Carry Flag, Parity Flag, Carry Flag).

8086 - 9 flags (Overflow Flag, Direction Flag, Interrupt Flag, Trap Flag, Sign Flag, Zero Flag, Auxiliary Carry Flag, Parity Flag, Carry Flag).

Comparison with 8085

- **Multiplication and Division:**
  8085 doesn't support whereas 8086 supports.
- **Operating Modes:**
  8085 supports only single operating mode whereas 8086 operates in two modes.
- **External Hardware:**
  8085 requires less external hardware whereas 8086 requires more external hardware.
- **Cost:**
  The cost of 8085 is low and 8086 is high.
- **Memory Segmentation:**
  In 8085, memory space is not segmented but in 8086, memory space is segmented.

8085 vs 8086

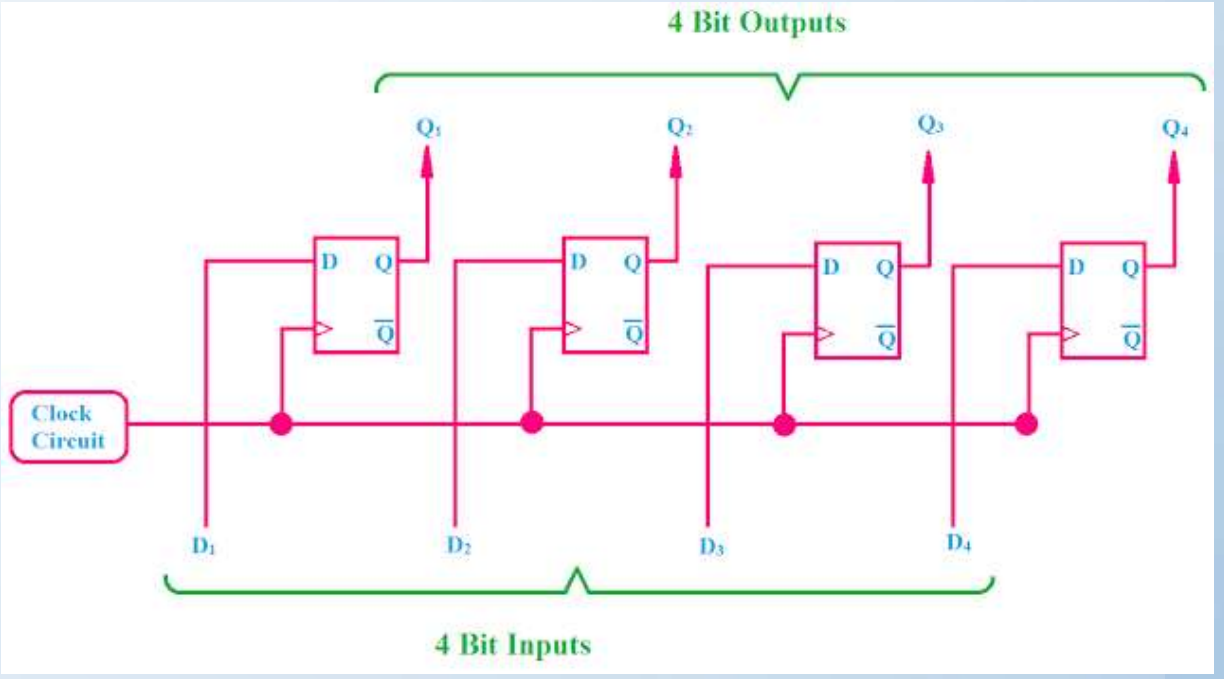
8085 microprocessor	8086 microprocessor
It is 8 bit microprocessor	It is 16 bit microprocessor
It has 16 bit address line	It has 20 bit address line
It has 8 bit data bus	It has 16 bit data bus
clock speed of 8085 microprocessor is 3 MHz	clock speed of 8086 microprocessor vary between 5,8 and 10 MHz for different versions.
It has 5 flags.	It has 9 flags.
It does not support pipelining.	It supports pipelining.
It operates on clock cycle with 50% duty cycle.	It operates on clock cycle with 33% duty cycle.
8085 microprocessor does not support memory segmentation.	8086 microprocessor supports memory segmentation.
It has less number of transistors compare to 8086 microprocessor. It is about 6500 in size.	It has more number of transistors compare to 8085 microprocessor. It is about 29000 in size.
It is accumulator based processor.	It is general purpose register based processor.
It has no minimum or maximum mode.	It has minimum and maximum modes.
In 8085, only one processor is used.	In 8086, more than one processor is used. Additional external processor can also be employed.
In this microprocessor type, only 64 KB memory is used.	In this microprocessor type, 1 MB memory is used.

Register Organization of 8086

What's a Register?

- A **Register is a circuit** consisting of Flip-Flops which can store more than one-bit data. The **register is** nothing but a sequential logic circuit in digital electronics.

- A **register** is a temporary storage area built into a CPU. It is the smallest amount of memory that is available to the processor to carry out arithmetic and logical operations.



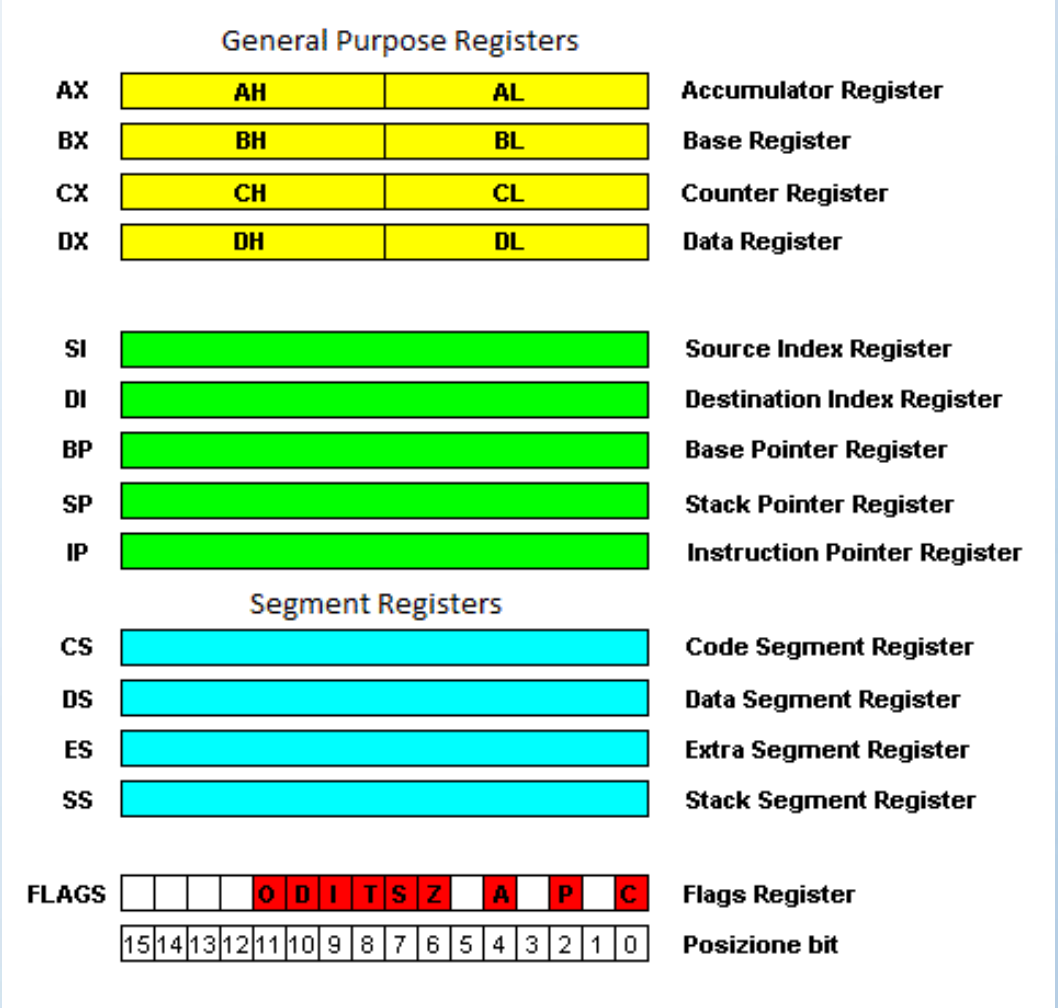
- **Register** is very fast and efficient than other memories like RAM, ROM or external memory.

- This is why registers occupied the top position in memory hierarchy model.

Registers of 8086

- The 8086 microprocessor has a total of fourteen registers that are accessible to the programmer.
- All these registers are 16-bit in size.
- The registers of 8086 are categorized into 5 different groups.

a) General registers
b) Segment registers
c) Index registers
d) Pointer registers
e) Status Register



a) General Purpose Registers

- The general purpose registers are used to perform arithmetic and logical operations. These registers are used by the program to code the microprocessor.
- All these registers are 16-bit in size.
- These all general registers can be used as either 8-bit or 16-bit registers.
- Four 16-bit registers are available in 8086 μP.
  - ✓ AX (Accumulator Register)
  - ✓ BX (Base Register)
  - ✓ CX (Counter Register)
  - ✓ DX (Data Register)

a) General Purpose Registers

i. AX (Accumulator Register):

- ✓ AX is used as 16-bit accumulator. The lower 8-bits of AX are designated to use as AL and higher 8-bits as AH. AL can be used as an 8-bit accumulator for 8-bit operation.
- ✓ This Accumulator used in arithmetic, logic and data transfer operations. For manipulation and division operations, one of the numbers must be placed in AX or AL.
- ✓ AL in this case contains the low order byte of the word, and AH contains the high order byte.

a) General Purpose Registers

ii. BX (Base Register):

- ✓ consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX. BL in this case contains the low order byte of the word, and BH contains the high order byte.
- ✓ This is the only general purpose register whose contents can be used for addressing the 8086 memory.
- ✓ The register BX is used as address register to form physical address in case of certain addressing modes (ex: indexed and register indirect).

a) General Purpose Registers

iii. CX (Counter Register):

- ✓ consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX. CL in this case contains the low order byte of the word, and CH contains the high order byte.
- ✓ The register CX is used default counter in case of string and **loop** instructions. Count register can also be used as a counter in **string manipulation** and **shift/rotate** instruction.

Ex:

- ✓ The instruction **LOOP START** automatically decrements CX by 1 without affecting the flags and checks if [CX] = 0.
- ✓ If it is zero, 8086 executes the next instruction; otherwise the 8086 branches to the label **START**.

a) General Purpose Registers

iv. DX (Data Register):

- ✓ consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX. DL in this case contains the low order byte of the word, and DH contains the high order byte.
- ✓ It is used with AX to hold 32-bit values during multiplication and division operation (holds the remainder of the result).
- ✓ Data register can also be used as a port number in I/O operations.

b) Segment Registers

- The 8086 architecture uses the concept of segmented memory. 8086 can able to access a memory capacity of up to 1 megabyte.
- This 1 megabyte of memory is divided into 16 logical segments. Each segment contains 64 Kbytes of memory.
    - ✓ Why 64KB memory ?
- There are four segment registers to access this 1 megabyte of memory.
    - ✓ CS (Code Segment Register)
    - ✓ SS (Stack Segment Register)
    - ✓ DS (Data Segment Register)
    - ✓ ES (Extra Segment Register)

b) Segment Registers

i. CS (Code Segment Register):

- ✓ Code segment (CS) is a 16-bit register that is used for addressing memory location in the code segment of the memory (64Kb), where the executable program is stored.
- ✓ CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.

ii. SS (Stack Segment Register):

- ✓ Stack Segment (SS) is a 16-bit register that used for addressing stack segment of the memory (64kb) where stack data is stored.
- ✓ SS register can be changed directly using POP instruction.

b) Segment Registers

iii. DS (Data Segment Register):

- ✓ Data Segment (DS) is a 16-bit register that points the data segment of the memory (64kb) where the program data is stored.
- ✓ DS register can be changed directly using POP and LDS instructions.

iv. ES (Extra Segment Register)

- ✓ Extra Segment (ES) is a 16-bit register that also points the data segment of the memory (64kb) where the program data is stored.
- ✓ ES register can be changed directly using POP and LES instructions.

c) Index Registers

- The index registers can be used for arithmetic operations but their use is usually concerned with the memory addressing modes of the 8086 microprocessor (indexed, base indexed and relative base indexed addressing modes).

i. SI (Source Index Register):

- ✓ SI is a 16-bit register. This register is used to store the offset of source data in data segment. In other words the Source Index Register is used to point the memory locations in the data segment.

ii. DI (Destination Index Register):

- ✓ DI is a 16-bit register. This is destination index register performs the same function as SI. There is a class of instructions called string operations that use DI to access the memory locations in Data or Extra Segment.

d) Pointer Registers

- Pointer Registers contains the offset of data(variables, labels) and instructions from their base segments (default segments). 8086 microprocessor contains three pointer registers.

i. IP (Instruction Pointer Register):

- ✓ The Instruction Pointer is a register that holds the address of the next instruction to be fetched from memory.
- ✓ It contains the offset of the next word of instruction code instead of its actual address

d) Pointer Registers

ii. SP (Stack Pointer Register):

- ✓ It hold the offset address of the top of the stack. Stack is a set of memory locations operating in LIFO (Last In First Out) manner.

iii. BP (Base Pointer Register):

- ✓ Base Pointer register also points the same stack segment. Unlike SP, we can use BP to access random locations in stack memory.

e) Status Register

- The status register also called as **flag register**. The 8086 flag register contents indicate the results of computation in the ALU. It also contains some flag bits to control the CPU operations.

