Dear students,

Welcome to the world of Microprocessors!

You are about to learn, how these little chips became the central force behind the computer revolution. Brain child of Alan Turing, nurtured by genius minds like Von Neumann, Maurice Wilkes, and materialized by corporations like Intel, Samsung, Apple… these chips transformed the way the world computed.

From traffic signals to air traffic control, drones to satellites, fitness bands to pace makers, Microprocessors have brought to you, ever improving standards of health, travel, entertainment, communication etc.

In this book, you begin with learning the 8086 Architecture.
You then learn 8086 in depth including its Memory module, Instruction set, Programming and interfacing with its set of coprocessors and peripherals.

You will then proceed to learning powerful microprocessors like Pentium.
You will see the speed advantage achieved in Pentium using superscalar architecture and on chip cache memories. You will then witness the brilliance of Branch Prediction Algorithm which minimized the biggest drawback of pipelining.

Before we start, allow me to take the opportunity to thank my mother,
Prof. Veena D. Acharya.
A teacher all her life, she was the primary inspiration for me to pursue this noble profession.
Her blessings are reflected in the enthusiasm shown in this book and in my lectures.

Bharat D. Acharya.
B. E. Computer Science.
Founder, Bharat Acharya Education.
Teaching Microprocessors & Microcontrollers since 2000.

INTRODUCTION TO MICROPROCESSORS

Drones, stump vision cameras, mobile phones, RFID sensors, autonomous cars, streaming servers, your favorite shopping website… all of these are fruits of a seed called **"Microprocessor"**, planted years ago in mid 1970s and in fact conceived even earlier in 1940s.

So what does this Microprocessor (henceforth called µP) actually do… Why do we need to learn it… And most importantly, after completing our education, how will this knowledge be useful to us…

Where do we use a µP? Is a µP used in a fan, or in a tube light, or in a switch board? No, none of them. Is it used in a mobile phone, a computer, a microwave oven, a washing machine? Yes, all of them! This is because they run on programs, and all those programs are executed by the microprocessor within these devices. **This is the main function of a µP, to execute programs**.

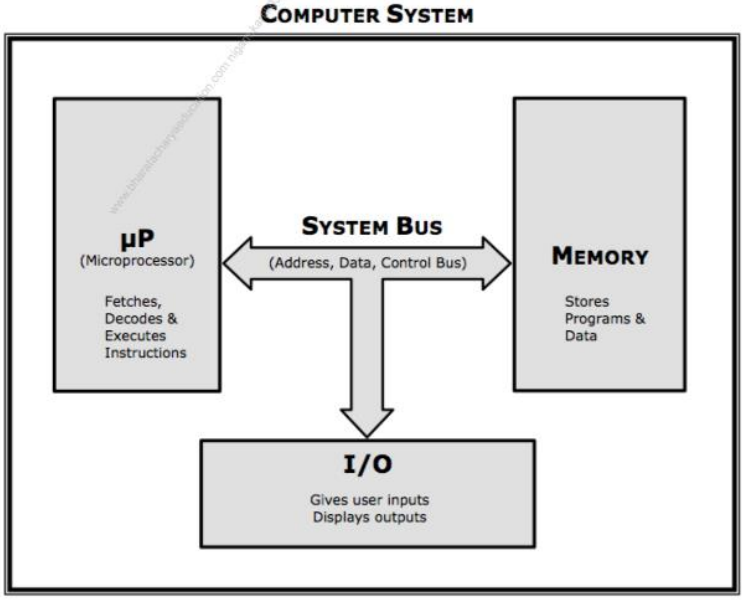
In our day to day encounters we come across several devices and appliances. If you feel any of them works on a program, you should most certainly realize, it must contain a microprocessor. Take a microwave oven as an example. The µP inside the oven isn't directly cooking the food. It is running programs that are responsible for rotating the dish, maintaining the correct temperature, counting the desired number of seconds, displaying the time remaining on the screen, and finally ringing the sweet alarm (ding!) informing us that the cooking is complete. All of these require programs, that are executed by the oven's µP. And who writes the programs? The engineer, yes that's you! It is this this combination of the engineers mind and the microprocessors execution abilities that has transformed the world in the past few decades and will continue to do so as both are getting ever so smart in the new age.

The simplest example of the use of a microprocessor, is in a **computer**. When you imagine a computer, lots of devices come to our mind, like the keyboard, mouse, printer, monitor and the big "box" also casually referred to by laymen as the CPU. Of course, you know their roles! The Keyboard and mouse are called input devices. Are they executing our programs? No! When we want to add two numbers, neither is the keyboard adding them nor is the mouse. So, are they required? Yes! To give inputs. That's their role. To provide inputs to the system. Similarly, the printer and the monitor are used to produce outputs. Hence these are called I/O devices (Input and Output devices). This leaves us with that big "box". Open it and you see an electric circuit board also called the motherboard. **On the motherboard, pretty much around the center lies a big chip, around the size of our palm, that's your µP**. You may notice in modern motherboards the µP is generally covered with a fan, to dissipate the heat generated by relentlessly performing millions and sometimes billions of operations per second. So how important is the µP? Well, remove it and our computer becomes a piece of junk! Every activity of your computer needs a program, lets get you in agreement with that. You watch a movie, play a song, surf the net, be on social media etc., all of these are programs. Executing them keeps the µP busy round the clock, hence the heat and the fan.

Let's say we are headed to the market right now, to buy the latest computer. Which microprocessor will you find inside? Yes, the Intel Core i7, or maybe Core i5 or i3, Intel also has recently released the Core i9 but its way too expensive to be mass produced as yet. So are we learning Core i7, i5, i3… not so early! These are a

BASIC ORGANIZATION OF A COMPUTER

A computer system, as we know it, consists of various components.
They can be broadly classified into three sections:
The Processor, Memory and I/O.



THE PROCESSOR – "µP"

The heart of the computer is its µP (Microprocessor). Current generation computers use processors like Intel Core i3, i5 or i7 and so on. They have come a long way from the initial processors that you are about to learn E.g.: 8085, 8086 etc.

Back in the day (1940s), when micro-electronics was not invented, processors looked very different and were certainly not "micro" in appearance. They were created using huge arrays of physical switches which were operated manually and often occupied large rooms.

In the following decades, with the invention of micro-electronics, scientists managed to embed thousands of microscopic switches (transistors) inside a small chip, and called it a "**Micro-processor**".

Over the years, microprocessors grew in strength. From housing a few thousand transistors (8085) to containing more than a billion transistors (Core i7), the computational power has been increasing exponentially. Having said that, some of the basics still remain the same.

To put it simply, **the main function of a µP is to Fetch, Decode and Execute instructions.**

Instructions are a part of programs. Programs are stored in the memory. First, **µP fetches an instruction** from the memory. It then **decodes the instruction**. This means, it "understands" the binary pattern of the instruction, also called its opcode. Every instruction when stored in the memory is in its unique binary form, which indicates the operation to be performed. **This is called its opcode**. Upon decoding the opcode, µP understands the operation to be performed and hence "executes" the instruction. This entire process is called an **"Instruction cycle"**. This process is repeated for the next instruction. Like this, one by one, all instructions of a program are executed.
Of course, by new concepts like **pipelining**, **multitasking**, **multiprocessing** etc., this procedure has become very advanced and efficient today. You will get to learn all of them, in the due course of this ever-intriguing subject.

