Interrupt

Friday, August 29, 2025 6:43 PM

8086 Interrupt System

An **interrupt** is an event that temporarily halts the normal execution of the CPU so it can respond to a different, higher-priority task. After servicing this task through an **Interrupt Service Routine (ISR)**, the processor returns to the program it was executing.

Functions performed during an interrupt:

1. Flags register is pushed onto the stack.
2. Code Segment (CS) register is pushed.
3. Instruction Pointer (IP) is pushed.
4. IP is loaded from memory location: (Interrupt Type) × 4.
5. CS is loaded from (Interrupt Type × 4) + 2.
6. **TF (Trap Flag)** and **IF (Interrupt Enable Flag)** are cleared.
7. Execution jumps to the ISR.
8. The ISR ends with **IRET**, which pops IP, CS, and Flags from the stack, resuming the main program.

Classification of Interrupts in 8086

The 8086 microprocessor supports **256 interrupts**, identified by their **Type Codes (00H to FFH)**. Each interrupt vector requires **4 bytes**:

- 2 bytes for IP (offset)
- 2 bytes for CS (segment)

Thus, the **Interrupt Vector Table (IVT)** occupies memory addresses 00000H – 003FFH.

- **Types 0 to 4 → Predefined Interrupts**
  - **Type 0**: Divide Error (Division by zero).
  - **Type 1**: Single Step (Trap Flag debugging).
  - **Type 2**: Non-Maskable Interrupt (NMI).
  - **Type 3**: Breakpoint interrupt.
  - **Type 4**: Overflow (triggered by INTO instruction).

- **Types 5 to 31 → Reserved by Intel**
  - Kept aside for possible future processor extensions.
  - Not generally used by programmers.

- **Types 32 to 255 → Available for Maskable Interrupts**
  - These can be triggered by software (**INT n** instruction).
  - Or assigned to external hardware devices through the **INTR pin** (with help of an 8259 PIC).
  - Provide flexibility for user-defined ISRs.

1. Predefined Interrupts (Types 0 to 4)

a. Type 0 – Division by Zero

- Occurs when a division by zero instruction is executed.
- Non-maskable and generated automatically.

b. Type 1 – Single Step

- Generated when Trap Flag (TF) = 1.
- Executes an interrupt after every instruction (used in debugging).

c. Type 2 – Non-Maskable Interrupt (NMI)

- Triggered by the **NMI pin** (edge-triggered: Low → High).
- Cannot be disabled, used for critical failures like power loss.
- Loads ISR from addresses 00008H (IP) and 0000AH (CS).

d. Type 3 – Breakpoint Interrupt

- Generated by the one-byte instruction **INT 3**.
- Used for program debugging (allows inserting breakpoints).

e. **Type 4 – Overflow Interrupt**
   - Triggered when the **Overflow Flag (OF)** is set and the **INTO** instruction is executed.

2. **User-Defined Software Interrupts**
   - The **8086** allows the programmer to generate an interrupt by executing the **two-byte instruction INT n**, where n is the type number (0–255).
   - This instruction is **not maskable** by the **Interrupt Enable Flag (IF)**.
   - Purpose:
     - INT n can be used to **test an ISR (Interrupt Service Routine)** without requiring an actual external interrupt.
     - All **256 type codes (0–255)** can be used.
   - If a **predefined interrupt** is not being used in a system, its associated type code can also be reassigned to a **user-defined software interrupt**.

