Notes on: 8085 Microprocessor

1.) Introduction to Microprocessors

Introduction to Microprocessors

The **microprocessor** is a foundational concept in computer engineering, representing the heart of nearly every digital device we interact with today. To understand complex computer systems, it's essential to grasp what a microprocessor is, how it works, and its role within a larger system. Our focus here will be on the core principles, often illustrated by the Intel 8085, a classic 8-bit microprocessor that serves as an excellent pedagogical tool for these fundamentals.

1- What is a Microprocessor?

• A microprocessor, often abbreviated as MPU, is essentially a Central Processing Unit (CPU) fabricated on a single integrated circuit (IC) chip. Historically, CPUs occupied multiple circuit boards or even entire rooms. The advent of integrated circuit technology, particularly Large Scale Integration (LSI) and Very Large Scale Integration (VLSI), made it possible to condense all the essential functions of a CPU onto a single silicon chip.

• Its primary function is to execute instructions, perform arithmetic and logical operations, and manage the flow of data within a computer system. Think of it as the 'brain' of any digital device, responsible for carrying out all computations and coordinating all activities.

• The term **micro** in microprocessor refers to its physical size, being a single, small chip, rather than its processing power, which can be immense in modern microprocessors. It is a general-purpose programmable device that reads binary instructions from memory, accepts binary data as input, processes the data according to the instructions, and provides results as output.

• The fundamental architecture of a microprocessor is based on Von Neumann architecture or Harvard architecture principles, where instructions and data are processed sequentially. It achieves its versatility by being programmable, meaning its behavior can be altered by changing the sequence of instructions (the software program) it executes.

2- Brief Historical Context for the 8085

• The journey to the single-chip microprocessor began with early computers built from vacuum tubes, which were massive and consumed enormous power. The invention of the transistor and subsequently the integrated circuit revolutionized electronics, allowing for significant miniaturization and increased complexity.

• Intel introduced the first commercially available microprocessor, the 4004, in 1971, which was a 4-bit processor. This was followed by the 8-bit 8008 and then the highly successful 8080 in 1974. The 8080 became a standard for many early personal computers.

• The Intel 8085, introduced in 1976, was a significant advancement over the 8080. It maintained backward compatibility with the 8080's instruction set but featured several key enhancements. Notably, it required only a single +5V power supply, simplifying system design compared to the 8080's three power supplies. It also integrated some clock generation circuitry on-chip, reducing the need for external components.

• The 8085 is an 8-bit microprocessor, meaning it can process 8 bits of data at a time. It was manufactured using N-MOS (N-type Metal-Oxide-Semiconductor) technology and typically operated at clock speeds around 3 MHz. Despite its age, the 8085 remains a popular choice for studying microprocessor fundamentals due to its relatively simple yet complete architecture, which clearly illustrates core concepts without the overwhelming complexity of modern processors. It provided a

capability to address up to 64 KB of memory, a substantial amount for its time.

3- Fundamental Components of a Generic Microprocessor

Every microprocessor, regardless of its specific design or generation, comprises several essential functional blocks that work in concert to execute programs. These blocks are primarily sequential logic circuits.

- 1- Arithmetic Logic Unit (ALU):
- The ALU is the digital circuit responsible for performing all arithmetic and logical operations. It is essentially the **calculator** of the microprocessor.
- Arithmetic operations include addition, subtraction, increment (adding one), and decrement (subtracting one). More complex operations like multiplication and division are often implemented by repeating these basic operations or using dedicated hardware for faster execution in more advanced processors.
- Logical operations include AND, OR, NOT, and XOR. These operations are fundamental for bit manipulation, decision-making, and control within programs.
- The ALU typically takes two input operands (data values) and an opcode (operation code) from the Control Unit, then performs the specified operation and outputs a result, often accompanied by status flags that indicate characteristics of the result (e.g., if the result was zero, negative, or if a carry occurred). These operands usually come from internal registers.