We begin learning with basic processors like **8085** or **8086**, but make no mistake, none of this is "outdated". Yes, your mobile phone or your computer today uses the most advanced processors (Apple's **A12 Bionic** etc.), but to run a **traffic light** or **TV remote** control you don't need a core i7 now, do you? And these are used by the millions across the world. They simply use processors of the same grade as an 8085 or an 8086, with different product numbers as they are made by various manufacturers.

MEMORY

Memory is used to **store information**.
It stores two kinds of information… **programs and data**. Yes, think about it! Everything that's stored in your computer's memory is either some program or some data. MS Word is a program, the Word Documents are data. Your Image Viewer is a program, the images are data. WhatsApp is a program, its messages are data. Instagram is a program, the feed on the wall is the data… and so on!

All programs and data are stored in the memory, in digitized form, where every information is represented in 1s and 0s called binary digits or simply bits.

There are various forms of memory devices.
The main memory also called primary memory consists of Ram and ROM.
Other memory devices like Hard disk, Floppy, CD/ DVD etc. are secondary storage devices.

Additionally there is also a high speed memory called Cache composed of SRAM.
For the majority portion of this book, you are dealing with the initial processors like 8086.
It will be in your best interest to think of Primary Memory only, whenever we speak of memory. That is because, secondary memory and high speed memories were implemented much later in the evolution of processors as the demand for mass storage and high speed performance started increasing. So, from now on in this book, unless specified otherwise, **the word memory refers to primary memory that is RAM and ROM.**

The memory is a series of locations.
Each location is identified by its own unique address.
Every memory location contains 1 Byte (8 bits) of data. There is a very good reason for this, and you will learn it when we discuss the topic of memory banking in 8086.

I/O Devices

I/O devices are used to enter programs and data as inputs and display or print the results as outputs.
We are all familiar with devices such as the keyboard, mouse, printer, monitor etc. Every form of computer system has a set of I/O devices for human interaction. A device like a touch screen performs dual functions of both input and output.

The µP, Memory and I/O are all connected to each other using the System Bus.

System Bus

A Bus is a set of lines. They are used to transfer information, obviously in binary form.
A line connected to Vcc will carry a logic 1 and if connected to Gnd it will carry logic 0.
Hence one line can transfer 1 bit. If we want multiple bits to be transmitted together, then we use a set of lines grouped together and that's called a bus.

The size of a bus refers to the number of lines it contains and is always given in terms of bits.
E.g.:: An 8-bit bus has 8 lines, a 16 bit bus has 16 lines and so on.

There are three types of busses… Address, Data and Control Bus. Collectively they are called the system bus. Lets take a closer look at them.

i) Address Bus

It carries the address where the operation has to be performed. Say the processor wants to write some data at a memory location. Firstly the processor will give the desired address on the address bus. This address will select a unique memory location. That's when data will be transferred with that location.
It is therefore obvious that **bigger the address bus, more is the number of memory locations** it can address and hence bigger the total size of the memory. Here is the relation between size of the address bus and size of the memory.
An address bus of 1 bit can give a total of two addresses: 0 and 1.
Hence can access a total memory of two locations.
A 2-bit address bus can generate a total of 4 addresses 00, 01, 10, 11 and hence can access a total of 4 locations.
3-bit address bus… 8 locations and so on.

So here is the rule,

An N-bit address bus can totally access 2^N locations.

As mentioned earlier, one memory location contains one byte of data.
This brings us to the following conclusion.
A processor with an N-bit address bus can access a memory of 2^N Bytes.

Lets solve a typical VIVA (oral exam) question.

Given the size of address bus, you have to figure out the size of the memory.

ADDRESS BUS (N - BIT)	MEMORY (2^N BYTES)
4 – bit	2^4 = 16 Bytes
6 – bit	2^6 = 64 Bytes
10 – bit	2^{10} = 1024 Bytes = 1 KB
16 – bit	$2^{16} = 2^6 \times 2^{10}$ = 64 x 1 K = 64 KB … (8085)
20 – bit	$2^{20} = 2^{10} \times 2^{10}$ = 1 K x 1 K = 1 MB … (8086)
30 – bit	$2^{30} = 2^{10} \times 2^{20}$ = 1 K x 1 M = 1 GB
32 – bit	$2^{32} = 2^2 \times 2^{30}$ = 4 x 1 G = 4 GB …. (80386 and Pentium)
40 – bit	$2^{40} = 2^{10} \times 2^{30}$ = 1K x 1G = 1 TB … (Typical Hard Disk)

In case you found it a little difficult after the 4th row, that is because you may have got a little confused with the powers of 2. This issue will persist all along the subject. The smarter thing to do is once for all, lets get this hurdle past us. Lets get all powers of 2 clearly sorted. You will realize in the due course of this subject, how helpful this small exercise will prove to be. Frankly speaking there's rarely a topic in this subject that is not connected with some power of 2. Lets sort this, totally!

Powers of 2

2^N	VALUE
2^0	0
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024 = 1K
2^{20}	2^{10} x 2^{10} = 1K x 1K = 1M
2^{24}	2^4 x 2^{20} = 16 x 1M = 16M
2^{28}	2^8 x 2^{20} = 256 x 1M = 256M
2^{30}	2^{10} x 2^{20} = 1K x 1M = 1G
2^{36}	2^6 x 2^{30} = 64 x 1G = 64G
2^{40}	2^{10} x 2^{30} = 1K x 1G = 1T

Similarly, there is one more very important fundamental that needs to be sorted out before we start learning the bigger concepts. You need to be sure of number representation and number conversions. You may have been familiar with various number systems like Decimal, Hexadecimal, Binary etc… Out of all of them, Hexadecimal and Binary are the two most widely used number systems in our course of learning this subject.

Every number we write, either in programs or in theory examples, is a hexadecimal number. When this number gets stored inside the computer, it is in binary form This simply means, you must be very well versed with hex-binary conversions as this will be needed while understanding various examples.

A single hexadecimal digit ranges from 0H… FH. This has 16 options. Hence, to represent the number in binary we need 4 bits as 2^4 =16. The following table shows this conversion.

HEX	BINARY
0 H	0000
1 H	0001
2 H	0010
3 H	0011
4 H	0100
5 H	0101
6 H	0110
7 H	0111
8 H	1000
9 H	1001
A H	1010
B H	1011
C H	1100
D H	1101
E H	1110
F H	1111

NO! You do not need to mug this up. There is a simple trick of "8421" that can get you any value from the above table. Consider the number 5 which is basically 4+1. Now consider the four binary bits corresponding to values 8, 4, 2 and 1. Since we need the equivalent of 5 which is 4 plus 1, we need 4 and 1 but we don't need 8 and 2 so in positions of 8 and 2 we put a "0" and in positions of 4 and 1 we put a "1". So we get 0101. Lets take the example of 0 H. We don't need any of them (8 or 4 or 2 or 1). So we put a "0" for all of them and hence get 0000. If we need F (which is 15), it is 8 + 4 + 2 + 1 so we need all of them and hence the binary equivalent is 1111. Take 9 as an example, 9 = 8 + 1. So we need 8 and 1 but not 4 and 2 so we put a "1" for 8 and 1 positions and a "0" for 4 and 2 positions giving us 1001. Hope you can now convert any hex digit into binary.

