Notes on: 8085 Assembly Language Programming_from_0

1.) Instruction format opcode & Operands

Understanding how a computer's Central Processing Unit (CPU), like the 8085 microprocessor, executes tasks begins with grasping its fundamental language: instructions. Every action the CPU performs is dictated by an instruction. In 8085 Assembly Language Programming, we use mnemonics (short codes) to represent these instructions, making them easier for humans to understand before they are converted into machine code (binary).

1. What is an Instruction?
   • An instruction is a command given to the microprocessor to perform a specific task.
   • Think of it as a single, atomic step in a larger program.
   • For the 8085, examples include **add two numbers** or **move data from one location to another.**

2. The Core Components of an Instruction
   • Every instruction typically has two main parts: the Opcode and the Operands.
   • These components define *what* operation to perform and *on what* data or location.

3. Opcode (Operation Code)
   • Definition: The Opcode is the part of an instruction that specifies *what* operation is to be performed by the CPU.
   • Purpose: It tells the 8085 microprocessor the type of task it needs to execute.
   • Analogy: If an instruction is a sentence, the opcode is the **verb.** It's the action word.
   • In 8085 Assembly Language: Opcodes are represented by mnemonics.
   • Examples:
   • MOV: Stands for **MOVe,** meaning to copy data.
   • ADD: Stands for **ADD,** meaning to perform addition.
   • SUB: Stands for **SUBtract,** meaning to perform subtraction.
   • INR: Stands for **INcrement,** meaning to increase a value by one.
   • Machine Code Representation: Each mnemonic opcode has a unique binary code that the 8085 CPU directly understands. When you write **MOV** in assembly, it gets converted to a specific 8-bit binary pattern for the CPU.

4. Operands
   • Definition: Operands are the data or the locations that the instruction will operate upon. They provide the necessary information for the opcode to complete its task.
   • Purpose: They tell the CPU *where* to find the data, *what* data to use, or *where* to put the result.
   • Analogy: Continuing our sentence analogy, if the opcode is the verb, the operands are the **nouns** or **objects** of the sentence. They are the subject or object of the action.
   • In 8085 Assembly Language: Operands can take several forms:
   • Registers: These are small, high-speed storage locations within the CPU itself.
   • Examples for 8085: A (Accumulator), B, C, D, E, H, L.
   • Example instruction: MOV A, B (Here, A and B are operands). This means **move data from register B to register A.**
   • Immediate Data: A specific value (constant) that is directly provided as part of the instruction itself.
   • Examples for 8085: A number like 05H (hexadecimal 5), 12H (hexadecimal 18).
   • Example instruction: ADI 05H (Here, 05H is an operand). This means **add the value 05H to the Accumulator.**
   • Memory Addresses: The location in the computer's memory where data is stored or retrieved.
   • While we won't detail addressing modes, understand that sometimes an instruction needs to access data from a specific memory location or store a result there.
   • Example instruction: LDA 2000H (Here, 2000H is an operand representing a memory address). This means **Load data from memory location 2000H into the Accumulator.**

5. Instruction Format: Combining Opcode and Operands

- The instruction format defines the complete structure of an instruction, specifying how the opcode and operands are arranged.
- This format is crucial because it allows the CPU to correctly interpret and execute each instruction.
- Not all instructions have operands, and some have one, two, or even more (though 8085 typically has zero, one, or two).
- Examples for 8085:
- Instruction: NOP (No OPeration)
- Opcode: NOP
- Operands: None
- Action: Does nothing, just consumes one machine cycle.
- Instruction: CMA (CoMplement Accumulator)
- Opcode: CMA
- Operands: None (implicitly operates on the Accumulator)
- Action: Inverts all bits in the Accumulator.
- Instruction: INR B (INcrement Register B)
- Opcode: INR
- Operands: B (register)
- Action: Increments the value in register B by 1.
- Instruction: MOV C, D (MOVe data from register D to C)
- Opcode: MOV
- Operands: C (destination register), D (source register)
- Action: Copies the content of register D into register C.
- Instruction: SUI 10H (SUbtract Immediate 10H from Accumulator)
- Opcode: SUI
- Operands: 10H (immediate data)
- Action: Subtracts the value 10H from the Accumulator.

6. Real-World Importance and Understanding

- When you write a program in 8085 assembly language, you are essentially creating a sequence of these opcode-operand combinations.
- The assembler (a program) translates your human-readable mnemonics and operands into the binary machine code format that the 8085 microprocessor understands.
- Understanding this structure is fundamental for:
- Writing correct and efficient assembly programs.
- Debugging programs by knowing exactly what each part of an instruction is supposed to do.
- Comprehending how the CPU processes information at its most basic level.
- Different instructions will have different 'lengths' in terms of the number of bytes they occupy in memory, depending on whether they have zero, one, or two operands, and what type of operands these are. This is part of the instruction format that dictates how the CPU fetches and decodes them.

7. Summary of Key Points:

- An instruction is a command for the CPU, consisting of an opcode and operands.
- The Opcode specifies *what* operation to perform (e.g., MOV, ADD). It's the action part.
- Operands specify *on what* or *with what* the operation is performed (e.g., registers, immediate data, memory addresses). They are the data or locations involved.
- In 8085 assembly, we use mnemonics for opcodes and specific register names, values, or labels for operands.
- The instruction format defines how these parts are structured for the CPU to correctly interpret them.
- Mastering this concept is crucial for effective 8085 assembly language programming and understanding microprocessor operation.

2.) Machine Language Instruction Format: 1-Byte, 2-Byte & 3-Byte

Machine Language Instruction Format: 1-Byte, 2-Byte & 3-Byte

Understanding how machine language instructions are structured in terms of their byte length is fundamental to grasping how a microprocessor like the 8085 operates. It directly impacts memory usage, program size, and the speed at which the CPU fetches and executes instructions.

- Context: Machine language instructions are the binary codes that the CPU directly understands. When you write an 8085 assembly language program, it is converted into these binary machine language instructions by an assembler. Each instruction consists of an opcode (operation code) and, often, one or more operands (data or addresses the operation works on). The number of bytes an instruction occupies determines how the CPU reads and processes it from memory.

1. 1-Byte Instructions

- Explanation: These are the simplest and shortest instructions. They consist only of an opcode. The operation they perform either does not require any explicit operand, or the operand is implicitly understood by the CPU (e.g., it always operates on the Accumulator register).
- Characteristics:
- They occupy only one byte in memory.
- They are fetched by the CPU in a single memory access cycle (opcode fetch).
- They are generally the fastest instructions to execute because they require minimal decoding and no additional data fetches.
- Real-World Understanding: Think of these as quick, self-contained commands. Like a traffic cop blowing a whistle – the action (stop/go) is universal and doesn't need extra information.
- 8085 Examples:
- HLT (Halt): Stops the CPU. The CPU knows what **halt** means without needing any further data.
- NOP (No Operation): Does nothing, often used for timing delays.
- CMA (Complement Accumulator): Flips all bits of the Accumulator. The operand (Accumulator) is implied.
- STC (Set Carry Flag): Sets the carry flag to 1. No other data is needed.
- RLC (Rotate Accumulator Left): Rotates the bits of the Accumulator left by one position.

2. 2-Byte Instructions

- Explanation: These instructions consist of an opcode followed by a single 8-bit operand. This operand can be 8-bit data or an 8-bit port address. The first byte is the opcode, and the second byte provides the necessary additional information.
- Characteristics:
- They occupy two bytes in memory.
- The CPU fetches the first byte (opcode) and then fetches the second byte (operand) in the subsequent memory access cycle.
- They are used when an operation requires an immediate 8-bit value or interaction with a specific 8-bit I/O port.
- Real-World Understanding: Imagine giving a command like **Give me 5 apples. Give me** is the operation, and **5** is the immediate 8-bit quantity.
- 8085 Examples:
- MVI R, data (Move Immediate 8-bit data to Register R):
- Example: MVI B, 05H
- Machine Code: 06H (opcode for MVI B), 05H (the 8-bit data)
- Here, '06H' is the opcode, and '05H' is the 8-bit data that needs to be moved into register B.
- ADI data (Add Immediate 8-bit data to Accumulator):
- Example: ADI 10H
- Machine Code: C6H (opcode for ADI), 10H (the 8-bit data)
- IN port (Input data from 8-bit port address):
- Example: IN 20H
- Machine Code: DBH (opcode for IN), 20H (the 8-bit port address)
- OUT port (Output data to 8-bit port address):
- Example: OUT F0H
- Machine Code: D3H (opcode for OUT), F0H (the 8-bit port address)

3. 3-Byte Instructions

- Explanation: These are the longest instructions, consisting of an opcode followed by a 16-bit operand. This 16-bit operand typically represents a 16-bit memory address or 16-bit data. The first byte is the opcode, the second byte is the low-order byte of the 16-bit operand, and the third byte is the high-order byte.
- Characteristics:
- They occupy three bytes in memory.
- The CPU fetches the opcode first, then the low-order byte, and finally the high-order byte of the operand in successive memory access cycles.
- The 8085 (like many microprocessors) uses a **little-endian** format for 16-bit operands, meaning the low-order byte is stored at the lower memory address immediately after the opcode, and the high-order byte is stored at the next higher memory address.
- They are necessary for operations that require a full 16-bit address (e.g., jumping to a specific memory location, loading/storing data at a specific address) or initializing 16-bit register pairs with immediate data.
- Real-World Understanding: This is like giving a command **Go to Main Street, House Number 1234. Go to** is the operation, and **Main Street, House Number 1234** is the full 16-bit address (or data) needed for the operation.
- 8085 Examples:
- JMP address (Jump to a 16-bit memory address):
- Example: JMP 2050H
- Machine Code: C3H (opcode for JMP), 50H (low-order byte), 20H (high-order byte)
- The CPU would fetch C3H, then 50H, then 20H.
- LXI Rp, data16 (Load 16-bit Immediate data into Register Pair Rp):
- Example: LXI H, 3000H
- Machine Code: 21H (opcode for LXI H), 00H (low-order byte), 30H (high-order byte)
- STA address (Store Accumulator contents at a 16-bit memory address):
- Example: STA 4000H
- Machine Code: 32H (opcode for STA), 00H (low-order byte), 40H (high-order byte)
- LDA address (Load Accumulator contents from a 16-bit memory address):
- Example: LDA 5000H
- Machine Code: 3AH (opcode for LDA), 00H (low-order byte), 50H (high-order byte)

- Why Different Instruction Lengths?
- Efficiency: The 8085 is designed to be efficient. It uses only as many bytes as are strictly necessary for an instruction. A simple operation does not need to waste memory with extra operand bytes if they are not required.
- Functionality: Different operations inherently require different amounts of information. Operations interacting with a full 64KB memory space (requiring a 16-bit address) cannot be done with a single byte or two bytes.
- Performance vs. Memory Trade-off: Shorter instructions are generally faster to fetch and decode but offer limited functionality. Longer instructions provide more power and flexibility but take longer to fetch from memory.

