# Intel 8086 Microprocessor:

## Features:

- Released in 1978 by Intel Corporation.
- It has **16-bit data bus** i.e. it can access 16-bit data in one operation. It's ALU and internal data registers are also 16-bit; hence 8086 is a **16-bit μp**.
- 8086 has a **20 bit address bus** and can access up to $2^{20}$ memory locations[00000H - FFFFFH] (1 MB)[16 times more than 8085].

$2^{20} = 1MB$ $2^{30} = 1GB$

$2^{10} = 1KB$ $2^{40} = 1TB$

- 8086 supports **Pipelining**
  - Fetching the next instruction while executing the current instruction is called Pipelining. 8086 prefetches up to 6 instruction bytes from memory and queues them in order to speed up instruction execution. Pipelining highly improves the performance of the system.

- 8086 supports **Memory Segmentation**:
  - Segmentation means dividing the memory into logical components. Here, the memory is divided into 4 segments: **Code, Data, Stack and Extra Segments.**

- 40-pin IC with +5V DC power supply.
- Available in 3 versions- 8086(Clock rate 5MHz), 8086-2(8 MHz) & 8086-1(10 MHz).
- 8086 is designed to operate in two modes, Minimum and Maximum.
- Have 117 different instructions.

## Internal Architecture of 8086 Microprocessor:

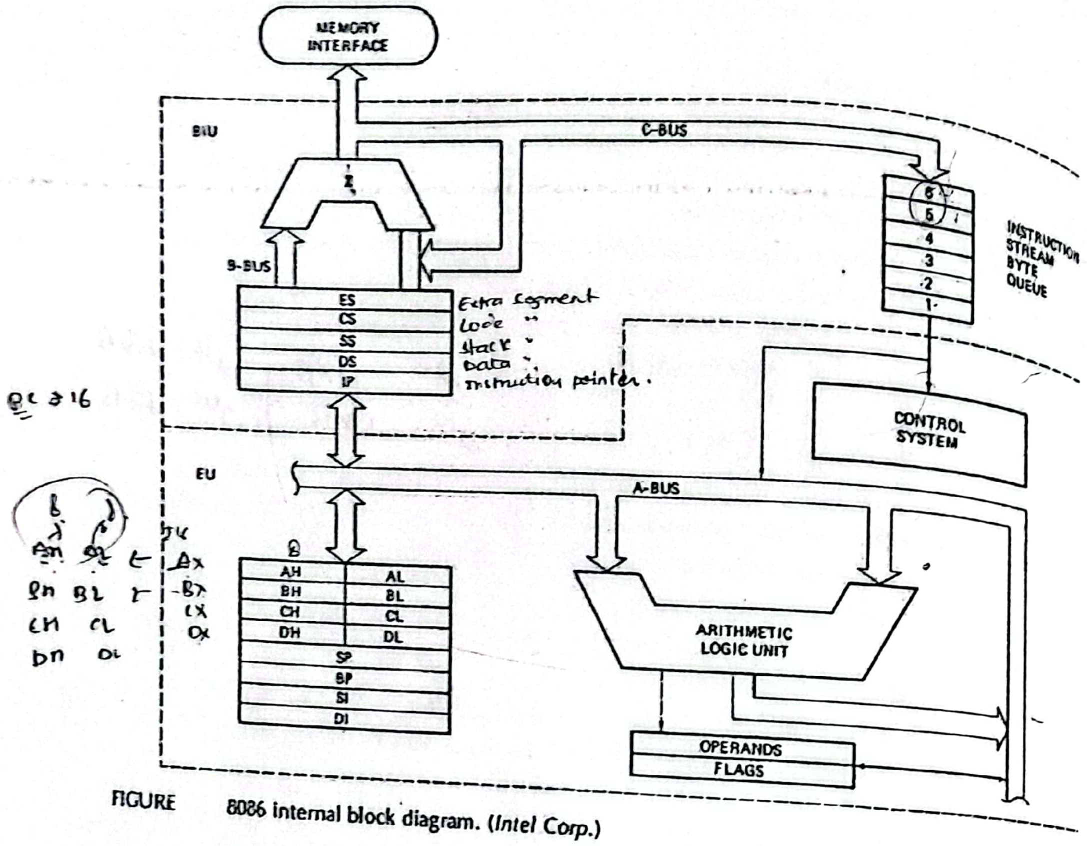


FIGURE 8086 internal block diagram. (*Intel Corp.*)

**Explanation:**

- 8086 is a 2-stage microprocessor and its architecture has two blocks: FIFO

  1. Bus Interface Unit (BIU) and

  2. Execution Unit (EU).

- 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.