Now consider a two digit number like 25H. To convert this to binary we need to substitute the 4 bit equivalents of 2 and 5 respectively. 2H is 0010 and 5H is 0101.

Hence the number 25H in binary will be 0010 0101.
Similarly a number like 74H will be 0111 0100 in binary. 89H will be 1000 1001 and so on.

As you noticed, the numbers 25H, 74H and 89H are all 2 digits in hexadecimal and hence need 8 bits in binary. Such numbers are called 8 bit numbers. The range of 8 bit and 16 bit numbers is mentioned below:

8 BIT NUMBERS (ALSO CALLED A "BYTE")

HEX	BINARY
00 H	0000 0000
01 H	0000 0001
...	...
68 H	0110 1000
...	...
FE H	1111 1110
FF H	1111 1111

16 BIT NUMBERS (ALSO CALLED A "WORD")

HEX	BINARY
0000 H	0000 0000 0000 0000
0001 H	0000 0000 0000 0001
...	...
4831 H	0100 1000 0011 0001
...	...
FFFE H	1111 1111 1111 1110
FFFF H	1111 1111 1111 1111

ii) DATA BUS

It carries data to and from the processor.
The size of data bus determines how much data can be transferred in one operation (cycle).
Bigger the data bus, faster the processor, as it can transfer more data in one cycle.

iii) CONTROL BUS

It Carries control signals like RD, WR etc.
These signals determine the kind of operation that will be performed on the system bus.

8086 Basics – Buses and Memory

Address Bus

- 8086 has a 20-bit address bus.
- This means it can access 2^{20} = 1 MB memory.
- **One memory location contains one byte of data.**
- A processor with an N-bit address bus can access a memory of 2^N Bytes.

Address Bus (N - bit)	Memory (2^N Bytes)
4 – bit	2^4 = 16 Bytes
6 – bit	2^6 = 64 Bytes
10 – bit	2^{10} = 1024 Bytes = 1 KB
16 – bit	$2^{16} = 2^6 \times 2^{10}$ = 64 x 1 K = 64 KB … (8085)
20 – bit	$2^{20} = 2^{10} \times 2^{10}$ = 1 K x 1 K = 1 MB … (8086)
30 – bit	$2^{30} = 2^{10} \times 2^{20}$ = 1 K x 1 M = 1 GB
32 – bit	$2^{32} = 2^2 \times 2^{30}$ = 4 x 1 G = 4 GB …. (80386 and Pentium)
40 – bit	$2^{40} = 2^{10} \times 2^{30}$ = 1K x 1G = 1 TB … (Typical Hard Disk)

In case you found it a little difficult after the 4th row, that is because you may have got a little confused with the powers of 2. This issue will persist all along the subject. The smarter thing to do is once for all, lets get this hurdle past us. Lets get all powers of 2 clearly sorted. You will realize in the due course of this subject, how helpful this small exercise will prove to be. Frankly speaking there's rarely a topic in this subject that is not connected with some power of 2. Lets sort this, totally!

Powers of 2

2^N	VALUE
2^0	0
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024 = 1K
2^{20}	$2^{10} \times 2^{10}$ = 1K x 1K = 1M
2^{24}	$2^4 \times 2^{20}$ = 16 x 1M = 16M
2^{28}	$2^8 \times 2^{20}$ = 256 x 1M = 256M
2^{30}	$2^{10} \times 2^{20}$ = 1K x 1M = 1G
2^{36}	$2^6 \times 2^{30}$ = 64 x 1G = 64G

Note

Bigger the address bus, more is the memory accessed by the processor.

DATA BUS

- 8086 has a 16-bit address bus.
- This means it can transfer 16-bit data in one cycle.
- Higher processors have bigger data buses.
- 80386 has a 32-bit data bus
- Pentium has a and 64-bit data bus.
- Hence they can access more data in one cycle

Note

Bigger the data bus, faster is the processor.

ALU

- 8086 has a 16-bit ALU.
- This means it can add/subtract 16-bits in one cycle.
- **Hence 8086 is called a 16-bit microprocessor. ← VIVA Question**

Note on 8088
Intel released a cheaper version of 8086 called 8088. It also has a 16-bit ALU but has an 8-bit data bus.

CONTROL BUS

There are many control signals which you will learn in due course of the subject, but the three main ones are…

- $\overline{RD}$ = 0 for read operations
- $\overline{WR}$ = 0 for write operations
- M/$\overline{IO}$ = 0 for I/O operations and 1 for Memory operations.

M/$\overline{IO}$	$\overline{RD}$	$\overline{WR}$	OPERATION
0	0	1	I/O Read
0	1	0	I/O Write
1	0	1	Memory Read
1	1	0	Memory Write

Special Note:

If you are learning this by piracy, then you are not my student. You are simply a thief!
#PoorUpbringing

Little Endian Rule:

Lower byte is stored at the lower address
Higher byte is stored at the higher address

MEMORY ACCESSED BY 8086

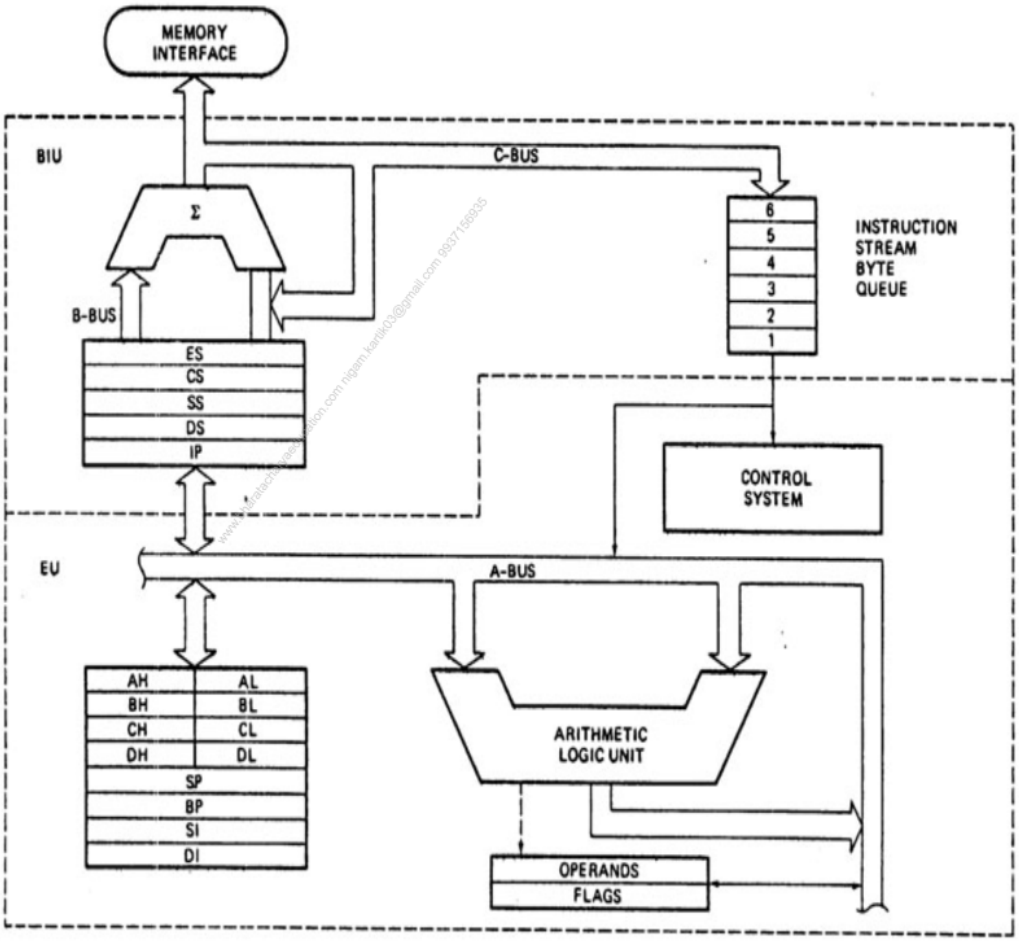
	Address	Data
First Location	00000H	00H ... FFH
	00001H	00H ... FFH
	00002H	00H ... FFH
	...	00H ... FFH
	FFFFEH	00H ... FFH
Last Location	FFFFFH	00H ... FFH
	← 20-bit →	← 8-bit →