- 2- Registers:
- Registers are small, high-speed memory locations located directly within the microprocessor. They are the fastest form of memory available to the CPU and are crucial for temporary storage of data, addresses, and control information during the execution of instructions. Accessing data from registers is significantly faster than accessing external memory.
- Different types of registers serve specific purposes:
- General Purpose Registers: These are used by the programmer to store temporary data during program execution. They provide flexible storage for intermediate results or operands.
- Program Counter (PC): This is a special-purpose register that always holds the memory address of the next instruction to be fetched from memory. It is critical for maintaining the sequential flow of a program. After an instruction is fetched, the PC is typically incremented to point to the subsequent instruction.
- Instruction Register (IR): After an instruction is fetched from memory, it is loaded into the Instruction Register. This register holds the instruction while it is being decoded and executed by the Control Unit.
- Status Register (or Flag Register): This register contains individual bits (flags) that are set or cleared based on the result of an arithmetic or logical operation performed by the ALU. For example, a zero flag might be set if an operation results in zero, or a carry flag if an addition operation generates a carry. These flags are essential for conditional branching in programs.

- 3- Control Unit (CU):
- The Control Unit is the orchestrator or **traffic cop** of the microprocessor. Its primary responsibility is to fetch instructions from memory, decode them, and generate the necessary control signals to execute the instructions.
- It interprets the instruction currently held in the Instruction Register, determining what operation needs to be performed and which components (ALU, registers, memory, I/O devices) are involved.
- Based on this interpretation, the Control Unit generates precise timing and control signals. These signals dictate when data should be moved, when the ALU should perform an operation, when memory should be read or written, and generally synchronize all operations within the microprocessor and between the microprocessor and external components. It ensures that all operations occur in the correct sequence and at the right moment.

4- How a Microprocessor Works - The Instruction Cycle

A program executed by a microprocessor is essentially a sequence of instructions stored in memory. The microprocessor continuously performs a fundamental three-step process known as the Instruction Cycle (or Fetch-Decode-Execute cycle) for each instruction:

- 1- Fetch:

• The cycle begins with the Control Unit using the address stored in the Program Counter (PC) to locate the next instruction in memory.
• The instruction (a specific binary code) is then read from that memory location and transferred via the data bus into the microprocessor, specifically into the Instruction Register (IR).
• Immediately after fetching, the Program Counter is typically incremented to point to the memory address of the next instruction in the sequence, preparing for the next fetch cycle.

• 2- Decode:
• Once the instruction is in the Instruction Register, the Control Unit takes over. It analyzes and interprets the binary code of the instruction.
• During this phase, the Control Unit determines what operation the instruction specifies (e.g., add, move data, jump to another part of the program) and identifies any operands (data or memory addresses) that the instruction requires. It essentially translates the binary instruction into a series of micro-operations.

• 3- Execute:
• Based on the decoded instruction, the Control Unit generates the appropriate timing and control signals. These signals direct the other components of the microprocessor (like the ALU and registers) and external devices (like memory or I/O) to perform the specified action.
• If the instruction is an arithmetic or logical operation, the Control Unit instructs the ALU to perform that operation using data from specified registers or memory locations.
• If it's a data transfer instruction, data is moved between registers, or between a register and memory, or between a register and an input/output device.
• If it's a control flow instruction (like a jump or branch), the Program Counter might be updated to a new address, altering the sequence of instruction execution.
• The result of the execution is then stored in a designated register or memory location, and the Status Register's flags are updated accordingly.
• This Fetch-Decode-Execute cycle repeats continuously, processing one instruction after another, until a halt instruction is encountered, or the power is turned off.