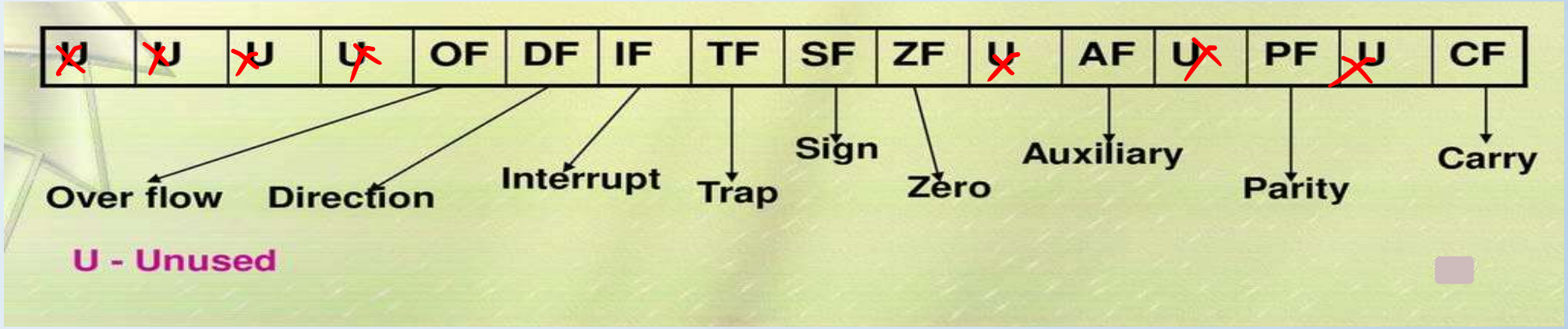
- In 8086, the status register contains

  ✓ 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



Flag Register

Auxiliary Carry Flag

This is set, if there is a carry from the lowest nibble, i.e, bit three during addition, or borrow for the lowest nibble, i.e, bit three, during subtraction.

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.

Sign Flag

This flag is set, when the result of any computation is negative

Zero Flag

This flag is set, if the result of the computation or comparison performed by an instruction is zero

Parity Flag

This flag is set to 1, if the lower byte of the result contains even number of 1's ; for odd number of 1's set to zero.

Over flow Flag

This flag is set, if an overflow occurs, i.e, if the result of a signed operation is large enough to accommodate in a destination register. The result is of more than 7-bits in size in case of 8-bit signed operation and more than 15-bits in size in case of 16-bit sign operations, then the overflow will be set.

Tarp Flag

If this flag is set, the processor enters the single step execution mode by generating internal interrupts after the execution of each instruction

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 highest address, i.e., auto incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e., auto incrementing mode.

Interrupt Flag

Causes the 8086 to recognize external mask interrupts; clearing IF disables these interrupts.

Registers - 8086

Size of Reg. = 16 Bit

Types of Reg. = 5 types

- General Reg.
  - AX
  - BX
  - CX
  - DX
  - (8 Bit Reg. — Lower Byte, Higher Byte)
- Seg. Reg.
  - CS
  - DS
  - ES
  - SS
  - (PA = Base add. + Starting add. = CS × 10H + ? add.)
- Pointer Reg.
- Index Reg.
- Status Reg.

Total Reg. in 8086 = 14

Length of Add. Bus 8086 = 20 Bit

Total memory = 2^{20} = 1MB

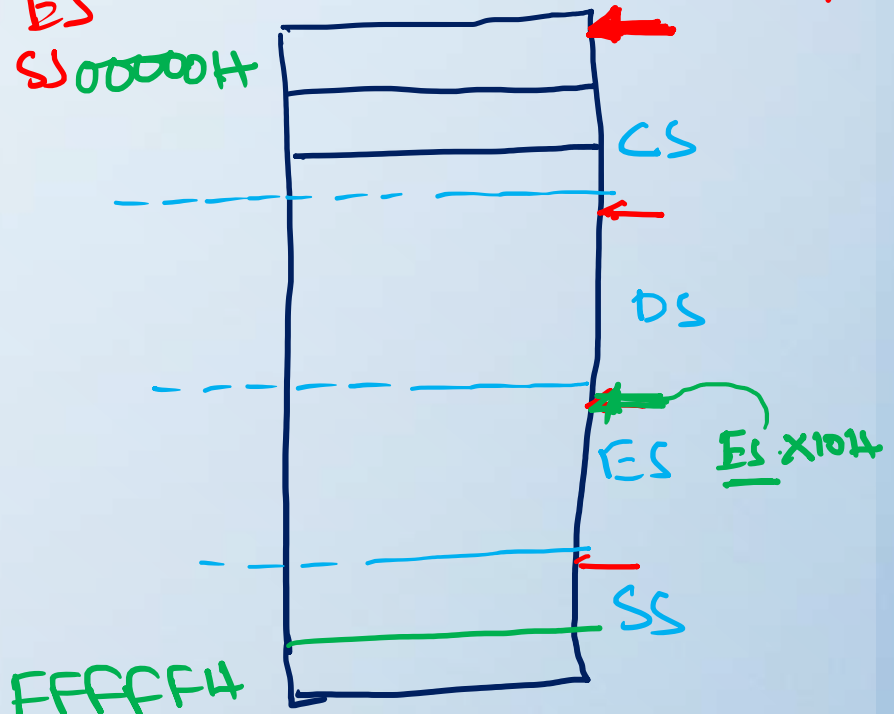
No. of Add. location = 2^{20} locations (20 Bits)

Lowest location = 00000H

Highest location = FFFFFH

PA = Base add. (Seg. Reg. × 10H) + Offset add.

Seg. Reg.



Total memory seg. (types) = 4 → CS, DS, ES, SS (64KB)

Total memory seg. = 16 seg.

SS = 1F30H

PA = Starting location of stack seg.

DS = 1254H

PA = Starting location of Data seg.

= 1254 X 10H = 12540H

DS (Starting loc)

Q. ES = 1234H

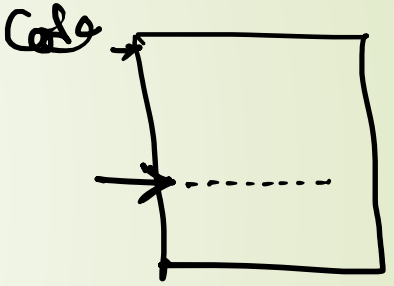
PA = Starting.

= 1234 X 10H = 12340H

= 13F300H

8086 μP Architecture

Pointer Reg :- Stack pointer (SP) } Stack
Base pointer (BP) }
Instruction pointer (IP) } code



PA = CS X 10H + IP

Index Reg.

SI → Data seg. (Any where in data seg.)

DI

PA = DS X 10H + SI

Extra → ES

PA = ES X 10H + DI

Stack Seg.

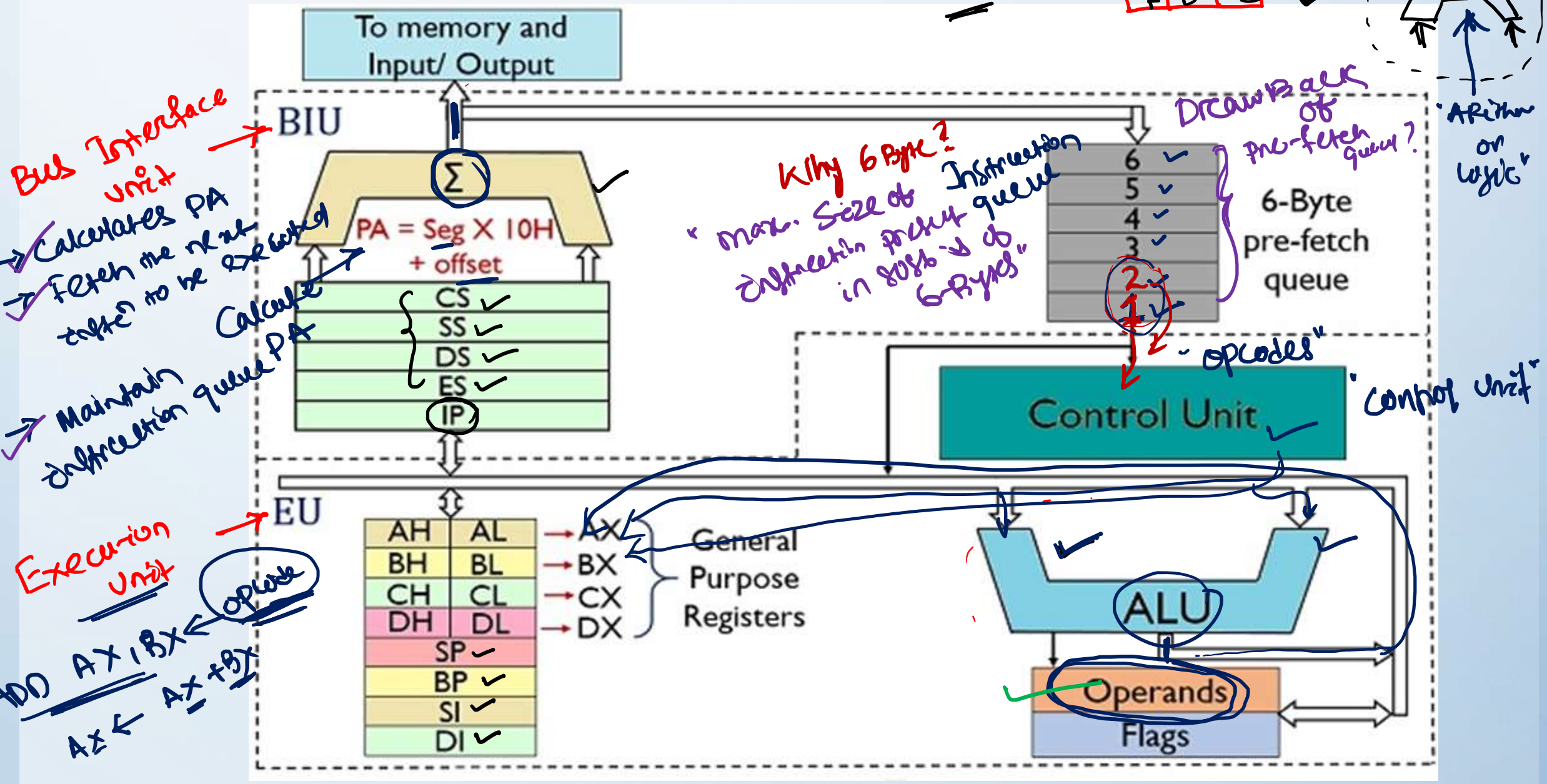
TOP of the stack

PA = SS X 10H + SP

Randomly any location in stack

PA = SS X 10H + BP

8086 Architecture



8086 – "16-Bit"

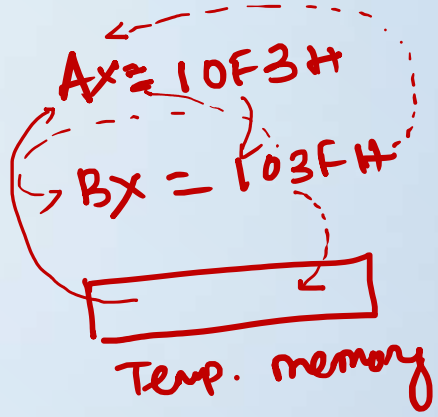
BIU – (Bus interface unit)

(*) Calculate the P.A. – CS, DS, ES, SS

"IP" (20 Bit)

(*) Fetch the next instruction to be executed.

