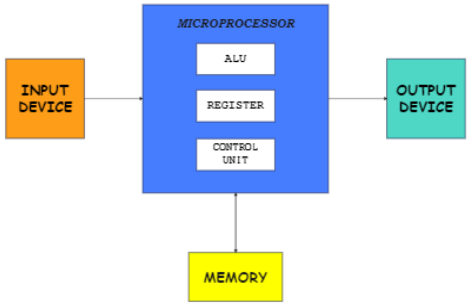
# Study Material on Microprocessors and 8086 Architecture (mid theory)

## 1. Introduction to Microprocessors

### Definition

A microprocessor is the central processing unit (CPU) of a computer, integrated onto a single integrated circuit (IC). It processes data by executing instructions stored in memory.




The 8086 microprocessor is a 16-bit processor developed by Intel. It follows the x86 architecture and supports segmented memory addressing, making it efficient in handling large programs.

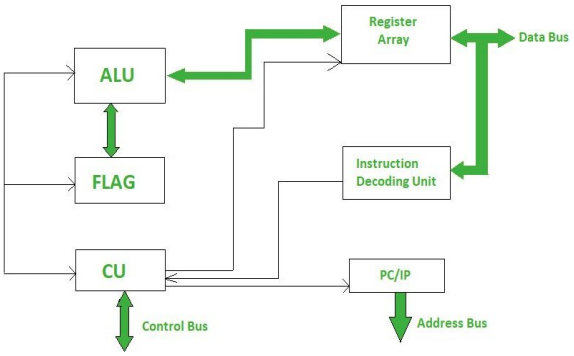
## Key Features:

- 16-bit processor
- 20-bit address bus (1MB memory addressing capability)
- 14 registers (general-purpose, segment, index, and flags)
- Supports pipelining through the **Bus Interface Unit (BIU)** and **Execution Unit (EU)**
- Supports both **minimum and maximum** mode operation

### Characteristics and How They Work

- **Program Counter (PC)**: A special register that stores the address of the next instruction to be executed. It increments automatically after each instruction fetch.
- **Internal Clock**: A timing device that synchronizes the execution of instructions. It determines the speed of operation.
- **Word Size**: The number of bits a microprocessor can process in one operation. A larger word size means greater processing power.
- **Instruction Set**: The set of machine-level commands a microprocessor can execute, defining its functionality and operations.

**Basic Components and How They Work**




1. **Arithmetic Logic Unit (ALU)** – Performs arithmetic (addition, subtraction) and logical (AND, OR, NOT) operations required for computations.
2. **Control Unit** – Directs the flow of data and instructions by decoding them and generating necessary control signals.
3. **Registers** – Small, fast storage locations inside the CPU that temporarily hold data and instructions for quick access.
4. **Memory** – Stores data and instructions, including RAM (temporary storage) and ROM (permanent storage for boot-up instructions).
5. **Input/Output (I/O) Devices** – Interfaces that allow data to be transferred between the CPU and external devices like keyboards and monitors.

**Functional Parts of CPU and How They Work**

- **Instruction Register** – Holds the instruction currently being executed, fetched from memory.
- **Decoder** – Converts instructions into machine-level commands understood by the CPU.
- **ALU** – Executes arithmetic and logical operations based on control signals.
- **Registers** – Store intermediate data and results to speed up processing.

**Types of Buses and How They Work**

- **Data Bus** – Transfers actual data between the CPU, memory, and peripherals (bidirectional).
- **Address Bus** – Carries the memory addresses for data transfer (unidirectional from CPU to memory).
- **Control Bus** – Transmits control signals that coordinate different parts of the computer (bidirectional).

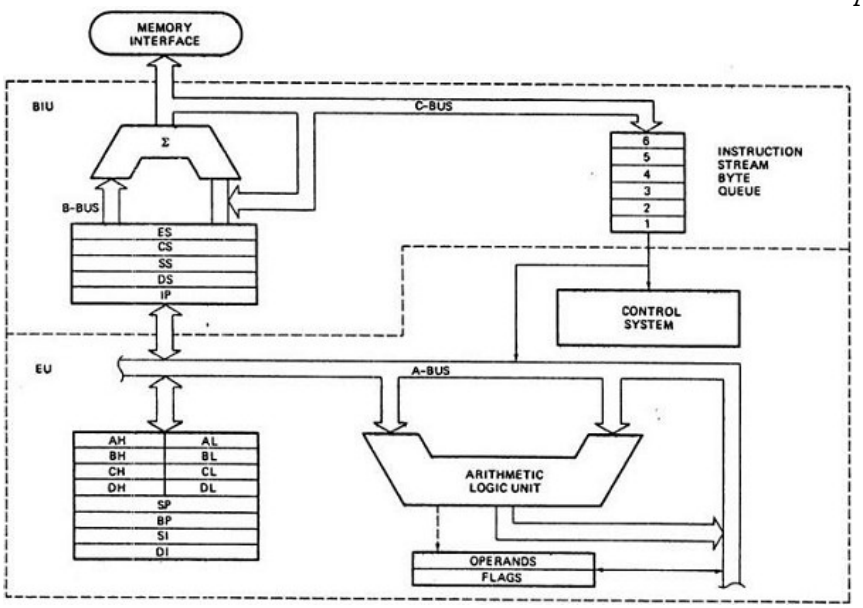
## 2. Intel 8086 Microprocessor

**Features**

- **16-bit Processor**: The CPU processes 16 bits of data at a time.

