Unit 2: Basic Computer Architecture

Unit 4: Assembly Language Programming

PART-II

8086 Microprocessor & Assembly Language Programming with 8086

8086 Microprocessor Syllabus

- Logical Block Diagram
- Segment Registers,
- Memory Segmentation
- Bus Interface Unit and Execution Unit
- Pipelining

Programming with Intel 8086 MP Syllabus

- Macro Assembler
- Assembling and Linking
- Assembler Directives, Comments
- Instructions: LEA, MUL, DIV, LOOP, AAA, DAA
- INT 21H Functions
  - 01H, 02H, 09H, 0AH, 4CH
- INT 10H Functions (Introduction Only)
  - 00H, 01H, 02H, 06H, 07H, 08H, 09H, 0AH

Features of Microprocessor 8086

1. **Buses:**
   - **Address Bus:** 8086 has a **20-bit address bus**, hence it can access 2^{20} Byte memory i.e. **1MB**. The **address range** for this memory is **00000H … FFFFFH**.
   - **Data Bus:** 8086 has a **16-bit data bus** i.e. it can access 16 bit data in one operation. Its ALU and internal data registers are also 16-bit. **Hence** 8086 is called as a **16-bit µP**.
   - **Control Bus:** The control bus carries the signals responsible for performing various operations such as **RD**, **WR** etc.
2. 8086 supports **Pipelining.**

   It is the process of **"Fetching the next instruction, while executing the current instruction"**. Pipelining improves performance of the system.
3. 8086 has **2 Operating Modes**.
   - **I. Minimum Mode** … here 8086 is the only processor in the system (uniprocessor).
   - **II. Maximum Mode** … 8086 with other co-processors like 8087-NDP **'Numeric Data Processor',** 8089-IOP **'Input Output Processor'** etc reads the message from memory, carries out the operation, and notifies the CPU when it has finished. Maximum mode is intended for multiprocessor configuration.

Features of Microprocessor 8086

4. 8086 provides **Memory Banks**.

   The entire memory of 1 MB is **divided into 2 banks of 512KB each**, in order to transfer 16-bits in 1 cycle. The banks are called **Lower Bank** (even) and **Higher Bank** (odd)

5. 8086 supports **Memory Segmentation**.

   Segmentation means dividing the memory into logical components. Here the memory is divided into **4 segments: Code, Stack, Data and Extra Segment**.

6. 8086 has **256 interrupts**.

   The ISR addresses for these interrupts are stored in the IVT (Interrupt Vector Table).

7. 8086 has a **16-bit IO address** ∴ it can access **216 IO ports** (216 = 65536 i.e. 64K IO Ports).

Features of Microprocessor 8086

- It has an **instruction queue**, which is capable of storing six instruction bytes from the memory resulting in faster processing.
- It was the first 16-bit processor having 16-bit ALU, 16-bit registers, internal data bus, and 16-bit external data bus resulting in faster processing.
- It is available in 3 versions based on the frequency of operation –
  - 8086 → 5MHz
  - 8086-2 → 8MHz
  - (c)8086-1 → 10 MHz
- It uses **two** stages of pipelining, i.e. **Fetch Stage** and **Execute Stage**, which improves performance.
- Fetch stage can prefetch up to 6 bytes of instructions and stores them in the queue.
- Execute stage executes these instructions.
- It has 256 vectored interrupts.
- It consists of 29,000 transistors.

Features of Microprocessor 8086

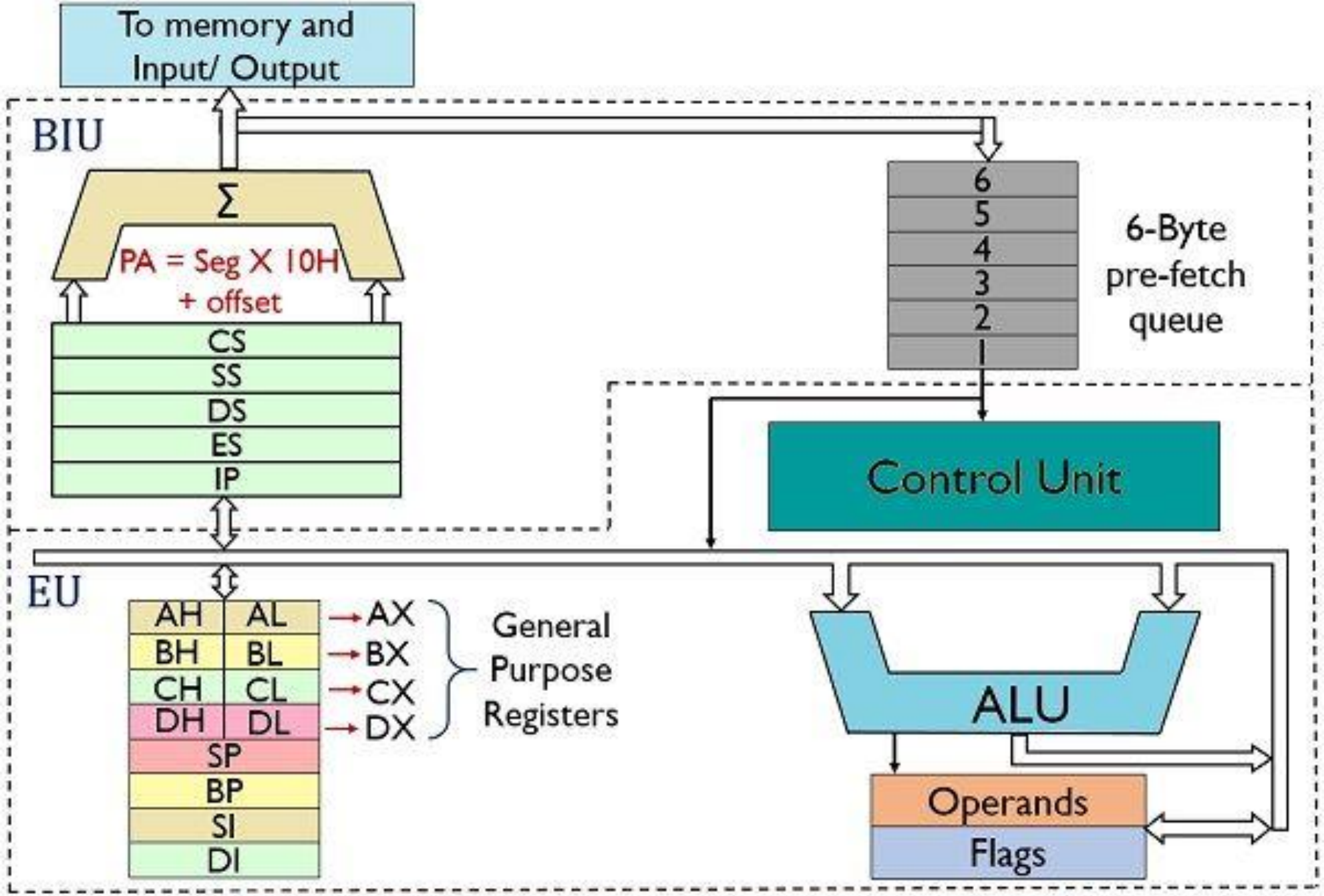
- Intel 8086 is an 16 bit microprocessor
- It is modified version of 8085.
- It is a 40 pin integrated circuit.
- Its operating frequencies are 5, 8, and 10 MHz.
- It has 20-bit address bus.
- It supports Pipelining
- It has almost 29000 transistors.

Comparison between 8085 & 8086 MP

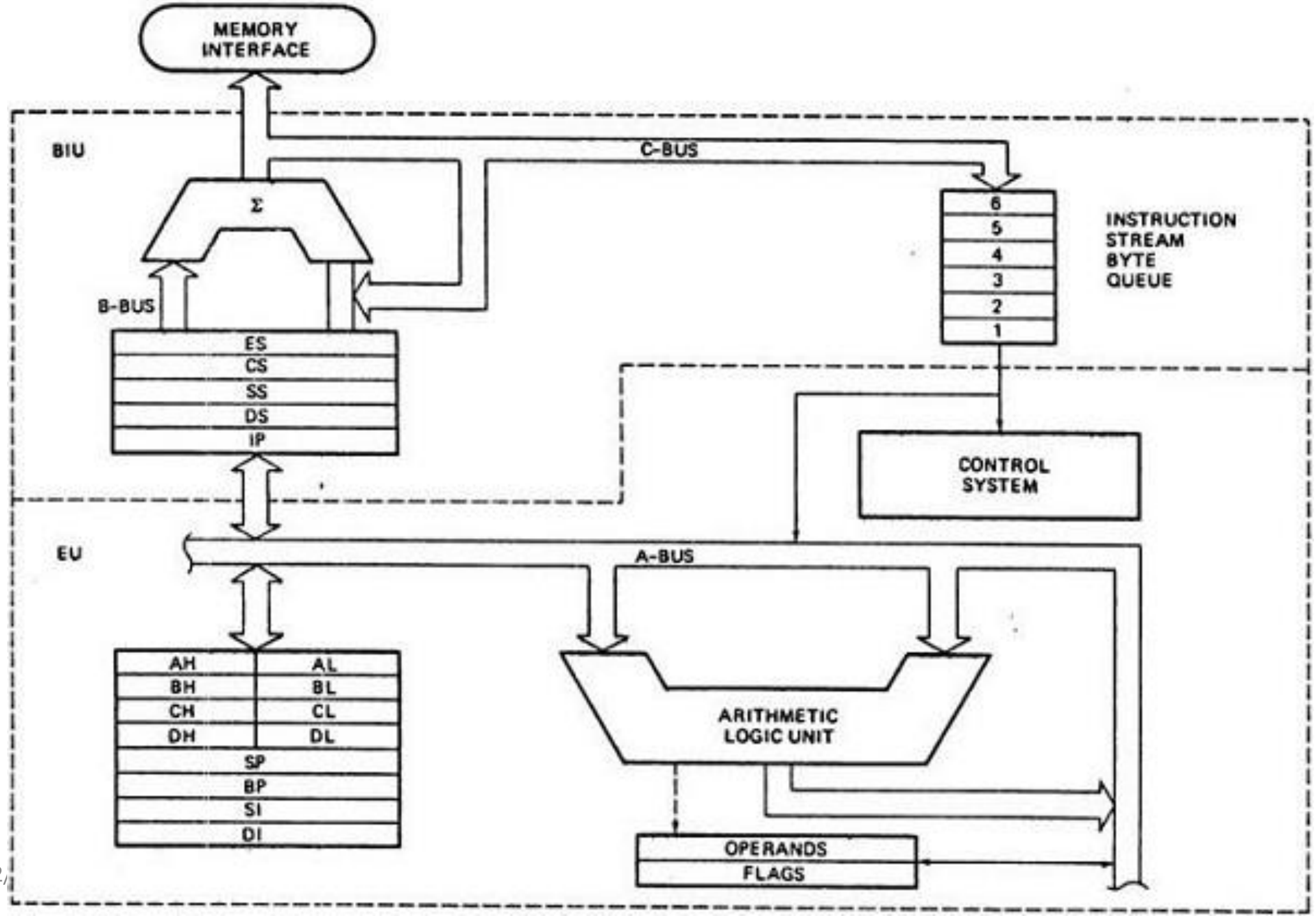
- **Size** – 8085 is 8-bit microprocessor, whereas 8086 is 16-bit microprocessor.
- **Address Bus** – 8085 has 16-bit address bus while 8086 has 20-bit address bus.
- **Memory** – 8085 can access up to 64Kb, whereas 8086 can access up to 1 Mb of memory.
- **Instruction** – 8085 doesn’t have an instruction queue, whereas 8086 has an instruction queue.
- **Pipelining** – 8085 doesn’t support a pipelined architecture while 8086 supports a pipelined architecture.
- **I/O** – 8085 can address 2^8 = 256 I/O's, whereas 8086 can access 2^16 = 65,536 I/O's.
- **Cost** – The cost of 8085 is low whereas that of 8086 is high.

Logical Block Diagram/Internal Architecture of 8086 Microprocessor

- The architecture of 8086 microprocessor is composed of 2 major units, the **BIU** i.e., **Bus Interface Unit** and **EU** i.e., **Execution Unit**.
- The figure below shows the block diagram of the architectural representation of the 8086 microprocessor:



Block Diagram of 8086 Microprocessor


MEMORY INTERFACE
BIU
C-BUS
Σ
B-BUS
6
5
4
3
2
1
INSTRUCTION STREAM BYTE QUEUE
ES
CS
SS
DS
IP
CONTROL SYSTEM
EU
A-BUS
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
SI
DI
ARITHMETIC LOGIC UNIT
OPERANDS
FLAGS

8086 MP Internal Components

❖As 8086 does 2-stage pipelining, its architecture is divided into two units:

1. **Bus Interface Unit (BIU)**
2. **Execution Unit (EU)**

Bus Interface Unit (BIU)

1. It provides the **interface** of 8086 **to** other devices.
2. It **operates w.r.t. Bus cycles.**
   This means it performs various machine cycles such as Mem Read, IO Write etc to transfer data with Memory and I/O devices.
3. It performs the following functions:
   a) It **generates** the 20-bit **physical address** for memory access.
   b) **Fetches Instruction** from memory.
   c) **Transfers data** to and from the **memory and IO**.
   d) **Supports Pipelining** using the 6-byte instruction queue.

Bus Interface Unit (BIU)

❖ **The main components of the BIU are as follows:**

a) Segment Registers:

1. CS Register

- CS holds the **base** (Segment) **address** for the **Code Segment**. All programs are stored in the Code Segment.
- It is **multiplied by 10H** (16d), to give the **20-bit physical address** of the **Code Segment**.
- E.g.: If **CS = 4321H** then CS x 10H = **43210H → Starting address** of Code Segment.
- CS register cannot be modified by executing any instruction except branch instructions.

