**Architecture of 8086**

**Introduction :**

The 8086 microprocessor is an 8-bit/16-bit microprocessor designed by Intel in the late 1970s. It is the first member of the x86 family of microprocessors, which includes many popular CPUs used in personal computers.

The architecture of the 8086 microprocessor is based on a complex instruction set computer (CISC) architecture, which means that it supports a wide range of instructions, many of which can perform multiple operations in a single instruction. The 8086 microprocessor has a 20-bit address bus, which can address up to 1 MB of memory, and a 16-bit data bus, which can transfer data between the microprocessor and memory or I/O devices.

The 8086 microprocessor has a segmented memory architecture, which means that memory is divided into segments that are addressed using both a segment register and an offset. The segment register points to the start of a segment, while the offset specifies the location of a specific byte within the segment. This allows the 8086 microprocessor to access large amounts of memory, while still using a 16-bit data bus.

The 8086 microprocessor has two main execution units: the execution unit (EU) and the bus interface unit (BIU). The BIU is responsible for fetching instructions from memory and decoding them, while the EU executes the instructions. The BIU also manages data transfer between the microprocessor and memory or I/O devices.

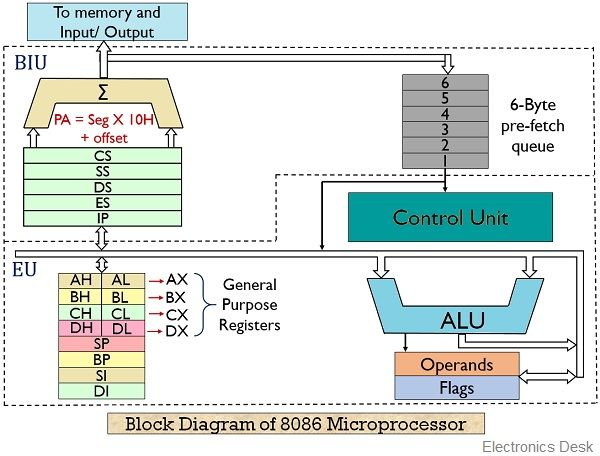
A **Microprocessor** is an Integrated Circuit with all the functions of a CPU. However, it cannot be used stand-alone since unlike a microcontroller it *has no memory or peripherals*.

[8086](https://www.geeksforgeeks.org/pin-diagram-8086-microprocessor/) does not have a RAM or ROM inside it. However, it has *internal registers* for storing intermediate and final results and interfaces with memory located outside it through the System Bus.

In the case of 8086, it is a 16-bit **Integer processor** in a 40-pin, Dual Inline Packaged IC.

The size of the internal registers(present within the chip) indicates how much information the processor can operate on at a time (*in this case 16-bit registers*) and how it moves data around internally within the chip, sometimes also referred to as the internal data bus.

8086 provides the programmer with 14 internal registers, each of 16 bits or 2 bytes wide.**The main advantage of the 8086 microprocessor is that it supports Pipelining.**



[**Memory segmentation:**](https://www.geeksforgeeks.org/memory-segmentation-8086-microprocessor/)

* In order to increase execution speed and fetching speed, 8086 segments the memory.
* Its 20-bit address bus can address 1MB of memory, it segments it into 16 64kB segments.
* 8086 works only with four 64KB segments within the whole 1MB memory.

The internal architecture of Intel 8086 is divided into 2 units: **The Bus Interface Unit (BIU)**, and **The Execution Unit (EU)**. These are explained as following below.

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

It provides the interface of 8086 to external memory and I/O devices via the System Bus. It performs various machine cycles such as memory read, I/O read, etc. to transfer data between memory and I/O devices.

BIU performs the following functions are as follows:

* It generates the 20-bit physical address for memory access.
* It fetches instructions from the memory.
* It transfers data to and from the memory and I/O.
* Maintains the 6-byte pre-fetch instruction queue(**supports pipelining**).

BIU mainly contains the **4 Segment registers**, the **Instruction Pointer**, a pre-fetch queue, and an **Address Generation Circuit**.

**Instruction Pointer (IP):**

* It is a *16-bit register*. It holds offset of the next instructions in the*Code Segment.*
* IP is incremented after every instruction byte is fetched.
* IP gets a new value whenever a branch instruction occurs.
* CS is multiplied by 10H to give the 20-bit physical address of the Code Segment.
* The address of the next instruction is calculated by using the formula CS x 10H + IP.

**Example:**

CS = 4321H ; IP = 1000H (Here Offset = Instruction Pointer(IP))

then CS x 10H = 43210H + offset = ***44210H***

This is the address of the next instruction.

**Code Segment register: (16 Bit register):**CS holds the base address for the Code Segment. All programs are stored in the Code Segment and accessed via the IP.

**Data Segment register: (16 Bit register):**DS holds the base address for the Data Segment.

**Stack Segment register: (16 Bit register):**SS holds the base address for the Stack Segment.

**Extra Segment register:** **(16 Bit register):**ES holds the base address for the Extra Segment.

*Please note that segments are present in memory and segment registers are present in Microprocessor.  
Segment registers store starting address of each segments in memory.*

**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:
* In Bus Interface Unit (BIU) the circuit shown by the Σ symbol is responsible for the calculation unit which is used to calculate the physical address of an instruction in memory.