- Real-World Significance for Computer Engineering Diploma Students:
- Program Size: Understanding instruction lengths helps in estimating the memory footprint of your assembly programs, which is crucial in embedded systems with limited memory.
- CPU Fetch-Decode-Execute Cycle: The CPU's instruction cycle directly depends on instruction length. A 3-byte instruction will require three memory read operations just to fetch the instruction bytes before it can even begin execution. This impacts the overall program execution speed.
- Memory Organization: Knowing the byte order (little-endian for 16-bit operands) is essential when debugging programs by inspecting memory contents directly.

- Summary of Key Points:
- Machine language instructions in 8085 can be 1, 2, or 3 bytes long.
- 1-Byte instructions: Opcode only, fastest, implied operands.
- 2-Byte instructions: Opcode + 8-bit operand (data or port address).
- 3-Byte instructions: Opcode + 16-bit operand (memory address or data), use little-endian order (low

byte first, then high byte).
- The choice of length depends on the complexity of the operation and the amount of data/address information required.
- Instruction length directly affects program size, memory access cycles, and CPU performance.

3.) 8085 Addressing Modes

Addressing modes in the 8085 microprocessor define how an instruction locates its operand. An operand is the data or the memory address that an instruction needs to perform its operation. Understanding these modes is essential for writing efficient and correct 8085 assembly language programs, as they dictate how the CPU retrieves the necessary information. Each instruction in the 8085 uses a specific addressing mode to determine where its data comes from.

The 8085 microprocessor supports five primary addressing modes:

1- Implicit (or Implied) Addressing Mode
2- Register Addressing Mode
3- Direct Addressing Mode
4- Register Indirect Addressing Mode
5- Immediate Addressing Mode

Let's explore each mode in detail:

1- Implicit (or Implied) Addressing Mode

- Explanation: In this mode, the operand is implicitly defined within the instruction itself. The instruction does not explicitly specify an operand because the operation inherently involves specific internal CPU registers or flags.
- How it works: The CPU knows exactly which register or flag to operate on based solely on the opcode. No additional operand is fetched from memory or specified in the instruction.
- Example: HLT
- This instruction halts the processor. It doesn't need any data or address from the programmer. The CPU just stops.
- Characteristics: It is the fastest addressing mode because no operand fetching is required. It uses 1-byte instructions.
- Real-world context: Often used for system control instructions or operations that exclusively act on the accumulator (like CMA - Complement Accumulator, which is also sometimes considered implicit because it always targets the accumulator).

2- Register Addressing Mode

- Explanation: The operand is located in one of the 8085's internal general-purpose registers (A, B, C, D, E, H, L). The instruction specifies the register containing the data.
- How it works: The instruction includes the names of the registers holding the source and/or destination data. The CPU accesses these registers directly.
- Example: MOV A, B
- This instruction moves the content of register B to register A. Both operands (source B and destination A) are registers.
- Characteristics: Very fast because data is within the CPU itself, requiring no memory access. Instructions using this mode are typically 1-byte.
- Real-world context: Frequently used for internal data manipulation, temporary storage, and fast arithmetic/logical operations between registers. This is crucial for optimizing code speed.

3- Direct Addressing Mode

- Explanation: The instruction specifies the exact 16-bit memory address where the operand is located. The address is explicitly provided as part of the instruction.

• How it works: The CPU fetches the instruction, then retrieves the 16-bit address from the instruction itself. It then uses this address to access the specified memory location to read or write data.
• Example: LDA 2050H
• This instruction loads the content of the memory location 2050H into the Accumulator. The address 2050H is part of the instruction.
• Characteristics: Slower than register modes because it requires memory access. Instructions using this mode are 3-bytes long (opcode + 2-byte address).
• Real-world context: Useful for accessing specific, fixed data locations in memory, such as configuration parameters, lookup tables, or I/O port addresses when memory-mapped I/O is used.

4- Register Indirect Addressing Mode

• Explanation: The instruction specifies a register pair (BC, DE, or HL) whose content acts as the 16-bit memory address of the operand. The register pair *points* to the memory location.
• How it works: The CPU takes the 16-bit value stored in the specified register pair and uses it as a memory address. It then accesses this memory location to get or put the data. The special symbol 'M' is often used to denote the memory location pointed to by the HL register pair.
• Example: MOV A, M
• This instruction moves the content of the memory location pointed to by the HL register pair into the Accumulator. The 'M' here refers to the memory location [HL].
• Example: STAX B
• This instruction stores the content of the Accumulator into the memory location pointed to by the BC register pair.
• Characteristics: Offers flexibility in accessing sequential data, often used for arrays or data blocks. It is faster than direct addressing as the address is already in a register. Instructions are typically 1-byte.
• Real-world context: Excellent for processing data arrays, stack operations (though specific stack instructions exist), or when the memory address needs to be calculated or modified during program execution (e.g., looping through a buffer).

5- Immediate Addressing Mode

• Explanation: The operand itself (the actual data value) is provided directly within the instruction. The instruction contains the data that needs to be used.
• How it works: The CPU fetches the instruction, and the next byte(s) immediately following the opcode are the data to be used. This data is then loaded into a register or used in an operation.
• Example: MVI A, 05H
• This instruction moves the immediate data 05H into the Accumulator. The value 05H is part of the instruction.
• Characteristics: Fast for loading constants. Instructions are either 2-bytes (opcode + 1-byte data) or 3-bytes (opcode + 2-byte data, for 16-bit data like LXI B, 2000H).
• Real-world context: Widely used for initializing registers with constant values, setting up counters, or loading fixed values for arithmetic or logical operations.

Summary of Key Points:

• Addressing modes define how the 8085 locates operands for instructions.
• Implicit mode: Operand is inherent to the instruction (e.g., HLT). Fastest, 1-byte.
• Register mode: Operand is in a CPU register (e.g., MOV A, B). Fast, 1-byte.
• Direct mode: 16-bit memory address is given in the instruction (e.g., LDA 2050H). Slower, 3-bytes.
• Register Indirect mode: A register pair holds the 16-bit memory address (e.g., MOV A, M or STAX B). Flexible, typically 1-byte.
• Immediate mode: The data itself is part of the instruction (e.g., MVI A, 05H). Good for constants, 2 or 3-bytes.
• Choosing the right addressing mode impacts program speed and memory usage.

These modes are fundamental to how all subsequent instruction types, such as data transfer, arithmetic, logical, and branching instructions, will find and manipulate their respective data.

4.) Data transfer Instructions

Data transfer instructions are fundamental operations in any microprocessor, including the 8085. They are responsible for moving (copying) data between different locations within the computer system without altering the data itself. Think of it like copying a file from one folder to another on your computer - the original file remains, and a duplicate is created in the new location. The data content itself doesn't change during the transfer.

These instructions are the backbone of all other operations because data must first be moved to a specific location (like a register or memory) before it can be processed by arithmetic, logical, or other instructions.

Here's a breakdown of data transfer instructions in the 8085 microprocessor:

1- Types of Data Transfer in 8085

The 8085 allows data transfer between:

- Registers to Registers: Copying data from one 8-bit general-purpose register (A, B, C, D, E, H, L) to another.
- Memory to Register: Copying data from an 8-bit memory location to a register.
- Register to Memory: Copying data from a register to an 8-bit memory location.
- Immediate Data to Register/Memory: Loading a constant 8-bit or 16-bit value directly into a register, register pair, or memory.
- Direct Memory Access: Loading or storing data from/to a specific memory address directly, often involving the Accumulator or the H-L pair.
- Register Pair Exchange: Swapping the contents of two 16-bit register pairs.

2- Key 8085 Data Transfer Instructions

A- MOV instruction (Move data between registers or register and memory)

This instruction is used to copy the content of a source to a destination. The source's content remains unchanged.

- MOV Rd, Rs (Register to Register Copy)
- Copies the 8-bit content of the source register (Rs) to the destination register (Rd).
- Example: MOV B, C
- Copies the content of register C into register B. (If C had 05H, B now also has 05H, C still has 05H.)

- MOV Rd, M (Memory to Register Copy)
- Copies the 8-bit content from the memory location addressed by the H-L register pair to the destination register (Rd).
- The H-L pair acts as a 16-bit memory pointer.
- Example: MOV A, M
- If H-L contains 2000H, the content of memory location 2000H is copied into the Accumulator (A).

- MOV M, Rs (Register to Memory Copy)
- Copies the 8-bit content of the source register (Rs) to the memory location addressed by the H-L register pair.
- Example: MOV M, D
- If H-L contains 2050H, the content of register D is copied into memory location 2050H.

B- MVI instruction (Move Immediate data to Register or Memory)

This instruction loads an 8-bit immediate value directly into a specified register or memory location.

- MVI R, 8-bit data (Move Immediate to Register)
- Loads the specified 8-bit data into register R.

- Example: MVI A, 3AH
- The Accumulator (A) is loaded with the value 3AH.

- MVI M, 8-bit data (Move Immediate to Memory)
- Loads the specified 8-bit data into the memory location addressed by the H-L register pair.
- Example: MVI M, 55H
- If H-L contains 2100H, the memory location 2100H is loaded with the value 55H.

C- LXI instruction (Load Register Pair Immediate)

This instruction loads a 16-bit immediate value into a specified 16-bit register pair (BC, DE, HL, or SP - Stack Pointer). This is often used to initialize memory pointers.

- LXI Rp, 16-bit data
- Rp can be B (for BC pair), D (for DE pair), H (for HL pair), or SP.
- The higher-order byte of data goes into the higher register of the pair, and the lower-order byte goes into the lower register.
- Example: LXI H, 3000H
- Register H gets 30H, and register L gets 00H. The H-L pair now points to memory address 3000H.
- Example: LXI SP, 20FFH
- The Stack Pointer (SP) is initialized to 20FFH.

D- LDA instruction (Load Accumulator Direct)

This instruction loads the 8-bit content of the specified 16-bit memory address directly into the Accumulator.

- LDA 16-bit address
- Example: LDA 2500H
- The content of memory location 2500H is copied into the Accumulator (A).