2. DS Register

- DS holds the **base** (Segment) **address** for the **Data Segment**.
- It is **multiplied by 10H** (16d), to give the **20-bit physical address** of the **Data Segment**.
- E.g.: If **DS = 4321H** then DS x 10H = **43210H → Starting address** of Data Segment.

Bus Interface Unit (BIU)

a) Segment Registers:

3. SS Register

- SS holds the **base** (Segment) **address** for the **Stack Segment**.
- It is **multiplied by 10H** (16d), to give the **20-bit physical address** of the **Stack Segment**.
- Eg: If **SS = 4321H** then SS x 10H = **43210H → Starting address** of Stack Segment

4. ES Register:

- ES holds the **base** (Segment) **address** for the **Extra Segment**.
- It is **multiplied by 10H** (16d), to give the **20-bit physical address** of the **Extra Segment**.
- Eg: If **ES = 4321H** then ES x 10H = **43210H → Starting address** of Extra Segmen**t**.

Bus Interface Unit (BIU)

b) Instruction Pointer (IP Register)

- It is a **16-bit register**.
- It **holds offset of** the **next instruction in** the **Code Segment**.
- Address of the **next instruction** is calculated as **CS x 10H + IP**. IP is **incremented after every instruction byte is fetched.**
- IP gets a new value whenever a branch occurs.
- **Physical address = Segment Address x 10h + Offset Address**

Bus Interface Unit (BIU)

c) Address Generation Circuit

- The BIU has a **Physical Address Generation Circuit.** It generates the 20-bit physical address using Segment and Offset addresses using the formula:

Physical address = Segment Address x 10h + Offset Address

- ***The Segment address is left shifted by 4 positions, this multiplies the number by 2^4 = 16 (i.e. 10h) and then the offset address is added.***
- **Example:** *If Segment address is **1234h** and offset address is **0005h**, then the physical address (12345h) is calculated as follows:*
  - *1234h = $(0001\ 0010\ 0011\ 0100)_{binary}$*
  - *Left shift by four positions and we get $(0001\ 0010\ 0011\ 0100\ \mathbf{0000})_{\mathbf{binary}}$ i.e. 12340h*
  - *Now add $(0000\ 0000\ 0000\ 0101)_{binary}$ i.e. 0005h and we get $(0001\ 0010\ 0011\ 0100\ 0101)_{binary}$ i.e. 12345h.*

Bus Interface Unit (BIU)

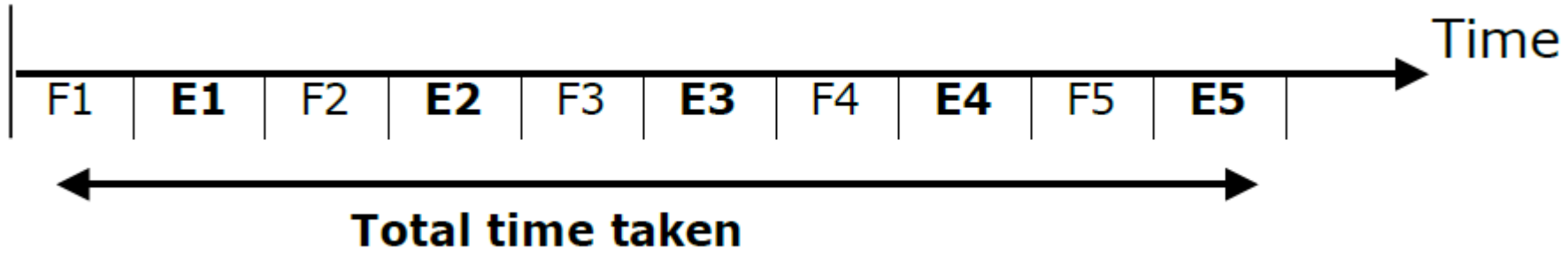
d) 6-Byte Pre-Fetch Queue (Pipelining)

- It is a **6-byte FIFO RAM** used to implement **Pipelining**.
- *Fetching the next instruction while executing the current instruction is called **Pipelining***.
- **BIU fetches** the next "**six instruction-bytes"** from the Code Segment and stores it into the queue. Execution Unit (EU) removes instructions from the queue and executes them.
- **The queue is refilled when at least two bytes are empty as 8086 has a 16-bit data bus.**
- Pipelining **increases** the **efficiency** of the µP.
- Pipelining **fails when** a **branch** occurs, as the pre-fetched instructions are no longer useful.
- Hence as soon as 8086 detects a branch operation, it clears/discards the entire queue. Now, the next six bytes from the new location (branch address) are fetched and stored in the queue and Pipelining continues.

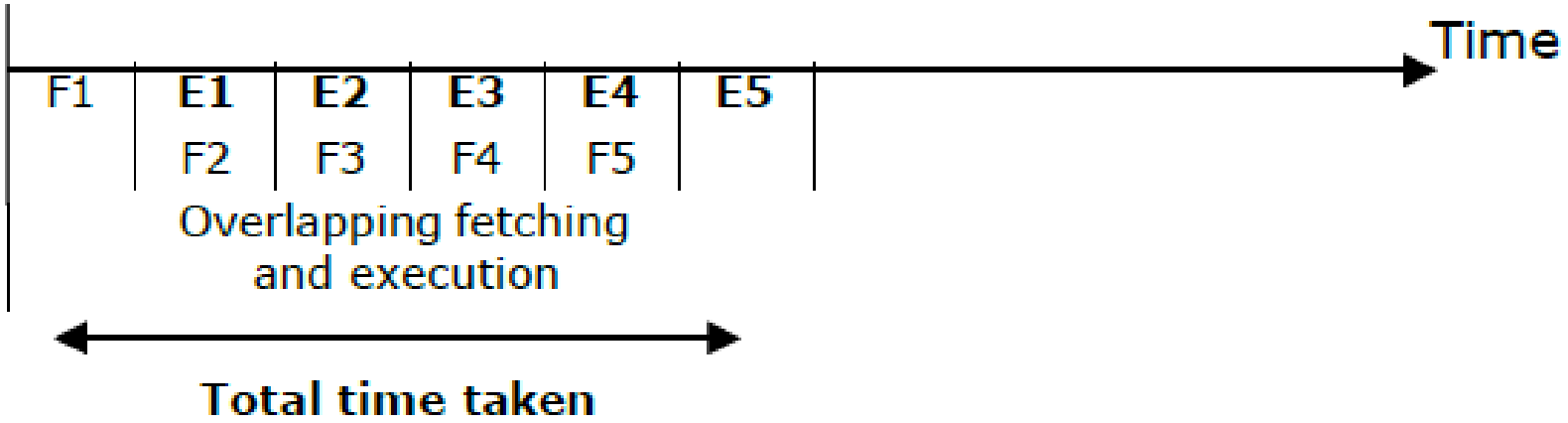
Bus Interface Unit (BIU)

d) 6-Byte Pre-Fetch Queue (Pipelining)

- **NON-PIPELINED PROCESSOR EG: 8085**



- **PIPELINED PROCESSOR EG: 8086**



Q. Explain the feature of pipelining and queue in 8086 architecture.

Answer:

1. The process of fetching the next instruction when the present instruction is being executed is called as pipelining.
2. Pipelining has become possible due to the use of queue.
3. BIU (Bus Interfacing Unit) fills in the queue until the entire queue is full.
4. BIU restarts filling in the queue when at least two locations of queue are vacant.
5. **Advantages of pipelining:**
   I. The execution unit always reads the next instruction byte from the queue in BIU. This is faster than sending out an address to the memory and waiting for the next instruction byte to come.
   II. In short pipelining eliminates the waiting time of EU and speeds up the processing. The 8086 BIU will not initiate a fetch unless and until there are two empty bytes in its queue. 8086 BIU normally obtains two instruction bytes per fetch.

Q. Explain the feature of pipelining and queue in 8086 architecture.

Answer:

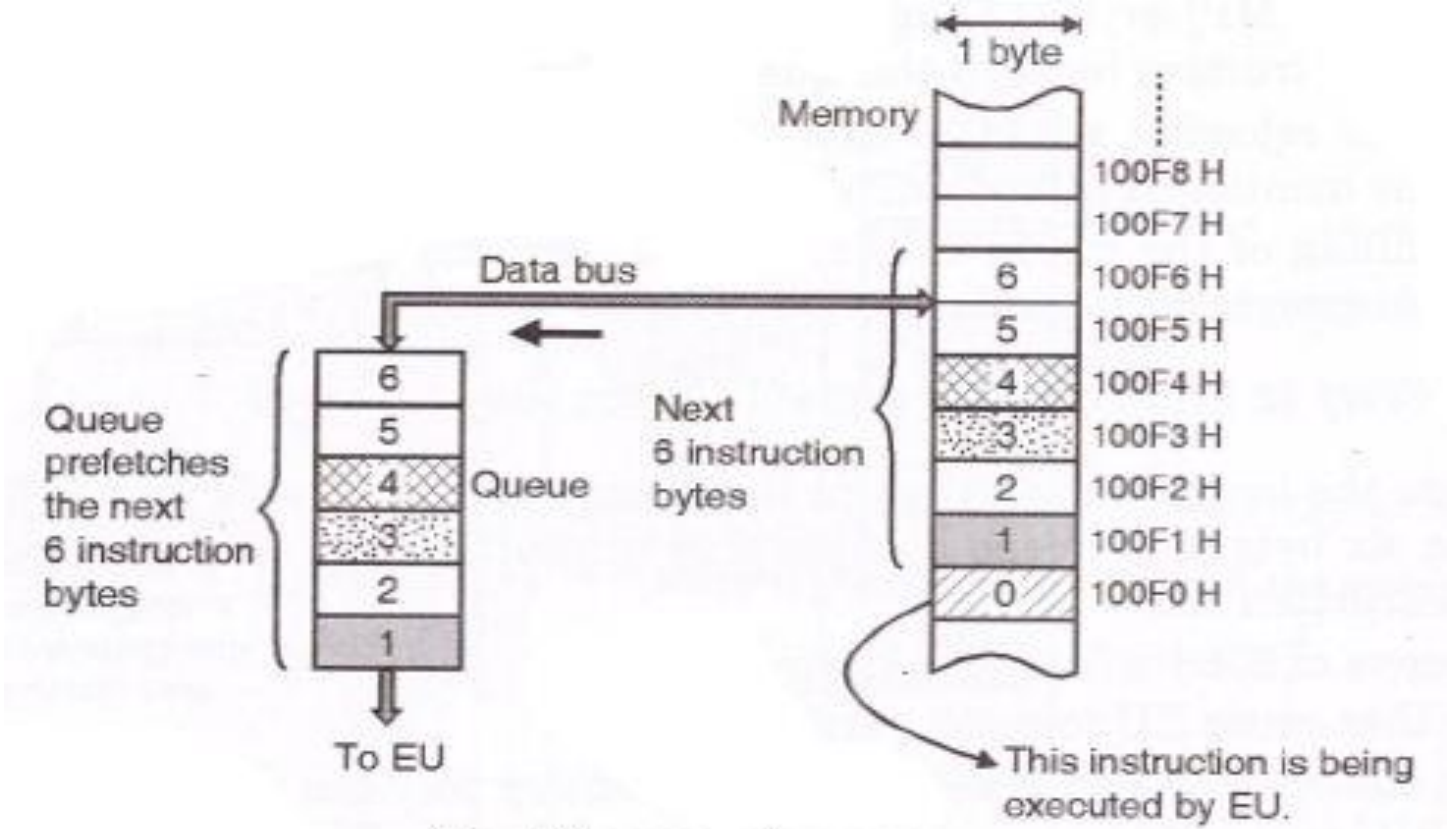
The Instruction Queue:

1. The execution unit (EU) is supposed to decode or execute an instruction.
2. Decoding does not require the use of buses.
3. When EU is busy in decoding and executing an instruction, the BIU fetches up to six instruction bytes for the next instructions.
4. These bytes are called as the pre-fetched bytes and they are stored in a first in first out (FIFO) register set, which is called as a queue.

Q. Explain the feature of pipelining and queue in 8086 architecture.

Answer:

Significance of Queue:



Significance of queue

Q. Explain the feature of pipelining and queue in 8086 architecture.

Answer:

Significance of Queue:

1. As shown in the above figure, while the EU is busy in decoding the instruction corresponding to memory location 100F0, the BIU fetches the next six instruction bytes from locations 100F1 to 100F6 numbered as 1 to 6.
2. These instruction bytes are stored in the 6 byte queue on the first in first out (FIFO) basis.
3. When EU completes the execution of the existing instruction and becomes ready for the next instruction, it simply reads the instruction bytes in the sequence 1, 2… from the Queue.
4. Thus the Queue will always hold the instruction bytes of the next instructions to be executed by the EU.

Execution Unit (EU)

1. It **fetches** instructions **from** the **Queue in BIU**, **decodes** and **executes them**.
2. It performs **arithmetic**, **logic** and **internal data transfer** operations.
3. It sends request signals to the BIU to access the external module.
4. It **operates w.r.t. T-States** (clock cycles).

Execution Unit (EU)

❖ **The main components of the EU are as follows:**

a) General Purpose Registers:

8086 has four 16-bit general-purpose registers **AX**, **BX**, **CX** and **DX**. These are **available** to the programmer, for storing values during programs. Each of these can be **divided** into two **8-bit registers** such as AH, AL; BH, BL; etc. Beside their general use, these registers also have some **specific functions.**

❑ ***AX*** *Register (16-Bits)*

It holds operands and results during **multiplication** and **division** operations. **All IO data transfers** using IN and OUT instructions use A reg (AL/AH or AX).