*Physical Address = Segment Address x 10H + Offset Address*

**6 Byte Pre-fetch Queue:**

* It is a 6-byte queue (FIFO).
* Fetching the next instruction (by BIU from CS) while executing the current instruction is called pipelining.
* Gets flushed whenever a branch instruction occurs.
* The pre-Fetch queue is of 6-Bytes only because the maximum size of instruction that can have in 8086 is 6 bytes. Hence to cover up all operands and data fields of maximum size instruction in 8086 Microprocessor there is a Pre-Fetch queue is 6 Bytes.
* The pre-Fetch queue is connected with the control unit which is responsible for decoding op-code and operands and telling the execution unit what to do with the help of timing and control signals.
* The pre-Fetch queue is responsible for pipelining and because of that 8086 microprocessor is called fetch, decode, execute type microprocessor. Since there are always instructions present for decoding and execution in this queue the speed of execution in the microprocessor is gradually increased.
* **When there is a 2-byte space in the instruction pre-fetch queue then only the next instruction will be pushed into the queue** otherwise if only a 1-byte space is vacant then there will not be any allocation in the queue. It will wait for a spacing of 2 bytes in subsequent queue decoding operations.
* Instruction pre-fetch queue works in a sequential manner so if there is any branch condition then in that situation pre-fetch queue fails. Hence to avoid chaos instruction queue is flushed out when any branch or conditional jumps occur.

**2.prefetch unit:**

The Prefetch Unit in the 8086 microprocessor is a component responsible for fetching instructions from memory and storing them in a queue. The prefetch unit allows the 8086 to perform multiple instruction fetches in parallel, improving the overall performance of the microprocessor.

The prefetch unit consists of a buffer and a program counter that are used to fetch instructions from memory. The buffer stores the instructions that have been fetched and the program counter keeps track of the memory location of the next instruction to be fetched. The prefetch unit fetches several instructions ahead of the current instruction, allowing the 8086 to execute instructions from the buffer rather than from memory.

This parallel processing of instruction fetches helps to reduce the wait time for memory access, as the 8086 can continue to execute instructions from the buffer while it waits for memory access to complete. This results in improved overall performance, as the 8086 is able to execute more instructions in a given amount of time.

The prefetch unit is an important component of the 8086 microprocessor, as it allows the microprocessor to work more efficiently and perform more instructions in a given amount of time. This improved performance helps to ensure that the 8086 remains competitive in its performance and capabilities, even as technology continues to advance.

**3. The Execution Unit (EU):**

The main components of the EU are General purpose registers, the ALU, Special purpose registers, the Instruction Register and Instruction Decoder, and the Flag/Status Register.

1. Fetches instructions from the Queue in BIU, decodes, and executes arithmetic and logic operations using the ALU.
2. Sends control signals for internal data transfer operations within the microprocessor.(Control Unit)
3. Sends request signals to the BIU to access the external module.
4. It operates with respect to T-states (clock cycles) and not machine cycles.

[8086 has four 16-bit general purpose registers](https://www.geeksforgeeks.org/general-purpose-registers-8086-microprocessor/) AX, BX, CX, and DX which store intermediate values during execution. Each of these has two 8-bit parts (higher and lower).

* **AX register:** **(Combination of AL and AH Registers)**  
  It holds operands and results during multiplication and division operations. Also an accumulator during String operations.
* **BX register: (Combination of BL and BH Registers)**  
  It holds the memory address (offset address) in indirect addressing modes.
* **CX register: (Combination of CL and CH Registers)**  
  It holds the count for instructions like a loop, rotates, shifts and string operations.
* **DX register: (Combination of DL and DH Registers)**  
  It is used with AX to hold 32-bit values during multiplication and division.

**Arithmetic Logic Unit (16-bit):**Performs**8 and 16-bit**arithmetic and logic operations.

**Special purpose registers (16-bit):** Special purpose registers are called Offset registers also. Which points to specific memory locations under each segment.

We can understand the concept of segments as Textbook pages. Suppose there are 10 chapters in one textbook and each chapter takes exactly 100 pages. So the book will contain 1000 pages. Now suppose we want to access page number 575 from the book then 500 will be the segment base address which can be anything in the context of microprocessors like Code, Data, Stack, and Extra Segment. So 500 will be segment registers that are present in Bus Interface Unit (BIU). And 500 + 75 is called an offset register through which we can reach on specific page number under a specific segment.

*Hence 500 is the segment base address and 75 is an offset address or (Instruction Pointer, Stack Pointer, Base Pointer, Source Index, Destination Index) any of the above according to their segment implementation.*

* **Stack Pointer:** Points to Stack top. Stack is in Stack Segment, used during instructions like PUSH, POP, CALL, RET etc.
* **Base Pointer:** BP can hold the offset addresses of any location in the stack segment. It is used to access random locations of the stack.
* **Source Index:** It holds offset address in Data Segment during string operations.
* **Destination Index:**It holds offset address in Extra Segment during string operations.

**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.

[**Flag/Status register (16 bits)**](https://www.geeksforgeeks.org/flag-register-8086-microprocessor/)**:**It has 9 flags that help change or recognize the state of the microprocessor.

**6 Status flags:**

1. Carry flag(CF)
2. Parity flag(PF)
3. Auxiliary carry flag(AF)
4. Zero flag(Z)
5. Sign flag(S)
6. Overflow flag (O)

Status flags are updated after every arithmetic and logic operation.

**3 Control flags:**

1. Trap flag(TF)
2. Interrupt flag(IF)
3. Direction flag(DF)

These flags can be set or reset using control instructions like CLC, STC, CLD, STD, CLI, STI, etc. The Control flags are used to control certain operations.

**4.Decode unit:**

The Decode Unit in the 8086 microprocessor is a component that decodes the instructions that have been fetched from memory. The decode unit takes the machine code instructions and translates them into micro-operations that can be executed by the microprocessor’s execution unit.

The Decode Unit works in parallel with the Prefetch Unit, which fetches instructions from memory and stores them in a queue. The Decode Unit reads the instructions from the queue and translates them into micro-operations that can be executed by the microprocessor.

The Decode Unit is an important component of the 8086 microprocessor, as it allows the microprocessor to execute instructions efficiently and accurately. The decode unit ensures that the microprocessor can execute complex instructions, such as jump instructions and loop instructions, by translating them into a series of simple micro-operations.