(*) Maintains the instruction queue.



EU – (Execution unit)

(*) Executes the instruction (ALU)

(*) Control unit – "Decodes the opcode"

[8086 – 15 Reg.s. – only 14]

↓

Available for Programing

"Operands = Register" : "Tempory Register" : "used by processor"

"16-Bit"

XCHG AX, BX

Copy value of BX to "Operands"

BX ← AX, AX ← Operands

Architecture:

- The total architecture can be divided into two parts:

BIU: Bus Interfacing Unit
Reads data from the memory or I/O ports and writes data into memory or I/O.

Contains: Instruction queue, Segment registers, Instruction pointer, Address adder.

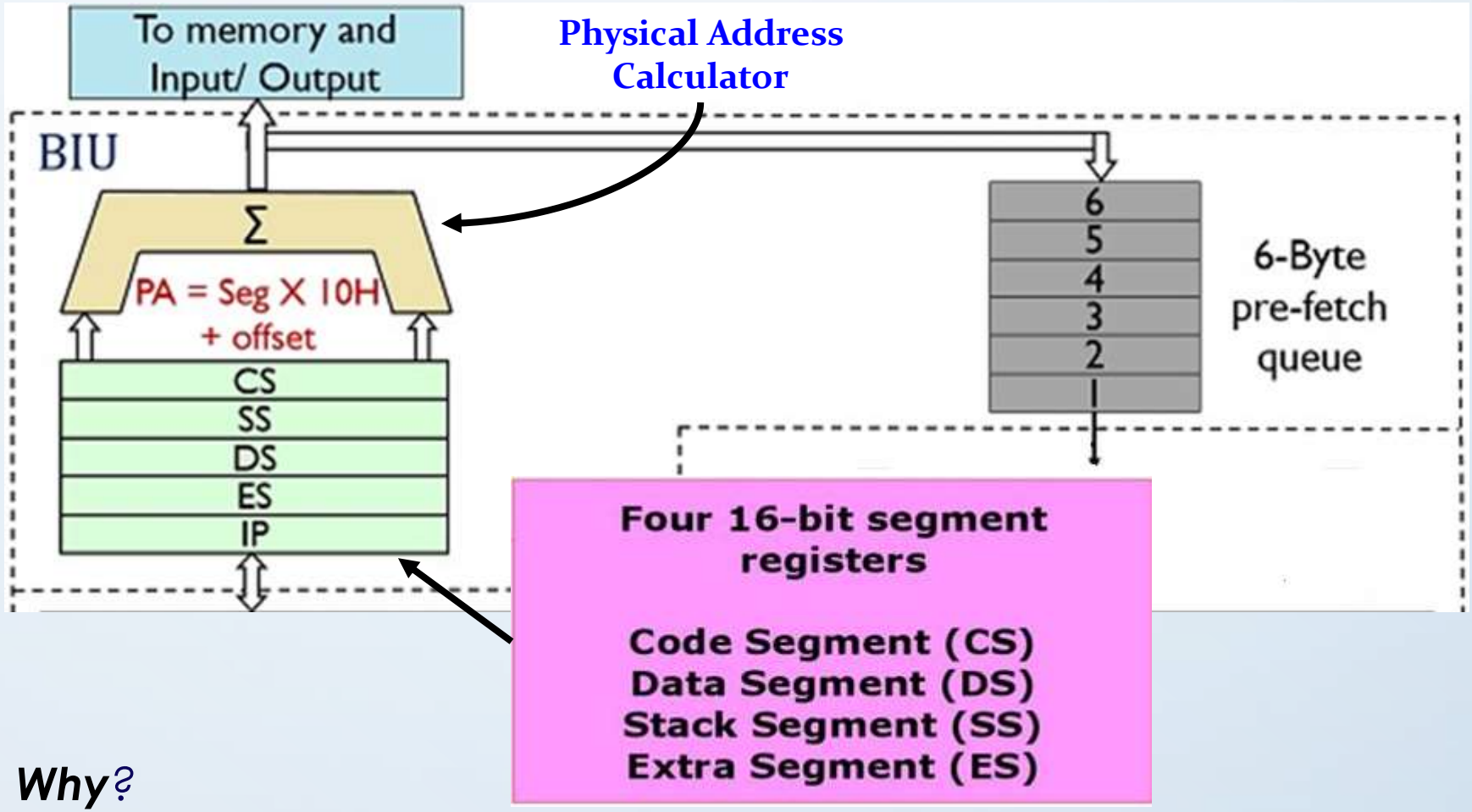
EU: Execution Unit
Executes data that has been already fetched from the memory.

Contains: Control circuitry, Instruction decoder, ALU, Pointer and Index register, Flag register.

Both units function separately

Why?
Both units operate asynchronously to give the 8086 an overlapping instruction fetch and execution mechanism which is called as Pipelining. This results in efficient use of the system bus and system performance.

BIU: Bus Interfacing Unit



Physical Address Calculator:

$PA = Seg.Add. \times 10_H + Offset\ Add.$

Where,
Seg. Add. = Segment Address
Offset Add. = Offset Address

Ex:
CS = 1000H
IP = 2345H

$$PA = CS. \times 10_H + IP$$
$$PA = 1000 \times 10_H + 2345$$
$$= 12345\ H$$

Why?

- *The 8086 μP can address $2^{20} = 1MB$ memory. Thus it has 20-bit address bus. Registers available in μP can hold 16-bit of data, thus require 2 registers to form 20-bit address.*
- *The memories are divided/segmented in to 4-groups each of $64KB$.*

BIU: Bus Interfacing Unit

Note:

- CS, SS, DS and ES in the BIU are not the segment memory (memory segment are part of memory, external to 8086); Remember these are segment register.

- CS Register: Contains the starting address of code segment.
- SS Register: Contains the starting address of stack segment.
- DS Register: Contains the starting address of data segment.
- ES Register: Contains the starting address of extra segment.

✓ IP Register: Contains the offset address of with in code segment. Instruction Pointer

✓ As the current instruction being decoded (or executed) the BIU loads the address of next instruction to be fetched, thus it increments the value of IP.

More on BIU

i. Fetches next instruction
ii. Calculates physical address
iii. Manages instruction queue.

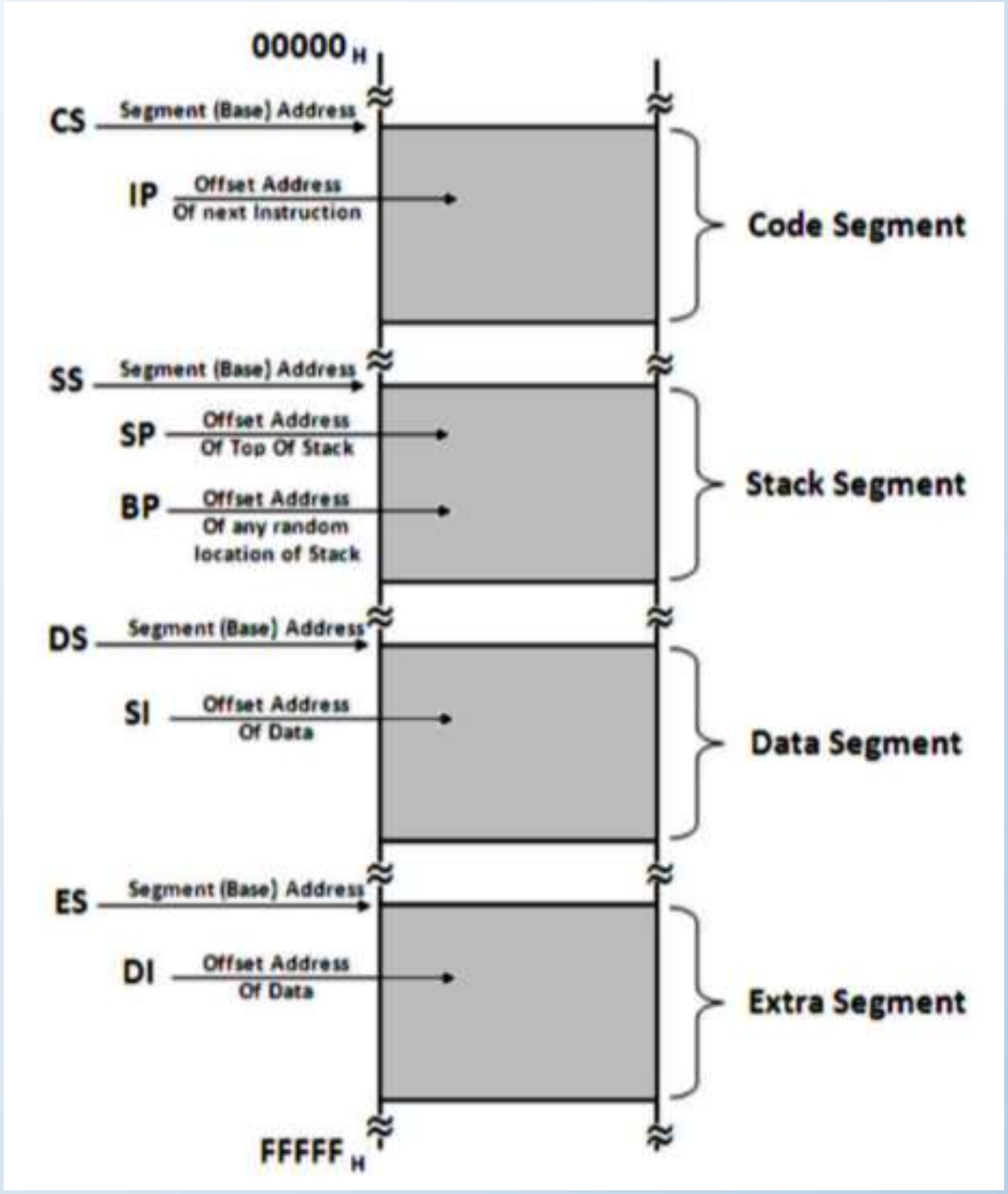
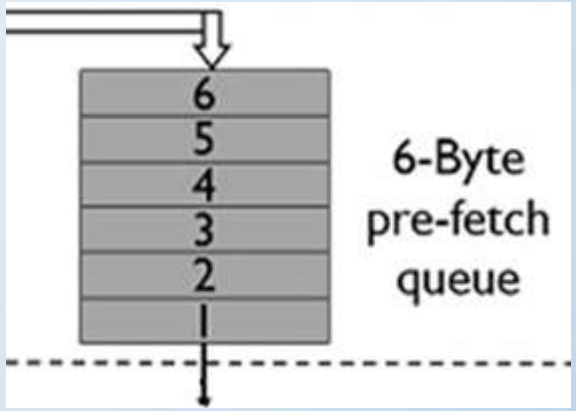


Fig: Memory Segment Available for 8086 μP

Instruction Queue

BIU: Bus Interfacing Unit

- The BIU uses a mechanism known as an instruction stream queue to implement a pipeline architecture.
- The instruction is get fetched through the data bus and stored in the queue; waiting to be executed next.



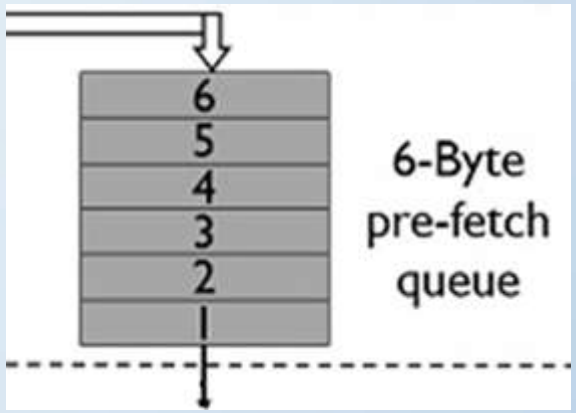
✓ What should be the size of Queue?