It functions as accumulator during **string operations**.

❑ ***BX*** *Register (16-Bits)*

Holds the **memory address** (offset address), in **Indirect Addressing modes**.

Execution Unit (EU)

a) General Purpose Registers:

- ***CX*** *Register (16-Bits)*

Holds **count** for instructions like: **Loop**, **Rotate**, **Shift** and **String** Operations.

- ***DX*** *Register (16-Bits)*

It is used with AX to hold **32 bit** values during **Multiplication** and **Division**.
It is used to **hold** the **address** of the **IO Port** in **indirect IO addressing** mode.

Execution Unit (EU)

b) Special Purpose Registers

- *Stack Pointer (**SP** 16-Bits)*

It is holds **offset address of** the **top of the Stack. Stack is a set of memory locations operating in LIFO manner. Stack is present in the memory in Stack Segment.**

SP is used with the SS Reg to calculate physical address for the Stack Segment. It used during instructions like PUSH, POP, CALL, RET etc. During PUSH instruction, SP is decremented by 2 and during POP it is incremented by 2.

- *Base Pointer (**BP** 16-Bits)*

BP can hold **offset address of** any location in the **stack segment.**

It is used to access random locations of the stack.

- *Source Index (**SI** 16-Bits)*

It is normally used to hold the **offset address** for **Data segment** but can also be used for other segments using Segment Overriding. It holds **offset address** of **source data** in Data Seg, during **String Operations.**

Execution Unit (EU)

b) Special Purpose Registers

- *Destination Index (**DI** 16-Bits)*

It is normally used to hold the **offset address** for **Extra segment** but can also be used for other segments using Segment Overriding. It holds **offset address** of **destination** in Extra Segment, during **String Operations.**

c) ALU (16-Bits)

It has a **16-bit ALU**. It performs 8 and 16-bit arithmetic and logic operations.

d) Operand Register

It is a 16-bit register used by the control register to hold the operands temporarily.

It is **not available** to the Programmer.

e) Instruction Register and Instruction Decoder (Present inside the Control Unit)

The **EU fetches an opcode from** the **queue into** the **Instruction Register**.

The **Instruction Decoder decodes** it and sends the information to the control circuit for execution.

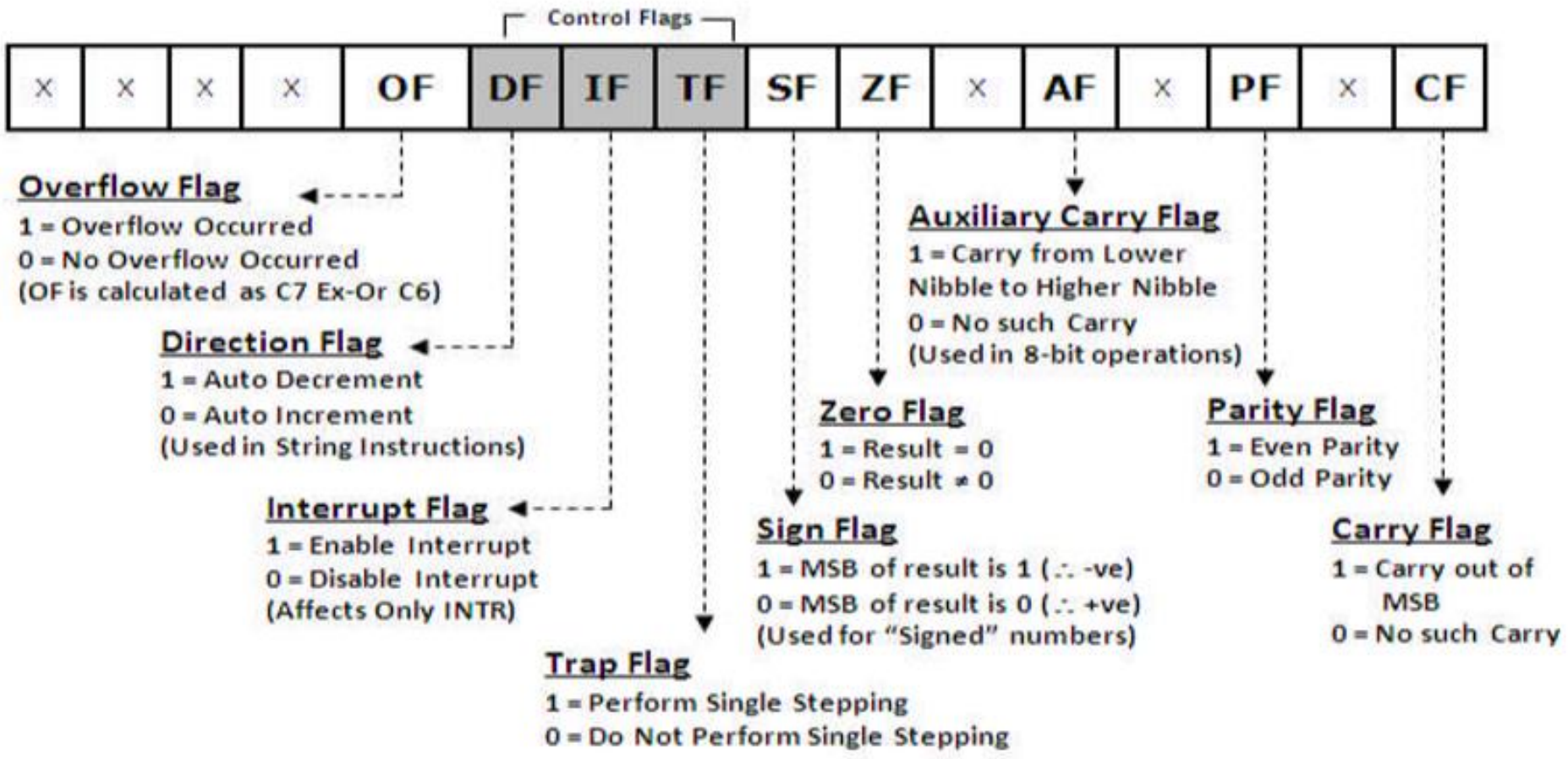
Execution Unit (EU)

f) Flag Register (16-Bits)

- It has 9 Flags.
- These flags are of two types: **6-Status** (Condition) Flags and **3-Control** Flags.
- **Status flags** are affected by the ALU, after every arithmetic or logic operation. They give the **status of the current result**.
- The **Control flags** are used to control certain operations. They are changed by the programmer.

Execution Unit (EU)

f) Flag Register (16-Bits)



Execution Unit (EU)

f) Flag Register (16-Bits)

I. STATUS FLAGS

*1) Carry flag (**CY**)*

It is **set** whenever there is a **carry** {or borrow} out of the MSB of a the result (D7 bit for an 8-bit operation D15 bit for a 16-bit operation)

*2) Parity Flag (**PF**)*

It is **set** if the result has **even parity**.

*3) Auxiliary Carry Flag (**AC**)*

It is **set** if a carry is generated out of the **Lower Nibble**. It is used only in 8-bit operations like DAA and DAS.

Execution Unit (EU)

f) Flag Register (16-Bits)

I. STATUS FLAGS

*4) Zero Flag (**ZF**)*

It is **set** if the result is **zero**.

*5) Sign Flag (**SF**)*

It is **set** if the **MSB** of the result is **1**.

For **signed** operations, such a number is treated as **–ve**.

*6) Overflow Flag (**OF**)*

It will be set if the **result of** a **signed operation** is **too large to fit** in the number of bits available to represent it. It can be **checked using** the **instruction INTO** (Interrupt on Overflow).

Execution Unit (EU)

f) Flag Register (16-Bits)

II. CONTROL FLAGS

*1. Trap Flag (**TF**)*

It is used to **set** the Trace Mode i.e. start **Single Stepping Mode**.

Here the µP is **interrupted after every instruction** so that, the **program** can be **debugged**.

*2. Interrupt Enable Flag (**IF**)*

It is used to mask (disable) or unmask (enable) the INTR interrupt.

*3. Direction Flag (**DF**)*

If this flag is **set, SI** and **DI** are in **auto-decrementing** mode in **String Operations**.

NEED FOR SEGMENTATION/ CONCEPT OF SEGMENTATION

1) Segmentation means **dividing** the memory into **logically different parts called segments**.
2) 8086 has a **20-bit address bus**, hence it can access 2^{20} Bytes i.e. **1MB** memory.
3) But this also means that **Physical address** will now be **20 bit**.
4) It is **not possible** to work with a **20 bit address** as it is **not** a **byte compatible** number. (20 bits is two and a half bytes).
5) To avoid working with this incompatible number, we **create a virtual model** of the memory.
6) Here the memory is **divided into 4 segments**: Code, Stack Data and Extra.
7) The **max size** of a segment is **64KB** and the **minimum size** is **16 bytes**.
8) Now programmer can access each location with a **VIRTUAL ADDRESS**.
9) The Virtual Address is a **combination** of **Segment Address and Offset Address**.
10) **Segment Address indicates where the segment is located in the memory (base address)**
11) **Offset Address gives the offset of the target location within the segment.**

NEED FOR SEGMENTATION/ CONCEPT OF SEGMENTATION

12)Since both, Segment Address and Offset Address are **16 bits each**, they both are **compatible numbers** and can be easily used by the programmer.
13)Moreover, **Segment Address is given only in the beginning** of the program, to initialize the segment. Thereafter, we **only give offset address**.
14)Hence we can access 1 MB memory using only a 16 bit offset address for most part of the program. This is the advantage of segmentation.
15)Moreover, dividing Code, stack and Data into different segments, makes the memory **more organized and prevents accidental overwrites** between them.
16)The **Maximum Size** of a segment is **64KB because offset addresses are of 16 bits**. 2^{16} = 64KB.
17)Moreover, dividing Code, stack and Data into different segments, makes the memory **more organized and prevents accidental overwrites** between them.
18)The **Maximum Size** of a segment is **64KB because offset addresses are of 16 bits**. 2^{16} = 64KB.
19)As max size of a segment is 64KB, programmer can create **multiple Code/Stack/Data segments** till the entire 1 MB is utilized, but **only one of each** type will be **currently active**.

NEED FOR SEGMENTATION/ CONCEPT OF SEGMENTATION

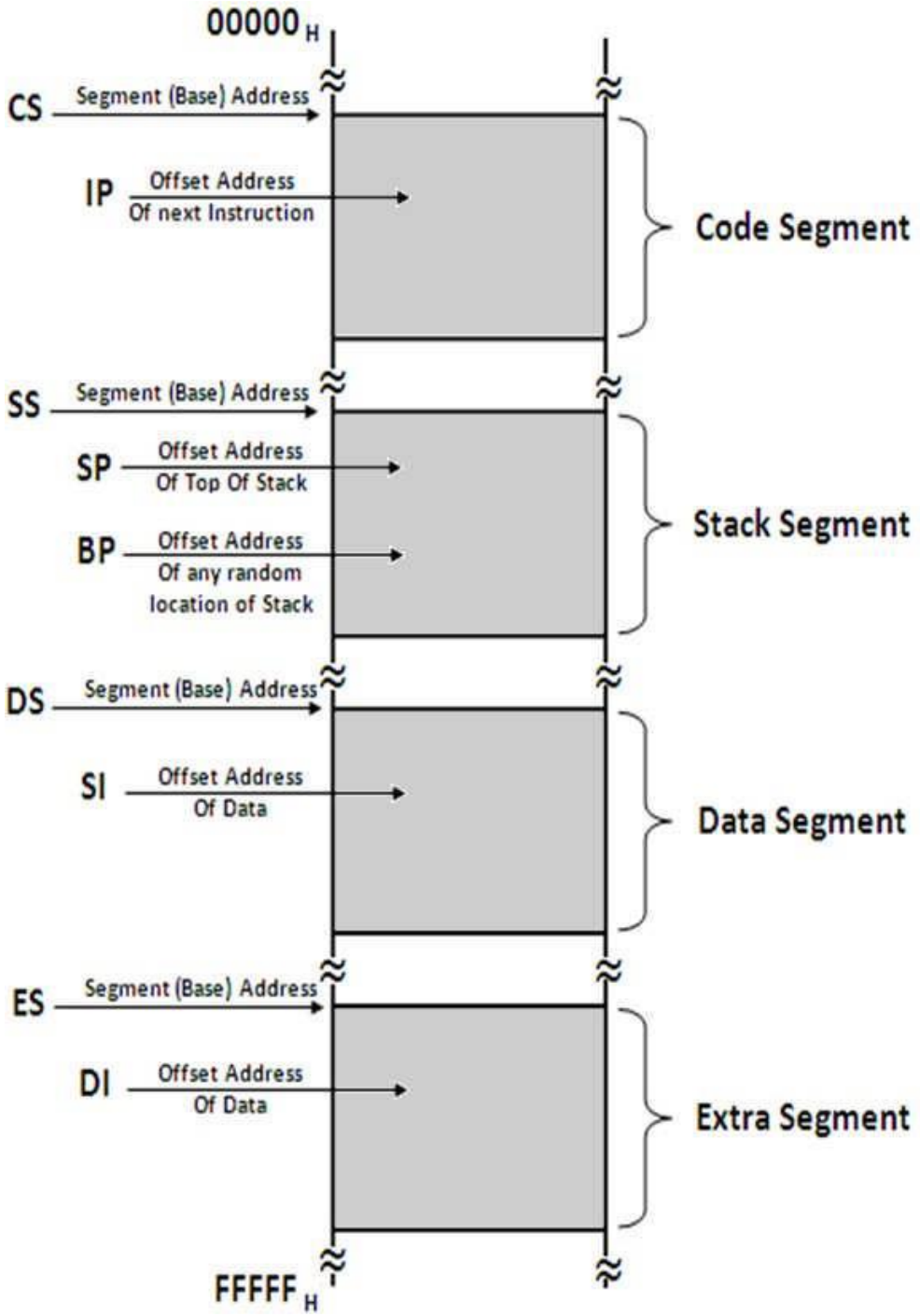
20)The physical address is calculated by the microprocessor, using the formula:

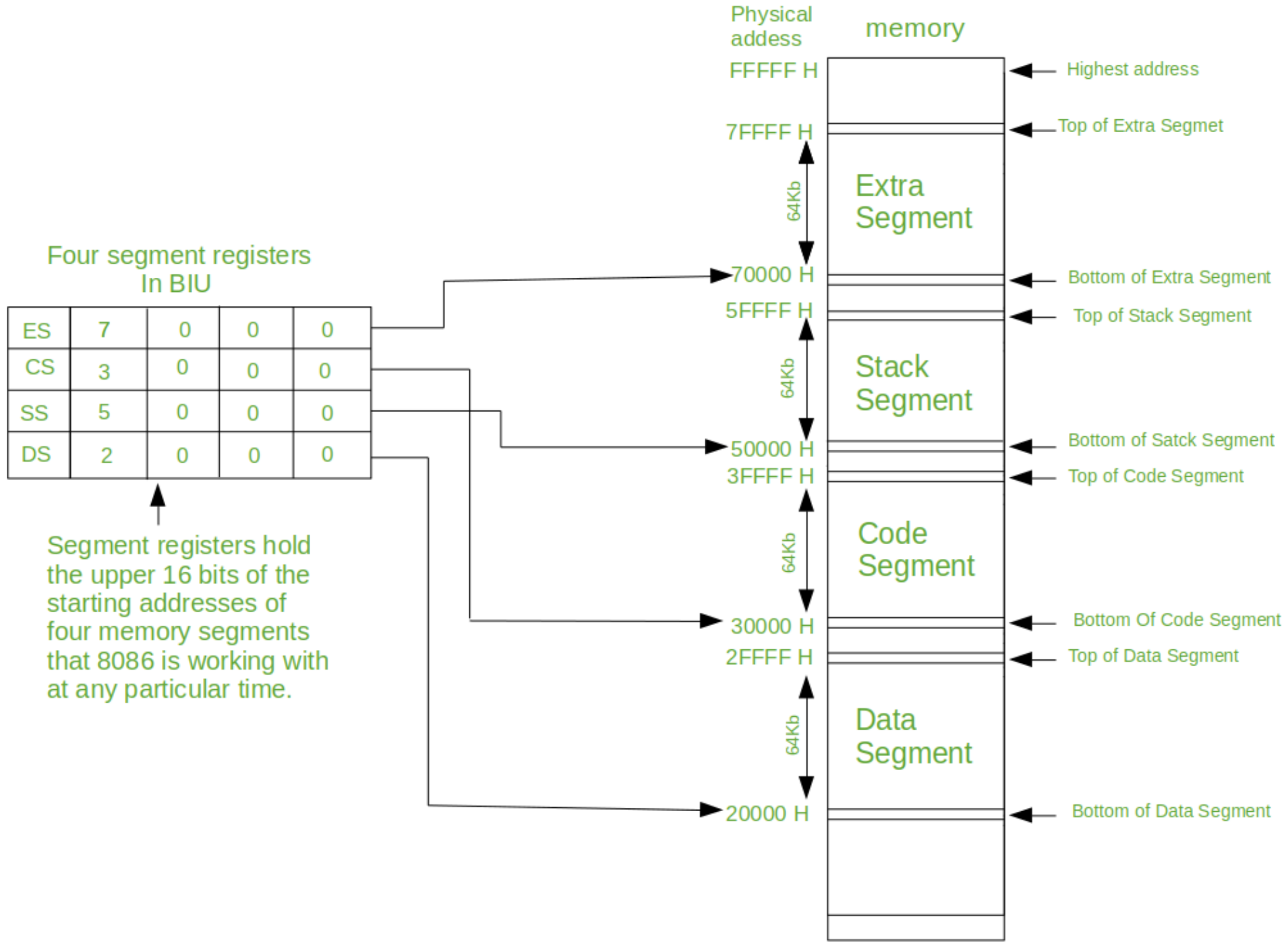
PHYSICAL ADDRESS = SEGMENT ADDRESS X 10H + OFFSET ADDRESS

21)Ex: if Segment Address = 1234H and Offset Address is 0005H then Physical Address = 1234H x 10H + 0005H = 12345H

22)This formula automatically ensures that the **minimum size of a segment is 10H bytes** (10H = 16 Bytes).

MEMORY SEGMENTATION IN 8086




Physical addess
memory
FFFFF H
Highest address
7FFFF H
Top of Extra Segmet
64Kb
Extra Segment
70000 H
Bottom of Extra Segment
5FFFF H
Top of Stack Segment
64Kb
Stack Segment
50000 H
Bottom of Satck Segment
3FFFF H
Top of Code Segment
64Kb
Code Segment
30000 H
Bottom Of Code Segment
2FFFF H
Top of Data Segment
64Kb
Data Segment
20000 H
Bottom of Data Segment
Four segment registers In BIU
ES 7 0 0 0
CS 3 0 0 0
SS 5 0 0 0
DS 2 0 0 0
Segment registers hold the upper 16 bits of the starting addresses of four memory segments that 8086 is working with at any particular time.

MEMORY SEGMENTATION IN 8086

1. Code Segment

- This segment is used to hold the **program** to be executed.
- **Instruction are fetched** from the Code Segment.
- CS register holds the 16-bit **base** address for this segment.
- **IP** register (Instruction Pointer) holds the 16-bit **offset** address.

2. Data Segment

- This segment is used to hold **general data**.
- This segment also holds the **source** operands during **string** operations.
- **DS** register holds the 16-bit **base** address for this segment.
- **BX** register is used to hold the 16-bit **offset** for this segment.
- **SI** register (Source Index) holds the 16-bit **offset** address during String Operations.

3. Stack Segment

- This segment holds the **Stack** memory, which operates in LIFO manner.
- **SS** holds its **Base** address.
- **SP** (Stack Pointer) holds the 16-bit **offset** address of the **Top** of the Stack.
- **BP** (Base Pointer) holds the 16-bit **offset** address during **Random Access**.

MEMORY SEGMENTATION IN 8086

4. Extra Segment

- This segment is used to hold **general data**
- Additionally, this segment is used as the **destination** during **String Operations**. **ES** holds the **Base** Address.
- **DI** holds the **offset** address during string operations.

Advantages of Segmentation:

1. It permits the programmer to access 1MB **using only 16-bit address**.
2. Its **divides** the **memory logically** to store Instructions, Data and Stack separately.

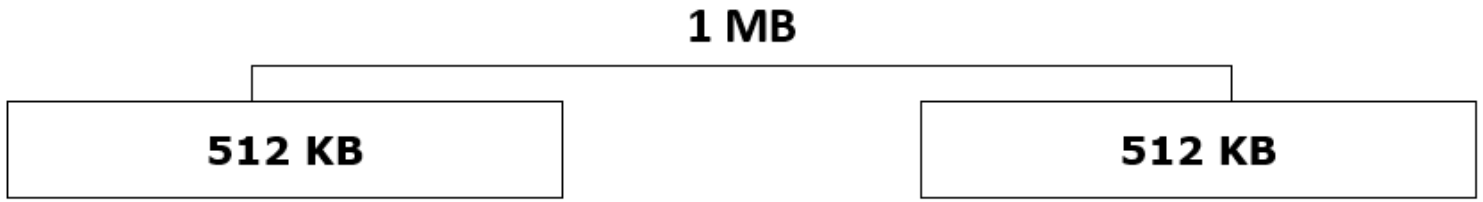
Disadvantage of Segmentation:

1. Although the total memory is 16*64 KB, **at a time only 4*64 KB memory can be accessed**.

Memory Banking in 8086

- As 8086 has a 16-bit data bus**,** it should be able to access 16-bit data **in one cycle**.
- To do so it needs to read from **2 memory locations**, as one memory location carries only one byte. 16-bit data is stored in two consecutive memory locations.
- However, if both these memory locations are in the same memory chip then they cannot be accessed at the same time, as the address bus of the chip cannot contain two address simultaneously.
- Hence, the memory of 8086 is divided into two banks each bank provides 8-bits.
- The division is done in such a manner that any two consecutive locations lie in two different chips. Hence each chip contains alternate locations.
- ∴ One bank contains all even addresses called the **"Even bank"**, while the other is called **"Odd bank"** containing all odd addresses.
- Generally for any 16-bit operation, the Even bank provides the lower byte and the ODD bank provides the higher byte. Hence the **Even bank** is also called the **Lower bank** and the **Odd bank** is also called the **Higher bank**.

Memory Banking in 8086



Odd Bank

- Also called as "Higher bank"
- Address range:

00001H
00003H
00005H
.
.
.
FFFFFH

- Selected when $\overline{BHE}$ = 0

Even Bank

- Also called as "Lower bank"
- Address range:

00000H
00002H
00004H
.
.
.
FFFFEH

- Selected when A_0 = 0

$\overline{BHE}$	A_0	Operation
0	0	R/W 16-bit from both banks
0	1	R/W 8-bit from higher bank
1	0	R/W 8-bit from lower bank
1	1	No Operation (Idle).

Memory Banking in 8086

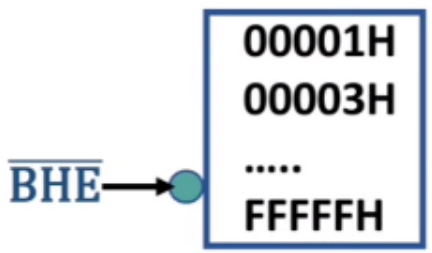
- 8086 has 16 bits of Data lines, so it should access 16 bits (2 Bytes) in one machine cycle.
- To perform 2 Bytes data transfer, So it needs to read 2 memory locations as one memory location has only one byte of data.
- If both of these memory locations are there in same memory then it can not be accessed by microprocessor.
- So, Microprocessor 8086 bisects the memory into two memory banks and each memory bank provides memory of one byte.
- The division is done in such a manner that any two consecutive memory locations lie in two memory chip. So, each memory chip holds alternate memory locations.
- Hence, One memory bank have all even address (Even Memory Bank) and other memory bank have all odd address (Odd Memory Bank).
- Generally, for any 16 bits operation, Even Memory Bank provides the lower Byte (Lower Bank) and Odd Memory Bank provides higher byte (Higher Bank). So we should write program from Even Memory locations.

Memory Banking in 8086

1 MB Memory [A19 – A0]

512 KB [ODD BANK]

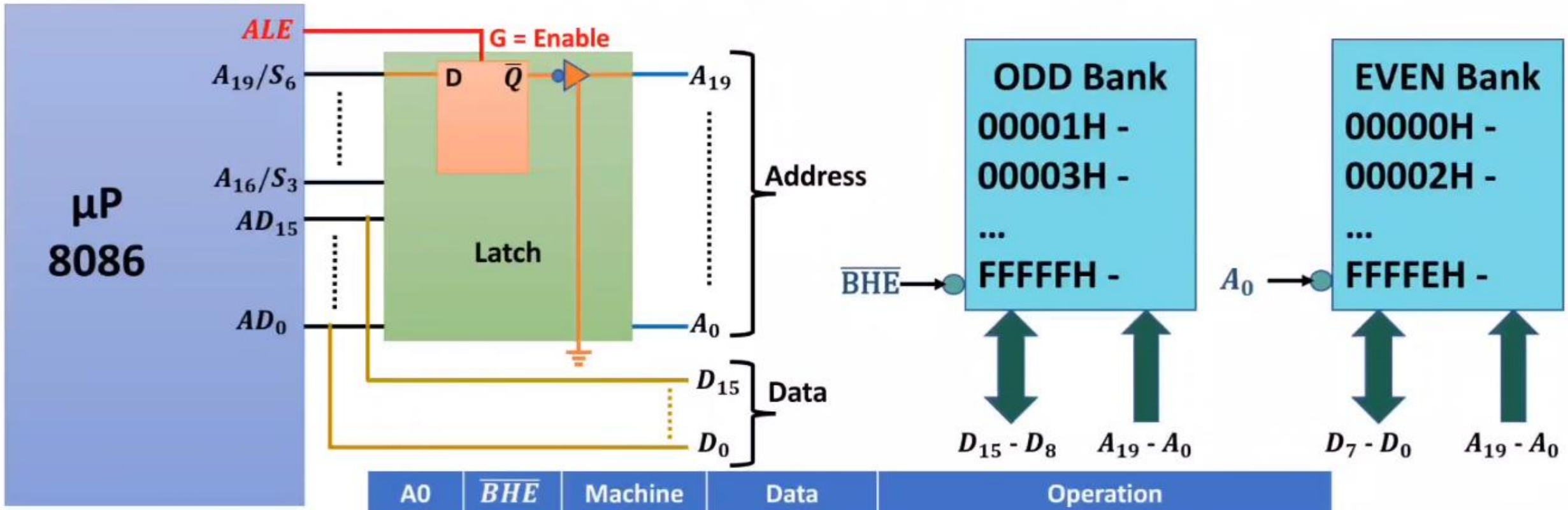
- It is also called higher bank
- These memory bank is selected when $\overline{BHE} = 0$
- Range of Address is



512 KB [EVEN BANK]

- It is also called Lower bank
- These memory bank is selected when $A_0 = 0$
- Range of Address is