- **Data Bus (16-bit) and Address Bus (20-bit)**: Can access 1MB of memory by addressing 2^20 locations.
- **Multiplexed Bus**: Address and data share the same lines, requiring demultiplexing.
- **Segmented Memory Architecture**: Divides memory into segments (Code, Data, Stack, Extra) for efficient access.
- **Prefetching Mechanism**: Fetches instructions in advance to speed up execution.

**Architecture**




## Bus Interface Unit (BIU) vs. Execution Unit (EU) in 8086 Microprocessor

The **8086 microprocessor** is designed with a **pipelined architecture**, dividing its operation into two main functional units:

1. **Bus Interface Unit (BIU)** – Responsible for memory and I/O communication.
2. **Execution Unit (EU)** – Handles instruction decoding and execution.

### 1. Bus Interface Unit (BIU)

The **BIU** manages interactions between the CPU and memory or I/O devices. It works independently from the execution process, allowing simultaneous instruction fetching and execution (pipelining).

#### Key Responsibilities of BIU:

- **Instruction Fetching:** Pre-fetches instructions from memory and stores them in a **6-byte instruction queue** to speed up execution.
- **Memory Address Calculation:** Computes **physical memory addresses** using **segment registers (CS, DS, SS, ES)** and the **Instruction Pointer (IP)**.
- **Bus Control & Data Transfer:** Controls the **data bus, address bus, and control bus** for memory read/write and I/O operations.
- **Instruction Queue Management:** Reduces CPU waiting time by keeping a queue of prefetched instructions.
- **Generates Control Signals:** Controls memory and peripheral access using signals like **RD (Read), WR (Write), and INTA (Interrupt Acknowledge)**.

**Components of BIU:**

- **Segment Registers (CS, DS, SS, ES):** Helps in memory segmentation.
- **Instruction Pointer (IP):** Holds the address of the next instruction to be fetched.
- **Instruction Queue:** Temporary storage for prefetched instructions (6 bytes).
- **Bus Control Logic:** Manages communication with memory and I/O devices.

---

### 2. Execution Unit (EU)

The **EU** is responsible for decoding and executing instructions that have been fetched by the BIU. It processes data using various registers and the **Arithmetic Logic Unit (ALU)**.

**Key Responsibilities of EU:**

- **Instruction Decoding:** Converts machine code into control signals for execution.
- **Arithmetic & Logic Operations:** Performs calculations like addition, subtraction, AND, OR, XOR, etc., using the **ALU**.
- **Register Operations:** Uses general-purpose registers (**AX, BX, CX, DX**) to store operands and results.
- **Flag Register Management:** Updates the **flag register** based on operation results (e.g., Zero Flag, Carry Flag).
- **Memory & Stack Management:** Uses **SP (Stack Pointer)** and **BP (Base Pointer)** for stack operations.

**Components of EU:**

- **Arithmetic Logic Unit (ALU):** Performs arithmetic and logical operations.
- **Instruction Decoder:** Interprets opcode and determines the required operation.
- **General-Purpose Registers (AX, BX, CX, DX):** Store data for computations.
- **Pointer & Index Registers (SP, BP, SI, DI):** Assist in memory addressing and stack management.
- **Flag Register:** Stores the status of the last executed instruction.

---

### 3. Interaction Between BIU and EU (Pipelining Effect)

- The **BIU fetches instructions** while the **EU decodes and executes them**.
- The **instruction queue (6 bytes)** helps reduce wait time for instruction fetching.
- If the **instruction queue is full**, the BIU pauses fetching until the EU executes some instructions.
- If the **instruction queue is empty**, the EU waits for new instructions, slowing execution.
- This **overlapping execution and fetching (pipelining)** increases processing efficiency.

**[In short]**

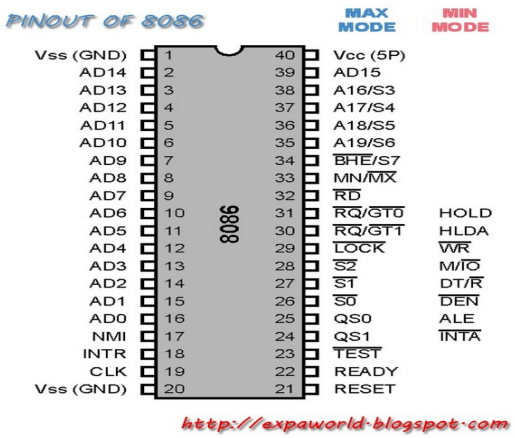
1. **Bus Interface Unit (BIU) and How It Works**

    - Manages data transfer between memory and CPU.
    - Prefetches instructions and stores them in an instruction queue.
    - Computes the physical address by combining segment and offset addresses.
    - Generates bus control signals to coordinate data flow.