Special Note:

If you are learning this by piracy, then you are not my student. You are simply a thief!
#PoorUpbringing

Author: Bharat Acharya
Sem IV – Electronics
Mumbai 2018

ARCHITECTURE OF 8086



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.

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** (16_d), to give the **20-bit physical address** of the **Code Segment**.
Eg: If **CS = 4321H** then CS × 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** (16_d), to give the **20-bit physical address** of the **Data Segment**.
Eg: If **DS = 4321H** then DS × 10H = **43210H → Starting address** of Data Segment.

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

b) Instruction Pointer (IP register)
It is a **16-bit register**.
It **holds offset of** the **next instruction in** the **Code Segment**.

3) SS Register

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

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.

c) Address Generation Circuit

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

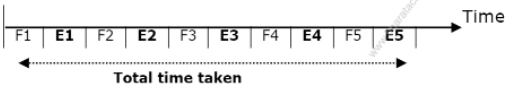
Physical address = Segment Address x 10h + Offset Address

Viva Question: Explain the real procedure to obtain the Physical Address?
The Segment address is left shifted by 4 positions, this multiplies the number by 16 (i.e. 10h) and then the offset address is added.
Eg: 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 $(0001\ 0010\ 0011\ 0100\ \mathbf{0000})_{binary}$ 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.

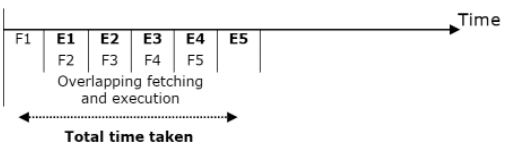
d) 6-Byte Pre-Fetch Queue {Pipelining – 4m}

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 atleast 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.

NON-PIPELINED PROCESSOR EG: 8085



PIPELINED PROCESSOR EG: 8086



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). ☺ **For doubts contact Bharat Sir on 98204 08217**

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**.

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.

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. #Please refer Bharat Sir's Lecture Notes for this ...

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.**

*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 Seg, during **String Operations.**

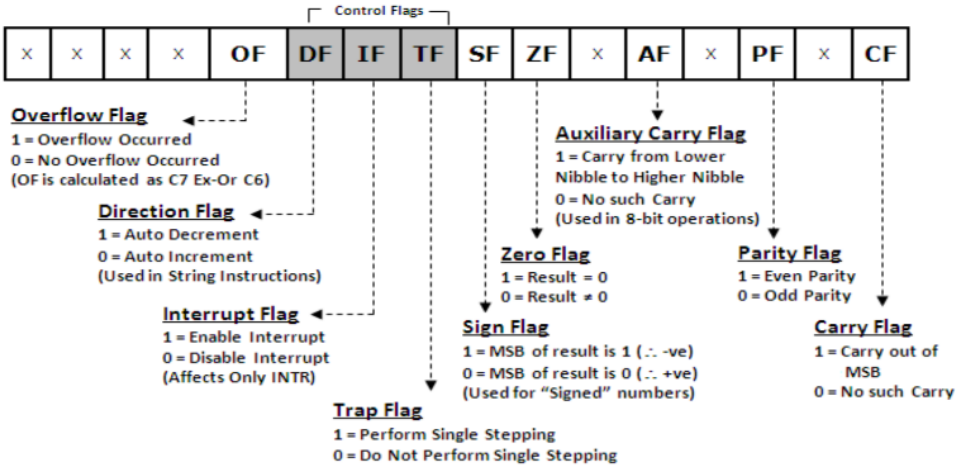
c) ALU (16-Bits)
It has a **16-bit ALU**. It performs 8 and 16-bit arithmetic and logic operations.

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.

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.



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.

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). #Please refer Bharat Sir's Lecture Notes for this ...

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**.

Videos | Books | Classroom Coaching
E: bharatsir@hotmail.com
M: 9820408217

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. ☺ For doubts contact Bharat Sir on 98204 08217
- 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**.

1 MB

512 KB	512 KB

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:

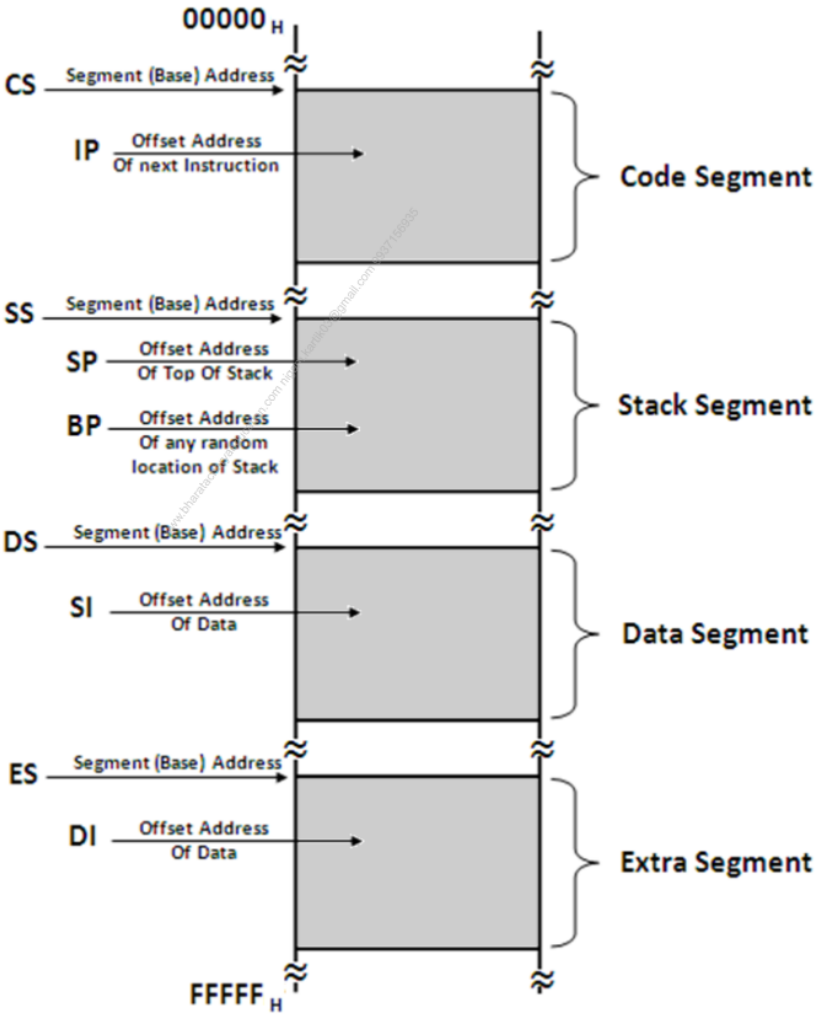
00000H
00002H
00004H
.
.
.
FFFFEH

- 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).

Author: Bharat Acharya
Sem IV – Electronics
Mumbai 2018

MEMORY SEGMENTATION IN 8086



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.**
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) 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**.
18) The physical address is calculated by the microprocessor, using the formula:

    PHYSICAL ADDRESS = SEGMENT ADDRESS X 10H + OFFSET ADDRESS

19) Ex: if Segment Address = 1234H and Offset Address is 0005H then
    Physical Address = 1234H x 10H + 0005H = 12345H
20) This formula automatically ensures that the **minimum size of a segment is 10H bytes** (10H = 16 Bytes).

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.

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.

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**.

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.

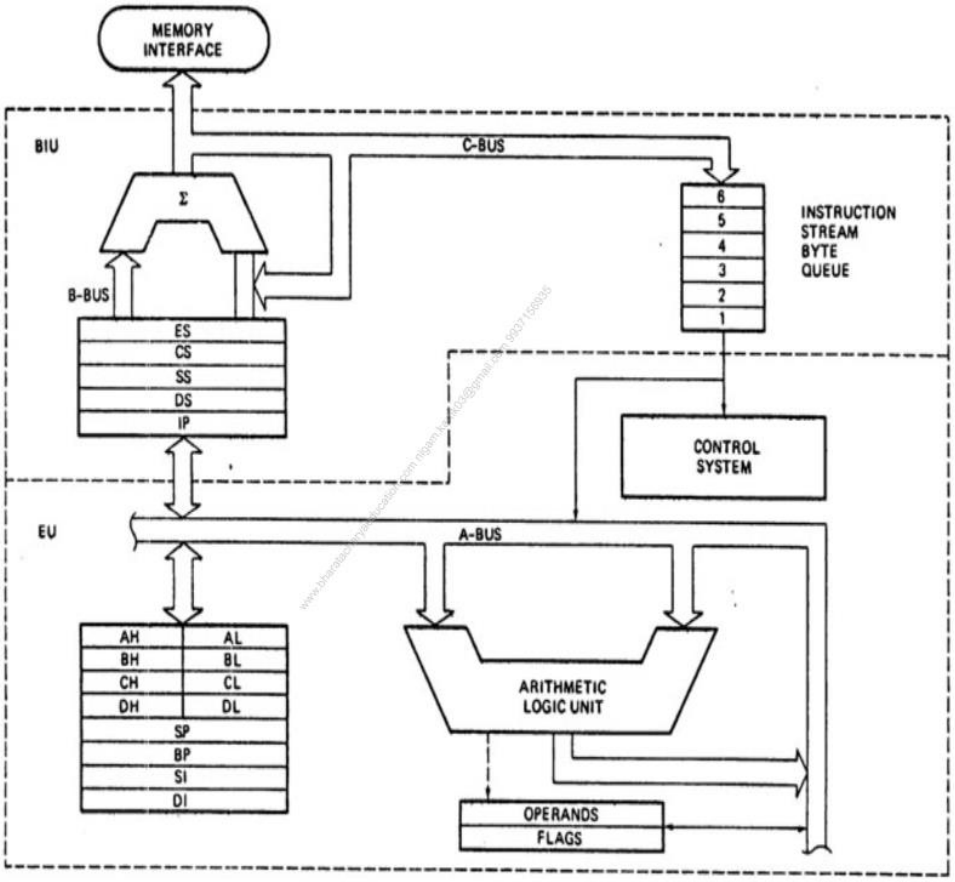
Advantages of Segmentation:

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**.

ARCHITECTURE OF 8086



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.

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** (16_d), to give the **20-bit physical address** of the **Code Segment**.
Eg: If **CS = 4321H** then CS × 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** (16_d), to give the **20-bit physical address** of the **Data Segment**.
Eg: If **DS = 4321H** then DS × 10H = **43210H → Starting address** of Data Segment.

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

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.

c) Address Generation Circuit

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

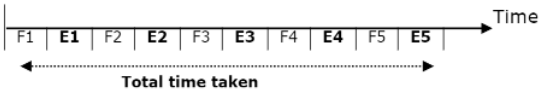
Physical address = Segment Address x 10h + Offset Address

Viva Question: Explain the real procedure to obtain the Physical Address?
The Segment address is left shifted by 4 positions, this multiplies the number by 16 (i.e. 10h) and then the offset address is added.
Eg: 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 $(0001\ 0010\ 0011\ 0100\ \mathbf{0000})_{binary}$ 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.

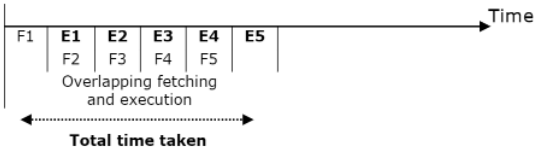
d) 6-Byte Pre-Fetch Queue {Pipelining – 4m}

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 atleast 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.

NON-PIPELINED PROCESSOR EG: 8085



PIPELINED PROCESSOR EG: 8086



Videos | Books | Classroom Coaching
E: bharatsir@hotmail.com
M: 9820408217

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). ☺ **For doubts contact Bharat Sir on 98204 08217**

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**.

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.

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. #Please refer Bharat Sir's Lecture Notes for this ...