5- Interacting with the Outside World - The System Bus Concept

A microprocessor cannot operate in isolation. It needs to communicate with external components like memory (to store programs and data) and Input/Output (I/O) devices (to interact with the user or other systems). This communication is facilitated by a set of electrical pathways known as the System Bus. Think of the system bus as a set of highways connecting the microprocessor to its peripheral components. The system bus is generally divided into three main types:

• 1- Address Bus:
• The address bus is a unidirectional set of electrical lines. This means information flows only in one direction: from the microprocessor to memory or I/O devices.
• Its purpose is to carry the memory address or the I/O port address that the microprocessor wants to access for a read or write operation. When the microprocessor needs to fetch an instruction or read/write data, it places the specific address of the desired location onto the address bus.
• The width of the address bus (i.e., the number of individual lines) determines the maximum amount of memory or I/O locations the microprocessor can directly access. For example, if a microprocessor has an N-bit address bus, it can address 2^N unique memory locations. An 8-bit microprocessor like the 8085 typically features a 16-bit address bus, allowing it to address 2^16 = 65,536 (64 KB) unique memory locations.

• 2- Data Bus:
• The data bus is a bidirectional set of electrical lines. This means information can flow in both directions: from the microprocessor to memory/I/O (for writing data) and from memory/I/O to the microprocessor (for reading data).
• Its purpose is to carry the actual data that is being transferred between the microprocessor and memory or I/O devices. When the microprocessor fetches an instruction or reads data, the data travels from memory/I/O to the microprocessor via the data bus. When the microprocessor writes data, it travels from the microprocessor to memory/I/O via the data bus.
• The width of the data bus (e.g., 8-bit, 16-bit, 32-bit) determines how much data can be transferred in

a single operation. An 8-bit microprocessor has an 8-bit data bus, meaning it can transfer 8 bits (one byte) of data at a time.

• 3- Control Bus:
• The control bus is a collection of various individual control signals, which can be unidirectional or bidirectional.
• These signals are generated by the Control Unit of the microprocessor and are used to manage, synchronize, and regulate all operations within the system. They provide timing information and indicate the type of operation currently being performed.
• Examples of control signals include:
• Read (RD) signal: Indicates that the microprocessor wants to read data from memory or an I/O device.
• Write (WR) signal: Indicates that the microprocessor wants to write data to memory or an I/O device.
• Memory/IO Request signal: Differentiates between a memory access and an I/O access.
• Clock signal: Provides the timing pulse for all operations.
• Reset signal: Initializes the microprocessor to a known starting state.
• Interrupt Request signal: A signal from an I/O device requesting attention from the microprocessor.
• The control bus ensures that all components operate in a coordinated and synchronized manner, preventing conflicts and ensuring proper data flow.

6- Microprocessor vs. Microcontroller vs. CPU

While often used interchangeably in casual conversation, these terms have distinct meanings in computer organization:

• 1- CPU (Central Processing Unit): This is a general term for the electronic circuitry within a computer that executes instructions comprising a computer program. Historically, a CPU could be composed of many discrete components or multiple integrated circuits. It represents the core computational and control logic.

• 2- Microprocessor (MPU): A microprocessor is a CPU that is entirely contained on a single integrated circuit chip. It provides the core processing capability but typically requires external support chips to form a complete functional system. These external chips include memory (RAM and ROM/Flash), I/O interfaces (for peripherals), and often a clock generator. Examples include the Intel 8085, 8086, Pentium series, and AMD Ryzen processors. They are designed for general-purpose computing where flexibility and computational power are paramount.

• 3- Microcontroller (MCU): A microcontroller is a complete small computer on a single integrated circuit. Unlike a microprocessor, a microcontroller integrates a CPU (often a simpler one), a small amount of memory (both volatile RAM and non-volatile ROM/Flash for program storage), and various input/output peripherals (like timers, serial communication interfaces (UART, SPI, I2C), Analog-to-Digital Converters (ADCs), Digital-to-Analog Converters (DACs)) all onto a single chip. Microcontrollers are specifically designed for embedded applications, where they perform dedicated tasks with real-time constraints, often in devices that are not considered **computers** in the traditional sense, such as washing machines, remote controls, automotive systems, and IoT devices. They prioritize cost-effectiveness, low power consumption, and compact size over raw computational power.