E- STA instruction (Store Accumulator Direct)

This instruction stores the 8-bit content of the Accumulator directly into the specified 16-bit memory address.

- STA 16-bit address
- Example: STA 2800H
- The content of the Accumulator (A) is copied into memory location 2800H.

F- LHLD instruction (Load H-L pair Direct)

This instruction loads the H-L register pair from two consecutive 16-bit memory locations. The content of the specified address goes to register L, and the content of the next consecutive address (address + 1) goes to register H.

- LHLD 16-bit address
- Example: LHLD 3000H
- Content of memory location 3000H is copied to L.
- Content of memory location 3001H is copied to H.
- So, if 3000H had 50H and 3001H had 12H, then HL would become 1250H.

G- SHLD instruction (Store H-L pair Direct)

This instruction stores the H-L register pair into two consecutive 16-bit memory locations. The content of register L goes to the specified address, and the content of register H goes to the next consecutive address (address + 1).

- SHLD 16-bit address

- Example: SHLD 3000H
- Content of L is copied to memory location 3000H.
- Content of H is copied to memory location 3001H.
- If HL is 1250H, then 3000H gets 50H and 3001H gets 12H.

H- XCHG instruction (Exchange H-L with D-E)

This instruction exchanges the 16-bit content of the H-L register pair with the D-E register pair. It's a useful instruction for quickly swapping memory pointers or data blocks.

- XCHG
- Example: If HL has 1234H and DE has 5678H, after XCHG, HL will have 5678H and DE will have 1234H.

I- SPHL instruction (Move H-L to SP)

This instruction copies the 16-bit content of the H-L register pair into the Stack Pointer (SP). This is commonly used to initialize or re-point the stack.

- SPHL
- The Stack Pointer is a special-purpose register that always holds the address of the top of the stack.

J- PCHL instruction (Move H-L to PC)

This instruction copies the 16-bit content of the H-L register pair into the Program Counter (PC). This effectively causes the program execution to jump to the address contained in the H-L pair.

- PCHL
- The Program Counter always holds the address of the next instruction to be fetched and executed.

3- Important Considerations and Real-World Relevance

- No Data Modification: All data transfer instructions merely copy data. They do not perform any arithmetic or logical operations on the data, meaning the source content remains unchanged.
- Flag Register: Most data transfer instructions do not affect the Flag Register. This means they don't set or reset flags like Zero, Carry, Sign, etc., as their purpose is solely data movement.
- Memory Addressing: Instructions like MOV M, MVI M, LDA, STA, LHLD, SHLD directly interact with memory. This is crucial for storing and retrieving program data, variables, or system configurations.
- Initializing Pointers: LXI and XCHG are often used to initialize and manipulate 16-bit memory pointers (like HL, DE) that are critical for accessing larger blocks of data in memory or for setting up specific data structures.
- Setting Up Operations: Before you can add two numbers, you might first use MOV or MVI to load them into registers. Data transfer is always the first step.
- System Control: SPHL and PCHL are special data transfer instructions that affect system control registers (Stack Pointer and Program Counter), enabling changes in stack management or program flow.

Summary of Key Points:

- Data transfer instructions copy data without changing its value at the source.
- They are essential for setting up data for all other processing instructions.
- The 8085 supports moving 8-bit data between registers, between a register and memory, and loading immediate data.
- 16-bit data transfer involves register pairs, especially for memory addressing (HL, DE, BC) and the Stack Pointer (SP).
- Instructions like LDA, STA, LHLD, SHLD provide direct access to specific memory addresses.
- The XCHG instruction allows for efficient swapping of 16-bit register pair contents.
- SPHL and PCHL transfer data to the Stack Pointer and Program Counter, affecting system control.

• Most data transfer instructions do not affect the 8085's flag register.

5.) Arithmetical Instructions

Arithmetical Instructions in the 8085 Microprocessor

Arithmetical instructions are fundamental operations that allow the 8085 microprocessor to perform mathematical calculations. These instructions primarily deal with addition, subtraction, incrementing, and decrementing values. They are crucial for tasks ranging from simple data manipulation to complex algorithm execution.

1. Core Principles of 8085 Arithmetical Instructions

• Accumulator-Centric: Most 8085 arithmetic instructions involve the Accumulator (Register A) as one of the operands. The result of the operation is almost always stored back into the Accumulator. This means one operand is implicitly the Accumulator, and the other is specified in the instruction.

• Flag Register Impact: All arithmetic operations significantly affect the 8085's Flag Register. The flags (Sign, Zero, Auxiliary Carry, Parity, Carry) are updated based on the result of the operation. Understanding flag behavior is essential for conditional execution and multi-byte arithmetic.

2. Addition Instructions

These instructions add the content of a source operand to the Accumulator. The result is stored in the Accumulator.

• ADD R (Add Register to Accumulator)
• Operation: A <- A + R
• Adds the content of register R (B, C, D, E, H, L) to the Accumulator.
• Example: ADD B (Adds content of Register B to Accumulator A)
• Flags Affected: All (S, Z, AC, P, CY)

• ADD M (Add Memory to Accumulator)
• Operation: A <- A + (HL)
• Adds the content of the memory location pointed to by the HL register pair to the Accumulator.
• Example: ADD M (Adds content of memory location [H:L] to Accumulator A)
• Flags Affected: All (S, Z, AC, P, CY)

• ADI Data (Add Immediate Data to Accumulator)
• Operation: A <- A + Data
• Adds the 8-bit immediate data provided in the instruction to the Accumulator.
• Example: ADI 45H (Adds hexadecimal 45 to Accumulator A)
• Flags Affected: All (S, Z, AC, P, CY)

• ADC R (Add Register to Accumulator with Carry)
• Operation: A <- A + R + CY
• Adds the content of register R and the Carry Flag (CY) to the Accumulator. This is vital for multi-byte addition.
• Example: ADC B (Adds content of Register B and current Carry Flag value to Accumulator A)
• Flags Affected: All (S, Z, AC, P, CY)

• ADC M (Add Memory to Accumulator with Carry)
• Operation: A <- A + (HL) + CY
• Adds the content of the memory location pointed to by HL and the Carry Flag (CY) to the Accumulator.
• Example: ADC M (Adds content of memory location [H:L] and current Carry Flag value to Accumulator A)
• Flags Affected: All (S, Z, AC, P, CY)

- ACI Data (Add Immediate Data to Accumulator with Carry)
- Operation: A <- A + Data + CY
- Adds the 8-bit immediate data and the Carry Flag (CY) to the Accumulator.
- Example: ACI 10H (Adds hexadecimal 10 and current Carry Flag value to Accumulator A)
- Flags Affected: All (S, Z, AC, P, CY)

- Real-world use: Multi-byte addition
- To add two 16-bit numbers, you first add their lower 8-bits using ADD, then add their higher 8-bits using ADC. The carry from the lower byte addition is automatically included in the higher byte addition.
- Example: Add 16-bit number stored at MEM1 (lower byte) and MEM2 (higher byte) to another at MEM3 and MEM4.

```
LXI H, MEM1 ; HL points to 1st num lower byte
MOV A, M ; Get lower byte of 1st num
LXI D, MEM3 ; DE points to 2nd num lower byte
MOV B, M ; Get lower byte of 2nd num
ADD B ; Add lower bytes, result in A, Carry set if overflow
STA SUM_L ; Store lower byte of sum
INX H ; HL points to 1st num higher byte
MOV B, M ; Get higher byte of 1st num
INX D ; DE points to 2nd num higher byte
MOV C, M ; Get higher byte of 2nd num (Oops, should use M not C)
MOV E, C ; (Corrected: MOV B,M in previous line, here for clarity)
MOV C, E ; (No, should be just MOV E, M for second number's higher byte)
; Let's re-do the example concisely:
LHLD NUM1 ; Load NUM1 (16-bit) into HL
LXI B, NUM2 ; Load NUM2 (16-bit) into BC
MOV A, L ; A = LSB of NUM1
ADD C ; A = LSB(NUM1) + LSB(NUM2)
MOV L, A ; L = new LSB
MOV A, H ; A = MSB of NUM1
ADC B ; A = MSB(NUM1) + MSB(NUM2) + Carry from LSB addition
MOV H, A ; H = new MSB
SHLD SUM ; Store 16-bit sum from HL
```

3. Subtraction Instructions

These instructions subtract the content of a source operand from the Accumulator. The result is stored in the Accumulator. For subtraction, the Carry Flag (CY) acts as a Borrow Flag. If CY is set (1) after subtraction, it indicates a borrow was needed.

- SUB R (Subtract Register from Accumulator)
- Operation: A <- A - R
- Subtracts the content of register R (B, C, D, E, H, L) from the Accumulator.
- Example: SUB C (Subtracts content of Register C from Accumulator A)
- Flags Affected: All (S, Z, AC, P, CY)

- SUB M (Subtract Memory from Accumulator)
- Operation: A <- A - (HL)
- Subtracts the content of the memory location pointed to by HL from the Accumulator.
- Example: SUB M (Subtracts content of memory location [H:L] from Accumulator A)
- Flags Affected: All (S, Z, AC, P, CY)

- SUI Data (Subtract Immediate Data from Accumulator)
- Operation: A <- A - Data
- Subtracts the 8-bit immediate data from the Accumulator.
- Example: SUI 20H (Subtracts hexadecimal 20 from Accumulator A)
- Flags Affected: All (S, Z, AC, P, CY)

- SBB R (Subtract Register from Accumulator with Borrow)
- Operation: A <- A - R - CY

- Subtracts the content of register R and the Carry Flag (CY, acting as borrow) from the Accumulator. Essential for multi-byte subtraction.
- Example: SBB D (Subtracts content of Register D and current Carry Flag value from Accumulator A)
- Flags Affected: All (S, Z, AC, P, CY)

- SBB M (Subtract Memory from Accumulator with Borrow)
- Operation: A <- A - (HL) - CY
- Subtracts the content of the memory location pointed to by HL and the Carry Flag (CY) from the Accumulator.
- Example: SBB M (Subtracts content of memory location [H:L] and current Carry Flag value from Accumulator A)
- Flags Affected: All (S, Z, AC, P, CY)

- SBI Data (Subtract Immediate Data from Accumulator with Borrow)
- Operation: A <- A - Data - CY
- Subtracts the 8-bit immediate data and the Carry Flag (CY) from the Accumulator.
- Example: SBI 05H (Subtracts hexadecimal 05 and current Carry Flag value from Accumulator A)
- Flags Affected: All (S, Z, AC, P, CY)

- Real-world use: Multi-byte subtraction
- Similar to addition, you first subtract the lower 8-bits using SUB, then the higher 8-bits using SBB. The borrow generated from the lower byte subtraction is automatically included in the higher byte subtraction.