*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.**

*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 Seg, during **String Operations.**

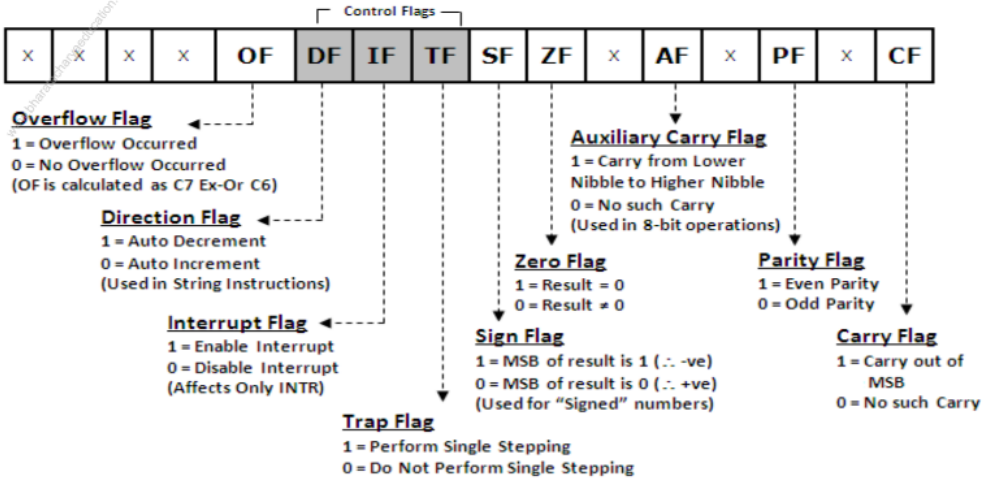
c) ALU (16-Bits)
It has a **16-bit ALU**. It performs 8 and 16-bit arithmetic and logic operations.

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.

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.



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.

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). #Please refer Bharat Sir's Lecture Notes for this ...

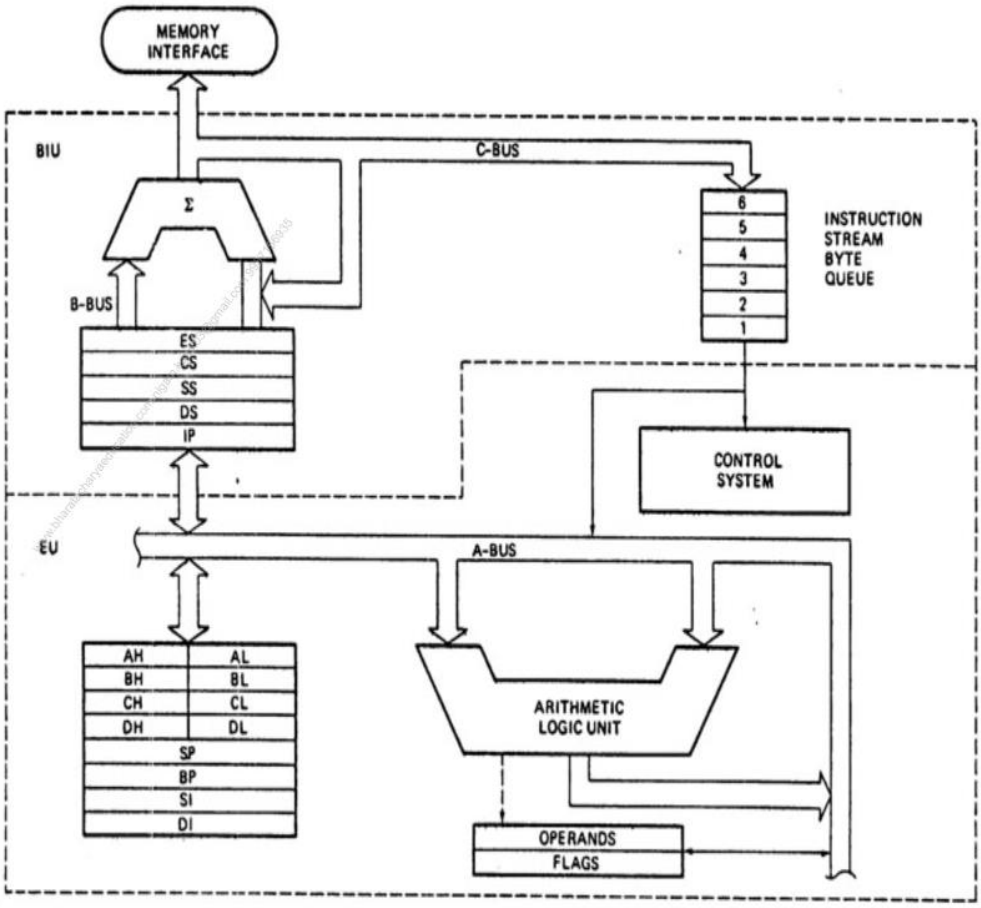
CONTROL FLAGS

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**.

ARCHITECTURE OF 8086



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.

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** (16_d), to give the **20-bit physical address** of the **Code Segment**.
Eg: If **CS = 4321H** then CS × 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** (16_d), to give the **20-bit physical address** of the **Data Segment**.
Eg: If **DS = 4321H** then DS × 10H = **43210H → Starting address** of Data Segment.

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

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.

c) Address Generation Circuit

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

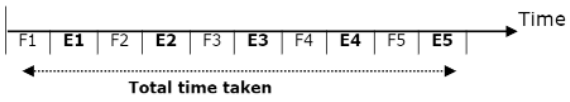
Physical address = Segment Address x 10h + Offset Address

Viva Question: Explain the real procedure to obtain the Physical Address?
The Segment address is left shifted by 4 positions, this multiplies the number by 16 (i.e. 10h) and then the offset address is added.
Eg: 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 $(0001\ 0010\ 0011\ 0100\ \mathbf{0000})_{binary}$ 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.

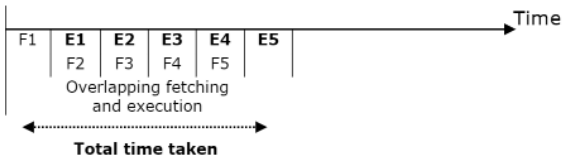
d) 6-Byte Pre-Fetch Queue {Pipelining – 4m}

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 atleast 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.

NON-PIPELINED PROCESSOR EG: 8085



PIPELINED PROCESSOR EG: 8086



Videos | Books | Classroom Coaching
E: bharatsir@hotmail.com
M: 9820408217

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). ☺ **For doubts contact Bharat Sir on 98204 08217**

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**.

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.

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. #Please refer Bharat Sir's Lecture Notes for this ...

*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.**

*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 Seg, during **String Operations.**

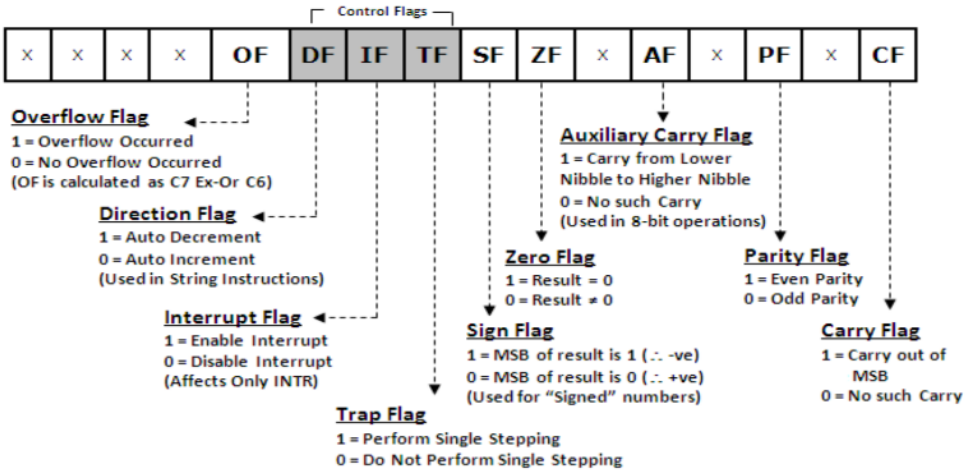
c) ALU (16-Bits)
It has a **16-bit ALU**. It performs 8 and 16-bit arithmetic and logic operations.