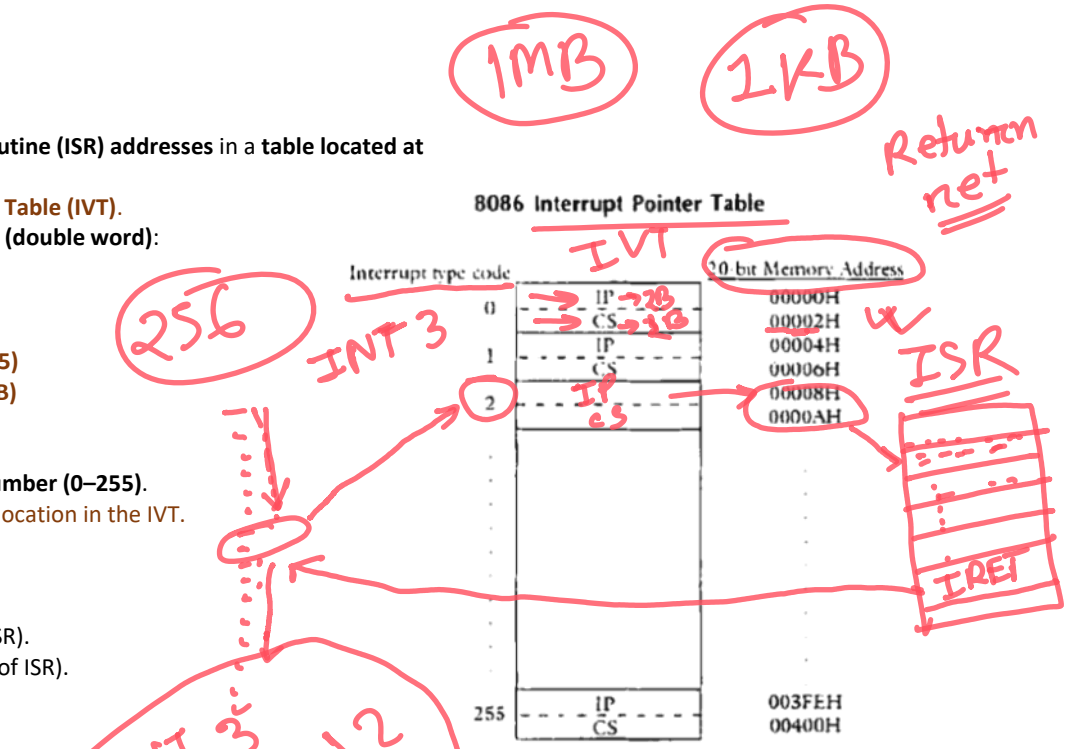
3. **User-Defined Hardware (Maskable Interrupts)**
   - Hardware interrupts are initiated through the **INTR pin** of the 8086.
   - These interrupts can be **enabled or disabled** using instructions:
     - STI → Set IF = 1 (Enable maskable interrupts).
     - CLI → Clear IF = 0 (Disable maskable interrupts).
   - If IF = 1 and **INTR is active (HIGH)** (with no higher priority interrupt pending), then after completing the current instruction, the 8086 generates:
     - **INTA' (Interrupt Acknowledge)** signal → LOW twice, each lasting about 2 cycles.
   - **INTR Sampling**
     - The 8086 samples the **INTR pin** at the **last clock cycle of each instruction**.
     - Special case: For instructions like **POP into a segment register (SS)**, interrupts are delayed until the next instruction completes.
       - This ensures a **32-bit pointer (SS:SP)** can be safely updated without being interrupted midway.

Interrupt Vector Table (IVT)

- The 8086 stores all **interrupt service routine (ISR) addresses** in a **table located at physical addresses 00000H – 003FFH**.
- This table is called the **Interrupt Vector Table (IVT)**.
- Each entry in the table requires **4 bytes (double word)**:
  - **2 bytes for IP (Instruction Pointer)**
  - **2 bytes for CS (Code Segment)**

Thus,

- Maximum interrupts = **256 types (0–255)**
- Total space = 256 × 4 = **1024 bytes (1 KB)**



Interrupt Type Codes

- Each interrupt is identified by a **type number (0–255)**.
- The **type code** determines the starting location in the IVT.
- Formula:

  Address in IVT=Type Code×4
- At this location:
  - The first **2 bytes** store IP (offset of ISR).
  - The next **2 bytes** store CS (segment of ISR).

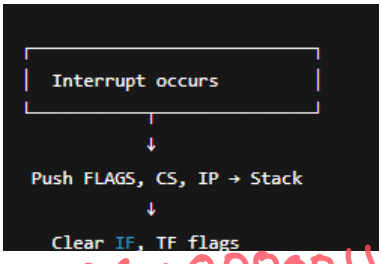
Example 1: INT 0 (Divide Error)

- Type code = 0
- IVT entry at address = 0 × 4 = **00000H**
- CS:IP fetched from [00000H–00003H]

Example 2: NMI (Type 2)

- Type code = 2
- IVT entry = 2 × 4 = **00008H**
- IP stored at 00008H–00009H
- CS stored at 0000AH–0000BH

So, when NMI occurs, the 8086 **executes INT 2** and jumps to the address specified at [00008H–0000BH].