Memory Banking in 8086

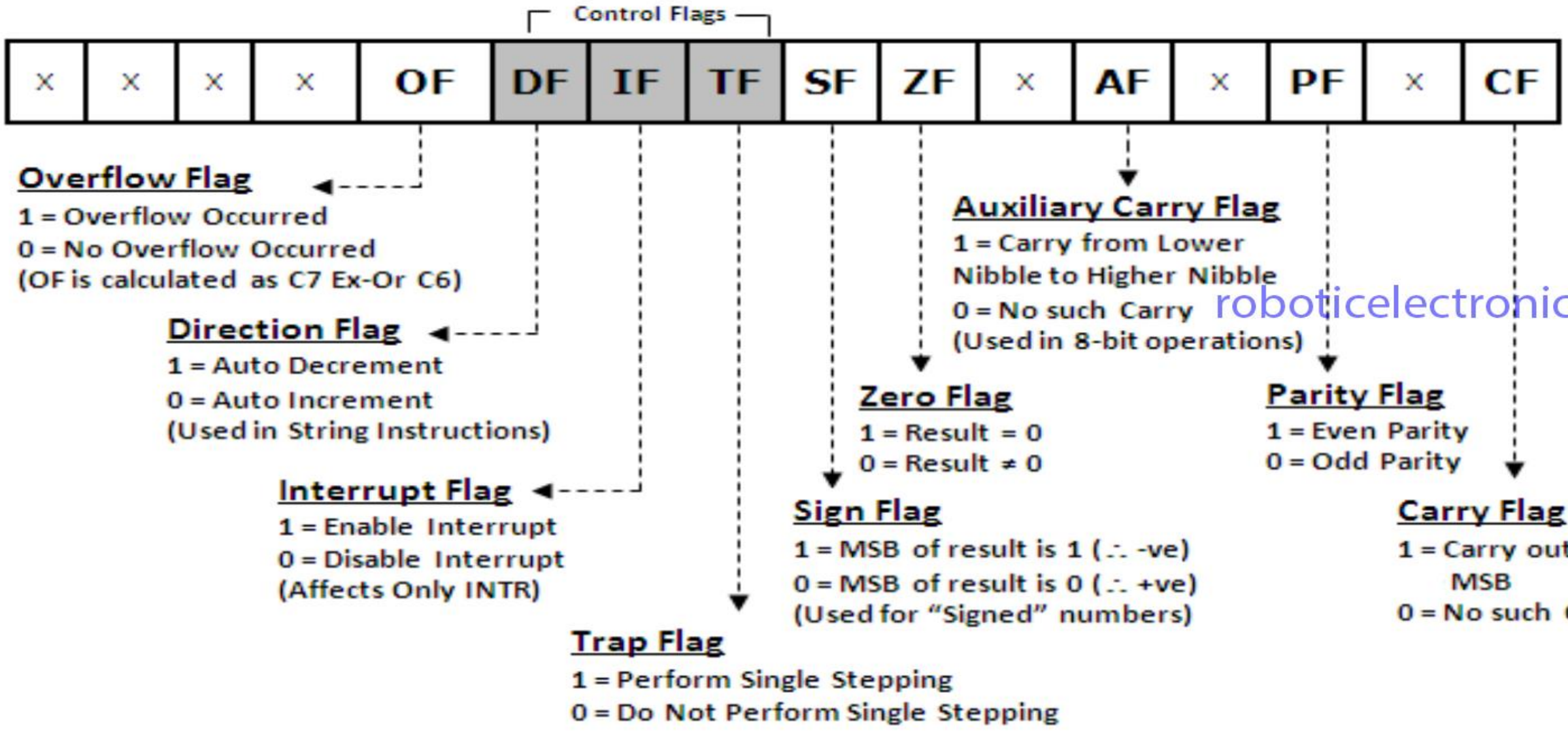


A0	$\overline{BHE}$	Machine Cycle	Data	Operation
0	0	1	D15-D0	Read a Word from Even Add
0	1	1	D7-D0	Read a Byte from Even Add
1	0	1	D15-D8	Read a Byte from Odd Add
1	0	2	D15-D8	Read a Word from Odd Add
0	1		D7-D0	

Flag Register of 8086

- As it has 16-bits, it has 16 flags. These 16 flags are classified as
  - 7 are don’t care flags.
  - 3 are control flags (accessible to programmers).
  - 6 are status flags (not accessible to programmers).

Flag Register of 8086



Flag Register of 8086

a) Status flags:

1. CF:

- It stands for carry flag.
- If CF = 1 ; it means carry is generated from the MSB.
- If CF = 0 ; no carry is generated out of MSB

CF = 1

```
   1 1
   1011 1100
 + 1001 0001
-------------
1  0100 1101
```

CF = 0

```
   1 1
   1011 1100
 + 0001 0001
-------------
   1100 1101
```

Flag Register of 8086

a) Status flags:

2. PF:

- It stands for parity flag.
- If PF = 1 ; it means it is even parity in the result (there are even numbers of 1's).
- If PF = 0 ; it means it is odd parity.

PF = 1

```
   1 1
  1011 1100
+ 1001 0001
-----------
1 0100 1101
```

PF = 0

```
   1 1
  1011 1100
+ 0001 0001
-----------
  1100 1101
```


In first example, there are 4 ones' which is even , in the second example there are 5 ones' which is odd in count.

Flag Register of 8086

a) Status flags:

3. AF:

AF stands for auxiliary flag. As 8-bits form a byte, similarly 4 bits form a nibble. So in 16 bit operations there are 4 nibbles.

If AF = 1 ; there is a carry out from lower nibble.

If AF = 0 ;no carry out of lower nibble.

Flag Register of 8086

a) Status flags:

4. OF:

- OF stands for overflow flag.
- The are two types of numbers
  - Signed
  - Unsigned

1. **Unsigned:**
   - This is a 8 bit positive number which ranges from 0 to 255. In hexadecimal it's range is from 00 to FF.
   - In the OF flag, it has nothing to do with unsigned numbers. Only signed numbers are considered in the OF flag.
2. **Signed:**
   - This is also a 8-bit number(can be 16-bit too) which is equally distributed among +ve and -ve numbers.
   - By considering MSB , it is decided whether it is a positive or negative number. If MSB=1 , it is a negative number else positive number.
   - Eg: 1011 0011 is -ve
     0111 1001 is +ve

Flag Register of 8086

a) Status flags:

4. OF:

1. **Signed:**
   - So it's positive range is from 0 to 127 [00 to 7F (in hexadecimal)], it consists of 128 +ve values.
   - It also has 128 negative numbers ranging from -1 to -128 [-01 to -80 (in hexadecimal)].
   - Whenever a negative number is saved, it is saved as it's 2's complement. To access the number we have to make its 2's complement again. A number is considered as negative whenever its MSB is 1, and its 2's complement is the actual negative number.
   - For e.g.: 1011 0011
   - It is a negative number and its 2's complement is 0100 1101, which is equivalent to 4D (hexadecimal). So it is -4DH.

Flag Register of 8086

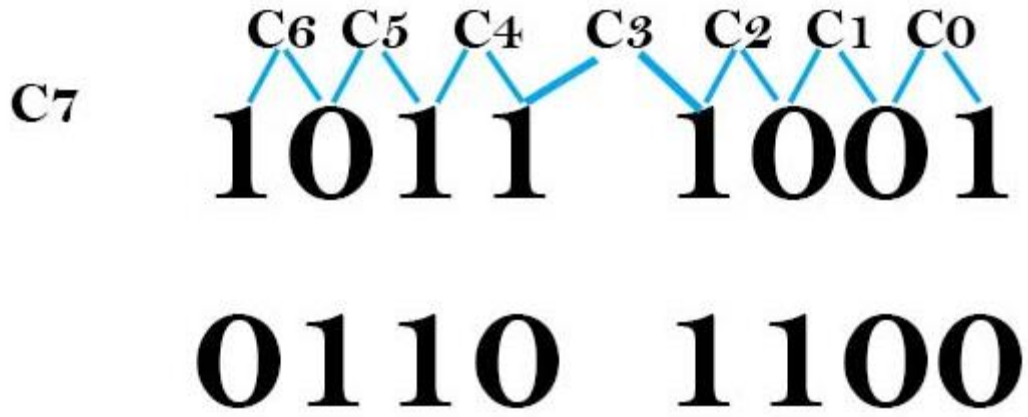
a) Status flags:

4. OF:

1. **Signed:**

- So there is a well defined limit for signed numbers (00 to 7F and -01 to 80). If the result or operands exceeds the limit, it is known as **overflow**. If there is overflow, then the MSB would show incorrect values, if there is overflow then
- MSB = 1 , then it is +ve number
- MSB = 0 , then it is -ve number. This is the reverse with the original case.
- **How does the processor come to know about overflow ?**

- XOR operation is performed on C_6 and C_7 carry.
- C_0 is carried from 0^{th} bit to 1^{st} bit and so on.
- If the **XOR** operation gives output as 1, then the operation is going out of the limit.



Flag Register of 8086

a) Status flags:

5. SF :

- This stands for sign flag. In short it copies the value of MSB. It shows wrong notation in the case of overflow.
- CF, AF, PF, ZF, OF, SF , these were status flags, and these keep changing after every arithmetic operation. And these flags are not controlled by the user, these are controlled by the ALU.

Flag Register of 8086

b) Control flag :

1. TF :

- This stands for trap flag. Generally processors give output after the complete program, but when TF = 1, output is given after every instruction.
- This is useful to check logical errors, when the program is too long.

TF=0

```
A = 5 + 3 ;
B = A + 1 ;
C = A + B ;
o/p
C = 17
```

TF=1

```
A = 5 + 3 ; o/p A = 8
B = A + 1 ; o/p B = 9
C = A + B ; o/p C = 17
```

Flag Register of 8086

b) Control flag :

2. IF :

- This is interrupt flag.
- IF = 1, then enable interrupts
- IF = 0, then interrupts are disabled. By default interrupts are disabled.

Flag Register of 8086

b) Control flag :

3. DF :

- This stands for direction flag. In the case string operation, by default address keeps incrementing for instruction execution. It means after execution of an instruction, whether the processor should execute next instruction or previous instruction.
- If DF = 0 ; address is auto incrementing (processor executes next instruction).
- If DF = 1 ; address is auto decrementing (processor executes previous instruction).

Bus Interface Unit (BIU)

- The Bus Interface Unit (BIU) manages the data, address and control buses.
- The BIU functions in such a way that it:
  - Fetches the sequenced instruction from the memory,
  - Finds the physical address of that location in the memory where the instruction is stored and
  - Manages the 6-byte pre-fetch queue where the pipelined instructions are stored.
- An 8086 microprocessor exhibits the property of pipelining the instructions in a queue while performing decoding and execution of the previous instruction.
- This saves the processor time of operation by a large amount. This pipelining is done in a **6-byte queue**.
- Also, the BIU contains **4 segment registers**. Each segment register is 16-bit. The segments are present in the memory and these registers hold the address of all the segments.
- These registers are as follows:

Bus Interface Unit (BIU)

1. **Code segment register**:
   - It is a 16-bit register and holds the address of the instruction or program stored in the code segment of the memory.
   - Also, the IP in the block diagram is the instruction pointer which is a default register that is used by the processor in order to get the desired instruction. The **IP contains the offset address** of the next byte that is to be taken from the code segment.
2. **Stack segment register**:
   - The stack segment register provides the starting address of the stack segment in the memory. Like in stack pointer, PUSH and POP operations are used in this segment to give and take the data to/from it.
3. **Data segment register**:
   - It holds the address of the data segment. The data segment stores the data in the memory whose address is present in this 16-bit register.

Bus Interface Unit (BIU)

4. **Extra segment register**:
   - Here the starting address of the extra segment is present. This register basically contains the address of the string data.
   - It is to be noteworthy that the physical address of the instruction is achieved by combining the segment address with that of the offset address.
5. **6-byte pre-fetch queue**:
   - This queue is used in 8086 in order to perform pipelining. As at the time of decoding and execution of the instruction in EU, the BIU fetches the sequential upcoming instructions and stores it in this queue.
   - The size of this queue is 6-byte. This means at maximum a 6-byte instruction can be stored in this queue. The queue exhibits **FIFO** behaviour., **first in first out**.

Execution Unit (EU)

❖The Execution Unit (EU) performs the decoding and execution of the instructions that are being fetched from the desired memory location.

1. **Control Unit**:
   - ❑ Like the timing and control unit in 8085 microprocessor, the control unit in 8086 microprocessor produces control signal after decoding the opcode to inform the general purpose register to release the value stored in it. And it also signals the ALU to perform the desired operation.
2. **ALU**:
   - ❑ The arithmetic and logic unit carries out the logical tasks according to the signal generated by the CU. The result of the operation is stored in the desired register.
3. **Flag:**
   - ❑ Like in 8085, here also the flag register holds the status of the result generated by the ALU. It has several flags that show the different conditions of the result.
4. **Operand**:
   - ❑ It is a temporary register and is used by the processor to hold the temporary values at the time of operation.
   - ❑ The reason behind two separate sections for BIU and EU in the architecture of 8086 is to perform fetching and decoding-executing simultaneously.