- ✓ 6-bytes of instruction are fetched and stored in queue.
- ✓ Instructions in μP can vary from 1 byte to 6 byte.
- ✓ As maximum size is 6-bytes, thus size of pre-queue is 6-byte.

- These pre-fetching instructions are held in its FIFO queue.
- With 16 bit data bus, the BIU fetches two instruction bytes in a single memory cycle.
- After a byte is loaded at the input end of the queue, it automatically shifts up through the FIFO to the empty location nearest the output. The EU accesses the queue from the output end.
- It reads one instruction byte after the other from the output of the queue.
- The intervals of no bus activity, which may occur between bus cycles are known as Idle state.

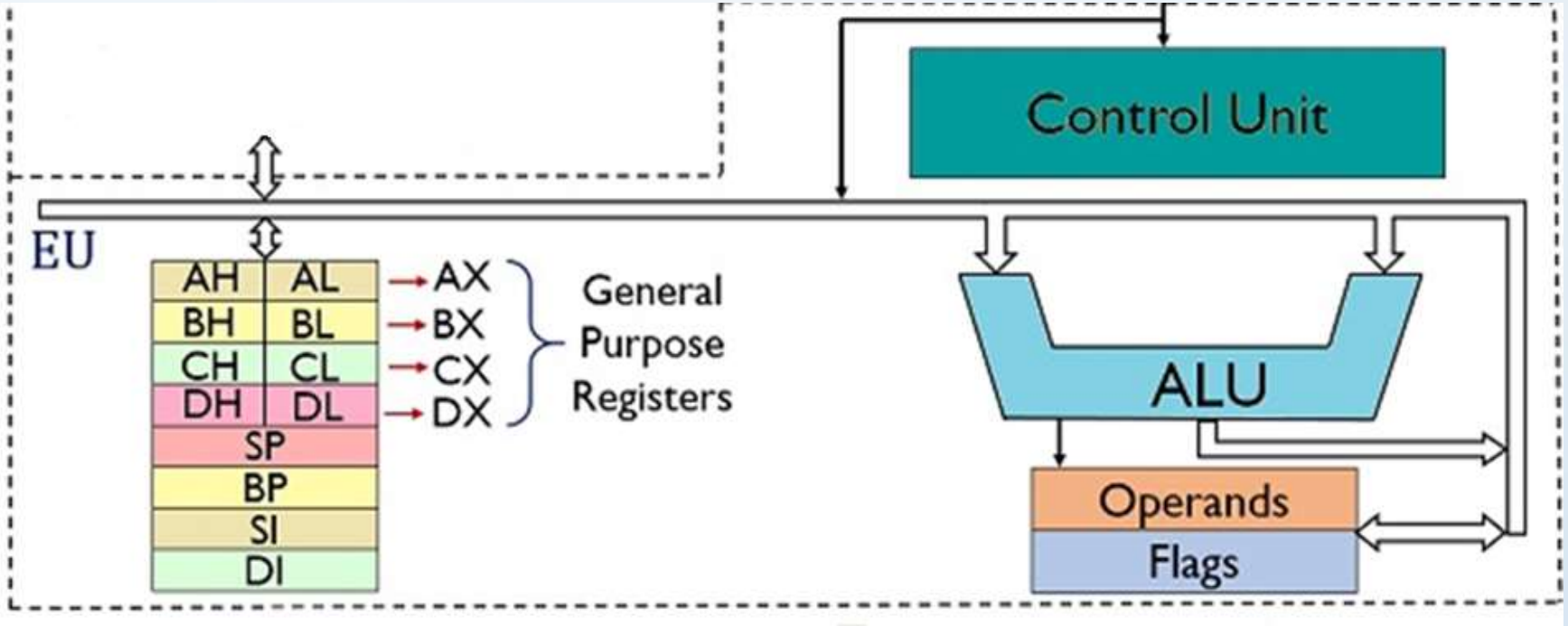
Drawback of Pipeline

- While executing the current instruction; BIU fetches six-bytes of instruction to maintain parallel processing.
- Parallel processing; Saves time
- However, when compiling a jump operation; the next instruction address may be far away from 6-bytes; Already pre-fetched.
- Thus, BIU has to discard all the previously stored data in queue.
- Need to store instruction; afresh. The instruction indicated by Jump statement.



- Thus pipeline creates delay during jump statement or ISR.

EU: Execution Unit

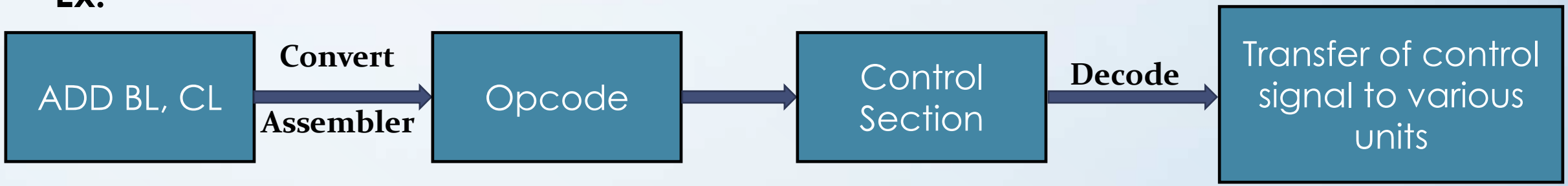


- **It executes the instruction that already fetched and stored in the pre-fetch queue.**
- **Before execution; needs instruction decoding; Control unit decodes the instruction.**
- **EU has no direct connection with system buses as shown in the above figure, it performs operations over data through BIU.**

- **ALU**
  - ✓ It handles all arithmetic and logical operations, like +, –, ×, /, OR, AND, NOT operations.
- **Flag Register**
  - ✓ It is a 16-bit register that behaves like a flip-flop, i.e. it changes its status according to the result stored in the accumulator. It has 9 flags and they are divided into 2 groups – Conditional Flags and Control Flags.

EU: Execution Unit

- **Ex:**



- **Decoding and Opcode:**

- **Ex:**

  i. **mov BL, CL ; CL: Source register and BL: Destination Register**
  ; Content of CL is copied in BL
  ; Assembler provides 'Opcode' for the entire instruction.

  ii. **mov CH, 12H ; mov CH is opcode and 12H is operand**
  ; The value 12H is stored in CH.

  iii. **mov AX, 3AF1H ; mov AX: Opcode and 3AF1: Operand**
  ; 16bit value will be stored in AX.

1. **General purpose register: Used by the ALU: All available for programming.**
   - **AX: AH and AL**
   - **BX: BH and BL**
   - **CX: CH and CL**
   - **DX: DH and DL**

2. **Operand Register: It is a temporary register: Not available for programming.**
   - ✓ **Used to store temporary results by the processor**
   - ✓ **16-bit**
   - ✓ **Ex: xchg BX, CX;**

   xchg: Exchange or swap the content of registers. *μP* stores the content of one register in Operand then does the copying operation in another.

8086 μP Memory Segmentation

00000H

20-Bit → 0000 0000 0000 0000 0000

⋮

→ 1111 1111 1111 1111 1111

FFFFFH

16-Bit address

2^{20} = 1 MB of memory

CPU

8-Bit

00000H

CS

DS

ES

SS

FFFFFH

memory

PA = ?

DS : BP =

16-Bit registers.

- CS
- DS
- ES
- SS

Register present inside 8086

ES : BP = ?

Starting position of memory : [Base Address]

Segment Register

CS, DS, ES, SS

- CS : IP ← PA
- DS : SI
- ES : DI
- SS : SP, BP

ES : DI = Extra

PA = Seg. Reg. : Offset Reg.

CS : SI = Code

Offset Reg : Offset Add

IP
SI
DI
SP
BP

Code : CS : IP = ?

Seg. : Offset Reg.

Seg. → DS : SI = Data

off.

SS : BP = Stack

Memory Segmentation

- ✓ The memory in 8086 μP based system is organized as segmented memory.
- ✓ The CPU is able to address 1MB of memory; with 20-bit address bus $= 2^{20} = 1MB$.
- ✓ The complete physical available memory may be divided into number of logical segments.
- ✓ *Logical Segments*: Since, the memory is virtually or logically divided into 4-divisions. Actually in memory there is no physical division/segmentation marked as boundary.
- ✓ 4-Divisions :
  - i. Code Segment
  - ii. Stack Segment
  - iii. Data Segment
  - iv. Extra Segment

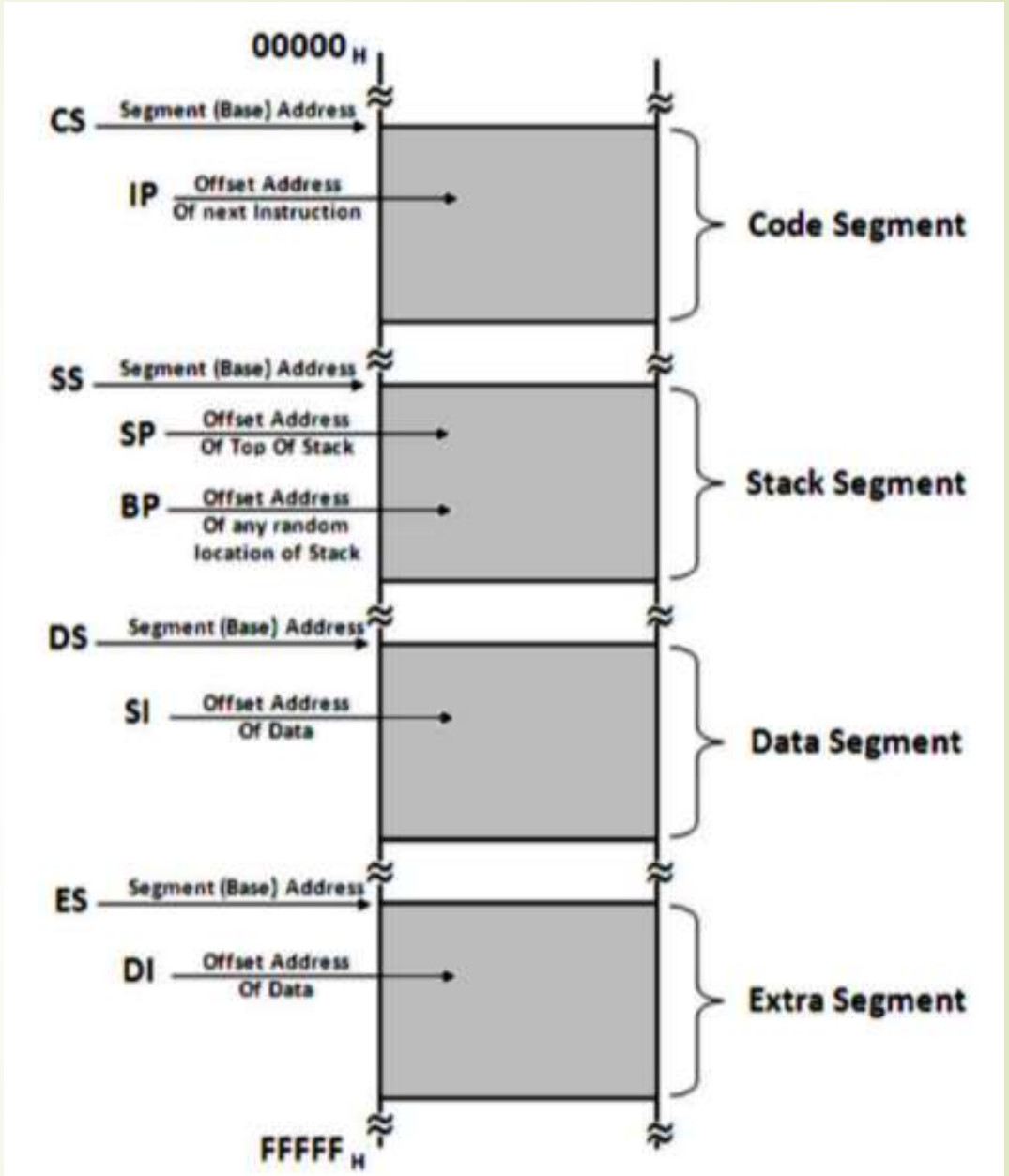


Fig: Memory Segment Available for 8086 μP

Memory Segmentation

i. Code Segment

- ✓ Programs are stored sequentially.
- ✓ After every instruction the address will be incremented.
- ✓ Stored in downward direction (Sequentially).