• The key distinction lies in integration: An MPU is primarily the processor itself, needing external components to become a system. An MCU is a **system-on-a-chip** with most essential components already integrated.

7- The 8085 Microprocessor - An Introductory Perspective

The Intel 8085 microprocessor holds a significant place in the history of computing and especially in computer engineering education. It is chosen for study not because it is a cutting-edge processor today, but because it elegantly encapsulates the fundamental principles of microprocessor operation in a manageable way.

• As an 8-bit microprocessor, the 8085 processes data in chunks of 8 bits. This relatively small data

width simplifies the understanding of data manipulation and memory organization. It operates using N-MOS technology and requires a single +5V power supply, which was a notable improvement at the time.

• The 8085's architecture and operation provide a clear, hands-on model for understanding the fetch-decode-execute cycle, the roles of different registers, the function of the ALU and Control Unit, and how the microprocessor interacts with memory and I/O devices via the system buses. Its design allows students to trace data flow and control signals without being overwhelmed by the complexities found in modern multi-core, pipelined, cached processors.

• By studying the 8085, one gains a solid conceptual foundation for more advanced topics in computer architecture. It offers practical insights into how a CPU addresses memory, reads and writes data, responds to external signals, and executes a program step-by-step. Its relatively simple instruction set and addressing modes make it easier to grasp the core concepts of assembly language programming and machine-level control. It lays the groundwork for understanding the role of hardware in executing software and provides the 'why' behind many abstract computer science concepts.

Summary of Key Points:

• A microprocessor is a CPU on a single integrated circuit, serving as the brain of a digital system by executing instructions and processing data.
• The 8085 is an 8-bit microprocessor from 1976, studied for its clear demonstration of fundamental concepts.
• Core internal components include the Arithmetic Logic Unit (ALU) for calculations, various Registers for temporary storage (like Program Counter, Instruction Register), and the Control Unit (CU) for orchestration and timing.
• Microprocessors operate through a continuous Fetch-Decode-Execute cycle, retrieving instructions from memory, interpreting them, and then carrying out the specified operations.
• Communication with external memory and I/O devices occurs via the System Bus, which comprises the Address Bus (for location), Data Bus (for data transfer), and Control Bus (for synchronization and type of operation).
• A Microprocessor (MPU) is primarily the processing unit requiring external components, while a Microcontroller (MCU) integrates CPU, memory, and I/O peripherals onto a single chip for embedded applications.
• The 8085's relative simplicity makes it an ideal platform for students to gain a detailed understanding of how a general-purpose processor functions at its most fundamental level.

2.) Evolution of Microprocessors

The Evolution of Microprocessors

The journey of microprocessors is a fascinating tale of continuous innovation, miniaturization, and increasing complexity, driven by the relentless demand for more computing power. A microprocessor, at its core, is the Central Processing Unit (CPU) of a computer, fabricated on a single integrated circuit (IC). Its evolution reflects fundamental shifts in computer architecture, manufacturing technology, and design philosophy, directly impacting how we interact with technology today. Understanding this evolution, especially leading up to key devices like the 8085, provides critical context for your studies in Computer Organization and Architecture.

1. The Genesis: Pre-Microprocessor Era

Before microprocessors, CPUs were built using discrete components like vacuum tubes, then individual transistors, and later Small-Scale and Medium-Scale Integrated Circuits (SSI/MSI) containing a few logic gates. These systems were large, expensive, consumed significant power, and were complex to design and maintain. The invention of the transistor and later the integrated circuit laid the groundwork for miniaturization, paving the way for the single-chip CPU.

2. First Generation: 4-bit Microprocessors (Early 1970s)

The primary goal was to put the entire CPU onto a single chip. This was a monumental technological leap.