- Contains:
    - **Instruction queue**: Temporarily holds prefetched instructions to speed up execution.
    - **Segment registers**: Stores memory segment addresses.
    - **Instruction pointer (IP)**: Holds the address of the next instruction.

2. **Execution Unit (EU) and How It Works**

- Decodes and executes instructions fetched by the BIU.
- Performs arithmetic and logic operations via the ALU.
- Updates flag registers based on the operation results.
- Contains:
    - **Control circuitry**: Directs execution steps.
    - **Instruction decoder**: Interprets opcode and determines the required operation.
    - **ALU**: Processes numerical and logical computations.
    - **General-purpose registers**: Temporarily store operands and results.
    - **Pointer and Index registers**: Facilitate memory addressing for data manipulation.
    - **Flag register**: Indicates the status of various operations.




Here's a table analyzing the **8086 microprocessor pinout** based on the provided image.

| Pin No. | Label | Description |
|---|---|---|
| 1 | Vss (GND) | Ground (0V reference) |
| 2-16 | AD14-AD0 | Multiplexed Address/Data bus |
| 17 | NMI | Non-Maskable Interrupt (High-priority interrupt) |
| 18 | INTR | Interrupt Request |
| 19 | CLK | System Clock Input |
| 20 | Vss (GND) | Ground (0V reference) |
| 21 | RESET | Resets the processor |
| 22 | READY | Wait-state signal for slow memory/peripherals |
| 23 | TEST | Used for debugging, halts CPU when low |
| 24-25 | QS0, QS1 | Queue status signals |
| 26-28 | S0, S1, S2 | Status signals for memory and I/O operation |
| 29 | LOCK | Bus lock signal to prevent access interference |

| Pin No. | Label | Description |
|---|---|---|
| 30-31 | RQ/GT0, RQ/GT1 | Request/Grant for bus control |
| 32 | RD | Read control signal for memory and I/O |
| 33 | MN/MX | Minimum/Maximum mode selection |
| 34 | BHE/S7 | Bus High Enable / Status signal |
| 35-38 | A19/S6 - A16/S3 | Address and Status signals |
| 39 | AD15 | Multiplexed Address/Data bus |
| 40 | Vcc (5V) | Power supply (+5V) |

## Additional Notes

- **Multiplexed Address/Data Bus (AD0-AD15):** These lines function as both address and data lines at different times.
- **Status & Control Signals (S0, S1, S2, QS0, QS1):** Used to indicate queue status, memory access, and bus operations.
- **Interrupts (NMI, INTR):** NMI is a high-priority interrupt that cannot be ignored, while INTR is a general interrupt request.
- **Mode Selection (MN/MX):** Determines if the CPU operates in **Minimum Mode (Single CPU)** or **Maximum Mode (Multiprocessor system).**
- **Memory Control Signals (RD, WR, M/IO, DT/R, DEN, ALE, INTA):** Control reading, writing, and bus interfacing.

## Registers in 8086 Microprocessor

The **8086 microprocessor** has a **16-bit register architecture**, meaning all registers are **16-bit (2 bytes) wide**. These registers are categorized into:

1. **General-Purpose Registers**
2. **Pointer and Index Registers**
3. **Segment Registers**
4. **Flag Register (Status Register)**

---

## 1. Detailed Explanation of General Purpose Registers in 8086

8086 has four **general-purpose registers (AX, BX, CX, DX)**, each **16-bit** in size. These can be **split into two 8-bit parts**:

- **Higher byte (H)** → Upper 8 bits
- **Lower byte (L)** → Lower 8 bits

Each register serves specific purposes in arithmetic, logic, and memory operations.

---

## General Purpose Registers and Their Functions

| Register | Size | Function |
|---|---|---|
| **AX** (Accumulator) | 16-bit | Used for **word multiplication, division, word I/O operations**, and arithmetic calculations. |
| **AL** (Lower byte of AX) | 8-bit | Used for **byte multiplication, division, byte I/O, and decimal arithmetic**. |
| **AH** (Higher byte of AX) | 8-bit | Stores **byte data** and used in **byte multiplication & division**. |
| **BX** (Base Register) | 16-bit | Stores **data and address information** for memory addressing. |
| **CX** (Count Register) | 16-bit | Used for **loop control, string operations, repeated shift & rotate operations**. |
| **CL** (Lower byte of CX) | 8-bit | Used for **variable shift and rotate operations** at the byte level. |
| **DX** (Data Register) | 16-bit | Used for **word multiplication, word division, and indirect I/O operations**. |
| **DX as an I/O register** | - | Holds **I/O addresses during I/O instructions**. In multiplication or division, **DX stores the higher order 16-bits of the result** while **AX stores the lower 16-bits**. |

---

## Key Takeaways

- **AX is the most used register** (for arithmetic, I/O, and multiplication).
- **BX is useful for indirect memory addressing** (base pointer).
- **CX is mainly for loops & shift operations**.
- **DX plays a special role in extended multiplication, division, and I/O operations**.