4. Increment Instructions
These instructions increase the value of a register or a memory location by 1.

- INR R (Increment Register Content by 1)
- Operation: R <- R + 1
- Increments the content of register R (A, B, C, D, E, H, L) by 1.
- Example: INR C (Increments content of Register C by 1)
- Flags Affected: All EXCEPT Carry Flag (S, Z, AC, P are affected, CY is NOT)
- Real-world use: Loop counters, pointer adjustments.

- INR M (Increment Memory Content by 1)
- Operation: (HL) <- (HL) + 1
- Increments the content of the memory location pointed to by HL by 1.
- Example: INR M (Increments content of memory location [H:L] by 1)
- Flags Affected: All EXCEPT Carry Flag (S, Z, AC, P are affected, CY is NOT)
- Real-world use: Modifying data in memory, loop counters if data is in memory.

5. Decrement Instructions
These instructions decrease the value of a register or a memory location by 1.

- DCR R (Decrement Register Content by 1)
- Operation: R <- R - 1
- Decrements the content of register R (A, B, C, D, E, H, L) by 1.
- Example: DCR E (Decrements content of Register E by 1)
- Flags Affected: All EXCEPT Carry Flag (S, Z, AC, P are affected, CY is NOT)
- Real-world use: Loop counters, pointer adjustments.

- DCR M (Decrement Memory Content by 1)
- Operation: (HL) <- (HL) - 1
- Decrements the content of the memory location pointed to by HL by 1.
- Example: DCR M (Decrements content of memory location [H:L] by 1)
- Flags Affected: All EXCEPT Carry Flag (S, Z, AC, P are affected, CY is NOT)
- Real-world use: Modifying data in memory, loop counters.

6. Double Precision Addition
The 8085 provides an instruction for 16-bit addition, which is particularly useful when working with memory addresses or 16-bit data.

- DAD Rp (Add Register Pair to HL Register Pair)
- Operation: HL <- HL + Rp
- Adds the content of the specified register pair Rp (BC, DE, or SP - Stack Pointer) to the HL register pair. The result is stored in HL.
- Example: DAD B (Adds content of BC to HL, stores in HL)
- Flags Affected: ONLY the Carry Flag (CY). (S, Z, AC, P are NOT affected)
- Real-world use: This is crucial for manipulating 16-bit memory pointers efficiently. For instance, if you have a base address in HL and an offset in BC, DAD B calculates the new effective address directly in HL. Useful for array indexing or moving blocks of data.

Summary of Key Points:
- 8085 arithmetic instructions primarily use the Accumulator as an operand and destination.
- All arithmetic operations update the Flag Register, which is vital for control flow (branching) and multi-byte arithmetic.
- Addition (ADD, ADI) and Subtraction (SUB, SUI) instructions operate on 8-bit data.
- ADC and SBB instructions are essential for performing multi-byte (e.g., 16-bit, 32-bit) arithmetic by incorporating the Carry/Borrow flag.
- Increment (INR) and Decrement (DCR) instructions modify values by 1 and affect all flags except the Carry Flag. They are often used for counting and address manipulation.
- DAD instruction performs 16-bit addition between a register pair and HL, only affecting the Carry Flag. This is powerful for address calculations and 16-bit data processing.

6.) Logical Instructions

Logical instructions are a fundamental set of operations in assembly language programming, including the 8085 microprocessor. They perform bit-wise logical operations on data, treating each bit independently. Unlike arithmetic instructions which deal with numerical values (addition, subtraction), logical instructions manipulate data at the individual bit level (0s and 1s). They are crucial for tasks like data validation, setting or clearing specific bits, and checking the status of individual bits within a byte.

The 8085 processor provides several logical instructions that operate primarily with the Accumulator register, which serves as a central hub for these operations.

Key Logical Operations

The core logical operations are based on Boolean algebra:

1- AND (Logical AND)
- Performs a bit-wise AND operation between two operands.
- If both corresponding bits are 1, the result bit is 1; otherwise, it's 0.
- Used for **masking** bits – clearing specific bits while retaining others.
- Example: 1010 AND 0011 = 0010

2- OR (Logical OR)
- Performs a bit-wise OR operation between two operands.
- If at least one of the corresponding bits is 1, the result bit is 1; otherwise, it's 0.
- Used for **setting** bits – forcing specific bits to 1 while retaining others.
- Example: 1010 OR 0011 = 1011

3- XOR (Exclusive OR)
- Performs a bit-wise XOR operation between two operands.
- If the corresponding bits are different (one is 0, the other is 1), the result bit is 1; otherwise, it's 0.

- Used for **toggling** bits (inverting them) or checking if two values are identical (XORing identical values results in 0).
- Example: 1010 XOR 0011 = 1001

4- NOT (Logical NOT / Complement)
- Inverts all the bits of a single operand. 0 becomes 1, and 1 becomes 0.
- Used to flip the state of all bits in a byte.
- Example: NOT 1010 = 0101

8085 Logical Instructions

The 8085 microprocessor implements these operations using specific instruction mnemonics. Most of these instructions operate with the Accumulator as one of the operands and store the result back into the Accumulator.

1- ANA r / ANA M / ANI data (Logical AND with Accumulator)
- Performs a bit-wise AND operation between the content of the Accumulator and the content of a specified register (r), memory location (M), or an 8-bit immediate data.
- The result is stored in the Accumulator.
- Flags affected: Zero (Z), Sign (S), Parity (P) are set/reset according to the result. Carry (CY) is always reset to 0. Auxiliary Carry (AC) is also set/reset (often cleared for ANA, set for ANI under specific conditions, generally not relied upon).
- Example:

MVI A, 0FH ; Accumulator = 0000 1111B
ANI 06H ; A = 0000 1111B AND 0000 0110B = 0000 0110B (06H)

2- ORA r / ORA M / ORI data (Logical OR with Accumulator)
- Performs a bit-wise OR operation between the content of the Accumulator and the content of a register (r), memory location (M), or an 8-bit immediate data.
- The result is stored in the Accumulator.
- Flags affected: Z, S, P are set/reset based on the result. CY and AC are always reset to 0.
- Example:

MVI A, 05H ; Accumulator = 0000 0101B
ORI 03H ; A = 0000 0101B OR 0000 0011B = 0000 0111B (07H)

3- XRA r / XRA M / XRI data (Exclusive OR with Accumulator)
- Performs a bit-wise XOR operation between the content of the Accumulator and the content of a register (r), memory location (M), or an 8-bit immediate data.
- The result is stored in the Accumulator.
- Flags affected: Z, S, P are set/reset based on the result. CY and AC are always reset to 0.
- Example:

MVI A, 0FH ; Accumulator = 0000 1111B
XRI FFH ; A = 0000 1111B XOR 1111 1111B = 1111 0000B (F0H) - Toggles all bits
XRA A ; Accumulator XOR Accumulator always results in 0, useful for clearing Accumulator and setting Z flag.

4- CMA (Complement Accumulator)
- Inverts all the bits in the Accumulator. No other register or memory location is involved.
- This is the 8085's **Logical NOT** operation.
- Flags affected: None. This instruction does not affect any flags.
- Example:

MVI A, 55H ; Accumulator = 0101 0101B
CMA ; A = 1010 1010B (AAH)

5- CMP r / CMP M / CPI data (Compare with Accumulator)
- Performs a subtraction of the operand (register r, memory M, or immediate data) from the Accumulator.
- Crucially, the result of this subtraction is not stored anywhere. Its primary purpose is to affect the flags based on the comparison.

- Flags affected: Z, S, P, CY, AC are set/reset based on the hypothetical subtraction.
- If A = operand, Z flag is set (result of A-operand is 0).
- If A < operand, CY flag is set (borrow occurs).
- If A > operand, CY flag is reset (no borrow).
- This instruction is fundamental for decision-making and flow control in programs.
- Example:

MVI A, 10H ; Accumulator = 10H
CPI 0AH ; Compares 10H with 0AH. A > operand, so Z=0, CY=0.
CPI 10H ; Compares 10H with 10H. A = operand, so Z=1, CY=0.
CPI 20H ; Compares 10H with 20H. A < operand, so Z=0, CY=1.

6- Rotate Instructions (Bit Manipulation)
These instructions shift bits within the Accumulator and involve the Carry flag. They are used for bit-level manipulation, serial data transfer, and implementing multiplication/division by powers of 2.

a. RLC (Rotate Accumulator Left)
- Each bit in the Accumulator is shifted one position to the left.
- The D7 (most significant) bit moves to the D0 (least significant) position and also copies into the Carry flag.
- Flags affected: Only CY is affected. S, Z, P, AC are not affected.
- Example: If A = 10110010B, CY = 0

RLC
A = 01100101B, CY = 1

b. RRC (Rotate Accumulator Right)
- Each bit in the Accumulator is shifted one position to the right.
- The D0 (least significant) bit moves to the D7 (most significant) position and also copies into the Carry flag.
- Flags affected: Only CY is affected. S, Z, P, AC are not affected.
- Example: If A = 10110010B, CY = 0

RRC
A = 01011001B, CY = 0

c. RAL (Rotate Accumulator Left Through Carry)
- Each bit in the Accumulator is shifted one position to the left.
- The D7 bit moves into the Carry flag.
- The content of the Carry flag moves into the D0 (least significant) position.
- Flags affected: Only CY is affected. S, Z, P, AC are not affected.
- Example: If A = 10110010B, CY = 1

RAL
A = 01100101B, CY = 1

d. RAR (Rotate Accumulator Right Through Carry)
- Each bit in the Accumulator is shifted one position to the right.
- The D0 bit moves into the Carry flag.
- The content of the Carry flag moves into the D7 (most significant) position.
- Flags affected: Only CY is affected. S, Z, P, AC are not affected.
- Example: If A = 10110010B, CY = 1

RAR
A = 11011001B, CY = 0

Flags Affected by Logical Instructions

Logical instructions significantly impact the 8085's Flag Register, specifically the Sign (S), Zero (Z), Parity (P), and Carry (CY) flags. The Auxiliary Carry (AC) flag is generally reset for most logical operations (ANA/ORA/XRA) but is not typically used or relied upon for their outcomes. CMA does not affect any flags. Rotate instructions only affect the CY flag.

- Sign Flag (S): Set if the most significant bit (D7) of the result is 1. Reset otherwise.