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.

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.



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.
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). #Please refer Bharat Sir's Lecture Notes for this ...

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**.

ADDRESSING MODES OF 8086

8086 provides different addressing modes for Data, Program and Stack Memory.

ADDRESSING MODES FOR DATA MEMORY {IMP}

I IMMEDIATE ADDRESSING MODE

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

```
Eg:  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.

```
Eg:  MOV CL, DL           ; Moves data of DL register into CL register
     MOV AX, BX           ; Moves data of BX register into AX register
```

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.

```
Eg:  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]

Eg:  MOV CX, [4320H]      ; Moves data from location 4320H and 4321H
                          ; in the data segment into CL and CH resp.
```

IV INDIRECT ADDRESSING MODES

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

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

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

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.**

```
Eg: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

Eg:   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
```

BASE INDEXED ADDRESSING MODE

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

```
Eg:   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

Eg:   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
```

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**.

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

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

V IMPLIED ADDRESSING MODE

In this addressing mode the operands are implied and are hence not specified in the instruction.
#Please refer Bharat Sir's Lecture Notes for this ...

Eg: **STC** *; Sets the Carry Flag.*

Eg: CLD *; Clears the Direction Flag.*

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.*
BX holds Offset Address for Data Segment.
BP holds Offset Address for Stack Segment.

4) *SI and DI are called Index Registers*

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

Data Transfer Instructions

1) **MOV Destination, Source**
Moves a byte/word **from** the **source to** the **destination** specified in the instruction.
Source: *Register, Memory Location, Immediate Number*
Destination: Register, Memory Location
Both, source and destination cannot be memory locations.
Eg: **MOV CX, 0037H** ; *CX ← 0037H*
MOV BL, [4000H] ; *BL ← DS:[4000H]*
MOV AX, BX ; *AX ← BX*
MOV DL, [BX] ; *DL ← DS:[BX]*
MOV DS, BX ; *DS ← BX*

2) **PUSH Source**
Push the **source** (word) **into** the **stack** and decrement the stack pointer by two.
The source MUST be a **WORD** (**16 bits**).
Source: *Register, Memory Location*
Eg: **PUSH CX** ; *SS:[SP-1] ← CH, SS:[SP-2] ← CL*
; *SP ← SP – 2*
PUSH DS ; *SS:[SP-1, SP-2] ← DS*
; *SP ← SP – 2*

3) **POP Destination**
POP a **word from** the **stack** into the given destination and increment the Stack Pointer by 2. The destination MUST be a **WORD** (16 bits).
Destination: *Register [EXCEPT CS], Memory Location*
Eg: **POP CX** ; *CH ← SS:[SP], CL ← SS:[SP+1]*
; *SP ← SP + 2*
POP DS ; *DS ← SS:[SP, SP+1]*
; *SP ← SP + 2*

***Please Note*: MOV, PUSH, POP** are the ONLY instructions that use the Segment Registers as operands {except CS}.

4) **PUSHF**
Push value of **Flag Register into stack** and decrement the stack pointer by 2.
Eg: **PUSHF** ; *SS:[SP-1] ← $Flag_H$, SS:[SP-2] ← $Flag_L$, SP ← SP – 2*

5) **POPF**
POP a **word from** the **stack into** the **Flag register**.
Eg: **POPF** ; *$Flag_L$ ← SS:[SP], $Flag_H$ ← SS:[SP+1], SP ← SP + 2*

6) **XCHG Destination, Source**
Exchanges a byte/word between the **source and** the **destination** specified in the instruction.
Source: *Register, Memory Location*
Destination: Register, Memory Location
Even here, both operands cannot be memory locations.
Eg: **XCHG CX, BX** ; *CX ←→ BX*
XCHG BL, CH ; *BL ←→ CH*

7) **XLATB / XLAT** (very important)
Move into AL, the **contents of** the **memory location** in Data Segment, **whose** effective **address** is **formed by** the **sum of BX and AL**.
Eg: **XLAT** ; *AL ← DS:[BX + AL]*
; i.e. if DS = 1000H; BX = 0200H; AL = 03H
; ∴ 10000 … DS × 16
; + 0200 … BX
; + 03 … AL
; =**10203** ∴ **AL ← [10203H]**

Note: the difference between XLAT and XLATB

In XLATB there is no operand in the instruction.
E.g.:: XLATB
It works in an implied mode and does exactly what is shown above.

In XLAT, we can specify the name of the look up table in the instruction
E.g.:: XLAT SevenSeg
This will do the translation form the look up table called SevenSeg.
In any case, the base address of the look up table must be given by BX.

8) **LAHF**
Loads AH with **lower byte** of the **Flag** Register.

9) **SAHF**
Stores the contents of **AH into** the **lower byte** of the **Flag** Register.

10) **LEA register, source**
Loads Effective Address (offset address) **of** the **source into** the **given register**.
Eg: **LEA BX, Total** ; *BX ← offset address of Total in Data Segment.*

11) **LDS destination register, source**
Loads the **destination register and DS** register **with offset** address **and segment address** specified by the **source**.
Eg: **LDS BX, Total** ; *BX ← {DS:[Total], DS:[Total + 1]},*
; *DS ← {DS: [Total + 2], DS:[Total + 3]}*

12) **LES destination register, source**
Loads the **destination register and ES** register with the **offset address and** the **segment address** indirectly specified by the **source**.
Eg: **LES BX, Total** ; *BX ← {DS:[Total], DS:[Total + 1]},*
; *ES ← {DS: [Total + 2], DS:[Total + 3]}*

I/O ADDRESSING MODES OF 8086 (5m – Important Question)

I/O addresses in 8086 can be either 8–bit or 16-bit

Direct Addressing Mode:

If we use **8-bit I/O address** we get a **range of 00H... FFH**.
This gives a total of **256 I/O ports.**
Here we use Direct addressing Mode, that is, the **I/O address is specified in the instruction.**

E.g.:: IN AL, 80H **; AL gets data from I/O port address 80H.**

This is also called **Fixed Port Addressing**.

Indirect Addressing Mode:

If we use **16-bit I/O address** we get a **range of 0000H... FFFFH**.
This gives a total of **65536 I/O ports.**
Here we use Indirect addressing Mode, that is, the **I/O address is specified by DX register.**

E.g.:: MOV DX, 2000H
IN AL, DX **; AL gets data from I/O port address 2000H given by DX.**

This is also called **Variable Port Addressing**.

13)IN destination register, source port
Loads the **destination register with** the contents of the **I/O port** specified by the source.

Source: It is an I/O port address.
If the address is 8-bit it will be given in the instruction by **Direct addressing mode.**
If it is a 16 bit address it will be given by DX register using **Indirect addressing mode.**

Destination: It has to be some form of "A" register, in which we will get data from the I/O device.
If we are getting 8-bit data, it will be AL or AH register.
If we are getting 16-bit data, it will be AX register.

Eg: **IN AL, 80H** *; AL gets 8-bit data from I/O port address 80H*
IN AX, 80H *; AX gets 16-bit data from I/O port address 80H*
IN AL, DX *; AL gets 8-bit data from I/O port address given by DX.*
IN AX, DX *; AX gets 16-bit data from I/O port address given by DX.*

14)OUT destination port, source register
Loads the destination I/O port with the contents of the source register.

Eg: **OUT 80H, AL** *; I/O port 80H gets 8-bit data from AL*
OUT 80H, AX *; I/O port 80H gets 16-bit data from AX*
OUT DX, AL *; I/O port whose address is given by DX gets 8-bit data from AL*
OUT DX, AX *; I/O port whose address is given by DX gets 16-bit data from AX*

Videos | Books | Classroom Coaching
E: bharatsir@hotmail.com
M: 9820408217

Data Transfer Instructions

1) MOV Destination, Source

Moves a byte/word **from** the **source to** the **destination** specified in the instruction.
Source: *Register, Memory Location, Immediate Number*
Destination: Register, Memory Location
Both, source and destination cannot be memory locations.

Eg: **MOV CX, 0037H** ; *CX ← 0037H*
MOV BL, [4000H] ; *BL ← DS:[4000H]*
MOV AX, BX ; *AX ← BX*
MOV DL, [BX] ; *DL ← DS:[BX]*
MOV DS, BX ; *DS ← BX*

2) PUSH Source

Push the **source** (word) **into** the **stack** and decrement the stack pointer by two.
The source MUST be a **WORD** (**16 bits**).
Source: *Register, Memory Location*

Eg: **PUSH CX** ; *SS:[SP-1] ← CH, SS:[SP-2] ← CL*
; *SP ← SP – 2*
PUSH DS ; *SS:[SP-1, SP-2] ← DS*
; *SP ← SP – 2*

3) POP Destination

POP a **word from** the **stack** into the given destination and increment the Stack Pointer by 2. The destination MUST be a **WORD** (16 bits).
Destination: *Register [EXCEPT CS], Memory Location*

Eg: **POP CX** ; *CH ← SS:[SP], CL ← SS:[SP+1]*
; *SP ← SP + 2*
POP DS ; *DS ← SS:[SP, SP+1]*
; *SP ← SP + 2*

***Please Note*: MOV, PUSH, POP** are the ONLY instructions that use the Segment Registers as operands {except CS}.

4) PUSHF

Push value of **Flag Register into stack** and decrement the stack pointer by 2.
Eg: **PUSHF** ; *SS:[SP-1] ← $Flag_H$, SS:[SP-2] ← $Flag_L$, SP ← SP – 2*

5) POPF

POP a **word from** the **stack into** the **Flag register**.
Eg: **POPF** ; *$Flag_L$ ← SS:[SP], $Flag_H$ ← SS:[SP+1], SP ← SP + 2*

6) XCHG Destination, Source

Exchanges a byte/word between the **source and** the **destination** specified in the instruction.
Source: *Register, Memory Location*
Destination: Register, Memory Location
Even here, both operands cannot be memory locations.

Eg: **XCHG CX, BX** ; *CX ←→ BX*
XCHG BL, CH ; *BL ←→ CH*

7) **XLATB / XLAT** (very important)
Move into AL, the **contents of** the **memory location** in Data Segment, **whose** effective **address** is **formed by** the **sum of BX and AL**.