• Intel 4004 (1971): Often credited as the world's first commercial microprocessor.
• Characteristics: It was a 4-bit processor, meaning it could process data in chunks of 4 bits at a time. It could address only 640 bytes of memory. It had 2300 transistors and operated at a clock speed of up to 740 kHz.
• Reasoning for Design: It was initially designed for Busicom's calculator, demonstrating the feasibility of integrating a complex CPU on one chip. Its limited word size was due to the technology constraints of the time – it was difficult to put more transistors on a single chip reliably.
• Impact: While basic, it proved that a single chip could perform CPU functions, opening doors for embedded control applications beyond simple calculators.

• Other early 4-bit processors: Rockwell PPS-4, National Semiconductor IMP-4. These processors were mainly used in niche applications like embedded controllers, traffic light controllers, and very basic automation systems.

3. Second Generation: 8-bit Microprocessors (Mid-1970s)

The limitations of 4-bit processors – primarily their small memory addressing capabilities and limited data processing size – quickly became apparent. There was a strong need for more powerful processors for broader applications.

• Intel 8008 (1972): An early 8-bit processor, developed for Datapoint terminals.
• Characteristics: Could process 8 bits of data at once and address 16 KB of memory. It still required many external support chips and operated at a relatively low speed.
• Reasoning for Design: Building on the 4004, the 8008 doubled the data bus width, which immediately allowed for more complex instructions and processing of larger data types, making it suitable for early terminal control.

• Intel 8080 (1974): A significant milestone, becoming an industry standard.
• Characteristics: An enhanced 8-bit processor capable of addressing 64 KB of memory. It had a much more robust instruction set than the 8008 and was significantly faster. It, however, required three different power supplies (+5V, -5V, +12V) and external clock generation.
• Reasoning for Design: The 8080 aimed to address the shortcomings of the 8008, offering a much more powerful and flexible architecture. Its increased address space was crucial for developing more sophisticated programs and operating systems (like CP/M). Its instruction set provided more powerful operations, reducing the number of instructions needed for common tasks.

• The Intel 8085 (1976): A refinement and improvement over the 8080, becoming a highly popular 8-bit processor, especially for embedded systems and educational purposes.
• Characteristics: The 8085 was still an 8-bit processor, but it introduced several key advancements:
• Single +5V Power Supply: This significantly simplified system design and reduced cost compared to the 8080's multiple power requirements.
• On-chip Clock Generator: It integrated the clock circuitry, only requiring an external crystal or RC network. This further reduced external components and simplified design.
• Increased Instruction Set: It was binary compatible with the 8080, meaning programs written for the 8080 could run on the 8085, but it added two new instructions (RIM and SIM) for serial communication and interrupt mask control.
• Multiplexed Address/Data Bus: To reduce pin count, the lower 8 bits of the address bus and the data bus were multiplexed. This was a design trade-off to allow for more control pins on the 40-pin package.
• Reasoning for Design: The 8085's design was driven by the need for a more integrated, simpler, and cost-effective 8-bit solution. It was intended to make microprocessor-based systems easier to design and more accessible, directly addressing the practical challenges faced with the 8080. Its features like on-chip clock and single power supply were direct responses to market demands for ease of use and reduced bill of materials.

• Impact: The 8085 became a cornerstone in many industrial and educational applications. Its simpler hardware requirements and robust instruction set made it an excellent choice for learning about microprocessor architecture and interfacing, laying the foundation for understanding more complex systems.

• Other 8-bit processors: Zilog Z80 (a highly successful 8080-compatible processor), Motorola 6800, MOS Technology 6502 (used in Apple II, Commodore 64). These processors also expanded the reach of computing into personal computers and consumer electronics.

4. Third Generation: 16-bit Microprocessors (Late 1970s - Early 1980s)

As applications became more complex, requiring larger programs and more data, the 64 KB memory limit of 8-bit processors became a bottleneck. The move to 16-bit addressed this by allowing significantly larger address spaces and processing larger data words.