CS:IP → CS = 0000H, IP → 000CH

CS = 00000H
IP = 000CH

So, when NMI occurs, the 8086 **executes INT 2** and jumps to the address specified at [00008H–0000BH].

```
Push FLAGS, CS, IP → Stack
          ↓
  Clear IF, TF flags
          ↓
Fetch CS:IP from IVT [Type x 4]
          ↓
 Jump to ISR (service routine)
          ↓
  ISR executes tasks
          ↓
     IRET instruction
          ↓
Pop IP, CS, FLAGS ← Stack
          ↓
Return to main program
```

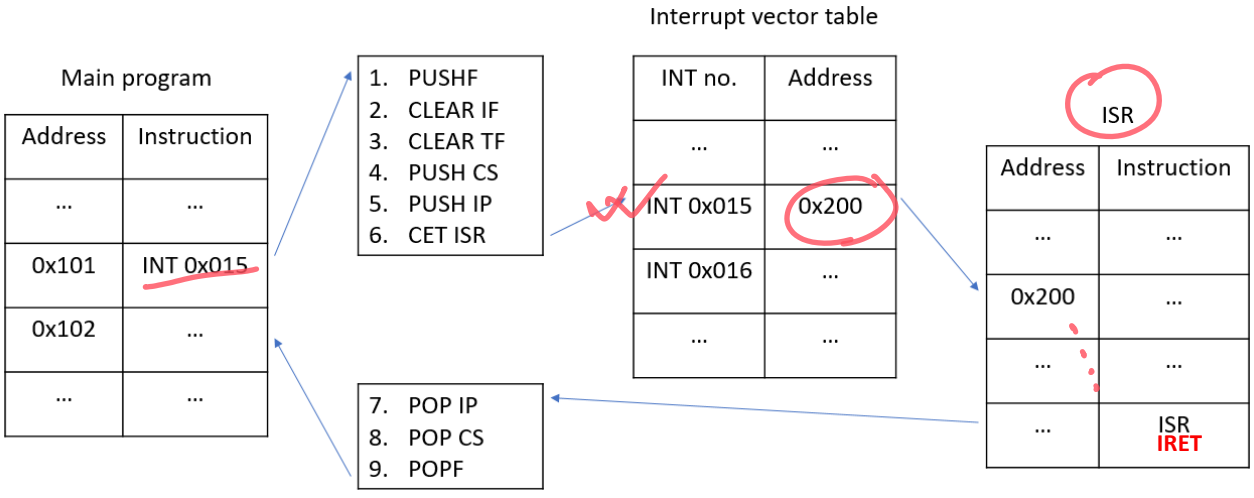
Execution Flow of Interrupts

When an interrupt occurs, the 8086 automatically performs these steps:

1. Pushes **FLAGS** onto the stack.
2. Pushes **CS** and **IP** onto the stack (to save return address).
3. Clears **IF** (Interrupt Flag) and **TF** (Trap Flag) → disables further interrupts temporarily.
4. Loads **new CS:IP** from the IVT for the given type code.
5. Begins execution of the **Interrupt Service Routine (ISR)**.

Returning from Interrupt

- The ISR must end with the **IRET (Interrupt Return)** instruction.
- IRET pops the saved **IP, CS, and FLAGS** from the stack.
- Execution resumes at the point where the program was interrupted.



Priority Interrupts (Interrupt and DMA Mode)

Recap: Interrupt-Initiated I/O

In interrupt-initiated I/O, when an input/output device is ready for data transfer, it generates an interrupt request signal to the CPU. Upon receiving the request, the CPU suspends the execution of the current instruction and services the request. However, when multiple devices generate interrupts simultaneously, a decision must be made regarding which request should be serviced first. This is achieved through a **priority interrupt system**, which defines an order of servicing among all devices.

Priority Interrupt System

The priority interrupt system establishes a hierarchy among devices so that simultaneous interrupt requests can be handled in a systematic manner. This system may be implemented through software or hardware methods.

Software Method: Polling

In the polling method, all interrupts branch to the same service program. This program checks devices one by one to identify which device generated the interrupt. The order of checking is determined by priority, beginning with the highest priority device and proceeding in descending order. Once the device is identified, a separate service routine for that device is executed.