- Zero Flag (Z): Set if the result of the operation is 00H. Reset otherwise.
- Parity Flag (P): Set if the result has an even number of 1s (even parity). Reset if it has an odd number of 1s (odd parity).
- Carry Flag (CY): For ANA, ORA, XRA, this flag is typically reset to 0. For CMP, it indicates a borrow. For rotate instructions, it participates in the rotation and reflects a bit shifted out.

Real-World Applications and Understanding

Logical instructions are vital for low-level programming and directly interact with hardware interfaces.

1- Bit Masking and Data Extraction:
- Using ANA/ANI: To isolate or clear specific bits. For example, to check the status of a specific sensor connected to a port, you might AND the port's value with a mask (e.g., 00000001B) to see only the status of the first bit, clearing all others.

2- Setting and Clearing Specific Bits:
- Using ORA/ORI: To force certain bits to 1. For example, to activate a device by setting a control register bit, you OR the current register value with a bitmask (e.g., 00001000B to set bit D3).
- Using XRA/XRI with a mask of all 1s (FFH): To toggle (invert) all bits in a byte.

3- Data Validation and Status Checking:
- Using CMP/CPI: To compare values without altering them, essential for conditional execution (e.g., **Is this input value within a valid range?**).
- Using XRA A (XOR Accumulator with itself): A common trick to quickly clear the Accumulator and simultaneously set the Zero flag, indicating an initial condition.

4- Bit Manipulation for Serial Communication or Graphics:
- Using Rotate instructions (RLC, RRC, RAL, RAR): For serial data transmission, where data is sent one bit at a time. Also used in algorithms for fast multiplication/division by powers of 2.

Summary of Key Points

- Logical instructions perform bit-wise operations (AND, OR, XOR, NOT, Rotate, Compare) on data, primarily using the Accumulator.
- They are fundamental for manipulating data at the individual bit level.
- ANA/ANI performs bit-wise AND, useful for masking and clearing bits.
- ORA/ORI performs bit-wise OR, useful for setting bits.
- XRA/XRI performs bit-wise XOR, useful for toggling bits or checking equality.
- CMA complements all bits in the Accumulator (Logical NOT).
- CMP/CPI compares the Accumulator with an operand, setting flags but not storing the result, crucial for conditional decision-making.
- Rotate instructions (RLC, RRC, RAL, RAR) shift bits within the Accumulator, often involving the Carry flag, important for serial data and bit manipulation.
- Flags (S, Z, P, CY) are significantly affected, providing status information for subsequent control flow decisions.
- Real-world uses include data masking, setting/clearing control bits, data validation, and serial data handling.

7.) Branching & Looping Instructions

In computer programming, instructions are typically executed sequentially, one after another, as they appear in memory. However, real-world programs often need to make decisions or repeat tasks. This is where Branching and Looping Instructions come into play, allowing the program flow to deviate from this linear execution. In 8085 Assembly Language Programming, these instructions are fundamental for creating dynamic and functional programs.

1. Introduction to Program Flow Control

- Sequential execution is the default. The Program Counter (PC) increments to the next instruction address.
- Branching and looping instructions modify the PC's value directly, altering the order of instruction execution.
- This ability to change execution path is crucial for implementing logic, conditions, and repetitions.

2. Branching Instructions

Branching instructions are used to transfer program control from one part of the program to another. This transfer can be unconditional (always happens) or conditional (happens only if a certain condition is met).

2.1. Unconditional Branching (JUMP Instructions)

- An unconditional jump instruction forces the program counter to load a new address specified in the instruction.
- Execution immediately continues from that new address, regardless of any conditions.
- Analogy: Taking a specific exit ramp from a highway, no matter what.

- 8085 Instruction: JMP (Jump)
- Format: JMP addr (e.g., JMP 2050H)
- Operation: The 16-bit address 'addr' is loaded into the Program Counter.
- Example:

```
1000H: MVI A, 05H
1002H: JMP 1008H ; Program jumps to address 1008H
1005H: MOV B, A ; This instruction will be skipped
1006H: HLT
1008H: MVI C, 0AH ; Execution starts here after the jump
100AH: HLT
```

2.2. Conditional Branching (Conditional JUMP Instructions)

- Conditional jump instructions transfer control to a new address only if a specific condition is true.
- These conditions are determined by the status of the Flag Register, which holds bits (flags) indicating the result of previous arithmetic or logical operations (e.g., Zero flag, Carry flag).
- If the condition is true, the jump occurs; otherwise, the program continues to the next sequential instruction.
- Analogy: Taking an exit ramp only if there's heavy traffic.

- 8085 Conditional Jump Instructions:
- Based on Zero Flag (Z):
- JZ addr: Jump if Zero (Z=1)
- JNZ addr: Jump if Not Zero (Z=0)
- Based on Carry Flag (CY):
- JC addr: Jump if Carry (CY=1)
- JNC addr: Jump if No Carry (CY=0)
- Based on Parity Flag (P):
- JPE addr: Jump if Parity Even (P=1)
- JPO addr: Jump if Parity Odd (P=0)
- Based on Sign Flag (S):
- JM addr: Jump if Minus (S=1, negative result)
- JP addr: Jump if Plus (S=0, positive result)

- Example (using JZ):

```
2000H: MVI A, 05H
2002H: SUI A, 05H ; A = A - 5 = 0. This sets the Zero Flag (Z=1).
2004H: JZ 200AH ; Since Z=1, program jumps to 200AH.
```

```
2007H: MVI B, 01H ; This instruction is skipped.
2009H: HLT
200AH: MVI C, 02H ; Execution starts here.
200CH: HLT
```

2.3. Subroutine Call and Return (Brief Mention)

- CALL addr and RET instructions are also a form of branching, used to implement subroutines (functions).
- CALL transfers control to a subroutine's starting address, and RET brings control back to the instruction immediately following the CALL.
- Their detailed operation involves saving the return address on the stack, which is a topic for future discussion. For now, understand them as structured forms of branching for modular programming.

3. Looping Instructions

Looping refers to the repetitive execution of a block of instructions a certain number of times or until a specific condition is met. The 8085 does not have a single **LOOP** instruction like some other processors; instead, loops are constructed using a combination of arithmetic/logical instructions, a counter, and conditional jump instructions.

- Core concept:

1. Initialize a counter.
2. Execute the block of instructions (the loop body).
3. Decrement the counter.
4. Check if the counter has reached zero (or a termination condition).
5. If not zero, jump back to repeat the loop body. If zero, exit the loop.

- Analogy: A manufacturing assembly line repeating the same set of steps for each product until the quota is met.

- Example of an 8085 Loop (Adds 01H to Accumulator 5 times):

```
3000H: MVI B, 05H ; Initialize B as loop counter (5 iterations)
3002H: MVI A, 00H ; Initialize Accumulator
3004H: LOOP_START:
3004H: ADI 01H ; Add 01H to A (loop body)
3006H: DCR B ; Decrement loop counter B
3007H: JNZ LOOP_START ; If B is not zero, jump back to LOOP_START
300AH: HLT ; Loop finishes when B becomes zero (Z flag set)
```

- Explanation:
- MVI B, 05H sets up a counter in register B for 5 iterations.
- The instructions ADI 01H and DCR B form the loop's core.
- DCR B decrements B and updates the Zero flag.
- JNZ LOOP_START checks the Zero flag. If B is not zero (Z=0), it jumps back to 3004H to repeat.
- When B becomes 0, DCR B sets Z=1, so JNZ fails, and the program proceeds to HLT.

4. Real-World Applications

- Decision Making: Conditional jumps are the basis of if-else statements in high-level languages (e.g., if a sensor reading is above a threshold, take action).
- Repetitive Tasks: Loops are used for tasks like processing arrays of data, polling input devices repeatedly, or performing calculations multiple times.
- Event Handling: Waiting for an event to occur (e.g., key press) often involves a loop that continually checks a status until the event happens.
- Error Handling: Jumping to an error routine if a certain condition (like a division by zero) is detected.

5. Summary of Key Points

• Branching and looping instructions alter the sequential flow of program execution.
• Unconditional JUMP (JMP) always transfers control to a new address.
• Conditional JUMP instructions (JZ, JNZ, JC, JNC, etc.) transfer control only if a specific flag condition is met.
• The 8085 constructs loops using a counter, a block of instructions, and a conditional jump (often JNZ or JZ).
• These instructions are fundamental for implementing decision-making and repetitive tasks in assembly language programming.

8.) Stack Instructions

Stack Instructions in 8085 Assembly Language Programming

The stack is a vital area in memory used for temporary data storage, especially during program execution flow changes like subroutine calls and interrupt handling. It operates on a Last-In, First-Out (LIFO) principle, similar to a stack of plates where the last plate added is the first one removed.

1. The Stack and Stack Pointer (SP)

• The stack is a reserved portion of RAM that functions as a temporary storage area.
• In the 8085 microprocessor, the stack grows downwards, meaning it starts from a higher memory address and expands towards lower memory addresses as data is added.
• The Stack Pointer (SP) is a dedicated 16-bit register in the 8085.
• Its sole purpose is to hold the memory address of the **top** of the stack. This is the last valid memory location where data was stored or the next available location for data.

2. Initializing the Stack Pointer (LXI SP, data16)

• Before using the stack, the Stack Pointer (SP) must be initialized to a valid 16-bit memory address within the RAM. This address typically points to the highest memory location of the allocated stack area.
• Instruction: LXI SP, data16 (Load eXtended Immediate into Stack Pointer)
• data16 is a 16-bit memory address.
• Operation: The 16-bit data (data16) is loaded directly into the Stack Pointer register.
• Example: LXI SP, CFFFH
• This instruction sets the Stack Pointer to address CFFFH. The stack will then grow downwards from this address. If the stack is allocated from C800H to CFFFH, CFFFH is the starting point (top of the stack when empty).
• Real-world: This is always the first step to set up the stack, defining where the processor should save temporary information.