**1) Bus Interface Unit (BIU):**

Contains

a) 6-byte Instruction Queue (Q)
b) The Segment Registers (CS, DS, ES, SS).

on ...

operate asynchronously to give the 8086 an over
fetch and exe

# Internal Architecture of 8086 Microprocess.

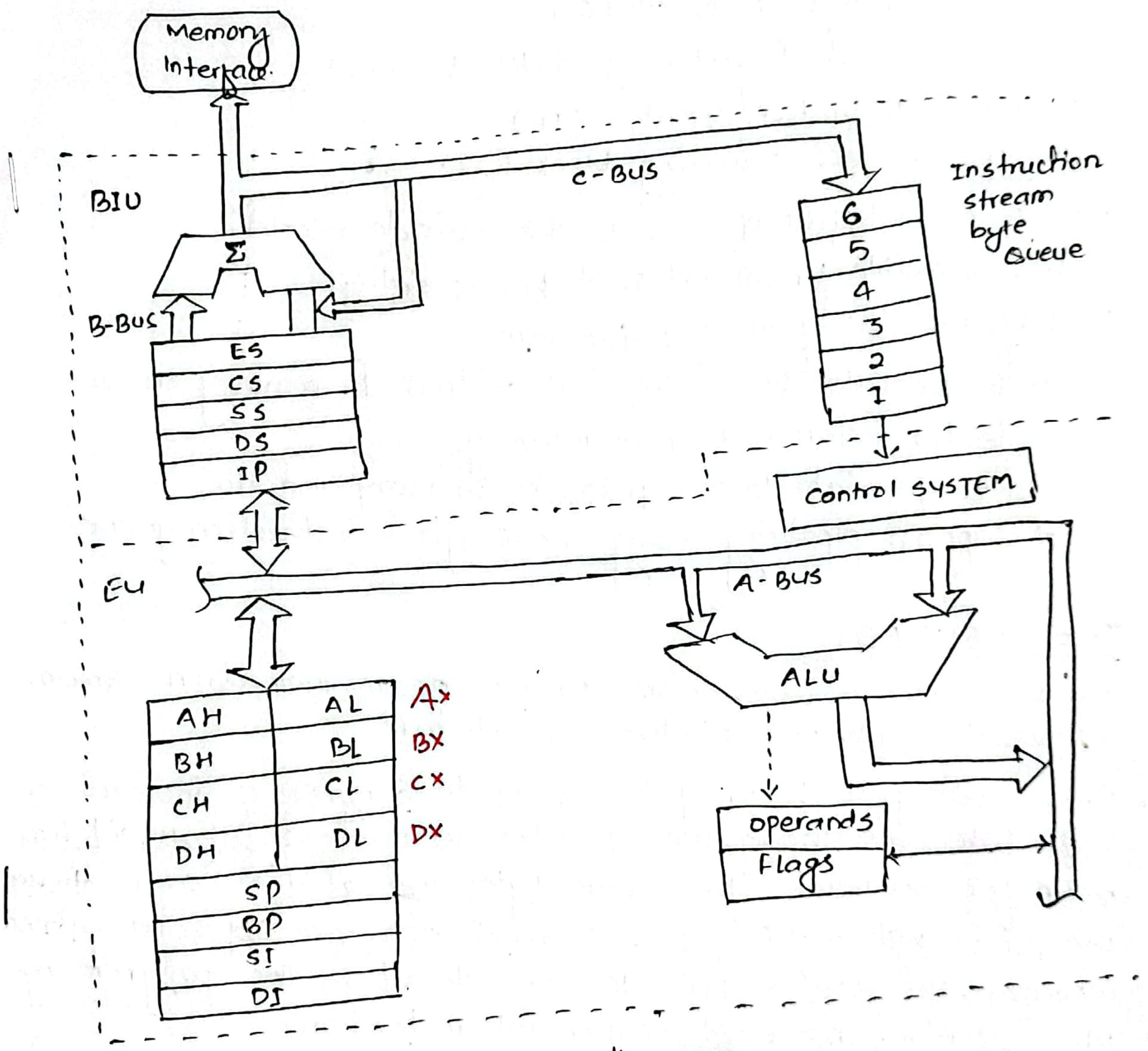


fig: 8086 Internal block diagram.