Advantages

- Simple to implement.
- No specialized hardware is required.

Disadvantages

- Relatively slow, as each device must be checked sequentially.
- Inefficient for real-time systems where a fast response is essential.

Hardware Method: Daisy Chaining

The daisy chaining method is a hardware-based approach to priority interrupts. All devices capable of requesting an interrupt are connected in a serial fashion, with the device of highest priority placed first, followed by devices of lower priority in order.

Working Principle

1. All devices share a common interrupt request line connected to the CPU.
2. If a device requests an interrupt, it pulls the request line to a low state.
3. The CPU acknowledges the interrupt by sending an acknowledge signal.
4. This signal enters the “Priority In” (PI) input of the first device.
    - If the device has not requested service, it passes the signal to the next device through its “Priority Out” (PO) output.
    - If the device has requested service, it consumes the acknowledge signal, places its interrupt vector address on the CPU data bus, and prevents the signal from propagating further.
5. Devices farther in the chain are only serviced if higher-priority devices do not claim the acknowledge signal.

Advantages

- Multiple devices can share a single interrupt line.
- Provides a simple mechanism for implementing hardware priority.

Disadvantages

- Devices farther in the chain experience longer response times.
- Troubleshooting is more difficult in systems with many chained devices.

Comparison of Methods

Software Polling

- Advantages: Simple implementation, no hardware required.
- Disadvantages: Slow, wastes CPU time, unsuitable for real-time tasks.

Priority Interrupts

- Advantages: Efficient handling of high-priority tasks, suitable for real-time systems, deterministic response time.
- Disadvantages: Risk of starvation for low-priority tasks, possible priority inversion if not managed correctly.

Daisy Chaining

- Advantages: Fewer interrupt lines required, hardware-based priority is straightforward.
- Disadvantages: Increased response time for lower-priority devices, more difficult to manage with many devices.

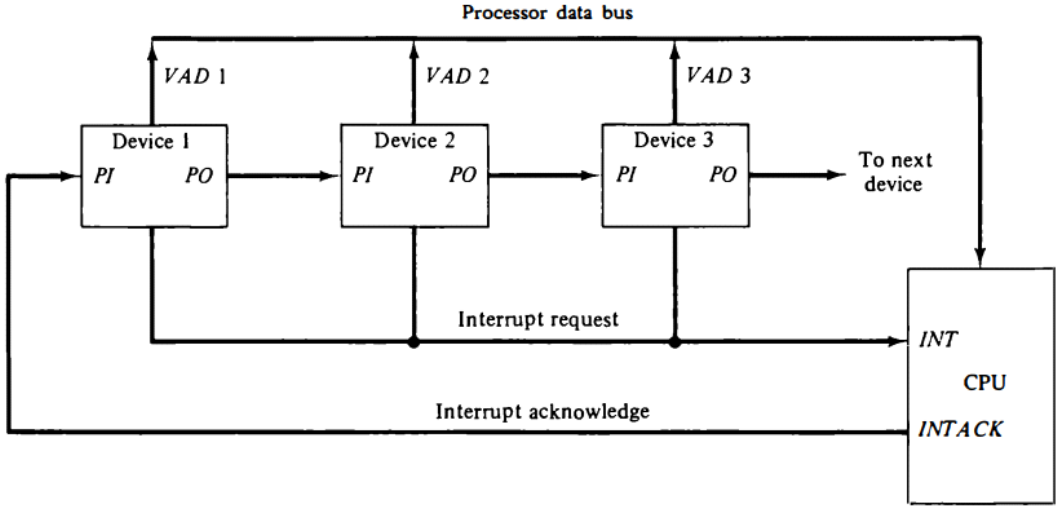


Figure 12 Daisy-chain priority interrupt.

Polled Interrupt

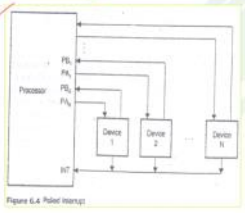


Figure 6.4 Polled interrupt

21

Daisy Chain Interrupt

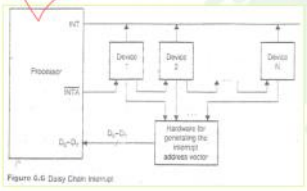


Figure 6.6 Daisy Chain interrupt

39