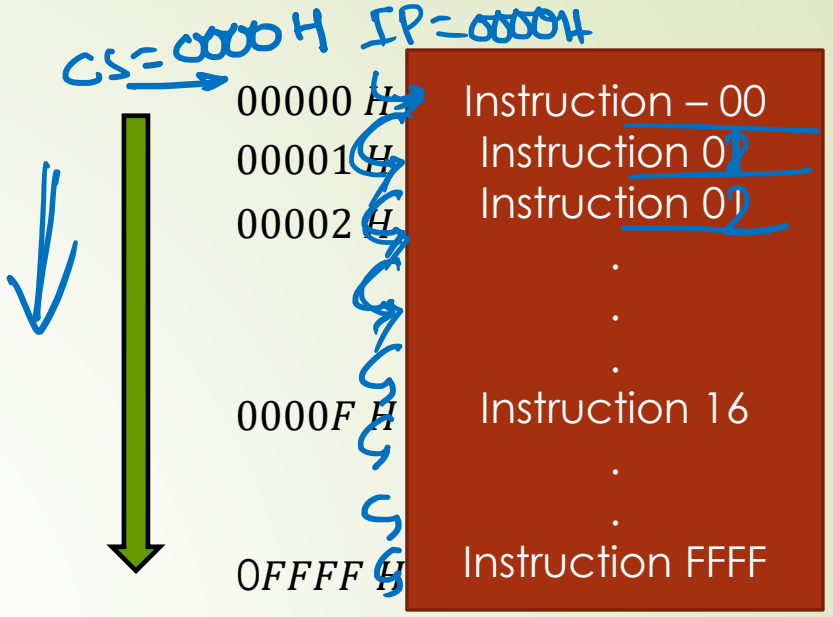


Fig: 8086: Code Segment: Fetching of Instruction

ii. Stack Segment

- ✓ Data is stored in a structured way,
- ✓ After every data stored in the stack (PUSH) the address goes decremented.
- ✓ LIFO.

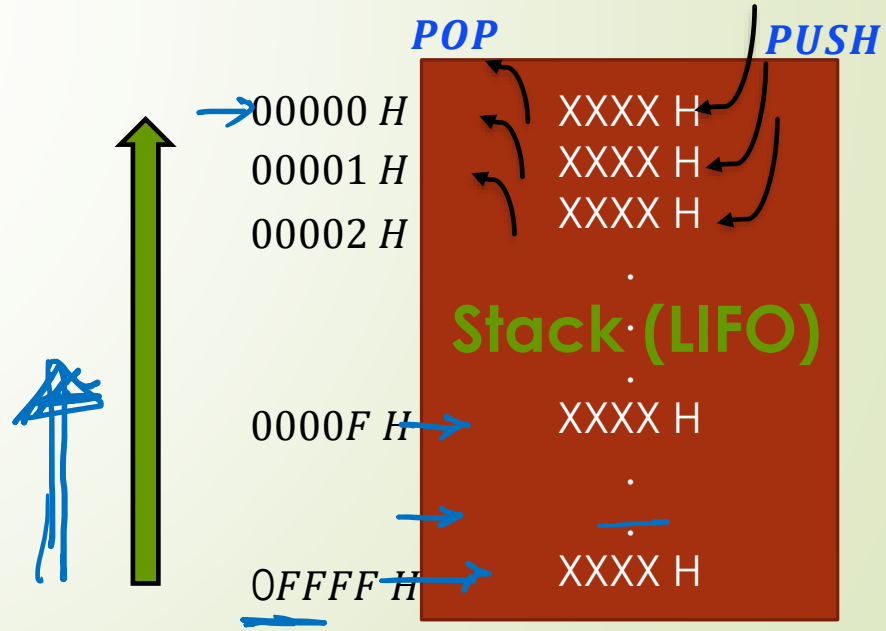


Fig: 8086: Stack Segment: Fetching of Data

Memory Segmentation

Stack Segment Register and Offset Register

- Every segment has one segment register and one offset register. However, stack segment has two offset registers:
  - SP (Stack Pointer): Points to the top of the stack.
  - BP (Base Pointer): Can move anywhere between stack.
- **Advantages of BP**:
  - Suppose we have 20 data; that is pushed. The address stars decrementing and pushes the values as shown in Fig.
  - Now, SP(old) changes at every push and the current location will be the top of the stack as indicated by SP(new).
  - Suppose, by mistake the second number i.e. pushed (X_2) is wrong. you can't directly access the memory location, randomly. However, you can pop all the 18 data from X_{20} to X_3 to get back the X_2 and change. This is impractical.
  - Therefore, we have another offset address called BP. It can move anywhere in stack segment.

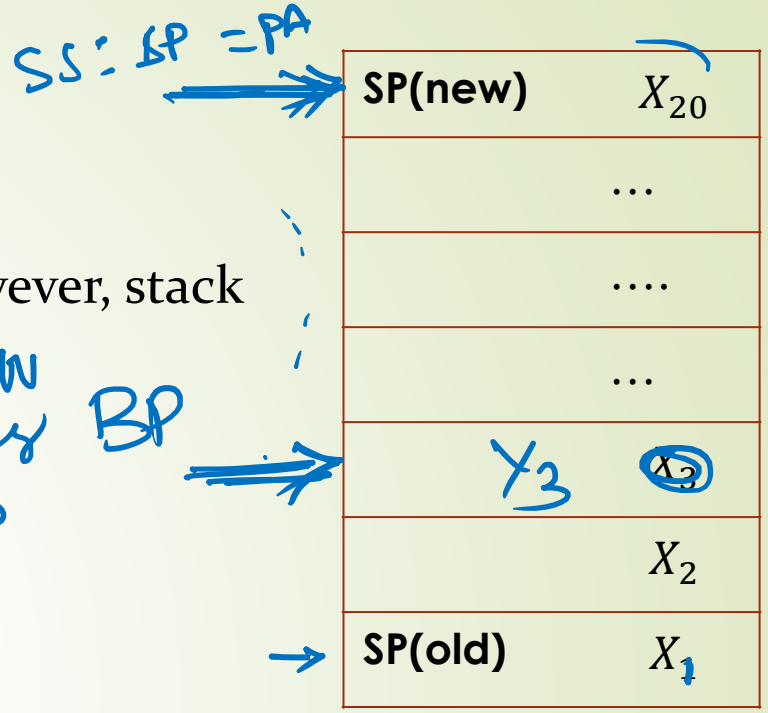


Fig: 8086: Stack Operation: Change in address location during each PUSH instruction.

Memory Segmentation

iii. Data Segment

- Data is stored randomly and in unstructured way.
  - Can be stored in upward directions.
  - Can be stored in downward directions.
  - Or at any random location.

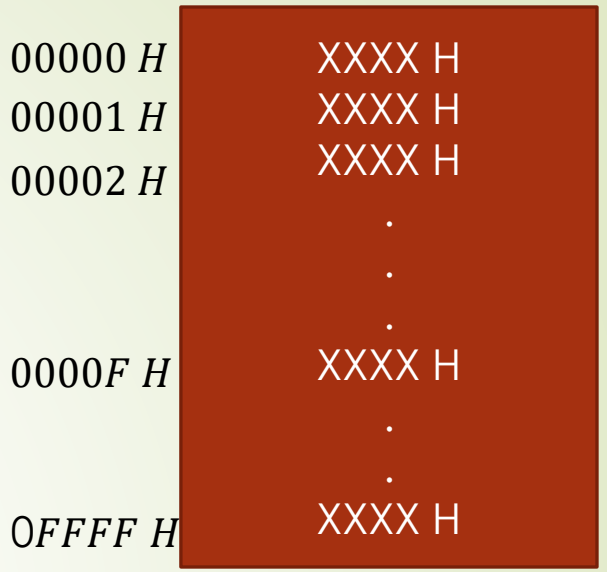
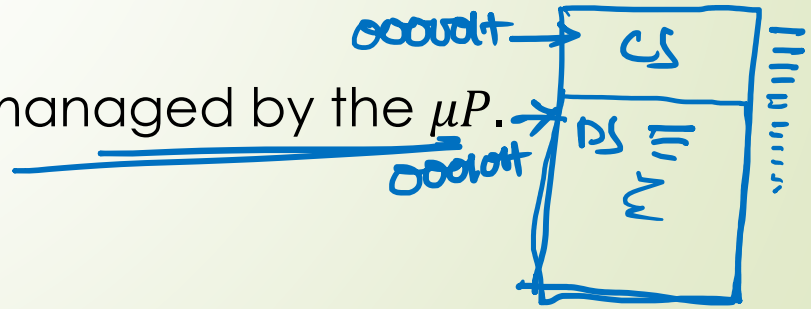


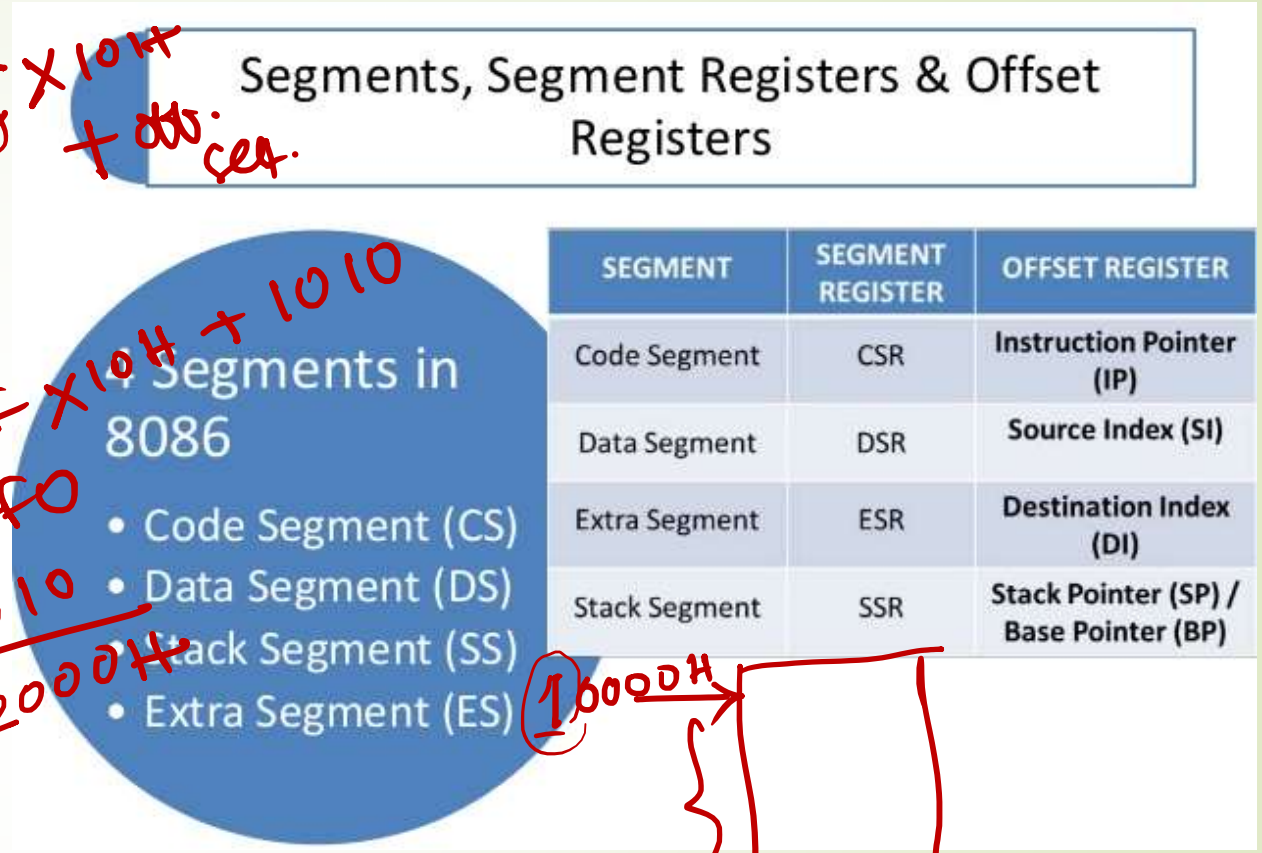
Fig: 8086: Data Segment: Fetching of Data

- **Who created segmentation?**
  - ✓ The programmer creates the memory segment. Before writing the code; the segments needs to be initialized.
  - ✓ Segments are created by the programmer and managed by the μP.
  - ✓ Advantages:
    - It prevents overwriting.
    - It creates or provides lots of memory locations to store data/code.