The Decode Unit is responsible for decoding instructions, performing register-to-register operations, and performing memory-to-register operations. It also decodes conditional jumps, calls, and returns, and performs data transfers between memory and registers.

The Decode Unit helps to improve the performance of the 8086 microprocessor by allowing it to execute instructions quickly and accurately. This improved performance helps to ensure that the 8086 remains competitive in its performance and capabilities, even as technology continues to advance.

**5.control unit :**

The Control Unit in the 8086 microprocessor is a component that manages the overall operation of the microprocessor. The control unit is responsible for controlling the flow of instructions through the microprocessor and coordinating the activities of the other components, including the Decode Unit, Execution Unit, and Prefetch Unit.

The Control Unit acts as the central coordinator for the microprocessor, directing the flow of data and instructions and ensuring that the microprocessor operates correctly. It also monitors the state of the microprocessor, ensuring that the correct sequence of operations is followed.

The Control Unit is responsible for fetching instructions from memory, decoding them, executing them, and updating the microprocessor’s state. It also handles interrupt requests and performs system management tasks, such as power management and error handling.

The Control Unit is an essential component of the 8086 microprocessor, as it allows the microprocessor to operate efficiently and accurately. The control unit ensures that the microprocessor can execute complex instructions, such as jump instructions and loop instructions, by coordinating the activities of the other components.

The Control Unit helps to improve the performance of the 8086 microprocessor by managing the flow of instructions and data through the microprocessor, ensuring that the microprocessor operates correctly and efficiently. This improved performance helps to ensure that the 8086 remains competitive in its performance and capabilities, even as technology continues to advance.

**The 8086 microprocessor uses three different buses to transfer data and instructions between the microprocessor and other components in a computer system. These buses are:**

**1.Address Bus:** The address bus is used to send the memory address of the instruction or data being read or written. The address bus is 16 bits wide, allowing the 8086 to address up to 64 kilobytes of memory.

**2.Data Bus:** The data bus is used to transfer data between the microprocessor and memory. The data bus is 16 bits wide, allowing the 8086 to transfer 16-bit data words at a time.

**3.Control Bus:** The control bus is used to transfer control signals between the microprocessor and other components in the computer system. The control bus is used to send signals such as read, write, and interrupt requests, and to transfer status information between the microprocessor and other components.

The buses in the 8086 microprocessor play a crucial role in allowing the microprocessor to access and transfer data from memory, as well as to interact with other components in the computer system. The 8086’s ability to use these buses efficiently and effectively helps to ensure that it remains competitive in its performance and capabilities, even as technology continues to advance.

**Execution of whole 8086 Architecture:**

1. All instructions are stored in memory hence to fetch any instruction first task is to obtain the Physical address of the instruction is to be fetched. Hence this task is done by Bus Interface Unit (BIU) and by Segment Registers. Suppose the Code segment has a Segment address and the Instruction pointer has some offset address then the physical address calculator circuit calculates the physical address in which our instruction is to be fetched.
2. After the address calculation instruction is fetched from memory and it passes through C-Bus (Data bus) as shown in the figure, and according to the size of the instruction, the instruction pre-fetch queue fills up. For example *MOV AX, BX*is 1 Byte instruction so it will take only the 1st block of the queue, and *MOV BX,4050H*  is 3 Byte instruction so it will take 3 blocks of the pre-fetch queue.
3. When our instruction is ready for execution, according to the FIFO property of the queue instruction comes into the control system or control circuit which resides in the Execution unit.***Here instruction decoding takes place***. The decoding control system generates an opcode that tells the microprocessor unit which operation is to be performed. So the control system sends signals all over the microprocessor about what to perform and what to extract from General and Special  Purpose Registers.
4. Hence after decoding microprocessor fetches data from GPR and according to instructions like ADD, SUB, MUL, and DIV data residing in GPRs are fetched and put as ALU’s input. and after that addition, multiplication, division, or subtraction whichever calculation is to be carried out.
5. According to arithmetic, flag register values change dynamically.
6. ***While Instruction was decoding and executing from step-3 of our algorithm, the Bus interface Unit doesn’t remain idle. it continuously fetches an instruction from memory and put it in a pre-fetch queue and gets ready for execution in a FIFO manner whenever the time arrives.***
7. So in this way, unlike the 8085 microprocessor, here the fetch, decode, and execution process happens in parallel and not sequentially. This is called*pipelining,* and because of the instruction pre-fetch queue, all fetching, decoding, and execution process happen side-by-side. Hence there is partitioning in 8086 architecture like Bus Interface Unit and Execution Unit to support Pipelining phenomena.

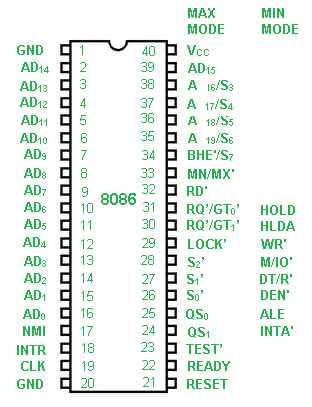
**Pin diagram of 8086 microprocessor**

The 8086 chip, presented by Intel in 1978, is a critical part in the development of processing innovation. Its pin outline is a basic part of figuring out its design and usefulness. The pin chart gives a visual portrayal of the chip’s outer associations and their individual capabilities, including power supply, ground, information, address, and control lines. Each pin is assigned for explicit jobs, for example, information movement, memory tending to, and framework control, which are fundamental for connecting with other equipment parts. Breaking down the 8086 pin outline is vital for planning and investigating frameworks that consolidate this persuasive microchip.