## Explanation:

↳ 8086 is a 2-stage microprocessor and its architecture has two blocks:

1. Bus Interface unit (BIU) and
2. Execution unit (EU)

↳ Both units operate asynchronously to give the 8086 an overlapping instruction fetch and execution mechanism which is called pipelining.

1. Bus Interface unit (BIU)

contains

a) 6-byte Instruction Queue (Q)

b) The segment Registers (CS, DS, ES, SS).

c) The Instruction Pointer (IP)

d) The address summing block (Σ)

↳ BIU is the interface of 8086 to the outside world.

↳ It is responsible for all external bus operation.

↳ It performs the following functions:

→ It generates 20-bit physical address for memory access

→ fetches instructions from memory

→ Transfer data to and from the memory and I/O.

→ supports pipelining using the 6-byte instruction queue.

THE QUEUE (Q)

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

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

↳ These pre-fetching instructions are held in its FIFO (first In first out) queue. with its 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.

## Instruction pointer and summing block

↳ The instruction pointer register contains a 16 bit offset address of instruction that is to be executed next.

↳ The IP always references the Code segment register (CS).

↳ The value contained in the instruction pointer is called as an offset because this value must be added to the base address of the code segment, which is available in the CS register to find the 20-bit physical address.

for eg: Memory address in 8086 = Base address x10 + offset address

let. Base address = 1000H
offset address = 2345H

Now,

20 bit physical address = (1000 x 10 + 2345) H
= (10000 + 2345) H
= 12345 H

↳ To form a 20 bit address of the next instruction, the 16 bit address of the IP is added (by the address summing block) to the address contained in the CS, which has been shifted four bits to the left.

## 2. Execution unit (EU)

↳ It fetches the instructions from the Instruction Queue, decodes and executes it.

↳ It performs arithmetic, logic, decision-making and data transfer operations.

↳ It sends request signals to the BIU to access the external module.

↳ Generates control signals.

The main components of EU are as follows:

## Segmented Memory :

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

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

↳ The complete physically available memory may be divided into a number of logical segments with each segment of 64 kbyte.

i> CS (Code Segment)

↳ This segment is used to hold the program to be executed.

↳ Instruction fetch operation is performed in this CS memory

↳ . Holds upper 16 bit of starting address ie. base address for the code.

ii> SS (Stack Segment)

↳ upper 16 bit of starting address for stack memory is kept in this register.

↳ stack Pointer (SP) in EU holds the 16 bit offset address of the Top of the stack.

↳ Physical address given by SS and SP.

iii> DS (Data Segment)

↳ Where data and operands are stored.

↳ This segment also holds the source operands during string instructions.

↳ DS register points current data segment.

↳ operands for most instructions are fetched from this segment.

iv> ES (Extra Segment):

↳ Data in access of 64 KB during some string instructions use ES to hold the destination data.

↳ ES and DI are used to determine 20 bit physical address.