Working of 8086 Microprocessor

- Basically, when an instruction is to be fetched from the memory, then firstly its physical address must be calculated and this is done at the BIU. The physical address of an instruction is given as:

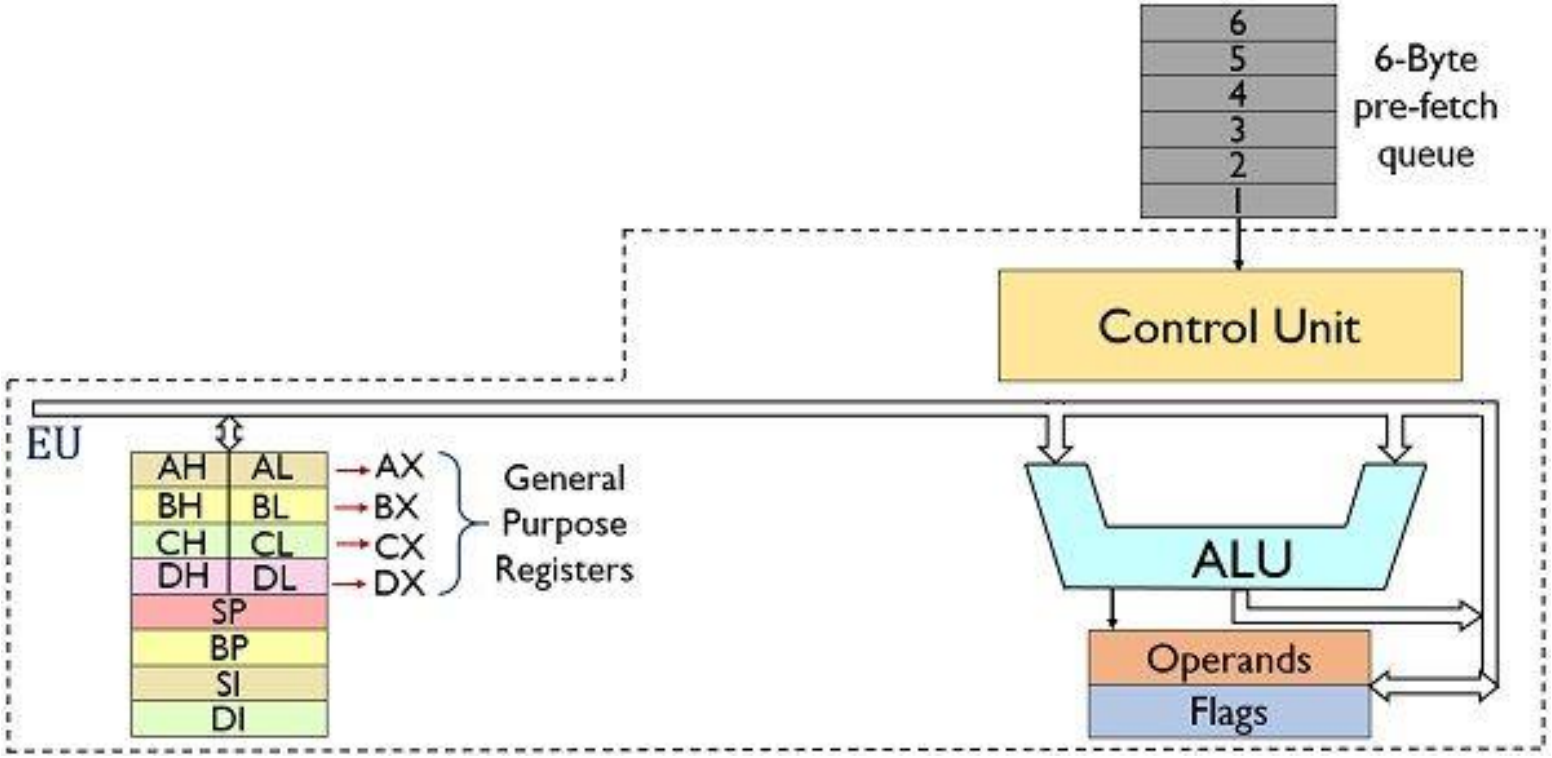
  PA = Segment address X 10 + Offset

- For example: Suppose the segment address is 2000 H and the offset address is 4356 H. So, the generated physical address is **24356 H**. Here, the code segment register provides the base address of the code segment which is combined with the offset address.

Working of 8086 Microprocessor

- The code segment contains the instructions.
- Each time an instruction is fetched the offset address inside the code segment gets incremented.
- So, once the physical address of an instruction is calculated by the BIU of the processor, it sends the memory location by the address bus to the memory.
- Further, the desired instruction at that memory location which is present in the form of the opcode is fetched by the microprocessor through the data bus.
- Suppose the instruction is **ADD BL, CL**. But, inside the memory, it will be in the form of an opcode. So, this opcode is sent to the control unit.
- The control unit decodes the opcode and generates control signals that inform the BL and CL register to release the value stored in it. Also, it signals the ALU to perform the ADD operation on that particular data.

Working of 8086 Microprocessor

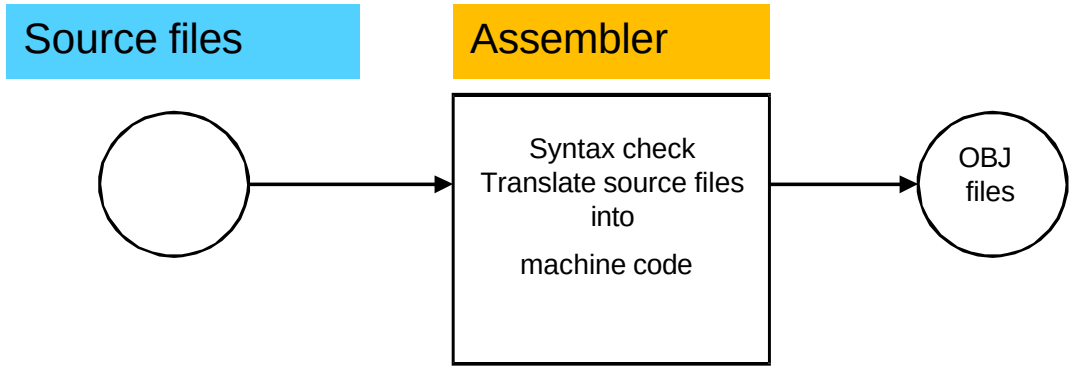


❖In any instruction, like **ADD BL, CL**. **BL** denotes the destination of the result of the add operation. This clearly shows that whatever, the operation is performed its result must be stored in the first register i.e., **BL** for this particular example.

Working of 8086 Microprocessor

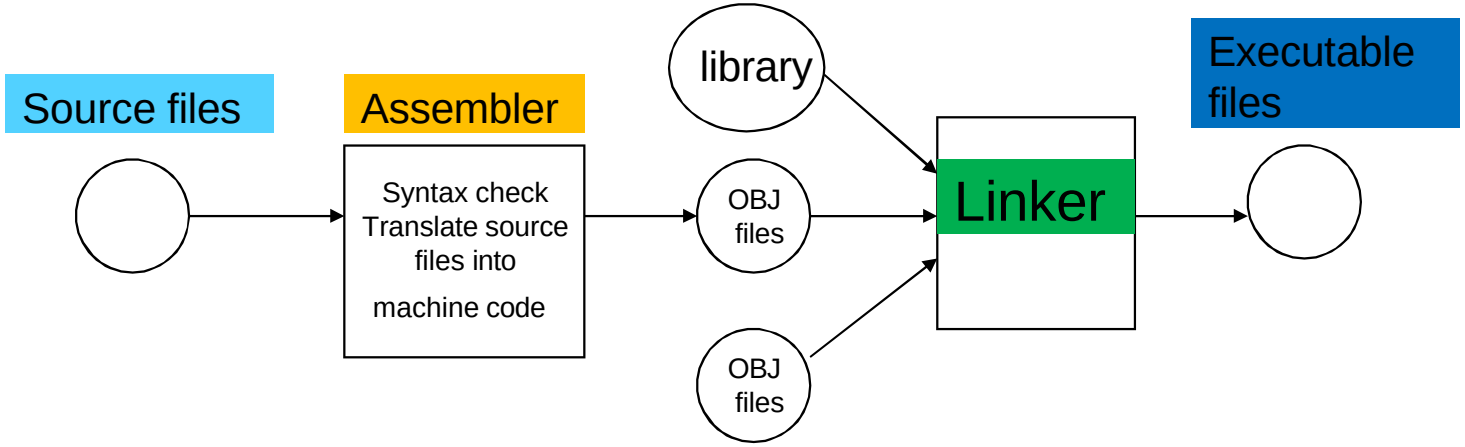
- Let us take another example: Consider an instruction; **ADD CL, 05H**.
  - This means that the operand which is 05H is to be added with the data present in the **CL** register and is stored in that particular register i.e., **CL**.
  - In such conditions, the operand is not provided to the control unit as only the opcode is required to be decoded by the **CU**. Hence the operand is directly provided to the ALU. Also, the status of this result is stored in the flag register. So, whenever, ALU carries out an operation, it simultaneously generates the result as well as its status.
  - In **BIU**, pipelining fails whenever there is branching in the instruction. This is because generally instructions are present in a sequential manner. But, sometimes the instructions are required to be executed un-sequentially.
  - However, in the queue, the instructions are stored sequentially. So, in case there exist a need for any random instruction to be decoded. The opcode stored in the queue will become invalid and must be cleared at that particular time.

Assembler



- An assembler is a program that converts source-code programs written in assembly language into object files in machine language.
- Popular assemblers include MASM (Macro Assembler from Microsoft), TASM (Turbo Assembler from Borland), NASM (Netwide Assembler for both Windows and Linux), and GNU assembler distributed by the free software foundation.

Linker



- A linker program combines your program's **object file created by the assembler with other object files and link libraries, producing a single executable program. You need a linker utility to produce executable files.**
- Two linkers: **LINK.EXE and LINK32.EXE are provided with the MASM 6.15 distribution to link 16-bit real-address mode and 32-bit protected-address mode programs respectively.**

Assembling and Linking

- Assembling and linking are two essential steps in the process of developing software for the Intel 8086 microprocessor.
- These steps are part of the overall process of converting human-readable assembly language code into executable machine code that the microprocessor can execute.

1. Assembling

- Assembling is the first step in the process. In this step, we use an assembler to convert the assembly language code into machine code in the form of object files. Each line of assembly code is translated into its corresponding machine code instruction. Symbolic labels, which represent memory addresses or constants, are resolved into actual addresses in memory.
- The output of the assembling process is typically one or more object files. These object files contain the machine code instructions and other information needed for later steps.

Assembling and Linking

2. Linking

- Linking is the second step, and it involves combining multiple object files along with any necessary libraries to create a complete executable program. When writing an assembly program, you might use external functions or routines defined in other object files or libraries. The linker's job is to resolve references to these external symbols and ensure that everything is correctly linked together.
- During the linking process, memory addresses are finalized, and any unresolved references (such as function calls to external modules) are resolved. The linker produces a single executable file that can be loaded and executed on the target microprocessor.

Macro Assembler

- A macro assembler in the context of the 8086 microprocessor is a type of assembler that supports the use of macros.
- Macros are a way to define reusable code templates that can be expanded inline within an assembly language program. They are similar to functions or procedures in high-level programming languages, but they are expanded directly at the assembly code level.
- Macros are just like procedures, but not really. Macros look like procedures, but they exist only until your code is compiled, after compilation all macros are replaced with real instructions.
- If you declared a macro and never used it in your code, compiler will simply ignore it.

Macro Assembler

```
; Macro definition
macro add_numbers a, b
       mov ax, a
       add ax, b
endm

add_numbers 10, 20 ; Macro invocation
```

MACRO & PROCEDURE in 8086

❖ PROC & ENDP

PROC Label

....

ENDP Label

- PROC is used to call procedure and ENDP is used to end the procedure.
- To call the procedure, we will have to use CALL instruction.
- Example : CALL Label

❖ MACRO & ENDM

MACRO Label

....

ENDM Label

- MACRO is used as function in program.
- To use the macro, we will have to use the Label only.
- Example : Label

PROCEDURE	MACRO
❖ To use PROCEDURE, we have to use CALL instruction.	❖ To use MACRO, we have to use LABEL only.
❖ By PROCEDURE, we perform subroutine.	❖ By MACRO, We insert function in code.
❖ PROCEDURE acquires size only once.	❖ MACRO may acquires size many times in program.
❖ PROCEDURE is based on CALL. So, it uses the STACK for PUSH and POP of subroutine.	❖ MACRO is function. We can use function with MACRO which is not using STACK.
❖ PROCEDURE executes branch.	❖ MACRO doesn't executes branch.
❖ It affects the pipelining as it is branch.	❖ It does not affects the pipelining.
❖ It is slower, but requires less memory space.	❖ It is Faster, but may needs more memory space.

Procedure Vs. Macro

S.N.	PROCEDURE (FUNCTION)	MACRO
1	A procedure (Subroutine/ Function) is a set of instruction needed repeatedly by the program. It is **stored as a subroutine and invoked from several places by the main program.**	A Macro is similar to a procedure but is not invoked by the main program. Instead, the **Macro code is pasted into the main program wherever the macro name is written in the main program.**
2	A subroutine is **invoked by a CALL** instruction and control returns by a RET instruction.	A Macro is simply accessed by **writing its name**. The entire macro code is pasted at the location by the assembler.
3	**Reduces the size** of the program.	**Increases the size** of the program .
4	**Executes slower** as time is wasted to push and pop the return address in the stack.	**Executes faster** as return address is not needed to be stored into the stack, hence push and pop is not needed.
5	**Depends on the stack**	**Does not depend on the stack**

Assembler Directives/Pseudo Opcodes

- An assembler directive is a message to the assembler that tells the assembler something it needs to know in order to carry out the assembly process;
- For example, an assemble directive tells the assembler where a program is to be located in memory.
- Following are 8086 assembler directives:
    1. The DB Directive
    2. The DW Directive
    3. The DD Directive
    4. The STRUCT (or STRUC) and ENDS directives (counted as one)
    5. The EQU Directive
    6. The Comment Directive
    7. ASSUME
    8. Extern
    9. Global
    10. OFFSET
    11. PROC
    12. GROUP
    13. INCLUDE

Assembler Directives/Pseudo Opcodes

1. **DB** – The DB directive is used to declare a BYTE – A BYTE is made up of 8 bits.