**What is 8086 Microprocessor?**

The 8086 microchip, created by Intel in 1978, is a 16-digit processor that noticeable a huge headway in figuring innovation. It includes a 16-cycle information transport and a 20-bit address transport, permitting it to address up to 1 MB of memory. The 8086 is known for its portioned memory engineering, which partitions memory into sections for more proficient access and the executives. It upholds a rich arrangement of guidelines and methods of activity, making it flexible for different applications. As a basic part in early PCs, the 8086 made ready for the improvement of later chips and impacted PC engineering plans.

Pin diagram of 8086 [microprocessor](https://www.geeksforgeeks.org/introduction-of-microprocessor/)is as given below:



Intel 8086 is a 16-bit HMOS microprocessor. It is available in 40 pin DIP chip. It uses a 5V DC supply for its operation. The 8086 uses a 20-line address bus. It has a 16-line data bus. The 20 lines of the address bus operate in multiplexed mode. The 16-low order address bus lines have been multiplexed with data and 4 high-order address bus lines have been multiplexed with status signals.

**AD0-AD15:**Address/Data bus. These are low order address bus. They are multiplexed with data. When AD lines are used to transmit memory address the symbol A is used instead of AD, for example A0-A15. When data are transmitted over AD lines the symbol D is used in place of AD, for example D0-D7, D8-D15 or D0-D15.

**A16-A19:**High order address bus. These are multiplexed with status signals.

**S2, S1, S0:**Status pins. These pins are active during T4, T1 and T2 states and is returned to passive state (1,1,1 during T3 or Tw (when ready is inactive). These are used by the 8288 bus controller for generating all the memory and I/O operation) access control signals. Any change in S2, S1, S0 during T4 indicates the beginning of a bus cycle.

| **S2** | **S1** | **S0** | **Characteristics** |
| --- | --- | --- | --- |
| 0 | 0 | 0 | Interrupt acknowledge |
| 0 | 0 | 1 | Read I/O port |
| 0 | 1 | 0 | Write I/O port |
| 0 | 1 | 1 | Halt |
| 1 | 0 | 0 | Code access |
| 1 | 0 | 1 | Read memory |
| 1 | 1 | 0 | Write memory |
| 1 | 1 | 1 | Passive state |

**A16/S3, A17/S4, A18/S5, A19/S6 :** The specified address lines are multiplexed with corresponding status signals.

| **A17/S4** | **A16/S3** | **Function** |
| --- | --- | --- |
| 0 | 0 | Extra segment access |
| 0 | 1 | Stack segment access |
| 1 | 0 | Code segment access |
| 1 | 1 | Data segment access |

**HE’/S7:**Bus High Enable/Status. During T1 it is low. It is used to enable data onto the most significant half of data bus, D8-D15. 8-bit device connected to upper half of the data bus use BHE (Active Low) signal. It is multiplexed with status signal S7. S7 signal is available during T2, T3 and T4.

**RD’:** This is used for read operation. It is an output signal. It is active when low.

**READY :** This is the acknowledgement from the memory or slow device that they have completed the data transfer. The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the microprocessor. The signal is active high(1).

**INTR :**Interrupt Request. This is triggered input. This is sampled during the last clock cycles of each instruction for determining the availability of the request. If any interrupt request is found pending, the processor enters the interrupt acknowledge cycle. This can be internally masked after resulting the interrupt enable flag. This signal is active high(1) and has been synchronized internally.

**NMI :**Non maskable interrupt. This is an edge triggered input which results in a type II interrupt. A subroutine is then vectored through an interrupt vector lookup table which is located in the system memory. NMI is non-maskable internally by software. A transition made from low(0) to high(1) initiates the interrupt at the end of the current instruction. This input has been synchronized internally.

**INTA :** Interrupt acknowledge. It is active low(0) during T2, T3 and Tw of each interrupt acknowledge cycle.

**MN/MX’ :** Minimum/Maximum. This pin signal indicates what mode the processor will operate in.

**RQ’/GT1′, RQ’/GT0′ :** Request/Grant. These pins are used by local bus masters used to force the microprocessor to release the local bus at the end of the microprocessor’s current bus cycle. Each of the pin is bi-directional. RQ’/GT0′ have higher priority than RQ’/GT1′.

**LOCK’ :** Its an active low pin. It indicates that other system bus masters have not been allowed to gain control of the system bus while LOCK’ is active low(0). The LOCK signal will be active until the completion of the next instruction.

**TEST’ :**This examined by a ‘WAIT’ instruction. If the TEST pin goes low(0), execution will continue, else the processor remains in an idle state. The input is internally synchronized during each of the clock cycle on leading edge of the clock.

**CLK :** Clock Input. The clock input provides the basic timing for processing operation and bus control activity. Its an asymmetric square wave with a 33% duty cycle.

**RESET :**This pin requires the microprocessor to terminate its present activity immediately. The signal must be active high(1) for at least four clock cycles.

**Vcc :** Power Supply( +5V D.C.)

**GND :** Ground

**QS1,QS0 :** Queue Status. These signals indicate the status of the internal 8086 instruction queue according to the table shown below:

| **QS1** | **QS0** | **Status** |
| --- | --- | --- |
| 0 | 0 | No operation |
| 0 | 1 | First byte of op code from queue |
| 1 | 0 | Empty the queue |
| 1 | 1 | Subsequent byte from queue |

**M/IO’:**This signal is used to distinguish between memory and I/O operations. The M Signal is Active high whereas the IO’ Signal is Active Low. When this Pin is High, the memory operations takes place. On the other hand, when the Pin is low, the Input/Output operations from the peripheral devices takes place.

**=DT/R :** Data Transmit/Receive. This pin is required in minimum systems, that want to use an 8286 or 8287 data bus transceiver. The direction of data flow is controlled through the transceiver.