• Intel 8086 (1978): A pivotal processor that started the x86 architecture, still dominant today.
• Characteristics: A 16-bit processor, capable of addressing 1 MB of memory (a huge jump from 64 KB). It introduced segmented memory architecture and had a more powerful instruction set with hardware multiply/divide. It still used a 20-bit address bus.
• Reasoning for Design: The 8086 was designed to cater to the growing demand for more powerful personal computers and workstations. The 1MB address space allowed for much larger programs and data sets. The segmented memory scheme was a creative way to extend addressing capabilities beyond 64KB using a 16-bit internal architecture.
• Intel 8088: A variant of the 8086 with an external 8-bit data bus, used in the original IBM PC to reduce system cost.

• Motorola 68000 (1979): Another very influential 16-bit processor (internally 32-bit), used in early Apple Macintosh and Amiga computers.

• Key characteristics of this generation: Wider data buses, larger address buses, more sophisticated instruction sets, beginning of concepts like pipelining (fetching next instruction while current one executes), and dedicated math coprocessors.

5. Fourth Generation: 32-bit Microprocessors (Mid-1980s - Early 1990s)

The 16-bit processors, while powerful, still had limitations regarding direct memory addressing for very large programs and multitasking efficiency. The shift to 32-bit was driven by the need for true multitasking operating systems, virtual memory, and more complex applications like graphical user interfaces (GUIs).

• Intel 80386 (1985): The first true 32-bit Intel processor.
• Characteristics: Full 32-bit data and address buses, enabling direct addressing of up to 4 GB of physical memory. It introduced protected mode, virtual memory management (allowing programs to think they have more memory than physically available), and advanced pipelining.
• Reasoning for Design: The 80386 was designed to be a mainframe-on-a-chip, supporting advanced operating systems like UNIX and Windows. Protected mode and virtual memory were crucial for robust multitasking and memory protection, allowing multiple programs to run concurrently without interfering with each other.

• Other 32-bit processors: Motorola 68020/030/040, early RISC (Reduced Instruction Set Computer) processors like SPARC and MIPS.

• Key characteristics: Full 32-bit architecture, virtual memory support, on-chip cache memory (small, fast memory for frequently accessed data), significantly deeper pipelining, and the rise of RISC architecture alongside CISC (Complex Instruction Set Computer).

6. Fifth Generation and Beyond: 64-bit and Multi-core Processors (Mid-1990s - Present)

This generation is characterized by a massive increase in clock speeds, transistor counts, and

architectural complexity, pushing towards parallel processing and enhanced multimedia capabilities.

- Intel Pentium series (1993 onwards) and AMD Athlon/Opteron.
- Characteristics: Initial 64-bit data path, then full 64-bit architecture (e.g., AMD Athlon 64, Intel EM64T), very deep pipelines, multi-level cache hierarchies (L1, L2, L3), instruction-level parallelism (superscalar execution), branch prediction, and specialized instruction sets (like MMX, SSE for multimedia).
- Multi-core processors: A significant shift from increasing clock speed to integrating multiple processing cores on a single chip, allowing true parallel execution of tasks.
- Reasoning for Design: The demand for faster multimedia processing, complex scientific simulations, and highly interactive software drove these advancements. The physical limits of increasing clock speed led to architectural innovations like multi-core designs to enhance performance through parallelism. Energy efficiency also became a critical design consideration.

- Key characteristics: 64-bit computing, multi-core architecture, hyper-threading, advanced power management, integration of graphics processing units (GPUs) and other specialized accelerators onto the CPU die, vast transistor counts (billions), and sophisticated memory controllers.