Memory Segmentation

- The maximum size of segment memory is 64KB. As 8086 have only 16-bit register (which are used to form Physical Address), it can address at a time $2^{16} = 64KB$ memory. Minimum size of a segment can be 16 bytes.
- The scheme used in the 8086 is called segmentation. Every address has two parts, a SEGMENT and an OFFSET (Segment: Offset)
- The segment indicates the starting of a 64 kilobyte portion of memory, in multiples of 16 (why?). The offset indicates the position within the 64k portion.



$$Absolute\ Memory\ Location = Segment\ Value * 10H + Offset\ Value$$

Seg. Reg. entry → Base address

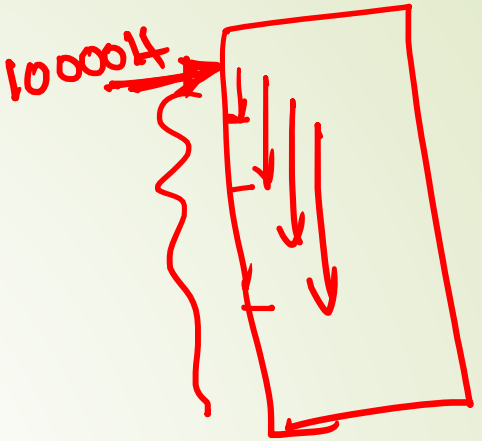
Offset Reg. entry → offset address → "16 bit"

CS = 1000H

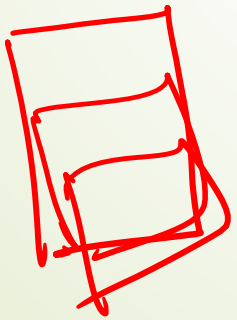
PA = Base address = CS x 10H = 1000 x 10H = 10000H

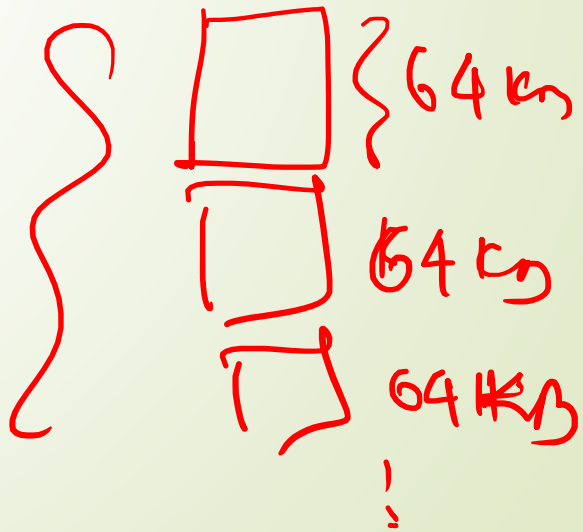
0000H ↕ FFFFH

2^{16} = 64KB



one seg.





Maximum Seg. Size ?

PA = Seg. Reg * offset Reg. = Seg. Reg x 10H + offset Address

Base Address

0000 0000 0000 0000

0000H ← 00000H

0001H ← 00010H } 10H = 16 Bytes

0002H ← 00020H

0000H ← } 10H = 16 Bytes

0010H ←

0020H ←

Memory Segmentation

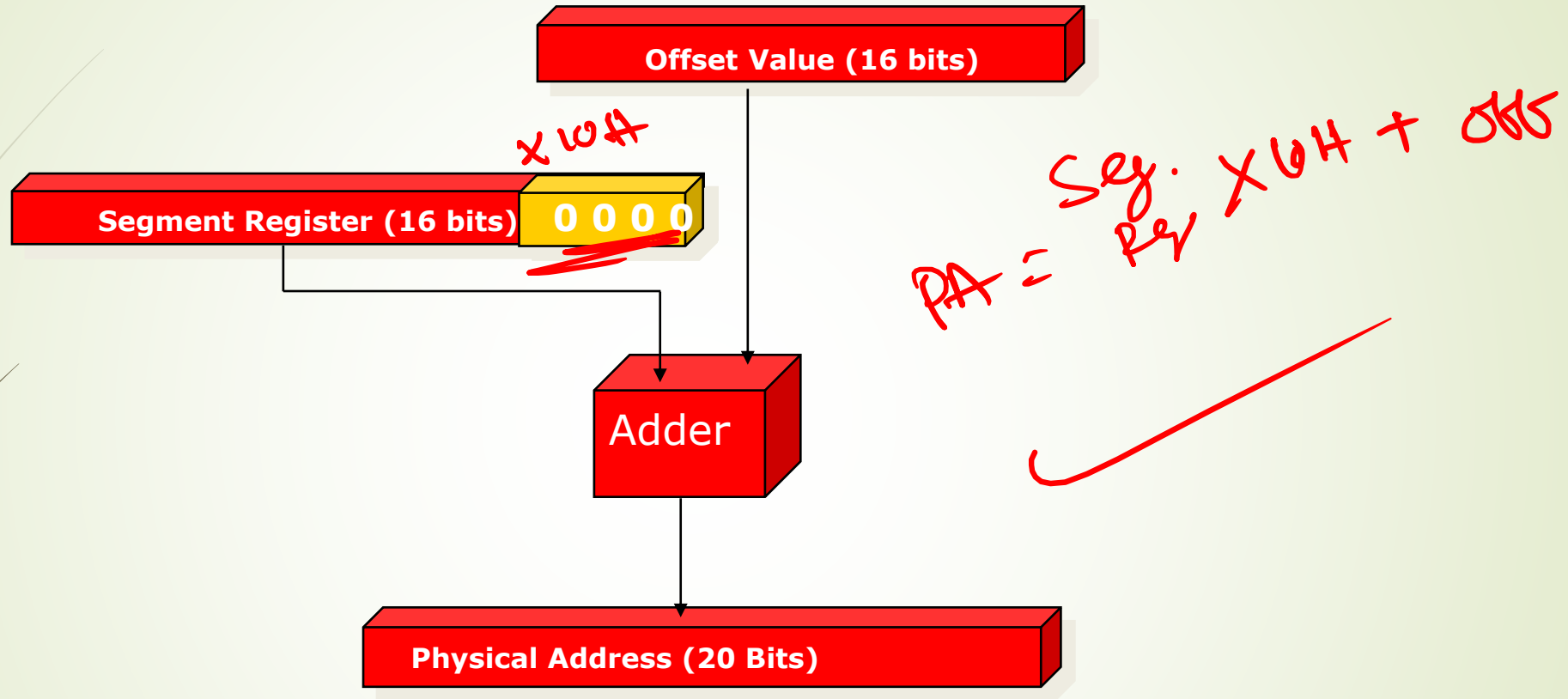


Fig: 8086: Memory Segment: Physical Address Calculation

Memory Segmentation

✓ Ex: Segment:Offset

i. Segment = F000 H

Offset = FFFD H

```
  F0000
+ FFFD
------
 FFFFD   or  1,048,573 (decimal)
```

✓ Ex: Se

ii. Segr

Offse

cimal)

✓ Ex: Seg

iii. Segment = FFFF H

Offset = FFFF H

```
+   FFFF
-------
 10FFEF   or   1,114,095 (decimal)
```

If a program tried to use a Segment:Offset pair that exceeded a 20-bit Absolute address (1MiB), the CPU would truncate the highest bit (an 8086/8088 CPU has only 20 address lines), effectively mapping any value over FFFFFh (1,048,575) to an address within the first Segment. Thus, 10FFEFh was mapped to FFEFh.

Memory Segmentation

Types of Segmentation:

- **Overlapping Segment –** A segment starts at a particular address and its maximum size can go up to 64kilobytes. But if another segment starts along this 64kilobytes location of the first segment, then the two are said to be *Overlapping Segment*.

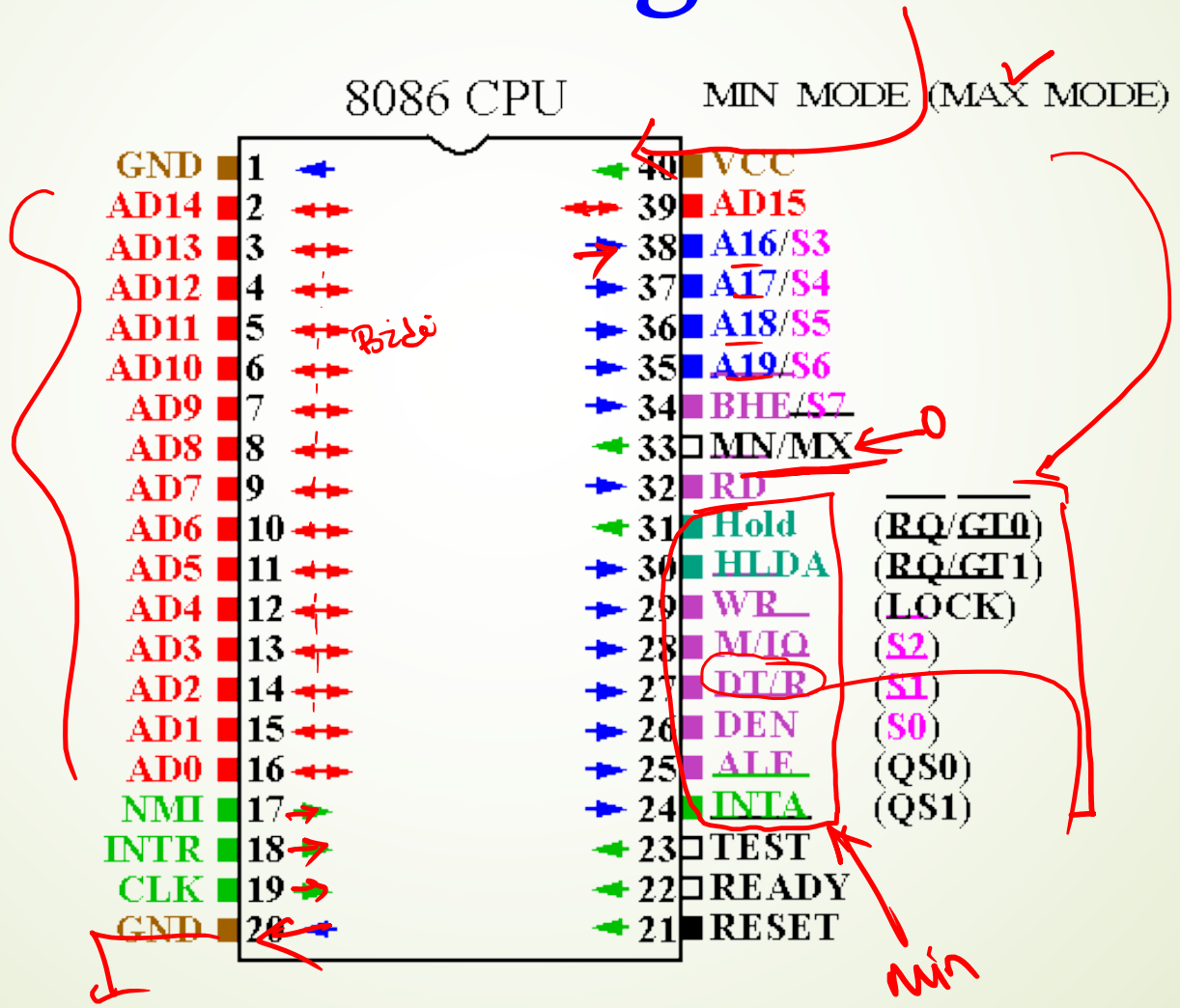
- **Non-Overlapped Segment –** A segment starts at a particular address and its maximum size can go up to 64kilobytes. But if another segment starts before this 64kilobytes location of the first segment, then the two segments are said to be *Non-Overlapped Segment*.
  - **2 Types:**
    - **Continuous non-overlapping**
    - **Discontinuous non-overlapping**