**DEN:** Data enable. This pin is provided as an output enable for the 8286/8287 in a minimum system which uses transceiver. DEN is active low(0) during each memory and input-output access and for INTA cycles.

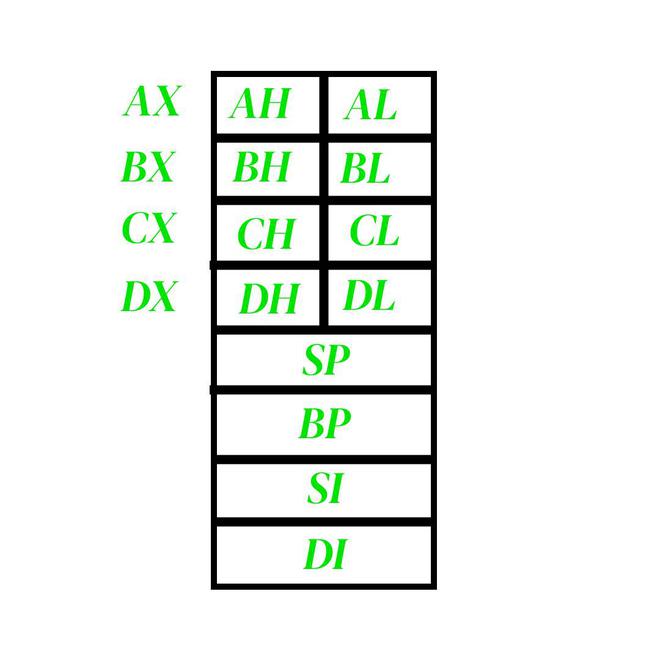
**HOLD/HOLDA:** HOLD indicates that another master has been requesting a local bus .This is an active high(1). The microprocessor receiving the HOLD request will issue HLDA (high) as an acknowledgement in the middle of a T4 or T1 clock cycle.

**ALE :**Address Latch Enable. ALE is provided by the microprocessor to latch the address into the 8282 or 8283 address latch. It is an active high(1) pulse during T1 of any bus cycle. ALE signal is never floated, is always integer.

**General purpose registers in 8086 microprocessor**

It is one of the most important chips ever created due to its part in the development of x86-based architecture. One significant aspect of this microprocessor is that it contains general registers. Efficiency and speed of computations in the processor are influenced by these registers since they determine arithmetic operations execution and data manipulations. Understanding such registers is important for code optimization as well as assembly language programming and system design.

The 8086 microprocessor contains a set of 16-bit general-purpose registers which are used for performing various arithmetic, logical, and data movement operations. Since these registers are flexible and can assume different combinations to perform various functions, they form the basic operation units of the processor itself.



**General-purpose registers are used to store temporary data within the microprocessor**

There are 8 general-purpose registers in the 8086 microprocessor.

**1. AX:** This is the accumulator. It is of 16 bits and is divided into two 8-bit registers AH and AL to also perform 8-bit instructions. It is generally used for arithmetical and logical instructions but in 8086 microprocessor it is not mandatory to have an accumulator as the destination operand. Example:

ADD AX, AX (AX = AX + AX)

**2. BX:**This is the base register. It is of 16 bits and is divided into two 8-bit registers BH and BL to also perform 8-bit instructions. It is used to store the value of the offset. Example:

MOV BL, [500] (BL = 500H)

**3. CX:** This is the counter register. It is of 16 bits and is divided into two 8-bit registers CH and CL to also perform 8-bit instructions. It is used in looping and rotation. Example:

MOV CX, 0005

LOOP

**4. DX:** This is the data register. It is of 16 bits and is divided into two 8-bit registers DH and DL to also perform 8-bit instructions. It is used in the multiplication and input/output port addressing. Example:

MUL BX (DX, AX = AX \* BX)

**5. SP:** This is the stack pointer. It is of 16 bits. It points to the topmost item of the stack. If the stack is empty the stack pointer will be (FFFE)H. Its offset address is relative to the stack segment.

It is AB and manage the present position of the top of the stack.

**Example**: To push or pop data from the stack in a function call, the SP is altered.

PUSH R1 ; Decrement the SP and then store the value of R1 that is at SP.

R1 = POP; R1 = increment SP; R1 = get value in SP

**6. BP –** This is the base pointer. It is of 16 bits. It is primarily used in accessing parameters passed by the stack. Its offset address is relative to the stack segment.

MOV AX, [BP+4] ; Access a parameter passed to the function

(4 bytes above BP in the Stack)

**7. SI –** This is the source index register. It is of 16 bits. It is used in the pointer addressing of data and as a source in some string-related operations. Its offset is relative to the data segment.

MOV AL, [SI] ; Move the byte at the address pointed to by SI into the AL register

(used as a pointer to source data).

**8.DI –** This is the destination index register. It is of 16 bits. It is used in the pointer addressing of data and as a destination in some string-related operations. Its offset is relative to the extra segment.

MOV [DI], AL; Move the byte in AL to the address pointed to by DI

(serves as the destination for the data).

**Flag register of 8086 microprocessor**

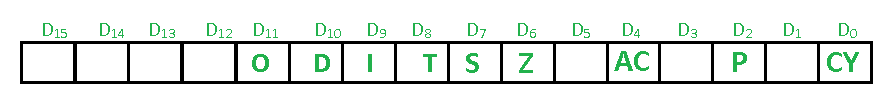
Prerequisite – [Flag register in 8085 microprocessor](https://www.geeksforgeeks.org/flag-register-8085-microprocessor/) The Flag register is a Special

**Introduction :**

The flag register is a 16-bit register in the Intel 8086 microprocessor that contains information about the state of the processor after executing an instruction. It is sometimes referred to as the status register because it contains various status flags that reflect the outcome of the last operation executed by the processor.