3. Pushing Data onto the Stack (PUSH RP)

• The PUSH instruction is used to store the contents of a 16-bit register pair onto the stack.
• Instruction: PUSH RP (Push Register Pair)
• RP can be B (BC register pair), D (DE register pair), H (HL register pair), or PSW (Program Status Word).
• PSW refers to the Accumulator (A) and the Flag register (F) pair. The Accumulator acts as the higher-order byte, and the Flag register acts as the lower-order byte.
• Operation (for PUSH RP, e.g., PUSH B):
• The Stack Pointer (SP) is decremented by 1 (SP <- SP - 1).
• The higher-order byte of the register pair (e.g., register B) is stored at the memory address pointed to by the new SP (M[SP] <- B).
• The Stack Pointer (SP) is again decremented by 1 (SP <- SP - 1).
• The lower-order byte of the register pair (e.g., register C) is stored at the memory address pointed to by the new SP (M[SP] <- C).
• Example: If BC = 1234H and SP = CFFFH

- PUSH B
- SP becomes CFFE H. Memory location CFFE H gets 12 H (B register content).
- SP becomes CFFD H. Memory location CFFD H gets 34 H (C register content).
- The new SP points to CFFD H.
- Real-world: PUSH instructions are crucial for saving the current state of registers before a subroutine call or an interrupt service routine. This ensures that the original values are preserved and can be restored later, allowing the main program to resume correctly.

4. Popping Data from the Stack (POP RP)

- The POP instruction is used to retrieve data from the stack into a 16-bit register pair.
- Instruction: POP RP (Pop Register Pair)
- RP can be B, D, H, or PSW.
- Operation (for POP RP, e.g., POP B):
- The lower-order byte of the register pair (e.g., register C) is loaded from the memory address pointed to by SP (C <- M[SP]).
- The Stack Pointer (SP) is incremented by 1 (SP <- SP + 1).
- The higher-order byte of the register pair (e.g., register B) is loaded from the memory address pointed to by SP (B <- M[SP]).
- The Stack Pointer (SP) is again incremented by 1 (SP <- SP + 1).
- Example: If SP = CFFD H, M[CFFD H] = 34 H, M[CFFE H] = 12 H
- POP B
- C register gets 34 H from M[CFFD H]. SP becomes CFFE H.
- B register gets 12 H from M[CFFE H]. SP becomes CFFF H.
- The new BC = 1234 H and SP points to CFFF H.
- Real-world: POP instructions are used to restore the original register values after a subroutine has completed or an interrupt has been serviced, bringing the processor back to its previous state.

5. Loading Stack Pointer from H and L Registers (SPHL)

- Instruction: SPHL (SP <- HL content)
- Purpose: This instruction transfers the 16-bit content of the HL register pair to the Stack Pointer (SP).
- Operation: The content of H register is moved to the higher-order byte of SP, and the content of L register is moved to the lower-order byte of SP.
- Example: If HL = D000 H
- SPHL
- SP will now contain D000 H.
- Real-world: This instruction is less commonly used than LXI SP but can be useful when the stack's starting address needs to be calculated or determined dynamically during program execution, rather than being a fixed constant.

6. Applications and Importance of Stack Instructions

- Subroutine Management: The stack is fundamental for handling CALL and RET instructions.
- When a CALL instruction is executed, the 8085 automatically PUSHes the 16-bit address of the instruction immediately following the CALL onto the stack. This is the **return address.**
- When a RET instruction is executed, the 8085 automatically POPs this return address from the stack and loads it into the Program Counter (PC), allowing the program to resume execution from where it left off.
- Interrupt Handling: When an interrupt occurs, the processor's current state (Program Counter, and often the Accumulator and Flags via PUSH PSW) is automatically or programmatically PUSHed onto the stack. This saves the execution context so it can be restored after the interrupt service routine completes, ensuring a smooth return to the interrupted program.
- Temporary Data Storage: Programmers can explicitly use PUSH and POP to temporarily save important register values that might be altered by a segment of code, and then restore them later. This is particularly useful for complex calculations or when using common registers for multiple purposes.
- Parameter Passing: The stack can be used to pass parameters to subroutines (by pushing them before CALL) or retrieve return values (by popping after RET), though this is more explicit in modern

architectures.

7. Stack Integrity and Common Pitfalls

- Stack Overflow: Occurs if too much data is PUSHed onto the stack, causing it to grow beyond its allocated memory area and overwrite other critical program data or code. This can lead to unpredictable program behavior or crashes.
- Stack Underflow: Occurs if a POP instruction is executed when the stack is empty or if more POPs are executed than PUSHes. This can lead to loading invalid data into registers.
- Mismatch PUSH/POP Operations: It is crucial to have a balanced number of PUSH and POP operations, and that they are performed in the correct order (LIFO). If you PUSH B, D, H, you must POP H, D, B to restore the original values correctly.

Summary of Key Points:

- The stack is a LIFO memory area, managed by the 16-bit Stack Pointer (SP) register.
- In 8085, the stack grows downwards in memory.
- LXI SP, data16 initializes the Stack Pointer to define the stack's starting address.
- PUSH RP saves a 16-bit register pair (BC, DE, HL, or PSW) onto the stack, decrementing SP twice.
- POP RP retrieves a 16-bit register pair from the stack, incrementing SP twice.
- SPHL loads the SP with the content of the HL register pair, useful for dynamic stack management.
- Stack instructions are fundamental for subroutine calls (saving return addresses), interrupt handling (saving processor context), and general temporary data storage.
- Maintaining stack integrity is crucial to avoid issues like stack overflow or underflow.

9.) I/O and Machine ControlInstructions

Input/Output (I/O) and Machine Control Instructions are crucial for any microprocessor to interact with the external world and manage its own operation. In the 8085 microprocessor, these instructions allow the CPU to communicate with peripheral devices and control its internal state.

1. Understanding Input/Output (I/O) Operations

- I/O refers to the process of a computer system exchanging data with the outside world. This includes receiving data from input devices (like keyboards, sensors, switches) and sending data to output devices (like displays, motors, printers, LEDs).
- Without I/O, a microprocessor would be an isolated computing unit, unable to perform any useful task that involves interaction with a user or the physical environment.
- The 8085 uses a technique called I/O-Mapped I/O. This means that I/O devices have separate address spaces from memory locations. It uses specific instructions to access these I/O ports.

1.1. I/O Ports

- An I/O port is a hardware interface on the microprocessor or a peripheral chip that allows the CPU to send or receive data from an external device.
- Each I/O port has a unique address, typically an 8-bit address in the 8085 system. This allows for up to 256 unique input ports and 256 unique output ports.
- Think of an I/O port as a specific mailbox. The CPU writes data into an output mailbox, and the external device reads it. The external device writes data into an input mailbox, and the CPU reads it.

1.2. 8085 I/O Instructions

- These instructions facilitate data transfer between the Accumulator (A register) and an I/O port.

- IN port_address (Input from Port)
- Description: This instruction reads 8-bit data from the specified input port and loads it into the Accumulator.

• The 'port_address' is an 8-bit address (00H to FFH) that identifies the specific input device.
• Analogy: The CPU reaching into a specific input mailbox (port) and taking out the message (data) to keep it in its hand (Accumulator).
• Example: To read data from input port 20H and store it in the Accumulator:

```
IN 20H
```

(If a switch connected to port 20H is pressed, the Accumulator might receive 01H. If not pressed, 00H.)

• OUT port_address (Output to Port)
• Description: This instruction takes the 8-bit data from the Accumulator and sends it to the specified output port.
• The 'port_address' is an 8-bit address (00H to FFH) that identifies the specific output device.
• Analogy: The CPU taking a message (data) from its hand (Accumulator) and putting it into a specific output mailbox (port) for an external device to read.
• Example: To send the value in the Accumulator to output port 21H:

```
MVI A, 01H ; Load Accumulator with 01H
OUT 21H ; Send 01H to port 21H
```

(If an LED is connected to port 21H, and bit 0 controls it, the LED would turn ON.)

• Real-world Example:
• Consider a washing machine. When you press the 'Start' button, it acts as an input device, and its state is read by an IN instruction. When the machine needs to turn on a motor or a water valve, an OUT instruction is used to send a command to the corresponding output port.

2. Machine Control Instructions

• These instructions are used to control the operation of the 8085 microprocessor itself, rather than moving data or performing arithmetic. They affect the CPU's internal state or behavior.

• HLT (Halt)
• Description: This instruction stops the CPU's execution. The program counter is incremented after fetching HLT, but no further instructions are executed. The CPU enters a low-power state.
• The CPU remains halted until an external interrupt (which will be covered in future topics) or a reset signal is received.
• Use case: Used at the end of a program to prevent further execution, or to put the system into a standby mode, waiting for an event.

• NOP (No Operation)
• Description: This instruction performs no operation. It simply consumes CPU cycles (typically 4 T-states) and increments the program counter.
• Use cases:
• Creating time delays: A sequence of NOPs can introduce a precise delay.
• Code padding: Used to fill space in memory, for example, when patching code or aligning segments.
• Debugging: Temporarily replacing an instruction with NOP to bypass it without changing the program structure.

• DI (Disable Interrupts)
• Description: This instruction disables the interrupt enable flip-flop in the 8085, preventing the CPU from responding to the INTR (Interrupt Request) signal. Other interrupts (like TRAP, RST 7.5, 6.5, 5.5) are also affected depending on the mask.
• Use case: Crucial for protecting **critical sections** of code where data integrity would be compromised if an interrupt handler modified shared resources. For example, during a multi-byte data transfer, you might disable interrupts to ensure the entire transfer completes without interruption.

• EI (Enable Interrupts)
• Description: This instruction enables the interrupt enable flip-flop, allowing the CPU to respond to interrupt requests (INTR).
• Use case: Used after a critical section or during system initialization to allow the CPU to respond to external events. It's often paired with DI.

- SIM (Set Interrupt Mask)
- Description: This instruction is used to configure the interrupt mask (for RST 7.5, 6.5, 5.5 interrupts) and also controls the Serial Output Data (SOD) line of the 8085. The mask bits are loaded from the Accumulator.
- It allows specific interrupts to be selectively enabled or disabled.
- Real-world context: If you have different sensors (e.g., fire, smoke, pressure), SIM allows you to tell the 8085 to only pay attention to the fire sensor interrupt for now, ignoring others.

- RIM (Read Interrupt Mask)
- Description: This instruction reads the current status of the interrupt masks (for RST 7.5, 6.5, 5.5), the pending interrupt bits, and the state of the Serial Input Data (SID) line into the Accumulator.
- It helps in determining which interrupts are currently enabled/disabled and if any interrupts are pending.

- Real-world Applications and Importance:
- Industrial Automation: PLCs (Programmable Logic Controllers) heavily rely on I/O instructions to read sensor data and control actuators (motors, valves). Machine control instructions like DI/EI are vital for ensuring real-time responsiveness and critical operation sequences.
- Embedded Systems: From smart appliances to medical devices, I/O is the bridge to the physical world. HLT is used for power management, and NOP for precise timing.
- Human-Machine Interfaces (HMI): Keyboards, touchscreens, and displays are all I/O devices managed by these instructions.