Memory Segmentation

Advantages of the Segmentation:

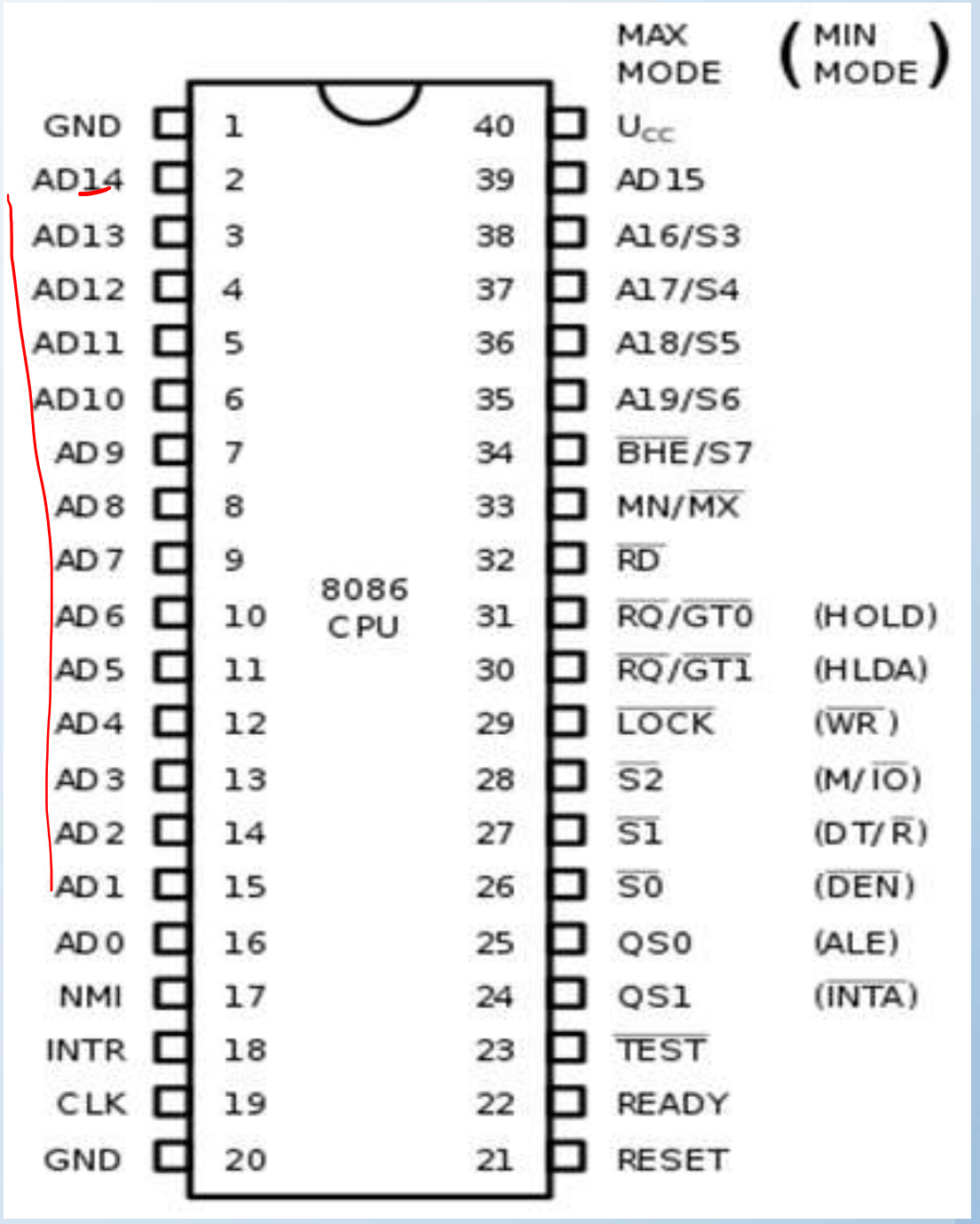
- ✓ It provides a powerful memory management mechanism.
- ✓ Data related or stack related operations can be performed in different segments.
- ✓ Code related operation can be done in separate code segments.
- ✓ It allows to processes to easily share data.
- ✓ It allows to extend the address ability of the processor, i.e. segmentation allows the use of 16 bit registers to give an addressing capability of 1 MB. Without segmentation, it would require 20 bit registers.
- ✓ It is possible to enhance the memory size of code data or stack segments beyond 64 KB by allotting more than one segment for each area.

8086 μP Signal Description: Pin Diagram



Pin Diagram

- 16-bit processor with three clock rates; 5 MHz, 8MHz and 10MHz; packaged in a 40 pin DIP.
- It can operates in a single processor or multi-processor configurations to achieve high performance.
- The 8086 signals can be categorized in 3-groups:
  - ✓ Signals having common functions.
  - ✓ Signals with special functions in minimum mode.
  - ✓ Signals with special functions in maximum mode.



Pin Diagram

- **Power supply (40)**

  It uses 5V DC supply at V_{CC} pin 40, and uses ground at V_{SS} pin 1 and 20 for its operation.

- **Clock signal (19)**

  Clock signal is provided through Pin-19. It provides timing to the processor for operations. Its frequency is different for different versions, i.e. 5MHz, 8MHz and 10MH**z.**

- **AD0-AD15 (2-16 & 39)**

  - ✓ Time multiplexed memory I/O address and data lines.
  - ✓ AD0-AD7 carries low order byte data and AD8-AD15 carries higher order byte data.
  - ✓ Address remain on the pins for '$\boldsymbol{T_1}$' state, while data is available on the data bus during T_2, T_3, T_w and T_4 states (These are clock cycle states).
  - ✓ These lines are active high and floats to a tristate during interrupt acknowledgement and local bus acknowledgement cycle.

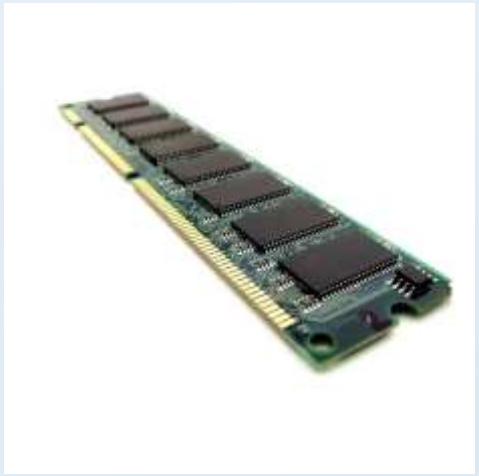
Pin Diagram

- **A19/S6, A18/S5, A17/S4, A16/S3: (Higher Address Lines and Status Signal) (35-38)**
  - ✓ During T_1 state these are most significant address line for memory operation.
  - ✓ During T_2, T_3, T_w and T_4 states, these lines are available for status information of I/O or memory operation.
  - ✓ If microprocessor is requested for interrupt and μP is ready to serve the interrupt, then '**S5**' flag bit is set. **S5** = interrupt enable flag.
  - ✓ The '**S3**' and '**S4**' together indicates which segment is presently used for memory access: [Refer Table]
  - ✓ '**S6**' is always low and is reserved for future use.

S3	S4	Indication
0	0	Alternate Data
0	1	Stack
1	0	Code or None
1	1	Data

Pin Diagram

- **$\overline{BHE}$/S7: (Bus High Enable and Status Signal) (34)**
  - ✓ The $\overline{BHE}$ is an active low pin; used to indicate transfer of higher byte data (D_{15}-D_{08})over the data bus, if the pin is selected low.
  - ✓ $\overline{BHE}$ low during **'T1'** and remain active for next cycles.
  - ✓ It is also used to drive chip select for odd-address memory bank or peripherals in combination with **'A0'** bit of address.
  - ✓ The status information (**S7**) is available during T_2, T_3, T_w and T_4 states. The signal is active low and is tristate during 'Hold'.



$\overline{BHE}$	A0	Indication
0	0	Whole Word (2Bytes)
0	1	Upper byte from or to odd address
1	0	Lower byte from or to even address
1	1	None

Pin Diagram

- **Read($\overline{RD}$) (32)**

  It is available at pin 32 and is used to read signal for Read operation.

- **Ready (22)**

  It is available at pin 22. It is an acknowledgement signal from I/O devices that data is transferred. It is an active high signal. When it is high, it indicates that the device is ready to transfer data. When it is low, it indicates wait state.

- **RESET (21)**

  It is available at pin 21 and is used to restart the execution. It causes the processor to immediately terminate its present activity. This signal is active high for the first 4 clock cycles to RESET the microprocessor.

- **GND (20)**

  Ground connection for circuit.

Pin Diagram

- **INTR (18)**

  It is available at pin 18. It is an interrupt request signal, which is sampled during the last clock cycle of each instruction to determine if the processor considered this as an interrupt or not.

- **NMI (17)**

  It stands for non-maskable interrupt and is available at pin 17. It is an edge triggered input, which causes an interrupt request to the microprocessor.

- **TEST (23)**

  This signal is like wait state and is available at pin 23. When this signal is high, then the processor has to wait for IDLE state, else the execution continues.

- **MN/MX (33)**

  It stands for Minimum/Maximum and is available at pin 33. It indicates what mode the processor is to operate in; when it is high, it works in the minimum mode and vice-versa.

Pin Diagram [Minimum mode operation]

- **M/$\overline{I/O}$ (Memory/Input Output) (28)**
  - ✓ When it is high, it indicates memory operation and when it is low indicates the I/O operation.
  - ✓ The line become active in previous '**T4**' state and remain active for current '**T4**' cycle.
  - ✓ It is tri-stated during local bus "hold Acknowledgement".
- **$\overline{INTA}$: Interrupt Acknowledgement (24)**
  - ✓ Low in this pin indicates that the processor has accepted the interrupt.
  - ✓ It is active low during T_2, T_3, T_w of each interrupt acknowledgement cycle.
- **ALE (25)**
  - ✓ It stands for address enable latch and is available at pin 25.
  - ✓ A positive pulse is generated each time the processor begins any operation. This signal indicates the availability of a valid address on the address/data lines.

Pin Diagram [Minimum mode operation]

- **DT/$\overline{R}$ (27)**
  - ✓ It stands for Data Transmit/Receive signal and is available at pin 27. It decides the direction of data flow through the trans-receiver.
  - ✓ Active High: Sends data from the processor.
  - ✓ Active Low: Receives data into processor.
  - ✓ Tristate during 'Hold acknowledgement'

- **$\overline{DEN}$ (Data Enable) (26)**
  - ✓ This pin indicates the availability of valid data over the data/address lines.
  - ✓ It is used to enable the transreceiver to separate the data from multiplexed address/data signal.
  - ✓ Tristate during 'Hold acknowledgement'

Pin Diagram [Minimum mode operation]

- **HOLD, HLDA (Hold, Hold Acknowledge) (31, 30)**
  - ✓ When another master request for bus line; it has to send a request signal '1' on the HOLD pin.
  - ✓ If the processor is now free to provide its bus to the master it will acknowledge on HLDA pin, in the middle of the clock cycle after completing the current instruction cycle.
  - ✓ When HOLD pin goes low it makes HLAD to low.

Pin Diagram [Maximum mode operation]

- **$\overline{S2}$, $\overline{S1}$ & $\overline{S0}$ (Status Line) (28, 27, 26)**
  - ✓ These status lines indicates types of operations being carried out by the processor.