The flag register is an important component of the 8086 microprocessor because it is used to determine the behavior of many conditional jump and branch instructions. The various flags in the flag register are set or cleared based on the result of arithmetic, logic, and other instructions executed by the processor.

The flag register is divided into various bit fields, with each bit representing a specific flag. Some of the important flags in the flag register include the carry flag (CF), the zero flag (ZF), the sign flag (SF), the overflow flag (OF), the parity flag (PF), and the auxiliary carry flag (AF). These flags are used by the processor to determine the outcome of conditional jump instructions and other branching instructions.

Purpose Register. Depending upon the value of result after any arithmetic and logical operation the flag bits become set (1) or reset (0).**Figure –** Format of flag register There are total 9 flags in 8086 and the flag register is divided into two types: **(a) Status Flags –** There are 6 flag registers in 8086 microprocessor which become set(1) or reset(0) depending upon condition after either 8-bit or 16-bit operation. These flags are conditional/status flags. 5 of these flags are same as in case of 8085 microprocessor and their working is also same as in 8085 microprocessor. The sixth one is the overflow flag. The 6 status flags are:

1. **Sign Flag (S)**
2. **Zero Flag (Z)**
3. **Auxiliary Carry Flag (AC)**
4. **Parity Flag (P)**
5. **Carry Flag (CY)** These first [five flags are defined here](https://www.geeksforgeeks.org/flag-register-8085-microprocessor/)
6. **Overflow Flag (O) –** This flag will be set (1) if the result of a signed operation is too large to fit in the number of bits available to represent it, otherwise reset (0). After any operation, if D[6] generates any carry and passes to D[7] OR if D[6] does not generates carry but D[7] generates, overflow flag becomes set, i.e., 1. If D[6] and D[7] both generate carry or both do not generate any carry, then overflow flag becomes reset, i.e., 0. **Example:** On adding bytes 100 + 50 (result is not in range -128…127), so overflow flag will set.

MOV AL, 50 (50 is 01010000 which is positive)

MOV BL, 32 (32 is 00110010 which is positive)

ADD AL, BL (82 is 10000010 which is negative)

1. Overflow flag became set as we added 2 +ve numbers and we got a -ve number.

**(b) Control Flags –** The control flags enable or disable certain operations of the microprocessor. There are 3 control flags in 8086 microprocessor and these are:

1. **Directional Flag (D) –** This flag is specifically used in string instructions. If directional flag is set (1), then access the string data from higher memory location towards lower memory location. If directional flag is reset (0), then access the string data from lower memory location towards higher memory location.
2. **Interrupt Flag (I) –** This flag is for interrupts. If interrupt flag is set (1), the microprocessor will recognize interrupt requests from the peripherals. If interrupt flag is reset (0), the microprocessor will not recognize any interrupt requests and will ignore them.
3. **Trap Flag (T) –** This flag is used for on-chip debugging. Setting trap flag puts the microprocessor into single step mode for debugging. In single stepping, the microprocessor executes a instruction and enters into single step ISR. If trap flag is set (1), the CPU automatically generates an internal interrupt after each instruction, allowing a program to be inspected as it executes instruction by instruction. If trap flag is reset (0), no function is performed.

**Uses of Flag register in 8086 microprocessor :**

The flag register in the 8086 microprocessor has several important uses, including:

1. Conditional branching: The flags in the flag register can be used to control conditional branching in assembly language programming. Conditional jump instructions allow a program to take different paths based on the state of the flags in the flag register.
2. Arithmetic and logic operations: The flag register is used to store the results of arithmetic and logic operations. The flags in the flag register provide information about the outcome of these operations, such as whether a result is negative or zero, or whether there was an overflow or carry.
3. Error detection and handling: The flag register can be used to detect errors and exceptions, such as overflow or divide-by-zero errors. This allows programs to handle these errors gracefully and to take appropriate corrective action.
4. Debugging: The flag register provides a convenient way to access important information about the status of the processor after executing an instruction. This information can be used to debug programs and to optimize performance.
5. Optimization: The flag register can be used to optimize the performance of assembly language programs by avoiding unnecessary instructions or reducing the number of conditional jumps required.

**Addressing modes in 8086 micro processor**

**Addressing modes** are important in assembly language programming as they define how data is located and accessed by instructions. These modes describe how an instruction specifies its operands, whether they are immediate values, memory addresses, or registers. They play important role in processor to fetch or store data for an operation, allowing us to manage read/write operations during program execution. Effective use of addressing modes makes assembly language code more efficient and flexible, directly impacting performance.

**What is Addressing Mode?** [**Addressing modes**](https://www.geeksforgeeks.org/addressing-modes/)specify how an instruction identifies the data, or operands. It operates on in [assembly language](https://www.geeksforgeeks.org/what-is-assembly-language/). This field indicates whether the operand is a direct value, a memory address, or stored in a register. Addressing modes are important during instruction execution, as they define the form of an operand and the way data is accessed, making sure proper data handling at the machine level.

**Types of Addressing Modes**

**Register Mode**: In this type of addressing mode both the operands are registers.

**Example:**

MOV AX, BX  
XOR AX, DX  
ADD AL, BL

**Immediate Mode**: In this type of addressing mode the source operand is a 8 bit or 16 bit data. Destination operand can never be immediate data.

**Example:**

MOV AX, 2000  
MOV CL, 0A  
ADD AL, 45  
AND AX, 0000

**Displacement or Direct Mode:** In this type of addressing mode the effective address is directly given in the instruction as displacement.