2. **DW** – The DW directive is used to declare a WORD type variable – A WORD occupies 16 bits or (2 BYTE).

3. **DD** – The DD directive is used to declare a DWORD – A DWORD double word is made up of 32 bits =2 Word's or 4 BYTE.

4. **STRUCT** and **ENDS** directives to define a structure template for grouping data items.

5. The **EQU** directive - is used to give name to some value or symbol. Each time the assembler finds the given names in the program, it will replace the name with the value or a symbol. The value can be in the range 0 through 65535.

6. **Extern** - It is used to tell the assembler that the name or label following the directive are some other assembly module.

7. **GLOBAL** - The GLOBAL directive can be used in place of PUBLIC directive .for a name defined in the current assembly module; the GLOBAL directive is used to make the symbol available to the other modules.

8. **SEGMENT** - It is used to indicate the start of a logical segment. It is the name given to the segment. Example: the code segment is used to indicate to the assembler the start of logical segment.

Assembler Directives/Pseudo Opcodes

9. **PROC: (PROCEDURE)** - It is used to identify the start of a procedure. It follows a name we give the procedure.

10. **NAME** - It is used to give a specific name to each assembly module when program consists of several modules.

11. **INCLUDE** - It is used to tell the assembler to insert a block of source code from the named file into the current source module. This shortens the source module.

12. **OFFSET** - It is an operator which tells the assembler to determine the offset or displacement of a named data item from the start of the segment which contains it. It is used to load the offset of a variable into a register so that variable can be accessed with one of the addressed modes.

13. **GROUP** - It can be used to tell the assembler to group the logical segments named after the directive into one logical group. This allows the contents of all he segments to be accessed from the same group.

Assembler Directives/ Pseudo Opcodes in 8086

✓ Basics of Assembler Directives

- ❑ Assembler directives are also referred as pseudo Opcodes. Assembler directives are not instructions, so they are not executed by MPU.
- ❑ It is used to define integers, characters, strings, segments etc.

❖ Segment & Ends

```
Name Segment
....
Name Ends
```

- Here Name is Label, with any name we can create segment.
- If Name = DATA, then with label "DATA", we have created segment [Segment is memory space].

❖ End

- End is assembler directive, which is used to end the program.
- Anything written after End will not be assembled by Assembler. So after end you can write key notes.

❖ Assume, Start: & End Start

```
Data Segment
....
Data Ends
Code Segment
        Assume CS: Code, DS: Data
        Start:
....
Code Ends
        End Start
End
```

- Assume directive tells 8086, which label is defined for which segment.
- CS is defined with Code and DS is defined with Data.
- "Start:" assembler directive is used to start assembly code and assembly code gets end at "End Start"

Assembler Directives/ Pseudo Opcodes in 8086

❖ PROC & ENDP

```
PROC Label
....
ENDP Label
```

- PROC is used to call procedure and ENDP is used to end the procedure.
- To call the procedure, we will have to use CALL instruction.
- Example : CALL Label

❖ MACRO & ENDM

```
MACRO Label
....
ENDM Label
```

- MACRO is used as function in program.
- To use the macro, we will have to use the Label only.
- Example : Label

❖ DB, DW, DD, DQ, DT & EQU

```
Data Segment
A DB 54H
B DW 3550H
C EQU 10H
Data Ends
```

- DB is Define Byte [8 bits of data]
- DW is Define Word [16 bits of data]
- DD is Define Double Word [32 bits of data]
- DQ is Define Quad Word [64 bits of data]
- DT is Define Ten Bytes [80 bits of data]
- EQU is used to define constant. It has fixed value.
- DB & DW generally used in 8086.
- DD, DQ & DT used in 8087.

```
Data Segment
A DB ?
B DW ?
Data Ends
```

- ? Is used to define variable, later on we can put the value by using it in program.

Assembler Directives/ Pseudo Opcodes in 8086

❖ DUP

```
Data Segment
A DB 05H DUP (0)
Data Ends
```

- DUP will Duplicate 0 five times. As per DB here we duplicate 8 bits of data five times.

❖ ALIGN / EVEN

```
Data Segment
ALIGN
A DW 05H DUP (2050H)
Data Ends
```

- ALIGN is used to store words from even address.
- It will take only one machine cycle to execute given data by microprocessor 8086.

❖ Byte PTR and Word PTR

```
MUL Byte PTR [2050H]
MUL Word PTR [2055H]
```

- Byte PTR defines memory location with Byte.
- Word PTR defines memory location with word.

❖ ORG

```
Data Segment
ORG 1000H
A DW 2505H DUP (0)
Data Ends
```

- It will define variable A from 1000H location.

❖ OFFSET

```
Data Segment
A DB 05H, 04H, 56H
Data Ends
MOV BX, OFFSET A
```

- OFFSET is used to load address of string in register.
- We can also use {LEA BX,A}

❖ Near and Far

- Near is used for Intra Segment branch [Same segment].
- Far is used for Inter Segment branch [Different Segment].

❖ Model Directives

- .Model Small [DS $\leq$ 64KB, CS $\leq$ 64KB]
- .Model Medium [DS $\leq$ 64KB, CS $\geq$ 64KB]
- .Model Large [DS $\geq$ 64KB, CS $\geq$ 64KB]

Comments

- The use of comments throughout a program can improve its clarity, especially in Assembly Language.
- A comment begins with **Semicolon(;)**.

- **Example: ADD AX, BX**

```
; Adds the contents of AX with BX
; and stores in AX
```

ADDRESSING MODES OF 8086

❖ An addressing mode refers to the method by which the processor accesses data or operands in memory.

I. IMMEDIATE ADDRESSING MODE

❑ In this mode the operand is specified in the instruction itself. Instructions are longer but the operands are easily identified.

```
E.g.: MOV CL, 12H      ; Moves 12 immediately into CL register
MOV BX, 1234H          ; Moves 1234 immediately into BX register
```

II. REGISTER ADDRESSING MODE

❑ In this mode operands are specified using registers.

❑ Instructions are shorter but operands cant be identified by looking at the instruction.

```
E.g.: MOV CL, DL       ; Moves data of DL register into CL register
MOV AX, BX             ; Moves data of BX register into AX register
```

ADDRESSING MODES OF 8086

III. DIRECT ADDRESSING MODE

- In this mode address of the operand is directly specified in the instruction.
- Here only the offset address is specified, the segment being indicated by the instruction.

```
E.g.: MOV CL, [4321H]      ; Moves data from location 4321H in the data
                           ; segment into CL
                           ; The physical address is calculated as
                           ; DS * 10H + 4321
                           ; Assume DS = 5000H
                           ; ∴ P A= 50000 + 4321 = 54321H
                           ; ∴ CL ← [54321H]
E.g.: MOV CX, [4320H]      ; Moves data from location 4320H and 4321H
                           ; in the data segment into CL and CH resp.
```

ADDRESSING MODES OF 8086

IV. INDIRECT ADDRESSING MODES

a) REGISTER INDIRECT ADDRESSING MODE

- In this mode the µP uses any of the 2 base registers BP, BX or any of the two index registers SI, DI to provide the offset address for the data byte.
- The segment is indicated by the Base Registers: **BX** -- Data Segment, **BP** -- Stack Segment

```
E.g.: MOV CL, [BX]      ; Moves a byte from the address pointed by BX in Data
                        ; Segment into CL.
                        ; Physical Address calculated as DS * 10H + BX

E.g.: MOV [BP], CL      ; Moves a byte from CL into the location pointed by BP
                        ; in Stack Segment.
                        ; Physical Address calculated as SS * 10H + BP
```

ADDRESSING MODES OF 8086

IV. INDIRECT ADDRESSING MODES

b) REGISTER RELATIVE ADDRESSING MODE

- In this mode the operand address is calculated using one of the base registers and a 8-bit or a 16-bit displacement.

```
E.g.: MOV CL, [BX+4]   ; Moves a byte from the address pointed by BX+4 in
                       ; Data Seg to CL.
                       ; Physical Address: DS * 10H + BX + 4H

E.g.: MOV 12H [BP], CL ; Moves a byte from CL to location pointed by BP+12H
                       ; in the Stack Seg.
                       ; Physical Address: SS * 10H + BP + 12H
```

ADDRESSING MODES OF 8086

IV. INDIRECT ADDRESSING MODES

c) BASE INDEXED ADDRESSING MODE

- Here, operand address is calculated as Base register plus an Index register.

```
E.g.: MOV CL, [BX+SI]  ; Moves a byte from the address pointed by BX+SI
                       ; in Data Segment to CL.
                       ; Physical Address: DS * 10H + BX + SI

E.g.: MOV [BP+DI], CL  ; Moves a byte from CL into the address pointed by
                       ; BP+DI in Stack Segment.
                       ; Physical Address: SS * 10H + BP + DI
```

ADDRESSING MODES OF 8086

IV. INDIRECT ADDRESSING MODES

d) BASE RELATIVE PLUS INDEX ADDRESSING MODE

- In this mode the address of the operand is calculated as Base register plus Index register plus 8-bit or 16-bit displacement.

```
E.g.: MOV CL, [BX+DI+20]      ; Moves a byte from the address pointed by
                              ; BX+SI+20H in Data Segment to CL.
                              ; Physical Address: DS * 10H + BX + SI+ 20H

E.g.: MOV [BP+SI+2000], CL    ; Moves a byte from CL into the location
                              ; pointed by BP+SI+2000H in Stack Segment.
                              ; Physical Address: SS * 10H + BP+SI+2000H
```

ADDRESSING MODES OF 8086

V. IMPLIED ADDRESSING MODE

- In this addressing mode the operands are implied and are hence not specified in the instruction.

E.g. : STC ; Sets the Carry Flag.

E.g. : CLD ; Clears the Direction Flag.

ADDRESSING MODES OF 8086

Important points for understanding addressing modes…

1) Anything given in square brackets will be an Offset Address also called Effective Address.

2) MOV instruction by default operates on the Data Segment; unless specified otherwise.

3) BX and BP are called Base Registers.

4) BX holds Offset Address for Data Segment. BP holds Offset Address for Stack Segment.

5) SI and DI are called Index Registers

6) The Segment to be operated is decided by the Base Register and NOT by the Index Register.

Addressing Modes in Microprocessor 8086

❖ **Addressing mode: The Various formats of specifying the operands are called addressing modes.**

✓ Immediate Addressing Mode

❑ In this Addressing mode, data (1byte/2bytes) specified in instruction is directly transferred into register.

Example:

```
MOV CL,15H      ; 15H will gets transferred to CL
MOV BX,1000H    ; 1000H will gets transferred to BX
```

✓ Register Addressing Mode

❑ In this Addressing mode, operands are specified in registers only.

Example:

```
MOV CL,DL       ; DL will gets copied into CL
MOV AX,BX       ; BX will gets copied into AX
```

✓ Direct Addressing Mode

❑ In this Addressing mode, address of operand is specified in instruction.

Example:

```
MOV CL,[1000H]  ; CL will get data from 1000H address
MOV CX,[1000H]  ; CL & CH will get data from 1000H &
                  1001H, respectively.
```

✓ Implied/Implicit Addressing Mode

❑ In this Addressing mode, the operand is implied in instruction.

Example:

```
STC             ; Set the carry flag
CLD             ; Clear Directional flag
```

Addressing Modes in Microprocessor 8086

✓ Indirect Addressing Mode

- In this Addressing mode, the address of operand is stored in registers.

❖ In 8086, with Indirect Addressing Mode following categories are there:

1. Register Indirect Addressing Mode
2. Register Relative Addressing Mode
3. Base Index Addressing Mode
4. Base Relative Plus Index Addressing Mode

1. Register Indirect Addressing Mode

- In this Addressing mode, operand address will be given by memory pointer.

Example:

```
MOV CL,[BX]     ; CL will take data pointed by BX Address
MOV [BP],CL     ; CL will be copied at Address pointed by BP (On stack)
```

2. Register relative Addressing Mode

- In this Addressing mode, operand address will be given by memory pointer + 8 bits or 16 bits displacement.

Example:

```
MOV CL,[BX+4]   ; CL will take data pointed by [BX+4] Address
```

3. Base Indexed Addressing Mode

- In this Addressing mode, operand address will be given by Base Register + Index register.

Example:

```
MOV CL,[BX+SI]  ; CL will take data pointed by [BX+SI] Address
```

4. Base Relative Plus Indexed Addressing Mode

- In this Addressing mode, operand address will be given by Base Register + Index register + 8 bits or 16 bits displacement .