## 2. Pointer and Index Registers (16-bit)

These registers are used for **stack operations, memory indexing, and string processing**.

| Register | Size | Function | How It Works |
|---|---|---|---|
| **SP** (Stack Pointer) | 16-bit | Stack Top Pointer | Keeps track of the **top of the stack** in memory. Points to the last pushed value. |
| **BP** (Base Pointer) | 16-bit | Stack Data Access | Helps in accessing data stored in the **stack segment (SS)** via offset values. |
| **SI** (Source Index) | 16-bit | String/Array Operations | Used as a source index for **string and memory operations**. Works with **DI** for block transfers. |
| **DI** (Destination Index) | 16-bit | String/Array Operations | Used as a destination index in **string manipulation and memory transfers**. Works with **SI** for block transfers. |

---

## 3. Segment Registers (16-bit)

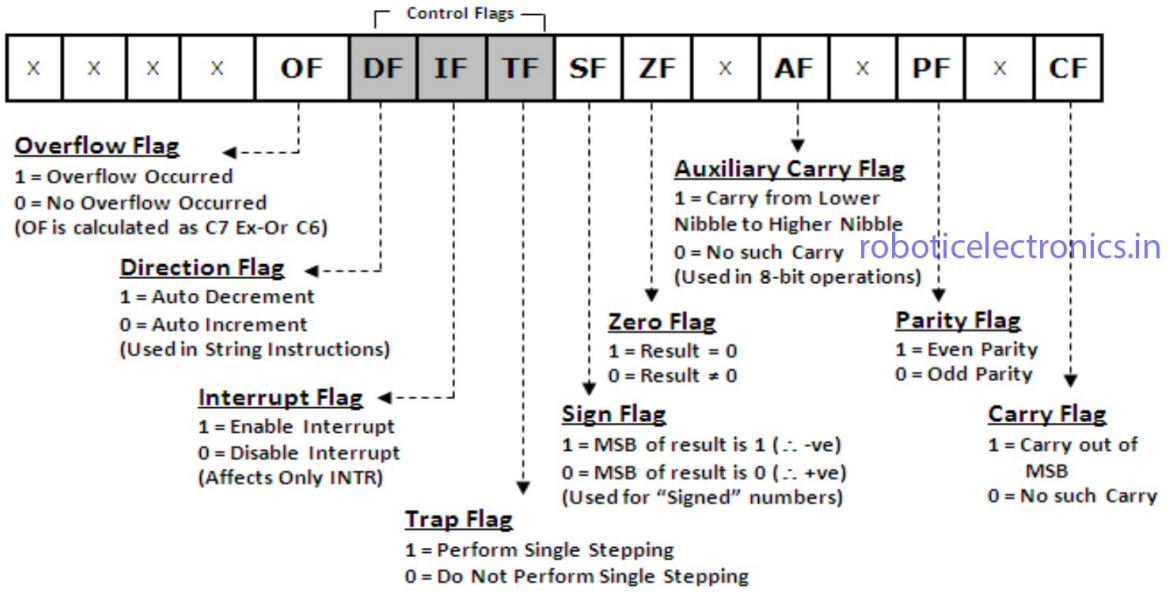
Segment registers store **base addresses** of different memory segments in the **segmented memory architecture** of the 8086 processor.

The **8086 microprocessor** follows a **segmented memory architecture**, meaning memory is divided into segments, each **64 KB** in size. The processor uses **four segment registers** to access these memory segments efficiently.

| Register | Size | Function | How It Works |
|---|---|---|---|
| **CS** (Code Segment) | 16-bit | Holds Executable Code | Stores the **memory address of the currently executing instructions**. The **IP (Instruction Pointer)** works with CS to fetch instructions. |
| **DS** (Data Segment) | 16-bit | Holds Program Data | Holds variables and constants used in the program. Combined with **SI/DI for indexed addressing**. |
| **SS** (Stack Segment) | 16-bit | Stores Stack Data | Defines the memory area used for the **stack** (temporary storage for function calls, local variables). Works with **SP and BP**. |
| **ES** (Extra Segment) | 16-bit | Additional Data Storage | Used for **additional memory operations**, often in **string processing and memory transfers**. |

---

## 4. Flag Register (16-bit) (Status Register)




## Explanation of Flags in the 8086 Microprocessor [in short]:

### 1. Status Flags (ALU Operation Results)

These flags reflect the outcome of arithmetic and logical operations.

| Flag | Name | Function |
|---|---|---|
| **CF** | Carry Flag | Set when an arithmetic operation generates a carry or borrow out of the most significant bit (MSB). Used in multi-byte arithmetic operations. |
| **PF** | Parity Flag | Set if the result of an operation has an even number of 1s (used for error detection in communication). |
| **AF** | Auxiliary Carry Flag | Set when there is a carry from the lower nibble (4 bits). Important for **BCD (Binary-Coded Decimal) arithmetic**. |
| **ZF** | Zero Flag | Set if the result of an operation is **zero**. Used for conditional branching (e.g., JZ – Jump if Zero). |
| **SF** | Sign Flag | Reflects the **MSB (Most Significant Bit)** of the result. If SF = 1, the result is **negative**; if SF = 0, the result is **positive**. |
| **OF** | Overflow Flag | Set if an arithmetic operation results in a value too large for the register (i.e., sign bit changes unexpectedly). |