Summary of Key Points:

- I/O instructions (IN, OUT) enable the 8085 to communicate with external hardware devices via I/O ports.
- I/O-Mapped I/O uses a separate address space for peripherals, accessed by dedicated I/O instructions.
- Machine control instructions (HLT, NOP, DI, EI, SIM, RIM) manage the internal operation and state of the 8085 CPU.
- HLT stops execution, NOP provides a delay, and DI/EI control the CPU's response to interrupts.
- SIM and RIM are advanced instructions for managing interrupt masks and serial data.
- These instructions are fundamental for building functional and responsive embedded systems that interact with the physical world.

10.) Classification of 8085 Interrupts and its priorities

1. Introduction to Interrupts

An interrupt is essentially a signal to the microprocessor, indicating that an event has occurred and requires immediate attention. It temporarily stops the CPU's current execution of the main program to handle this urgent event.

- Think of it like a doorbell ringing while you are reading a book. You pause your reading (the main program), go to answer the door (handle the event), and then return to your book exactly where you left off.
- The 8085 microprocessor uses interrupts to enhance its efficiency and responsiveness. Instead of constantly checking (polling) for external events, the CPU can focus on its main tasks until an interrupt signal informs it of something that needs attention.
- This mechanism is crucial for real-time systems, handling input from devices like keyboards, receiving data from sensors, or managing timers.

2. The Interrupt Service Routine (ISR)

When an interrupt occurs and is accepted by the CPU, the processor doesn't just randomly stop. It follows a defined procedure:

- First, the CPU completes the instruction it is currently executing.
- Then, it saves the current state of the main program, most importantly the address of the next instruction (stored in the Program Counter, PC) onto a special memory area called the Stack. Other important register values might also be saved by the programmer within the ISR.
- The CPU then jumps to a specific memory location where a special program, called the Interrupt Service Routine (ISR), is stored.
- The ISR contains the code specifically designed to handle the event that triggered the interrupt. For example, if a keyboard key was pressed, the ISR would read the key code.
- After the ISR finishes its task, the CPU restores the saved program state by retrieving the PC from the Stack.
- Finally, the CPU resumes execution of the main program from exactly where it was interrupted.

3. General Classification of 8085 Interrupts

8085 interrupts can be broadly classified into two main categories based on their origin:

- Hardware Interrupts:
- These interrupts are triggered by external devices or signals from the physical environment.
- They occur through dedicated pins on the 8085 microprocessor chip.
- Examples of 8085 hardware interrupt pins are TRAP, RST 7.5, RST 6.5, RST 5.5, and INTR.
- They are vital for responding to events like user input, data ready signals from peripheral devices, or timer overflows.

- Software Interrupts:
- These interrupts are triggered by a specific instruction within the main program itself, rather than by an external signal.
- In the 8085, these are the Restart (RST) instructions (RST 0 through RST 7). Note that these are different from the hardware RST pins.
- When an RST instruction is encountered, it behaves like a subroutine call that automatically pushes the PC onto the stack and jumps to a fixed, predefined memory address specific to that RST instruction.
- Software interrupts are often used for debugging purposes, implementing system calls, or creating software-controlled event handlers.

4. Further Classification of Hardware Interrupts

Hardware interrupts in the 8085 are further categorized based on their behavior and how they are handled:

- Maskable vs. Non-Maskable Interrupts:

- Maskable Interrupts:
- These interrupts can be selectively enabled or disabled (masked) by the programmer using specific software instructions.
- The 8085 provides instructions like EI (Enable Interrupts) and DI (Disable Interrupts) to control whether the CPU will respond to these interrupt requests.
- This allows the programmer to protect critical sections of code where an interruption might lead to data corruption or incorrect operation.
- RST 7.5, RST 6.5, RST 5.5, and INTR are maskable interrupts.

- Non-Maskable Interrupts (NMI):
- These interrupts cannot be disabled or ignored by software instructions. They are always recognized by the CPU, regardless of the interrupt enable status.
- NMIs are reserved for very critical, high-priority events that absolutely must be handled immediately to prevent system failure or data loss.
- Examples include power failure detection or major hardware errors.
- TRAP is the only non-maskable interrupt in the 8085 microprocessor.

- Vectored vs. Non-Vectored Interrupts:

- Vectored Interrupts:
- For vectored interrupts, the CPU automatically knows the starting memory address (called the vector address) of the Interrupt Service Routine (ISR) associated with that specific interrupt.
- When such an interrupt is accepted, the CPU directly jumps to this predefined vector address. This direct jump makes the interrupt response very fast.
- TRAP, RST 7.5, RST 6.5, and RST 5.5 are all vectored interrupts, each having its unique and fixed vector address in memory.

- Non-Vectored Interrupts:
- For non-vectored interrupts, the CPU does not automatically know the ISR's starting address.
- After acknowledging the interrupt, the interrupting device itself must provide the CPU with the memory address of the ISR, or an instruction (typically an RST instruction) that points to the ISR.
- This handshake process adds a small amount of overhead but offers more flexibility, as multiple devices can use the same interrupt pin (like INTR) and provide their own specific ISR addresses.
- INTR is the only non-vectored hardware interrupt in the 8085 microprocessor.

5. Priorities of 8085 Interrupts

When multiple interrupt requests arrive at the same time, or if a lower-priority interrupt is being serviced and a higher-priority one occurs, the 8085 microprocessor needs a mechanism to decide which interrupt to handle first. The 8085 has a fixed, predefined hardware priority scheme for its interrupt inputs.

- The priority order determines which interrupt will take precedence. A higher-priority interrupt can interrupt (pre-empt) a lower-priority interrupt that is currently being serviced.
- This fixed priority ensures that critical events are always attended to promptly, even if other less critical events are already in progress.

- The priority order for the 8085 hardware interrupts, from highest to lowest, is as follows:

1. TRAP (Highest priority, Non-Maskable)
2. RST 7.5 (Maskable, but has the next highest priority)
3. RST 6.5 (Maskable)
4. RST 5.5 (Maskable)
5. INTR (Lowest hardware priority among the dedicated interrupt pins, Maskable, Non-Vectored)

- This means:
- TRAP will always be handled first if it occurs simultaneously with any other interrupt. It can also interrupt any other interrupt's ISR.
- RST 7.5 can interrupt RST 6.5, RST 5.5, and INTR, but not TRAP.
- RST 6.5 can interrupt RST 5.5 and INTR, but not TRAP or RST 7.5.
- RST 5.5 can only interrupt INTR.
- INTR has the lowest priority and can be interrupted by any of the other hardware interrupts.

- Real-world analogy: Imagine an emergency response system. A fire alarm (TRAP) has the absolute highest priority and must be responded to immediately. A burglar alarm (RST 7.5) is next, followed by a smoke detector (RST 6.5), and then a simple motion sensor (INTR). Each takes precedence over those below it in the hierarchy.

6. Interrupt Handling Flow (Brief Overview)

The general sequence of events when the 8085 handles an interrupt:

- The CPU finishes the instruction it is currently executing.
- It then checks for any pending interrupt requests, evaluating their enable status (for maskable interrupts) and their hardware priority.
- If an interrupt is accepted based on its priority and mask status:
- The content of the Program Counter (PC) is automatically pushed onto the Stack to save the return address of the main program.

• For maskable interrupts, the CPU's internal Interrupt Enable flip-flop is typically reset (disabling further maskable interrupts) to prevent nested interrupts from occurring immediately.
• The CPU then fetches the vector address (for vectored interrupts) or receives an instruction from the interrupting device (for non-vectored INTR) to determine the starting address of the Interrupt Service Routine (ISR).
• The CPU jumps to the memory location of the ISR and begins executing it.
• Within the ISR, the programmer typically saves any important CPU registers that might be modified and then executes the code to handle the event.
• After completing its task, the ISR restores the saved registers.
• The programmer must explicitly use the EI (Enable Interrupts) instruction if they wish to allow further maskable interrupts to be processed.
• Finally, a RET (Return) instruction is executed at the end of the ISR. This instruction pops the saved PC value from the Stack back into the Program Counter.
• The CPU then resumes execution of the main program from where it was originally interrupted.

7. Summary of Key Points

• Interrupts are signals that cause the CPU to temporarily halt its main program and respond to an urgent event.
• They improve system efficiency and responsiveness by avoiding constant polling.
• The 8085 classifies interrupts into Hardware (external pins) and Software (RST instructions).
• Hardware interrupts are further categorized as Maskable (can be enabled/disabled by software) or Non-Maskable (always recognized, e.g., TRAP).
• They are also classified as Vectored (CPU knows ISR address) or Non-Vectored (device provides ISR address, e.g., INTR).
• The 8085 has a fixed hardware priority scheme for its interrupts: TRAP > RST 7.5 > RST 6.5 > RST 5.5 > INTR.
• An Interrupt Service Routine (ISR) is a dedicated code segment that handles the specific event, after which the main program resumes.

11.) 8085 Vectored interrupts: TRAP, RST7.5, RST 6.5, RST 5.5 and RST Instruction

Vectored Interrupts in 8085 Microprocessor

The 8085 microprocessor uses interrupts to handle events that require immediate attention. When an interrupt occurs, the CPU temporarily stops its current task, services the interrupt, and then resumes the original task. Vectored interrupts are a specific type where the CPU automatically knows the starting memory address of the Interrupt Service Routine (ISR) associated with that interrupt. This means the interrupting device does not need to provide the address.

Here's how the 8085 handles a vectored interrupt:
• The CPU finishes its current instruction execution.
• It saves the address of the next instruction (Program Counter value) onto the stack.
• It disables further maskable interrupts (TRAP is non-maskable and not affected).
• It then automatically jumps to a predefined memory location, known as the vector address.
• The ISR stored at this vector address is executed.
• The ISR ends with a RET (return from subroutine) instruction, which retrieves the saved PC from the stack and returns control to the main program.

The 8085 has five vectored interrupt inputs: TRAP, RST 7.5, RST 6.5, and RST 5.5. Additionally, the RST instruction acts as a software-vectored interrupt. These interrupts have a fixed priority order, with TRAP having the highest priority.