Example:

```
MOV CL,[BX+SI+4] ; CL will take data pointed by [BX+SI+4] Address.
```

Addressing Modes in 8086

- Addressing modes define how operands are specified in assembly language instructions.
- Here are the various addressing modes supported by the 8086:

1. **Immediate Addressing Mode:** In this mode, the operand is a constant value or immediate data. The value is directly specified in the instruction. For example:

```
MOV AX, 1234h ; Load immediate value 1234h into AX
```

2. **Register Addressing Mode:** The operand is a register. Data is directly accessed from or stored in a register. For example:

```
ADD AX, BX ; Add contents of BX register to AX register
```

3. **Memory Direct Addressing Mode:** The operand is a memory location specified by a direct memory address. For example:

```
MOV AL, [1234h] ; Load value from memory address 1234h into AL
```

4. **Memory Indirect Addressing Mode:** The operand is a memory location pointed to by a register. The register holds the memory address. For example:

```
MOV AX, [SI] ; Load value from memory address pointed to by SI into AX
```

Addressing Modes in 8086

6. **Based Indexed Addressing Mode:** Similar to indexed mode, but with an added displacement value. Useful for accessing elements in data structures with variable offsets.

```
MOV AX, [BX + SI + 10] ; Load value from memory address (BX + SI + 10) into AX
```

7. **Based Addressing Mode:** Uses a base register (usually BX or BP) to access memory. Often used in accessing data on the stack.

```
MOV AX, [BX] ; Load value from memory address pointed to by BX into AX
```

8. **Register Indirect Addressing Mode:** The operand is a register that holds a memory address. Used for more general memory access.

```
MOV AX, [BX] ; Load value from memory address pointed to by BX into AX
```

9. **Register Relative Addressing Mode:** Combines a base register and an index register to calculate the memory address. Allows more complex memory access calculations.

```
MOV AL, [BX + SI + 10] ; Load value from memory address (BX + SI + 10) into AL
```

Instructions: LEA

1. **LEA:** Load Effective Address
   - General form: **LEA Register, Source**
   - This instruction loads the effective address of destination operand into the source register.
   - This instruction determines the offset of the variable or memory location named as the source and puts this offset in the indicated 16-bit register.
   - No flags are affected.
   - Example:

```
LEA BX, NUM1
LEA BX, PRICES              ; Load BX with offset of PRICE in DS
LEA BP, SS: STACK_TOP       ; Load BP with offset of STACK_TOP in SS
LEA CX, [BX][DI]                        ; Load CX with EA = [BX] + [DI]
```

Instructions: MUL

2. MUL: Unsigned multiplication of byte or word

- General form: MUL source
- This instruction multiplies an unsigned byte by contents of AL or an unsigned word by contents of AX.
- The source can be a register or memory location. Immediate data cannot be used as source.
- When a byte is multiplied by AL, the result is put in AX.
- When a word is multiplied by AX, the result can be as large as 32 bits. The most significant word (upper 16 bits) of result is placed in DX. The least significant word (lower 16 bits) of result is placed in AX.
- If the most significant byte of 16 bit result or the most significant word of 32 bit result is 0, CF and OF will be 0. A, P, S and Z flags are undefined.
- Examples:

```
MUL BH              ; AX = AL * BH
MUL CX              ; DX : AX = AX * CX
MUL BYTE PTR [BX]
MUL WORD PTR [SI]
```

Instructions: IMUL

3. IMUL: Multiply signed numbers

- General form: IMUL source
- This instruction multiplies a signed byte by AL or a signed word by contents of AX.
- The source can be a register or memory location. Immediate data can not be used as source.
- When a byte is multiplied by AL, the signed result is put in AX.
- When a word is multiplied by AX, the signed result is put in registers DX and AX with upper 16 bits in DX and lower 16 bits in AX.
- If upper byte of 16 bit result or upper word of 32 bit result contains only sign bits (all 0s for positive result and all 1s for negative result) then CF = OF = 0 (reset).
- If upper byte of 16 bit result or upper word of 32 bit result contains part of the product, CF = OF = 1 (set).
- A, P, S, Z flags undefined.
- Examples:

```
IMUL BX
IMUL AX
IMUL WORD PTR [SI]
```

Instructions: DIV

4. **DIV:** Unsigned Divide

- General form: **DIV source**
- This instruction is used to divide an unsigned word by a byte or an unsigned double word by a word.
- The source can be a **register** or **memory location**.
- When a word is divided by byte, the word must be in AX. After division, AL will contain an 8 bit result (**quotient**) and AH will contain an 8 bit **remainder**.
- If a number is divided by zero or if result is greater than FFH, 8086 will automatically generate a type 0 interrupt.
- When a double word is divided by a word, the most significant word must be in DX and least significant word must be in AX. After division, AX will contain the 16 bit result and DX will contain 16 bit remainder.
- If a number is divided by 0 or if the result is greater than FFFFH, type 0 interrupt is generated.
- All flags are undefined after a DIV instruction
- **Examples:**

```
DIV BL ; AX / BL
DIV CX ; DX : AX / CX
DIV BYTE PTR [SI] ; AX / [SI]
```

Instructions: IDIV

5. IDIV: Signed division

- General form: **IDIV source register or memory**
- This instruction is used to divide a signed word by a signed byte or a signed double word by a signed word.
- When a signed word is divided by signed byte, the word must be in AX. After division, AL will contain signed result (quotient) and AH will contain signed remainder.
- If a number is divided by zero or if result is greater than 127 or result is less than -127, type 0 interrupt is generated.
- When a signed double word is divided by a signed word, the most significant word must be in DX and least significant word must be in AX. After division, AX will contain the 16 bit result and DX will contain 16 bit remainder.
- If a number is divided by 0, or if the result is greater than 7FFFH, or if the result is less than 8001H, type 0 interrupt is generated.
- All flags are undefined.
- Examples:

```
IDIV BL
IDIV BP
IDIV BYTE PTR[BX]
```

Instructions: LOOP, LOOPE, LOOPNE

6. **LOOP:** Loop unconditionally
   - ❑ General form: LOOP **label**
   - ❑ This instruction executes instruction from the label upto LOOP instruction CX times.
   - ❑ At each iteration, CX is automatically decremented.
   - ❑ No flags are affected.
   - ❑ Examples:

```
MOV AL, 00H
MOV CX, 0010
next: ADD AL, CL
LOOP next
```

7. **LOOPE / LOOPZ:**
   - ❑ Loop while CX ≠ 0 and ZF = 1 (set)
   - ❑ This instruction executes the loop when CX ≠ 0 and ZF = 1.
   - ❑ If ZF becomes 0 or CX = 0, the loop is terminated.

8. **LOOPNE / LOOPNZ:**
   - ❑ Loop while CX ≠ 0 and ZF = 0 (reset)
   - ❑ This instruction executes the loop when CX ≠ 0 and ZF = 0.
   - ❑ If ZF becomes 1 or CX = 0, the loop is terminated.

Instructions: AAA

9. AAA: ASCII Adjust after Addition

- General form: **AAA**
- This instruction is generally used after an **ADD** instruction. **AH** must be cleared before **ADD operation**.
- This instruction **converts** the contents of **AL** to **unpacked decimal digits**.
- This instruction examines the lower 4 bits of AL whether it contains a value in the range 0 to 9.
- If it is between 0 – 9 and AF=0, this instruction sets 4 higher bits of AL to zero.
- If lower 4 bits of AL are in the range 0 – 9 and AF=1, 06H is added to AL. The upper four bits of AL are cleared and AH is incremented by 1.
- If lower nibble (lower 4 bits) of AL is greater than 9, AL is incremented by 6 and AH is incremented by 1. The upper nibble of AL is cleared and AF=CF=1.
- Flags affected: A, C
- Examples:

1) **Let BL = 34H**
AL = 33H
then
ADD AL, BL ; AL = 33 + 34 = 67H
AAA ; AL = 07H

2) Let **BL = 34H**
AL = 36H
then
ADD AL, BL ; AL = 33 + 34 = 6AH
AAA ; Since lower 4 bits of AL = A > 9
therefore AL = AL + 06H = 10H
AL = 00H, AH = 01H

Instructions: AAS, AAM

10. **AAS:** ASCII Adjust after Subtraction

- General form: **AAA**
- This instruction corrects the result in AL register. This instruction is used after subtraction operation.
- If lower nibble of AL is greater than 9 or if AF = 1, AL is decremented by 6 and AH is decremented by 1. The CF and AF are set to 1.

11. **AAM:** ASCII Adjust after Multiplication

- General form: **AAM**
- This instruction converts the product available in unpacked BCD format.
- This instruction is used after multiplication operation in which two unpacked BCD operands are multiplied.
- Flags affected: **S, Z, P**
- Example:

```
MOV AL, 04
MOV BL, 09
MUL BL ; AX = 0024H
AAM ; AH = 03, AL = 06
```

Instructions: AAD

12.AAD: ASCII Adjust before Division

- General form: **AAM**
- This instruction converts two unpacked BCD digits in AH and AL to equivalent binary number and stores it in AL.
- Flags modified: **S, Z, P**
- This instruction is used **before** DIV instruction.
- Example:

  Let AX = 0508

  AAD ; AL = 3AH

Instructions: DAA

13. **DAA**: Decimal Adjust Accumulator

- General form: **DAA**
- This instruction is used to convert the result of addition of two packed BCD numbers to a valid BCD numbers.
- The result has to be only in **AL**.
- If lower nibble of **AL** is **greater than 9** or if AF = 1, it will **add 06** to lower nibble in **AL**.
- After adding 06, if upper nibble of **AL** is **greater than 9** or if CF=1, this instruction **adds 60** to **AL**.
- Flags affected: S, Z, A, P, C
- Following examples explains this instruction.

i) Let **AL = 53, CL = 29**

ADD AL, CL ; AL = 53 + 29 = 7C

DAA ; C > 9

7C + 06 = 82

ii) Let **AL = 73, CL = 29**

ADD AL, CL ; AL = 73 + 29 = 9C

DAA ; C > 9

9C + 06 = A2

A > 9

A2 + 60 = 02 in AL and CF = 1

Instructions: DAS

14.DAS: Decimal Adjust after Subtraction

- General form: **DAS**
- This instruction is used after subtracting two packed BCD numbers. The result of subtraction must be in AL.
- If lower nibble of AL > 9 or the AF = 1 then this instruction will subtract 6 from lower nibble of AL.
- If the result in upper nibble is now greater than 9 or if carry flag was set, the instruction will subtract 60 from AL.
- Examples:

1) Let **AL = 75, BH = 46**

SUB AL, BH ; AL = 75 – 46 = 2F

DAS ; AL = AL – 06 = 2F – 06 = 29

2)Let **AL = 49, BH = 72**

SUB AL, BH ; AL = 49 – 72 = D7 with CF = 1

Since **D > 9**

AL = AL – 60 = D7 – 60 = 77 with CF = 1

For More 8086 Instructions

VISIT: https://www.dropbox.com/scl/fi/4dq277oaduo4i4gnggvqg/Instruction-Set-of-8086-Microprocessor-with-Addressing-Modes.pdf?rlkey=pxhu1r3zsiczvqy5y5j2pap6l&dl=0

INT 21H (DOS Interrupt) Functions

- **INT 21h** is the assembly language mnemonic for the Intel CPU command used to perform a System call to the operating system MS-DOS.
- MS-DOS actually used quite a few different software interrupt vectors for system calls, but INT 21h was the main one.

Task	Method	Comment
How to **input** a **character** from the screen	**Mov AH, 01H** **INT 21H**	Takes the user input character from the screen. Returns the ASCII value of the character in AL register. If AL=0, then a control key was pressed.
How to **input** a **string** from the screen	**Mov AH, 0AH** **LEA DX, string** **INT 21H**	0AH is the parameter for the input string function. The string will be stored from the offset address given by DX.
How to **display** a **character** on the screen	**Mov AH, 02H** **Mov DL, char** **INT 21H**	02H is the parameter for the display char function. DL should contain the char to be displayed.
How to **display** a **string** on the screen	**Mov AH, 09H** **LEA DX, string** **INT 21H**	09H is the parameter for the display string function. DX should contain the offset address of the output string.
How to **terminate** the program	**Mov AH, 4CH** **Mov AL, 00H** **INT 21H**	4CH is the parameter for the terminate function. The return code is placed by the system in AL register. If AL is 00h then the program terminated without an error.

Function number	Description
01h e.g. mov ah,01h int 21h	Keyboard input with echo: This operation accepts a character from the keyboard buffer. If none is present, waits for keyboard entry. It returns the character in AL.
02h e.g. mov ah,02h int 21h	Display character: Send the character in DL to the standard output device console.
09h e.g. mov ah,09h int 21h	String output: Send a string of characters to the standard output. DX contains the offset address of string. The string must be terminated with a '$' sign.
0Ah e.g. mov ah,0Ah int 21h	String input
4Ch e.g. mov ax,4C00h int 21h	Terminate the current program (mov ah,4Ch int 21h is also used.)

INT 10H Functions

- It is a video display vector interrupt used for Video display services (BIOS services).

Function number	Description
00h	Set video mode
01h	Set cursor size
02h	Set cursor position
06h	Scroll window up
07h	Scroll window down
08h	Read character and attribute of cursor
09h	Display character and attribute at cursor
0Ah	Display character at cursor

8086 Instruction Sets

- 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

- These instructions are used to transfer the data from the source operand to the destination operand. Following are the list of instructions under this group −
- **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.

Data Transfer 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 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.
- **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.

Arithmetic Instructions

- These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc. Following is the list of instructions under this group –
- **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 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.

Arithmetic Instructions

- **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 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.

Bit Manipulation Instructions

- These instructions are used to perform operations where data bits are involved, i.e. operations like logical, shift, etc. Following is the list of instructions under this group –
- **Instructions to perform logical operation**
  - **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.
- **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.

Bit Manipulation Instructions

- **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.

String Instructions

- String is a group of bytes/words and their memory is always allocated in a sequential order. Following is the list of instructions under this group −
  - **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)

- 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.

Program Execution Transfer Instructions (Branch and Loop Instructions)

- **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
  - **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

Processor Control Instructions

- These instructions are used to control the processor action by setting/resetting the flag values. Following are the instructions under this group −
  - **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.

Iteration Control 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

Interrupt Instructions

- These instructions are used to call the interrupt during program execution.
  - **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

8086 ALP Structure

```
include ‘emu8086.inc’
.model small
.stack 100h ; default stack size is 1024 bytes
.data
      ;initialize data variables
.code
      main proc
            .
            .
            .
      main endp
end main
```