**Example:**

MOV AX, [DISP]  
MOV AX, [0500]

**Register Indirect Mode:**In this addressing mode the effective address is in SI, DI or BX.

**Example:** Physical Address = Segment Address + Effective Address

MOV AX, [DI]  
ADD AL, [BX]  
MOV AX, [SI]

**Based Indexed Mode:** In this the effective address is sum of base register and index register.

Base register: BX, BP  
Index register: SI, DI

**Indexed Mode:**In this type of addressing mode the effective address is sum of index register and displacement.

**Example:**

MOV AX, [SI+2000]  
MOV AL, [DI+3000]

**Based Mode:**In this the effective address is the sum of base register and displacement.

**Example:**

MOV AL, [BP+ 0100]

**Based Indexed Displacement Mode:**In this type of addressing mode the effective address is the sum of index register, base register and displacement.

**Example:**

MOV AL, [SI+BP+2000]

**String Mode:** This addressing mode is related to string instructions. In this the value of SI and DI are auto incremented and decremented depending upon the value of directional flag.

**Example:**

MOVS B  
MOVS W

**Input/Output Mode:** This addressing mode is related with input output operations.

**Example:**

IN A, 45  
OUT A, 50

**Relative Mode:**In this the effective address is calculated with reference to instruction pointer.

**Example:**

JNZ 8 bit address  
IP=IP+8 bit address

**Advantages of Addressing Modes**

* By using different addressing modes, you can access data in registers or memory and as immediate values which makes CPU instructions more versatile.
* The register-based and indexed mode make better use of memory, especially in the case of loop operations / arrays.
* For example, modes such as register addressing will make the instructions run faster because they read data directly from [registers](https://www.geeksforgeeks.org/general-purpose-registers/) rather than memory.

**Disadvantages of Addressing Modes**

* Indirect or indexed addressing modes: here the program logic becomes harder to see and maintain as not all instructions are using them.
* Like displacement or relative addressing modes, that could add possibly memory overhead which is near in the event of large value displacements.
* Other addressing modes are more closely linked to the CPU architecture, making it less easy for assembly code to be ported across different hardware platforms.

**Arithmetic instructions in 8086 micro processor**

Arithmetic Instructions are the instructions which perform basic arithmetic operations such as addition, subtraction and a few more. Unlike in 8085 microprocessor, in 8086 microprocessor the destination operand need not be the accumulator.

Following is the table showing the list of arithmetic instructions:

| **OPCODE** | **OPERAND** | **EXPLANATION** | **EXAMPLE** |
| --- | --- | --- | --- |
| ADD | D, S | D = D + S | ADD AX, [2050] |
| ADC | D, S | D = D + S + prev. carry | ADC AX, BX |
| SUB | D, S | D = D – S | SUB AX, [SI] |
| SBB | D, S | D = D – S – prev. carry | SBB [2050], 0050 |
| MUL | 8-bit register | AX = AL \* 8-bit reg. | MUL BH |
| MUL | 16-bit register | DX AX = AX \* 16-bit reg. | MUL CX |
| IMUL | 8 or 16 bit register | performs signed multiplication | IMUL CX |
| DIV | 8-bit register | AX = AX / 8-bit reg. ; AL = quotient ; AH = remainder | DIV BL |
| DIV | 16-bit register | DX AX / 16-bit reg. ; AX = quotient ; DX = remainder | DIV CX |
| IDIV | 8 or 16 bit register | performs signed division | IDIV BL |
| INC | D | D = D + 1 | INC AX |
| DEC | D | D = D – 1 | DEC [2050] |
| CBW | none | converts signed byte to word | CBW |
| CWD | none | converts signed byte to double word | CWD |
| NEG | D | D = 2’s compliment of D | NEG AL |
| DAA | none | decimal adjust accumulator | DAA |
| DAS | none | decimal adjust accumulator after subtraction | DAS |
| AAA | none | ASCII adjust accumulator after addition | AAA |
| AAS | none | ASCII adjust accumulator after subtraction | AAS |
| AAM | none | ASCII adjust accumulator after multiplication | AAM |
| AAD | none | ASCII adjust accumulator after division | AAD |

Here D stands for destination and S stands for source.  
D and S can either be register, data or memory address.

**Logical instructions in 8086 microprocessor**

**Introduction :**

Logical instructions in the 8086 microprocessor are instructions that perform logical operations on data stored in registers or memory locations. These instructions can manipulate bits within a byte, set or clear individual bits, or perform Boolean operations such as AND, OR, XOR, and NOT.

Some of the commonly used logical instructions in the 8086 microprocessor include:

1. AND – Performs a bitwise logical AND operation between two operands and stores the result in the destination operand.
2. OR – Performs a bitwise logical OR operation between two operands and stores the result in the destination operand.
3. XOR – Performs a bitwise logical XOR (exclusive OR) operation between two operands and stores the result in the destination operand.
4. NOT – Performs a bitwise logical NOT (negation) operation on the operand and stores the result in the destination operand.
5. TEST – Performs a bitwise logical AND operation between two operands, but does not store the result. Instead, it sets the condition code flags based on the result of the operation.

Logical instructions are used in many applications, including bit manipulation, data encryption, and data compression. They are an important part of the instruction set of the 8086 microprocessor and are used extensively in assembly language programming.

Logical instructions are the instructions which perform basic logical operations such as AND, OR, etc. In 8086 microprocessor, the destination operand need not be the accumulator. Following is the table showing the list of logical instructions:

| **OPCODE** | **OPERAND** | **DESTINATION** | **EXAMPLE** |
| --- | --- | --- | --- |
| AND | D, S | D = D AND S | AND AX, 0010 |
| OR | D, S | D = D OR S | OR AX, BX |
| NOT | D | D = NOT of D | NOT AL |
| XOR | D, S | D = D XOR S | XOR AL, BL |
| TEST | D, S | performs bit-wise AND operation and affects the flag register | TEST [0250], 06 |
| SHR | D, C | shifts each bit in D to the right C times and 0 is stored at MSB position | SHR AL, 04 |
| SHL | D, C | shifts each bit in D to the left C times and 0 is stored at LSB position | SHL AX, BL |
| ROR | D, C | rotates all bits in D to the right C times | ROR BL, CL |
| ROL | R, C | rotates all bits in D to the left C times | ROL BX, 06 |
| RCR | D, C | rotates all bits in D to the right along with carry flag C times | RCR BL, CL |
| RCL | R, C | rotates all bits in D to the left along with carry flag C times | RCL BX, 06 |

Here D stands for destination, S stands for source and C stands for count. They can either be register, data or memory address.