1. TRAP Interrupt

- Pin Name: TRAP
- Type: Non-maskable and edge/level sensitive. This means it cannot be disabled by software and responds to both a rising edge and a high level.
- Priority: Highest among all 8085 interrupts.
- Vector Address: 0024H.
- Characteristics:
- It is the only non-maskable interrupt, meaning it cannot be ignored by the CPU once activated. This makes it crucial for critical system failures.
- It is triggered by a rising edge and remains active as long as the input is high. However, to prevent re-triggering, the signal must go low and then high again for a new interrupt.
- Real-world use:
- Used for critical events like power failure detection or system reset. For example, if the system detects an impending power loss, TRAP can be triggered to save vital data to non-volatile memory before the power completely fails, ensuring data integrity.
- It can also be used for emergency shutdowns in industrial control systems.

2. RST 7.5 Interrupt

- Pin Name: RST 7.5
- Type: Maskable and rising edge sensitive. It can be enabled or disabled by software and responds only to a positive transition (low to high).
- Priority: Second highest among vectored interrupts (after TRAP).
- Vector Address: 003CH.
- Characteristics:
- It's a maskable interrupt, meaning its operation can be controlled using the EI (Enable Interrupts) and DI (Disable Interrupts) instructions, along with the SIM (Set Interrupt Mask) instruction.
- It has an internal flip-flop that is set by the rising edge, even if interrupts are disabled. This flip-flop ensures the interrupt request is remembered and processed once interrupts are enabled.
- Real-world use:
- Often used for time-sensitive events like timer overflows. For instance, a real-time clock chip might trigger RST 7.5 when a specific time interval elapses, signaling the CPU to update a display or perform a scheduled task.
- Could be used for critical sensor inputs where immediate action is needed, but the system designer wants the option to temporarily ignore it.

3. RST 6.5 Interrupt

- Pin Name: RST 6.5
- Type: Maskable and level sensitive. It can be enabled or disabled by software and responds as long as the input is high.
- Priority: Third highest among vectored interrupts.
- Vector Address: 0034H.
- Characteristics:
- Like RST 7.5, it is maskable via EI/DI and SIM instructions.
- It is level-triggered; the interrupt is active as long as the RST 6.5 pin is held high. The CPU will continuously request the interrupt if the signal remains high and interrupts are enabled.
- Real-world use:
- Suitable for status monitoring where a peripheral indicates it needs attention, and it holds the line high until serviced. For example, a printer signaling that its buffer is full or a device requesting more data.
- Could be used for general-purpose input signals from external devices that need CPU intervention.

4. RST 5.5 Interrupt

- Pin Name: RST 5.5
- Type: Maskable and level sensitive. It can be enabled or disabled by software and responds as long as the input is high.
- Priority: Fourth highest among vectored interrupts.
- Vector Address: 002CH.

- Characteristics:
- It shares characteristics with RST 6.5, being maskable and level-triggered.
- Its masking and enabling are also controlled by the SIM and RIM (Read Interrupt Mask) instructions.
- Real-world use:
- Similar to RST 6.5, it is used for general-purpose external events or peripheral service requests that are level-triggered.
- For example, an external keypad controller indicating a key press, or a data acquisition system signaling new data is available for processing.

5. RST Instruction

- Type: Software Interrupt. It is an instruction executed within a program, not triggered by a hardware signal.
- Format: RST n, where 'n' is a number from 0 to 7.
- Vector Address: 8 * n. The CPU jumps to this calculated address.
- RST 0: 0000H
- RST 1: 0008H
- RST 2: 0010H
- RST 3: 0018H
- RST 4: 0020H
- RST 5: 0028H
- RST 6: 0030H
- RST 7: 0038H
- Characteristics:
- Unlike hardware interrupts, RST instructions are initiated by the program itself.
- When an RST instruction is encountered, the CPU behaves similarly to a hardware interrupt: it pushes the current PC onto the stack and jumps to the calculated vector address.
- Functionally, RST is similar to a CALL instruction to a fixed address but takes less memory (1 byte) and executes faster.
- Real-world use:
- System calls: Operating systems or monitor programs use RST instructions to provide a set of services (e.g., input/output routines, memory management) that user programs can invoke.
- Debugging: Debuggers might insert RST instructions at specific points in a program to trigger a breakpoint, allowing the programmer to inspect registers and memory.
- Subroutine calls to frequently used routines, especially in ROM-based systems where fixed entry points are beneficial.

Summary of Key Points:
- Vectored interrupts in 8085 automatically direct the CPU to a fixed, predefined memory location for their service routine.
- TRAP is the highest priority, non-maskable, and edge/level triggered, for critical system events.
- RST 7.5 is maskable and edge-triggered, good for time-sensitive tasks.
- RST 6.5 and RST 5.5 are maskable and level-triggered, suitable for general-purpose peripheral requests.
- The RST instruction is a software-initiated interrupt, behaving like a fast CALL to a fixed vector address, used for system calls or debugging.
- All vectored interrupts save the Program Counter on the stack and disable further maskable interrupts before jumping to their respective vector addresses.
- An Interrupt Service Routine always ends with a RET instruction to return control to the main program.

12.) 8085 Non-Vectored Interrupts: INTR

The 8085 microprocessor features several interrupt inputs to allow external devices to request its attention. Among these, the INTR (Interrupt Request) pin stands out as the only non-vectored interrupt.

Understanding INTR is crucial for designing systems where peripherals need flexible ways to interact with the CPU.

1. What is INTR?
INTR is a general-purpose hardware interrupt request line on the 8085 microprocessor. It is an active HIGH, level-triggered input. When an external device wants to interrupt the CPU, it asserts a HIGH signal on the INTR pin.

2. Distinction: Non-Vectored
- Unlike vectored interrupts (TRAP, RST7.5, RST6.5, RST5.5), where the 8085 automatically jumps to a predefined memory location (vector address) upon receiving the interrupt, INTR does not have such a fixed address.
- **Non-vectored** means the CPU does not intrinsically know where to go to service the interrupt. Instead, the external interrupting device must provide this information to the CPU.

3. Key Characteristics of INTR
- Maskable: The INTR interrupt can be enabled or disabled by software using the EI (Enable Interrupts) and DI (Disable Interrupts) instructions. If interrupts are disabled, the CPU will ignore INTR requests.
- Lowest Priority: Among all 8085 interrupts, INTR has the lowest priority. If multiple interrupts occur simultaneously, higher priority interrupts (like TRAP or RST7.5) will be serviced first.
- Level-Triggered: The INTR line must remain HIGH until acknowledged by the CPU. If it goes LOW before acknowledgment, the interrupt request might be missed.

4. The INTR Handshake Process
Servicing an INTR request involves a special handshake mechanism between the 8085 CPU and the interrupting peripheral:

- a. Peripheral Request: An external device needing CPU attention asserts a HIGH signal on the 8085's INTR pin.
- b. CPU Checks: The 8085 continuously monitors its INTR line. If INTR is HIGH and interrupts are enabled (via EI instruction), the CPU completes its current instruction.
- c. PC Save: The 8085 saves the address of the next instruction (Program Counter, PC) onto the stack. This is crucial for returning to the main program later.
- d. Interrupt Acknowledge (INTA): The 8085 then sends out an active LOW signal on its INTA (Interrupt Acknowledge) pin. This signal informs the interrupting peripheral that its request has been accepted and the CPU is ready to receive instructions.
- e. Instruction Provision by Peripheral: Upon receiving the INTA signal, the external peripheral must place an 8-bit instruction on the 8085's data bus (AD7-AD0). This instruction tells the 8085 where to go for the Interrupt Service Routine (ISR).
- Typically, this instruction is an RST n (Restart) instruction (e.g., RST 0, RST 1, ..., RST 7) or a CALL instruction (e.g., CALL 4000H).
- f. CPU Executes Provided Instruction: The 8085 reads the 8-bit instruction from the data bus and executes it as if it were part of its normal program flow.
- If it's an RST n instruction: The CPU calculates the vector address (n * 8) and jumps to that specific memory location to execute the ISR. For example, if RST 3 (opcode C7H) is provided, the CPU jumps to address 0018H (3 * 8).
- If it's a CALL instruction: The CPU pushes the current PC onto the stack (again, for return) and then jumps to the 16-bit address specified by the CALL instruction. This requires two more INTA cycles to fetch the 16-bit address bytes.
- g. ISR Execution: The Interrupt Service Routine (ISR) associated with the interrupting device then executes. This routine performs the necessary tasks to handle the peripheral's request.
- h. Return to Main Program: The ISR must end with a RET (Return from Subroutine) instruction. This instruction pops the saved PC value from the stack back into the Program Counter, allowing the 8085 to resume execution of the main program from where it was interrupted.

5. Role of External Hardware
- Since INTR is non-vectored, the responsibility of providing the correct RST or CALL instruction lies with external hardware, often an interrupt controller chip (like the 8259A, though not directly with 8085, the concept applies) or custom logic.

- This external logic determines which device interrupted and then places the appropriate instruction on the data bus during the INTA cycle.

6. Advantages of INTR
   - Flexibility: Multiple external devices can share the single INTR line. The external hardware can arbitrate these requests and provide different RST or CALL instructions, directing the CPU to different ISRs for each device.
   - Scalability: Allows for a larger number of interrupt sources than the limited number of fixed-vector interrupts.
   - Dynamic Vectoring: The **vector** (target address) is not fixed but is dynamically provided by the interrupting device, offering greater control.

7. Disadvantages of INTR
   - Requires External Hardware: Implementing INTR requires additional external logic to generate the RST/CALL instruction, which increases hardware complexity and cost.
   - Slower Response: The handshake process (INTA, fetching instruction) makes it slightly slower to respond compared to fixed-vector interrupts like TRAP.

8. Real-World Example/Application
   - Keyboard Input: Imagine a system where multiple keys can be pressed. When a key is pressed, it might assert INTR. External logic then identifies which key was pressed and sends an RST instruction corresponding to that key's handling routine.
   - Printer Ready Signal: A printer might assert INTR when it's ready to receive more data. The external logic sends an RST instruction that directs the CPU to a routine to send the next batch of data to the printer.
   - Data Acquisition Systems: Multiple sensors might need to interrupt the CPU when they have new data. Using INTR and external decoding logic, each sensor can effectively direct the CPU to its specific data processing routine.

Summary of Key Points:
- INTR is the 8085's maskable, lowest-priority, level-triggered, non-vectored interrupt.
- **Non-vectored** means the CPU relies on external hardware to provide the address of the Interrupt Service Routine (ISR).
- The process involves a handshake: Peripheral asserts INTR -> 8085 acknowledges with INTA -> Peripheral provides an 8-bit instruction (usually RST n or CALL addr) on the data bus.
- The 8085 executes this provided instruction to jump to the ISR.
- It offers flexibility and scalability but requires external hardware logic.
- Essential for systems requiring dynamic interrupt handling from various peripherals.