---

**2. Control Flags (Processor Control)**

These flags control CPU operations and behaviors.

| Flag | Name | Function |
|---|---|---|
| **DF** | Direction Flag | Controls **string operations** (e.g., MOVS, CMPS). If DF = 0, operations process **forward** (auto-increment). If DF = 1, operations process **backward** (auto-decrement). |
| **IF** | Interrupt Enable Flag | When set (IF = 1), **CPU responds to maskable interrupts** (external hardware requests). When cleared (IF = 0), interrupts are ignored. |
| **TF** | Trap Flag | Enables **single-step debugging** mode. When TF = 1, the CPU executes **one instruction at a time** and generates an interrupt for debugging. |

---

## Summary

- The **status flags** indicate the outcome of arithmetic/logic operations.
- The **control flags** modify CPU behavior (interrupts, string processing, debugging).
- Some bits in the flag register are **unused**.

This flag system helps in **decision-making, branching, and debugging** during program execution.

The **DETAILED "flag register"** is **16-bit wide**, but only **9 flags** are used to indicate the **status of ALU operations**.

- **Control Flags:** Direct CPU operations (e.g., Interrupt Enable).
- **Status Flags:** Reflect the outcome of arithmetic/logical operations (e.g., Zero Flag, Carry Flag).

- 3 are control flags ( accessible to programmers ).
- 6 are status flags ( not accessible to programmers ).

**Status flags:**

**CF:**

It stands for **carry flag.**

Holds the carry after addition or the borrow after subtraction. Also indicates some error conditions, as dictated by some programs and procedures .

If CF = 1 ; it means carry is generated from the MSB.

If CF = 0 ; no carry is generated out of MSB.

```
CF = 1                  CF = 0
  1 1                     1 1
  1011 1100               1011 1100
+ 1001 0001             + 0001 0001
-----------             -----------
1 0100 1101               1100 1101
```

**PF:**

It stands for **parity flag.**

THE NUMBER OF ONE'S COUNT IN RESULT

If PF = 1 ; it means it is even parity in the result ( there are even numbers of 1's ).

If PF = 0 ; it means it is odd parity.

```
PF = 1                  PF = 0
  1 1                     1 1
  1011 1100               1011 1100
+ 1001 0001             + 0001 0001
-----------             -----------
1 0100 1101               1100 1101
```



In example 1, there are 4 ones' which is even , in the second example there are 5 ones' which is odd in count.

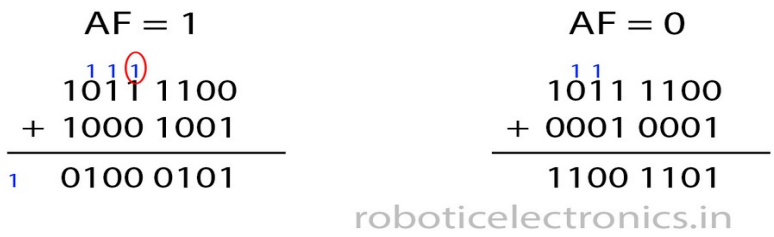
**AF:**

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

**Holds the carry (half – carry) after addition or borrow after subtraction between bit positions 3 and 4 of the result (for example, in BCD addition or subtraction.)**

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

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




**ZF:**

This is **zero flag**. Whenever the output is **0** this flag is 1.

Shows the result of the arithmetic or logic operation. ZF=1; result is zero. ZF=0; The result is 0

If ZF = 1 ; output is zero.

If ZF = 0 ; output is non zero.

**OF:**

OF stands for **overflow flag.**

Overflow occurs when signed numbers are added or subtracted. An overflow indicates the result has exceeded the capacity of the Machine

The are two types of numbers

- Signed
- Unsigned

**Unsigned:**

This is a 8 bit positive number which ranges from 0 to 255. In hexadecimal it's range is from 00 to FF. In the OF flag, it has nothing to do with unsigned numbers. Only signed numbers are considered in the OF flag.

**Signed:**

This is also a 8-bit number( can be 16-bit too ) which is equally distributed among +ve and -ve numbers. By considering MSB , it is decided whether it is a positive or negative number. If MSB=1 , it is a negative number or else negative number.

Eg: 1011 0011 is -ve

0111 1001 is +ve

So it's positive range is from 0 to 127 [ 00 to 7F (in hexadecimal ) ], it consists of 128 +ve values.

It also has 128 negative numbers ranging from -1 to -128 [ -01 to -80 ( in hexadecimal ) ].

Whenever a negative number is saved, it is saved as it's 2's complement. To access the number we have to make its 2's complement again. A number is considered as negative whenever its MSB is 1, and its 2's complement is the actual negative number.