```
Eg: XLAT                ; AL ← DS:[BX + AL]
                        ; i.e. if DS = 1000H; BX = 0200H; AL = 03H
                        ; ∴ 10000   … DS × 16
                        ; +  0200   … BX
                        ; +    03   … AL
                        ; =10203 ∴ AL ← [10203H]
```

Note: the difference between XLAT and XLATB

In XLATB there is no operand in the instruction.
E.g.:: XLATB
It works in an implied mode and does exactly what is shown above.

In XLAT, we can specify the name of the look up table in the instruction
E.g.:: XLAT SevenSeg
This will do the translation form the look up table called SevenSeg.
In any case, the base address of the look up table must be given by BX.

8) **LAHF**
Loads AH with **lower byte** of the **Flag** Register.

9) **SAHF**
Stores the contents of **AH into** the **lower byte** of the **Flag** Register.

10) **LEA register, source**
Loads Effective Address (offset address) **of** the **source into** the **given register**.

```
Eg: LEA BX, Total       ; BX ← offset address of Total in Data Segment.
```

11) **LDS destination register, source**
Loads the **destination register and DS** register **with offset** address **and segment address** specified by the **source**.

```
Eg: LDS BX, Total       ; BX ← {DS:[Total], DS:[Total + 1]},
                        ; DS ← {DS: [Total + 2], DS:[Total + 3]}
```

12) **LES destination register, source**
Loads the **destination register and ES** register with the **offset address and** the **segment address** indirectly specified by the **source**.

```
Eg: LES BX, Total       ; BX ← {DS:[Total], DS:[Total + 1]},
                        ; ES ← {DS: [Total + 2], DS:[Total + 3]}
```

I/O ADDRESSING MODES OF 8086 (5m – Important Question)

I/O addresses in 8086 can be either 8–bit or 16-bit

Direct Addressing Mode:

If we use **8-bit I/O address** we get a **range of 00H... FFH**.
This gives a total of **256 I/O ports.**
Here we use Direct addressing Mode, that is, the **I/O address is specified in the instruction.**

E.g.:: IN AL, 80H **; AL gets data from I/O port address 80H.**

This is also called **Fixed Port Addressing**.

Indirect Addressing Mode:

If we use **16-bit I/O address** we get a **range of 0000H... FFFFH**.
This gives a total of **65536 I/O ports.**
Here we use Indirect addressing Mode, that is, the **I/O address is specified by DX register.**

E.g.:: MOV DX, 2000H
IN AL, DX **; AL gets data from I/O port address 2000H given by DX.**

This is also called **Variable Port Addressing**.

13)IN destination register, source port
Loads the **destination register with** the contents of the **I/O port** specified by the source.

Source: It is an I/O port address.
If the address is 8-bit it will be given in the instruction by **Direct addressing mode.**
If it is a 16 bit address it will be given by DX register using **Indirect addressing mode.**

Destination: It has to be some form of "A" register, in which we will get data from the I/O device.
If we are getting 8-bit data, it will be AL or AH register.
If we are getting 16-bit data, it will be AX register.

Eg: **IN AL, 80H** *; AL gets 8-bit data from I/O port address 80H*
IN AX, 80H *; AX gets 16-bit data from I/O port address 80H*
IN AL, DX *; AL gets 8-bit data from I/O port address given by DX.*
IN AX, DX *; AX gets 16-bit data from I/O port address given by DX.*

14)OUT destination port, source register
Loads the destination I/O port with the contents of the source register.

Eg: **OUT 80H, AL** *; I/O port 80H gets 8-bit data from AL*
OUT 80H, AX *; I/O port 80H gets 16-bit data from AX*
OUT DX, AL *; I/O port whose address is given by DX gets 8-bit data from AL*
OUT DX, AX *; I/O port whose address is given by DX gets 16-bit data from AX*