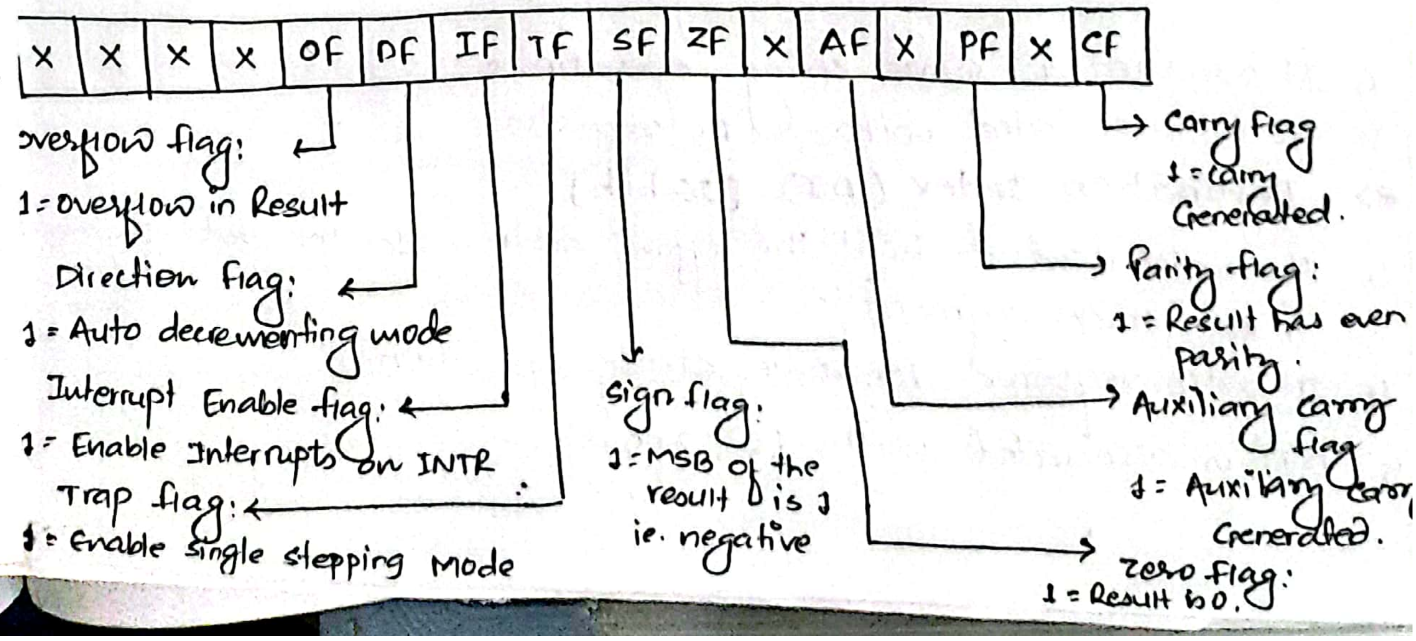
f> ALU (Arithmetic and Logic Unit) [16-bit]
→ It has a 16 bit ALU. It performs 8/16 bit binary arithmetic and logic operations.

g> Operand Register:
→ It is a 16 bit register used by control register to hold the operands temporarily.
→ It is not available to the Programmer.

h> Instruction Register and Instruction Decoder
→ 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.

i> Flag Register
> A flag is a flip flop which indicates some conditions produced by the execution of an instruction or controls certain operations of the EU.

> In 8086 The EU contains
* a 16 bit flag register
* 9 of the 16 are active flags and remaning 7 are undefined.
* 6 flags indicate some conditions - status flags
* 3 flags - control flags.



a) General Purpose Registers:

↳ 8086 has four 16-bit general purpose register AX, BX, CX and DX.

↳ These are available to the programmer for arithmetic, logic and data transfer operations.

↳ Each of these registers can be divided into two 8-bit registers like - AH-AL, BH-BL, CH-CL and DH-DL

| | | | | |
|---|---|---|---|---|
| ie. | AX = | AH-AL | = | Accumulator |
| | BX = | BH-BL | = | Base Register |
| | CX = | CH-CL | = | Counter Register |
| | DX = | DH-DL | = | Data Register. |

(16 bit) = (8bit + 8bit)

b) Stack Pointer (SP) [16 bits]:

↳ Holds the offset address of the top most elements of the stack in the stack segment.

↳ It is used by Instructions like PUSH, POP, CALL, RET etc.

c) Base Pointer (BP) [16 bits]:

↳ It is used to hold the offset address of the stack in Random Access Mode.

↳ It can be used with SS Register to access the stack rando

d) Source Index (SI) [16-bits]:

↳ Normally used to hold the offset address for the data segment but can be used for the stack segment.

↳ It required for some string operations.

↳ SI is associated with DS register.

e) Destination Index (DI) [16-bits]

↳ It is also used to hold the offset address for the data segme or the stack segment.

↳ It also required for some string operation.

↳ DI is associated with ES register.

## 3.2. Addressing modes of 8086

↳ Addressing mode is way of defining operands.
↳ It is the way in which processor can access data.