For eg: 1011 0011

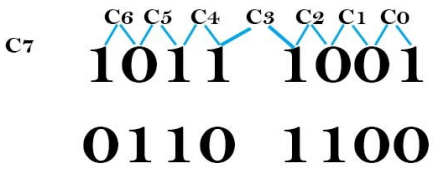
It is a negative number and its 2's complement is 0100 1101, which is equivalent

to 4D ( hexadecimal ) . so it is $-4D_H$.

So there is a well defined limit for signed numbers ( 00 to 7F and -01 to 80 ). If the result or operands exceeds the limit, it is known as overflow. If there is overflow, then the MSB would show incorrect values, if there is overflow then

MSB = 1 , then it is +ve number

MSB = 0 , then it is -ve number. This is the reverse with the original case.




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

**SF:**

This stands for **sign flag**. In short it copies the value of MSB. It shows wrong notation in the case of overflow.

**holds the sign of the result after an arithmetic/logic instruction execution. SF=1; negative, SF=0**

CF, AF, PF, ZF, OF, SF , these were status flags, and these keep changing after every arithmetic operation. And these flags are not controlled by the user, these are controlled by the ALU.

## Control flag:

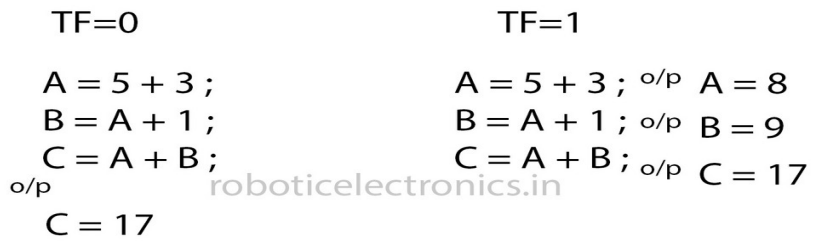
There are three types of control flags, and by **default all are zero**.

**TF:**

This stands for **trap flag**. Generally processors give output after the complete program, but when TF = 1, output is given after every instruction.

**Enables the trapping through an on-chip debugging feature.**

This is useful to check logical errors, when the program is too long.




**IF:**

This is **interrupt flag.**

**It selects either the increment or decrement mode for DI and or SI registers during the string instructions.These are programmable.**

IF = 1, then enable interrupts

IF = 0, then interrupts are disabled. By default interrupts are disabled.

**DF:**

This stands for **direction flag.** In the case string operation, by default address keeps incrementing for instruction execution. It means after execution of an instruction, whether the processor should execute next instruction or previous instruction.

**It selects either the increment or decrement mode for DI and or SI registers during the string instructions.These are programmable.**

If DF = 0 ; address is auto incrementing ( processor executes next instruction ).

If DF = 1 ; address is auto decrementing ( processor executes previous instruction ).

The image represents the **8086 Flag Register**, which is **16-bit wide**. However, not all bits are used; some are marked as **U (Unused)**. The flag register consists of **status flags** and **control flags**, which influence or indicate the outcome of CPU operations.

---

## The Queue (Q) in 8086 Microprocessor

The **Queue (Q)** in the 8086 microprocessor is an **instruction prefetch queue** used to improve execution speed through **pipelining**. It is part of the **Bus Interface Unit (BIU)** and plays a key role in fetching and storing instructions before execution.

---

## Key Features of the Queue in 8086

1. **Size:** The queue is **6 bytes** long in **8086**.
2. **Purpose:** It **preloads** instructions from memory, reducing waiting time for the **Execution Unit (EU)**.
3. **Working Mechanism:**
    - The **BIU fetches** instructions from memory and stores them in the queue.
    - The **EU executes** instructions from the queue one by one.

    - If the queue is **empty**, the EU waits for new instructions to be fetched.
4. **Advantage:** Improves performance by **overlapping instruction fetching and execution** (pipelining).
5. **Limitations:**
    - The queue is **flushed** when a **branch or jump** instruction is encountered because it changes the instruction flow.

---

## How It Works (Step-by-Step Execution)

1. **BIU fetches** instructions and fills the queue.
2. **EU takes** instructions from the queue and executes them.
3. If the queue becomes **empty**, BIU refills it.
4. If a **jump or branch** occurs, the queue is flushed, and fetching restarts from the new location.

---

The **Queue (Q) in 8086** optimizes instruction execution by ensuring a steady flow of instructions to the EU. It enables **faster execution** using pipelining but has to be flushed when jumps or branches occur.

## Basic Syntax of 8086 Assembly Language

8086 assembly language follows a structured format where each line of code consists of up to four parts:

```
[Label:]  Mnemonic  [Operands]  [;Comment]
```

- **Label** → (Optional) Used for jump locations.
- **Mnemonic** → Instruction (e.g., MOV, ADD, SUB).
- **Operands** → Data being processed (registers, memory, or constants).
- **Comment** → (Optional) Explanation of the instruction (`;` starts a comment).