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Indication
0	0	0	Interrupt Ack.
0	0	1	Read I/O Port
0	1	0	Write I/O Port
0	1	1	Halt
1	0	0	Code Access
1	0	1	Read Memory
1	**1**	**0**	**Write Memory**
1	1	1	Passive

Pin Diagram [Maximum mode operation]

- $\overline{LOCK}$ **(29)**
  - ✓ Low on this pin prevents the master to use system bus.
  - ✓ Output Signal; Lock instruction is used to indicate the master that the processor is preventing to access the system bus.
- $\overline{RQ}/\overline{GT0}$, $\overline{RQ}/\overline{GT1}$ **(Request/Grant) (31, 30)**
  - ✓ These are the Request/Grant signals used by the other processors requesting the CPU to release the system bus. When the signal is received by CPU, then it sends acknowledgment. RQ/GT0 has a higher priority than RQ/GT1.
- $QS1$, **QS0 (Queue Status) (24, 25)**
  - ✓ It provides the status of the code pre-fetch queue.

QS1	QS0	Indication
0	0	No Operation
0	1	First byte of opcode from queue
1	0	Empty Queue
1	1	Subsequent byte from the queue

8086 μP Addressing Modes

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

Addressing Mode

- The 8086 processors let you access memory in many different ways. The 8086 memory addressing modes provide flexible access to memory, allowing you to easily access variables, arrays, records, pointers, and other complex data types.

- It is the manner in which an operand is given to the instruction. Or

- The different ways in which a source operand is denoted in an instruction are known as addressing modes.

- Ex:
  - ✓ If you want to add two numbers the; how you are going to tell the processor to work with the operand.

Addressing Mode

1. Register Addressing 2. Immediate Addressing	Group I : Addressing modes for register and immediate data
3. Direct Addressing 4. Register Indirect Addressing 5. Based Addressing 6. Indexed Addressing 7. Based Index Addressing 8. String Addressing	Group II : Addressing modes for memory data
9. Direct I/O port Addressing 10. Indirect I/O port Addressing	Group III : Addressing modes for I/O ports
11. Relative Addressing	Group IV : Relative Addressing mode
12. Implied Addressing	Group V : Implied Addressing mode

Register Addressing Mode

- This mode involves the use of registers. These registers hold the operands.
- This mode is very fast as compared to others because CPU doesn't need to access memory. CPU can directly perform an operation through registers.

- **Ex:**

```
MOV     AL,  BL     ; copy the data of BL register to AL
MOV     CX, BX      ; copy the data of BX register to CX

INC     BX          ; BX := BX + 1
```

Immediate Addressing Mode

- In this, an 8 bit or 16 bit data is specified as part of the instruction
- Here, CPU doesn't need to search for the data as data is directly given in the instruction. Thus the operation is carried out immediately.

Example:

MOV DL, 08H

The 8-bit data (08_H) given in the instruction is moved to DL

(DL) $\leftarrow$ 08_H

MOV AX, 0A9FH

The 16-bit data ($0A9F_H$) given in the instruction is moved to AX register

(AX) $\leftarrow$ $0A9F_H$

Direct Addressing Mode

- The effective address of memory location at which the data operand is stored is given.
- The effective address is 16 bit number directly written in the instruction.
- The instruction consists of a register and an offset address (indicated by square bracket). To compute physical address, shift left the ***DS*** register and add the ***offset address*** into it.

Example:

MOV CL, [2000H] ; Copy the content of memory location: 'PA = (DS*10H) + 2000H'

MOV CX, [2000H] ; Copy the content of memory location: CL ← 'PA = (DS*10H) + 2000H'
; Copy the content of memory location: CH ← 'PA = (DS*10H) + 2001H'

```
ORG 100h
MOV AX, 2162H   ;copies hexadecimal value 2162h to AX
MOV DS, AX      ;copies value of AX into DS
MOV CX, 24      ;copies decimal value 24 into CX
MOV [481], CX  ;stores the data of CX to memory address 2162:01E1
MOV BX, [481]  ;load data from memory address 2162:01E1 into BX
RET             ;stops the program
```

- **Effective Address and Registers**:

✓ The offset of a memory operand is called the operand's effective address (EA).

✓ It is an unsigned 16 bit no. That expresses the operands distance in byte from the beginning of the segment

✓ 8086 has Base register and Index register

✓ So **EU** calculates **EA** by summing a Displacement, Content of **Base register** and Content of **Index register**.

$$\boldsymbol{EA} = \{\boldsymbol{Base\ Register}\} + \{\boldsymbol{Index\ Register}\} + \{\boldsymbol{8\ or\ 16\ Bit\ Displacement}\}$$

$$= \begin{Bmatrix} BX \\ BP \end{Bmatrix} + \begin{Bmatrix} SI \\ DI \end{Bmatrix} + \begin{Bmatrix} 08\ or\ 16\ Bit \\ Displacement \end{Bmatrix}$$

✓ Based on this EA combinations four indirect addressing modes are available:

1. Register Indirect
2. Register Relative
3. Based Index
4. Relative Based Index

Register Indirect Addressing Mode

- The register indirect addressing mode uses the offset address which resides in one of these three registers i.e., BX, SI, DI.
- The sum of offset address and the DS value shifted by one position generates a physical address.
- If BX, SI or Di is used as base register then DS will be used as segment register.
- If BP is used then by default it will work on stack segment (SS).

Example:

```
MOV     BX,  5000H  ; Copy the data 5000 in BX register

MOV     CX, [BX]    ; Copy the content of memory location: CL ← 'PA = (DS*10H) + [BX]H'
                    ; Copy the content of memory location: CH ← 'PA = (DS*10H) + [BX]+1H'
```

```
ORG 100h
MOV AX, 0708h    ;set AX to hexadecimal value of 0708h.
MOV DS, AX       ;copy value of AX to DS
MOV CX, 0154h    ;set CX to hexadecimal value of 0154h.
MOV SI, 42Ah     ;set SI to 42Ah.
MOV [SI], CX     ;copy contents of CX to memory at 0708:042Ah
RET              ;returns to operating system
```

Register Relative Addressing Mode

- In this addressing mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any register BX, BP, SI or DI in the default (either DS or ES) segment.

- MOV BX, 1000H

  MOV AX, 50H[BX] ; Copy the content of memory location: DS*10H + [BX]+50H in AL

  ; Copy the content of memory location: DS*10H + [BX]+50H+1 in AH

- MOV [BX+5], DX

```
ORG 100h
MOV AX, 0708h    ;set AX to hexadecimal value of 0708h.
MOV DS, AX       ;copy value of AX to DS
MOV CX, 0154h    ;set CX to hexadecimal value of 0154h.
MOV BX, 42Ah     ;set BX to 42Ah.
MOV [BX+5],DX    ;copy contents of DX to memory at 0708:042Fh
RET              ;returns to operating system
```

Base Indexed Addressing Mode

- The effective address of the data is formed by adding content of the base register (BX or BP) to the content of an index register (SI or DI). The default segment for data may be DS or ES.

- MOV AX, [BX][SI]

  MOV [BX][DI], AX ; [Operation will be carried out in data segment]: Copy the content of AL into the memory location DS*10H + [BX] + [DI] and AH in DS*10H + [BX] + [DI] +1

- MOV CL, [BP+SI]; [Operation will be carried out in Stack segment]: Copy the content of memory location SS*10H + [BP] + [SI] in CL.

Relative Base Indexed Addressing Mode

- The effective address of the data is formed by adding content of the base register (BX or BP), the content of an index register (SI or DI) and an 8-bit or 16-bit displacement. The default segment for data may be DS or ES.

- MOV AX, 50H[BX][SI]

  MOV [BX+DI+1FH], AX ; [Operation will be carried out in data segment]: Copy the content of AL into the memory location DS*10H + [BX] + [DI] + 1FH and AH in DS*10H + [BX] + [DI] +21H

- MOV CL, [BP+SI+20]; [Operation will be carried out in Stack segment]: Copy the content of memory location SS*10H + [BP] + [SI] + 20 in CL.

Implied Addressing Mode

- Instructions using this mode has no operands.
- The instruction itself will specify the data to be operated by the instruction.

- STC ; Set Carry Flag, CF := 1.
- CLC; Clear the carry flag. CF := 0.
- DAA; Decimal Adjust after Addition.

I/O Addressing Mode

- Used to access date from standard I/O mapped devices or ports.
- In direct addressing, an 8-bit port address is directly specified in the instruction.

Example: IN AL, [09H] ;Content of port with address 09H is moved to AL register.

- In indirect port addressing mode, the instruction will specify the name of the register which holds the port address. In 8086, the 16-bit port address is stored in DX register.

Example: OUT [DX], AX ;Content of AX is moved to port with address specified by DX register.

8086 μP Instruction Set

Instructions of 8086 μP

8086 instructions can be categories in 8 groups.

1. **Data Transfer/copy Instructions**
2. **Arithmetic and Logical Instructions**
3. **Shift Rotate Instructions**
4. **Loop Instructions**
5. **Machine Control Instructions**
6. **Flag Manipulation Instructions**
7. **Branch Instructions**
8. **String Instructions**

Instructions of 8086 μP

1. Data Transfer/copy Instructions

- These instructions are used to transfer the data from the source operand to the destination operand.
- Instruction to transfer a word
  - **MOV** – Used to copy the byte or word from the provided source to the provided destination.
  - **PUSH** – 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

Instructions of 8086 μP

1. Data Transfer/copy Instructions

- Instructions for input and output port transfer
  - **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.
- Instructions to transfer the address
  - **LEA** – Used to bad the address of operand into the provided register.
  - **LDS** – Used to bad DS register and other provided register from the memory
  - **LES** – Used to bad ES register and other provided register from the memory.

Instructions of 8086 μP

1. Data Transfer/copy Instructions

- Instructions to transfer flag registers
  - **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.

Instructions of 8086 μP

2. Arithmetic and logical Instructions

- These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.
- Instructions to perform addition
  - **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

Instructions of 8086 μP

2. Arithmetic and logical Instructions

- Instructions to perform subtraction
  - **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.
- Instruction to perform multiplication
  - **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.

Instructions of 8086 μP

2. Arithmetic and logical Instructions

- Instructions to perform division
  - **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.
  - **CBW** – Used to fill the upper byte of the word with the copies of sign bit of the lower byte.
  - **CWD** – Used to fill the upper word of the double word with the sign bit of the lower word.

Instructions of 8086 μP

2. Arithmetic and logical Instructions

- These instructions are used to perform operations where data bits are involved, i.e. operations like logical also called as Bit Manipulation instructions.

- Instructions to perform logical operation
  - **NOT** – Used to invert each bit of a byte or word.
  - **AND** – Used for multiplying each bit in a byte/word with the corresponding bit in another byte/word.
  - **OR** – Used for adding 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.

Instructions of 8086 μP

3. Shift/Rotate Instructions [Bit Manipulation]

- Instructions to perform shift operations
  - **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.
- Instructions to perform rotate operations
  - **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.

Instructions of 8086 μP

4. Loop Instructions

- These instructions are used to execute the given instructions for number of times. Following is the list of instructions under this group −
  - **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

Instructions of 8086 μP

5. Branch Instructions

- These instructions are used to transfer/branch the instructions during an execution. It includes the following instructions –
- Instructions to transfer the instruction during an execution without any condition –
  - **CALL** – Used to call a procedure and save their return address to the stack.
  - **RET** – Used to return from the procedure to the main program.
  - **JMP** – Used to jump to the provided address to proceed to the next instruction.
  - Instructions to transfer the instruction during an execution with some conditions –
  - **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

Instructions of 8086 μP

5. Branch Instructions

- **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
- **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

Instructions of 8086 μP

6. Flag Manipulation Instructions

- These instructions are used to control the processor action by setting/resetting the flag values.
  - **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.

Instructions of 8086 μP

7. Machine Control Instructions

- These instructions are used to control the machine status.
  - **Wait** – Wait for the test input pin to go low.
  - **HLT** – Halt the Processor
  - **NOP** – No operation (Till 4 clock cycles).
  - **ESC** – Escape to external device like numeric coprocessor. Bus control will be given to other processors
  - **LOCK** – Bus lock instruction prefix.

Instructions of 8086 μP

8. String Manipulation Instruction

- String is a group of bytes/words and their memory is always allocated in a sequential order.
  - ✓ **REP** – Used to repeat the given instruction till $CX \neq 0$.
  - ✓ **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.

8086 μP: Introduction to Programming

Thank You
For Your Attention