Types:
i> Immediate Addressing Mode
ii> Direct Addressing Mode
iii> Register Addressing mode
iv> Register Indirect Addressing mode.
v> Register Relative Addressing Mode.
vi> Base + Index Addressing Mode, and
vii> Base Relative + Index Addressing mode.

### i) Immediate Addressing Mode:
→ operand is specified in the instruction itself.
→ eg: MOV AL, 39H
ADD AX, 4836H

Note: MVI is not used in 8086

### ii> Direct Addressing Mode:
→ operand offset is given in the instruction as a 16 bit displacement.

eg: ADD AL, [0301H]; Adds the contents of offset address 0301H to AL.

Let the content of DS = 4000H then content of (4000H + 0301H = 40301H is added to the content of AL and result is stored in AL).

### iii> Register Indirect Addressing Mode:
↳ operand is placed in one of the register BX (not CX, DX), BP, SI, DI as specified in the instruction.

eg: (i) MOV AX, [BX]; memory address = content of BX
If [BX] = 0301H and

Let,
[0301 H] = 53H
[0302H] = 95H then,
[AX] = 9553H

Note: MOV A, M in 8085 is equivalent with MOV AX, [BX] in 8086.

ii> ADD AL, [SI] ; content of memory location addressed by SI is added to content of AL and result placed in AL

iv> Register Addressing Mode:

↳ operand is placed in one of the 16/8 bit general purpose register. (Registers must be of same size).

eg: MOV AL, BL
MOV AX, CX ; content of CX is transferred to Accumulat
MOV SI, DI ; content of DI is copied to SI register.

v> Register Relative Addressing mode:

↳ operands offset is the sum of an 8-bit displacement and the content of base register BX or index registers

eg: MOV AX, [BX + 05H] : if BX = 0301H , offset = 0301H + 05H = 0306 H
MOV [SI + 20H], BL.

Note: Relative : "extra displacement"

vi> Base + Index Addressing Mode:

↳ uses base registers (BP or BX) and one index registers (SI or DI) to indirectly address memory.

↳ Base register = beginning location of memory
Index register = relative position of an element in an arr

So, Offset = [BX or BP] + [SI or DI]

eg: (i) ADD AX , [BX + SI] ;

$$\begin{tikzcd} BX \arrow[d] & SI \arrow[d] \\ \text{Base} & \text{Index} \end{tikzcd}$$

(ii) MOV BL, [BP+SI] ; if PA = 40000H
[BX] = 1250H,
[SI] = 8214H then,

Actual address = (40000 + 1250 + 8214) H
= 49464 H , so

BL [49464 H], after execution of this instruction.

Vii) Base Relative + Index Addressing mode:

↳ combines the features of 'Base + index' and 'register relative' addressing modes.

Here, operand offset is given by,

offset = [BX or BP] + [SI or DI] + 8/16 bit displacement

eg: MOV AX, [BX + SI + 05H] : 05H is 8-bit displacement
MOV AX, [BX+SI + 1235H]: 1235H is 16 bit displacement

MOV [DI + BP + 20H], AX.

## 3.3. ALP Development Tools:
(ASSEMBLY LANGUAGE PROGRAM DEVELOPMENT TOOLS)

→ Pd
→ 16

a. EDITOR:

An editor is a software tool that allows user, the construction of an assembly language program by providing a set of commands. By using this tool, the user can type and modify the program in assembly code. The program in ~~assembly~~ created by an editor is termed as source program.

b. ASSEMBLER:

An assembler program is used to translate the assembly language mnemonics for instructions to corresponding binary codes. when you run the assembler, it reads the source file of your program from the disk where you have saved it after editing.

- on the first pass through the source programs, the assembler determines the displacement of named data items, the offset of labels, etc and puts this information in a symbol table.
- On the second pass through the source program, the assembler produces the binary code for each instruction and insert the offsets, etc. that it calculated during the first pass.
- The assembler generates 2 files on the floppy disk or hard disk. The first file is called object file (.obj).
- The second file generated by assembler is called the assembler list file and is given extension (.lst).

c. LINKER:

↳ A linker is a program used to join several object files into one large object file.

↳ The linker produces a link file which contains the binary codes for all the combined modules. The linker also produces a link map file which contains the address information about the linked files (.exe).