**Why use Logical instructions in 8086 microprocessor ?**

Here are some reasons why logical instructions are used in the 8086 microprocessor:

1. Bit manipulation: Logical instructions allow the programmer to manipulate individual bits within a byte, which is useful in many applications, including device control, signal processing, and graphics programming.
2. Data encryption: Logical instructions are used in data encryption algorithms to scramble data by performing bitwise operations on the data.
3. Data compression: Logical instructions are used in data compression algorithms to reduce the size of the data by removing redundant or irrelevant bits.
4. Boolean operations: Logical instructions are used to perform Boolean operations such as AND, OR, XOR, and NOT, which are used in many applications, including control systems, signal processing, and mathematical operations.
5. Condition code flags: Logical instructions can set the condition code flags in the flags register, which can be used to control program flow or make decisions based on the result of the logical operation.

**Advantages of Logical instructions in 8086 microprocessor :**

1. Efficient data manipulation
2. Boolean operations
3. Condition code flags
4. Data encryption and compression
5. Faster execution

**Dis-advantages of Logical instructions in 8086 microprocessor :**

1. Limited precision
2. Limited functionality
3. Register limitations
4. Complex programming
5. Debugging challenges

**Data transfer instructions in 8086 microprocessor**

**Introduction :**

Data transfer instructions in the 8086 microprocessor are used to move data between memory locations, registers, and input/output (I/O) devices. These instructions are essential for manipulating data within a program, as well as for communicating with external devices.

Data transfer instructions are a fundamental part of programming in the 8086 microprocessor, and are used extensively in applications ranging from simple data manipulation to complex I/O device communication and string processing.

Data transfer instructions are the instructions which transfers data in the microprocessor. They are also called copy instructions.

**Types of Data transfer instructions :**

**1. Move instructions:**

These instructions are used to move data from one memory location to another or between a memory location and a register. They include the following instructions:

* MOV: Moves data from a source operand to a destination operand.
* XCHG: Swaps the contents of two operands.
* XLAT: Translates a byte in memory using a lookup table pointed to by the contents of the AL register.
* LEA: Loads a 16-bit offset address into a register.

**2. Load instructions:**

These instructions are used to load data from a memory location or I/O device into a register. They include the following instructions:

* LDS: Loads a 16-bit pointer value from a memory location into a register pair and loads the 8-bit value from the next memory location into another register.
* LSS: Loads a 16-bit pointer value from a memory location into a register pair and loads the 16-bit value from the next memory location into another register.
* LXI: Loads a 16-bit value into a register pair.
* MOV with memory operand: Loads data from a memory location into a register.

**3. Store instructions:**

These instructions are used to store data from a register into a memory location or I/O device. They include the following instructions:

* MOV with memory operand: Stores data from a register into a memory location.
* STA: Stores the contents of the accumulator register (AL or AX) in memory.
* STAX: Stores the contents of a register pair (BC, DE, or HL) in memory using either the indirect addressing
* mode or the direct addressing mode.
* SHLD: Stores a 16-bit data word from registers H and L in memory using the direct addressing mode.
* PUSH: Stores the contents of a register onto the stack.

**4. Input/Output instructions:**

These instructions are used to communicate with external input/output (I/O) devices. They include the following instructions:

* IN: Reads a byte or word of data from an I/O port into a register.
* OUT: Writes a byte or word of data from a register to an I/O port.
* INS: Reads a block of data from an I/O port into a memory location.
* OUTS: Writes a block of data from a memory location to an I/O port.

**5. String instructions:**

 These instructions are used for manipulating strings of data, such as moving, copying, or comparing strings. They operate on consecutive bytes or words in memory, and can be used for fast and efficient string processing. Some examples of string instructions include:

* MOVS: Moves a byte or word from a source location to a destination location, and updates the index registers to point to the next byte or word.
* CMPS: Compares a byte or word in memory to a byte or word in a register, and updates the index registers accordingly.
* LODS: Loads a byte or word from a memory location into a register, and updates the index registers to point to the next byte or word.
* STOS: Stores a byte or word from a register into a memory location, and updates the index registers to point to the next byte or word.

Following is the table showing the list of data transfer instructions:

| **OPCODE** | **OPERAND** | **EXPLANATION** | **EXAMPLE** |
| --- | --- | --- | --- |
| MOV | D, S | D = S | MOV AX, [SI] |
| PUSH | D | pushes D to the stack | PUSH DX |
| POP | D | pops the stack to D | POP AS |
| PUSHA | none | put all the registers into the stack | PUSHA |
| POPA | none | gets words from the stack to all registers | POPA |
| XCHG | D, S | exchanges contents of D and S | XCHG [2050], AX |
| IN | D, S | copies a byte or word from S to D | IN AX, DX |
| OUT | D, S | copies a byte or word from D to S | OUT 05, AL |
| XLAT | none | translates a byte in AL using a table in the memory | XLAT |
| LAHF | none | loads AH with the lower byte of the flag register | LAHF |
| SAHF | none | stores AH register to lower byte of the flag register | SAHF |
| PUSHF | none | copies the flag register at the top of the stack | PUSHF |
| POPF | none | copies a word at the top of the stack to the flag register | POPF |

Here D stands for destination and S stands for source.   
D and S can either be register, data or memory address.