---

## 1. Basic Structure of an 8086 Program

```
.MODEL SMALL     ; Memory model (SMALL, MEDIUM, LARGE)
.STACK 100H      ; Stack segment size
.DATA            ; Data segment
    message DB 'Hello, World!$'  ; String for output

.CODE            ; Code segment
MAIN PROC        ; Procedure start
    MOV AX, @DATA
    MOV DS, AX   ; Initialize data segment

    MOV DX, OFFSET message  ; Load message address
    MOV AH, 09H   ; DOS interrupt for string output
    INT 21H       ; Call DOS interrupt

    MOV AH, 4CH   ; DOS terminate program function
    INT 21H       ; Call DOS interrupt
MAIN ENDP        ; Procedure end
END MAIN         ; Program ends
```

## 2. Basic Assembly Instructions Syntax

### a. Data Transfer Instructions

```
MOV  DEST, SOURCE   ; Move data from source to destination
XCHG REG1, REG2     ; Swap two registers
```

Example:

```
MOV AL, 5    ; Load value 5 into AL register
MOV BX, AX   ; Copy AX value to BX
```

### b. Arithmetic Instructions

```
ADD DEST, SOURCE   ; DEST = DEST + SOURCE
SUB DEST, SOURCE   ; DEST = DEST - SOURCE
MUL SOURCE         ; Multiply AL/AX with SOURCE
DIV SOURCE         ; Divide AX by SOURCE
```

Example:

```
MOV AL, 10
ADD AL, 5    ; AL = 10 + 5
```

### c. Logical Instructions

```
AND DEST, SOURCE  ; Bitwise AND
OR DEST, SOURCE   ; Bitwise OR
XOR DEST, SOURCE  ; Bitwise XOR
NOT DEST          ; Bitwise NOT
```

Example:

```
MOV AL, 0Fh
AND AL, 07h   ; AL = AL AND 07h
```

### d. Control Flow Instructions

```
JMP LABEL    ; Unconditional jump
JE LABEL     ; Jump if equal (ZF=1)
JNE LABEL    ; Jump if not equal (ZF=0)
CALL PROC    ; Call subroutine
RET          ; Return from subroutine
```

Example:

```
CMP AL, 5
JE LABEL    ; If AL == 5, jump to LABEL
```

### e. Stack Instructions

```
PUSH REG    ; Push value onto stack
POP REG     ; Pop value from stack
```

Example:

```
PUSH AX
POP BX     ; Move AX value to BX via stack
```

---

## 3. Basic Assembly Program (Adding Two Numbers)

```
.MODEL SMALL
.STACK 100H
.DATA
    NUM1 DB 5
    NUM2 DB 3
    RESULT DB ?

.CODE
MAIN PROC
    MOV AX, @DATA
    MOV DS, AX  ; Initialize data segment

    MOV AL, NUM1  ; Load first number
    ADD AL, NUM2  ; Add second number
    MOV RESULT, AL  ; Store result

    MOV AH, 4CH   ; Exit program
    INT 21H
MAIN ENDP
END MAIN
```

---

## Key Takeaways

- **Assembly code is divided into segments** (`DATA`, `CODE`, `STACK`).
- **MOV is used for data transfer**, arithmetic/logical operations require specific instructions.
- **INT 21H is used for DOS interrupts (like printing output)**.
- **JMP and CMP help in decision-making**.

This syntax and structure apply to **MASM (Microsoft Assembler)** and **TASM (Turbo Assembler)** for 8086.

## Practice Section: Enhanced 8086 Assembly Tasks

### Explanation of Basic Assembly Tasks

- **Task 1 (Move Data Between Registers)**: This moves a value into AX and copies it to BX.
- **Task 2 (Addition of Two Numbers)**: Loads values into AL and BL, then adds them.
- **Task 3 (Loop for Counting)**: Uses CX as a loop counter and decrements it until zero.
- **Task 4 (Store Data in Memory)**: Stores AL's value into a memory address.
- **Task 5 (String Copy)**: Moves bytes from one memory location to another using SI and DI.

### Additional Practice Tasks

#### Task 6: Subtract Two Numbers

```
MOV AL, 10H    ; Load AL with 16
MOV BL, 05H    ; Load BL with 5
SUB AL, BL     ; Subtract BL from AL (AL = AL - BL)
```

**Task 7: Multiplication**

```
MOV AL, 4H      ; Load AL with 4
MOV BL, 3H      ; Load BL with 3
MUL BL          ; Multiply AL by BL (AX = AL * BL)
```

**Task 8: Division**

```
MOV AX, 08H     ; Load AX with 8
MOV BL, 2H      ; Load BL with 2
DIV BL          ; Divide AX by BL (AL = Quotient, AH = Remainder)
```

**Task 9: Logical AND Operation**

```
MOV AL, 0F0H    ; Load AL with F0H
MOV BL, 0FH     ; Load BL with 0FH
AND AL, BL      ; Perform AL AND BL (bitwise operation)
```

**Task 10: Logical OR Operation**

```
MOV AL, 0F0H    ; Load AL with F0H
MOV BL, 0FH     ; Load BL with 0FH
OR AL, BL       ; Perform AL OR BL (bitwise operation)
```

# Additional all the keywords of asm

**Yes! There are more keywords in 8086 assembly language, categorized under different functionalities. Here's a more comprehensive breakdown:**

---

## 1. Data Transfer Instructions

These instructions move data between registers, memory, or I/O ports.

- **MOV** → Move data from one location to another
- **PUSH** → Push data onto the stack
- **POP** → Remove data from the stack
- **XCHG** → Exchange data between registers/memory
- **IN** → Input data from a port
- **OUT** → Output data to a port
- **LEA** → Load Effective Address
- **LDS** → Load DS register and pointer
- **LES** → Load ES register and pointer

---

## 2. Arithmetic Instructions

Used for mathematical calculations.

- **ADD** → Addition
- **ADC** → Add with carry
- **SUB** → Subtraction
- **SBB** → Subtract with borrow

- **MUL** → Unsigned multiplication
- **IMUL** → Signed multiplication
- **DIV** → Unsigned division
- **IDIV** → Signed division
- **INC** → Increment by 1
- **DEC** → Decrement by 1
- **NEG** → Two's complement (negate value)
- **CMP** → Compare two values

---

## 3. Logical and Bitwise Instructions

Used for logical operations on bits.

- **AND** → Bitwise AND
- **OR** → Bitwise OR
- **XOR** → Bitwise XOR
- **NOT** → Bitwise NOT (complement)
- **TEST** → AND operation without storing the result

---

## 4. Shift and Rotate Instructions

Used to shift or rotate bits within registers/memory.

- **SHL** → Shift left (Multiply by 2)
- **SHR** → Shift right (Divide by 2)
- **SAL** → Shift arithmetic left
- **SAR** → Shift arithmetic right
- **ROL** → Rotate left
- **ROR** → Rotate right
- **RCL** → Rotate through carry left
- **RCR** → Rotate through carry right

---

## 5. Control Transfer Instructions (Branching & Jumping)

Used to change the flow of execution.

- **JMP** → Unconditional jump
- **CALL** → Call a procedure
- **RET** → Return from a procedure
- **INT** → Interrupt call
- **IRET** → Return from an interrupt

### Conditional Jump Instructions

- **JZ** / **JE** → Jump if Zero / Equal
- **JNZ** / **JNE** → Jump if Not Zero / Not Equal

- **JC** → Jump if Carry
- **JNC** → Jump if No Carry
- **JO** → Jump if Overflow
- **JNO** → Jump if No Overflow
- **JS** → Jump if Sign
- **JNS** → Jump if No Sign
- **JP** / **JPE** → Jump if Parity Even
- **JNP** / **JPO** → Jump if Parity Odd
- **JA** / **JNBE** → Jump if Above
- **JAE** / **JNB** → Jump if Above or Equal
- **JB** / **JNAE** → Jump if Below
- **JBE** / **JNA** → Jump if Below or Equal

---

## 6. Loop Instructions

Used for repeated execution of code blocks.

- **LOOP** → Loop until CX = 0
- **LOOPE** / **LOOPZ** → Loop while Zero Flag (ZF = 1)
- **LOOPNE** / **LOOPNZ** → Loop while Zero Flag (ZF = 0)

---

## 7. String Instructions

Used for handling arrays, strings, and memory blocks.

- **MOVS** → Move string data
- **LODS** → Load string data
- **STOS** → Store string data
- **CMPS** → Compare strings
- **SCAS** → Scan string data
- **REP** → Repeat string operation
- **REPE** / **REPZ** → Repeat while Zero Flag is set
- **REPNE** / **REPNZ** → Repeat while Zero Flag is not set

---

## 8. Processor Control Instructions

Used to control CPU behavior and status.

- **HLT** → Halt processor
- **NOP** → No operation
- **STI** → Set Interrupt flag (Enable interrupts)
- **CLI** → Clear Interrupt flag (Disable interrupts)
- **STC** → Set Carry flag
- **CLC** → Clear Carry flag
- **CMC** → Complement Carry flag

- **WAIT** → Wait for Test Input pin to be active
- **ESC** → Escape to external processor (e.g., co-processor)

---

## 9. Segment & Memory Management Instructions

These help in managing memory and segmentation.

- **ASSUME** → Assign segment registers
- **SEGMENT** → Define segment
- **ENDS** → End segment
- **ORG** → Origin (set start address)
- **EQU** → Assign a constant value
- **PTR** → Pointer (byte/word specification)

---

## 10. Macro & Procedure Instructions

Used to define reusable code blocks.

- **MACRO** → Define a macro
- **ENDM** → End a macro
- **PROC** → Define a procedure
- **ENDP** → End a procedure

---

## 11. Debugging & Execution Control Instructions

Used for debugging and special execution control.

- **TRAP (TF)** → Enable single-step execution mode
- **INTO** → Interrupt on overflow
- **LOCK** → Lock bus during an operation
- **XLAT** → Translate byte from table (lookup table)

---

## Example: Simple 8086 Assembly Code

```
MOV AX, 0A00H     ; Load AX with 0A00H
MOV BX, 0200H     ; Load BX with 0200H
ADD AX, BX        ; Add BX to AX
JZ ZERO_LABEL     ; Jump if result is zero
HLT               ; Halt execution
ZERO_LABEL: NOP   ; No operation
```

---