Summary of Key Evolutionary Trends:

- Increased Word Size (4-bit -> 8-bit -> 16-bit -> 32-bit -> 64-bit): Directly impacts the amount of data processed per cycle and the range of numbers that can be handled.
- Expanded Memory Addressing Capabilities: From bytes to gigabytes, enabling larger programs and more complex data structures. The 8085's 64 KB was significant for its time, but quickly outgrown.
- Higher Clock Speeds: Leading to faster execution of instructions.
- Miniaturization and Integration: More transistors on a chip, leading to more components being integrated onto the CPU (e.g., clock generator in 8085, memory controller, cache, GPU in modern CPUs).
- Enhanced Instruction Sets: From simple operations to complex multimedia and vector instructions.
- Architectural Improvements: Pipelining, caching, virtual memory, multiprocessing, multi-core design – all aimed at improving performance, efficiency, and system robustness.
- Power Efficiency: A growing concern, leading to sophisticated power management techniques.
- Cost Reduction: Initial high cost of early CPUs steadily decreased, making computing accessible.

The 8085, while an 8-bit processor, represents a crucial step in this evolution. It perfected the 8-bit design by integrating key components and simplifying system requirements, making microprocessor technology more practical and widespread. This paved the way for the more powerful 16-bit and 32-bit processors that followed, which in turn formed the backbone of the personal computer revolution and the advanced computing systems we use today. Understanding its position helps appreciate the continuous drive for innovation in the field of computer architecture.

3.) 8085 Microprocessor Architecture

8085 Microprocessor Architecture

The 8085 is an 8-bit microprocessor, meaning it processes data in 8-bit chunks. Its architecture refers to the internal organization of its various functional units and how they are interconnected to perform its primary tasks: fetching instructions, decoding them, and executing them. Understanding the architecture is fundamental to programming and interfacing with the 8085, providing a blueprint of its capabilities and limitations by illustrating how the internal components are structured and communicate.

1. Conceptual Overview and Core Function

The 8085 acts as the Central Processing Unit (CPU) of a microcomputer system. Its fundamental role is to sequentially execute a set of instructions, known as a program, stored in memory. This execution involves a continuous cycle of:

- Fetch: Retrieving an instruction (opcode) from a designated memory location.

• Decode: Interpreting the binary code of the fetched instruction to understand the operation it represents and the operands it requires.
• Execute: Performing the operation specified by the instruction, which might involve data manipulation, memory access, or I/O communication.
The internal architecture of the 8085 is specifically designed to efficiently support this cycle, ensuring a smooth flow of data and control signals.

2. Major Functional Blocks (High-level)

The 8085 comprises several key functional blocks that work in harmony. While their detailed internal workings involve complex logic gates and circuits, understanding their architectural roles is crucial for grasping the overall design.

• Arithmetic Logic Unit (ALU):
• This is the computational core of the 8085. Its primary function is to perform all arithmetic operations (such as addition, subtraction, increment, decrement) and logical operations (such as AND, OR, XOR, NOT, compare).
• The ALU typically takes two 8-bit operands, usually one from the Accumulator and another from an internal register or temporary data register, and places the 8-bit result back into the Accumulator.
• Its existence defines the microprocessor's ability to manipulate data and perform computations, making it central to any processing task. The design of the ALU directly influences the types of instructions the 8085 can execute for data transformation.

• Register Array:
• The 8085 contains a set of internal memory locations called registers. These registers are crucial for temporary storage of data, addresses, and status information during program execution.
• They provide extremely fast access to data, much faster than accessing external memory. This speed advantage is a key architectural reason for their inclusion, as they reduce the need for slower main memory accesses during intermediate steps of computation.
• The specific types and arrangement of these registers form a significant part of the 8085's programming model, directly influencing how instructions are formulated and data is handled.

• Timing and Control Unit:
• This unit acts as the orchestrator of all internal and external operations of the 8085. It is responsible for generating the necessary control signals and timing pulses.
• Internally, it manages data flow between registers, enables ALU operations, and controls the instruction fetch-decode-execute cycle.
• Externally, it generates signals like Memory Read (MEMR), Memory Write (MEMW), I/O Read (IOR), I/O Write (IOW), and various status signals to coordinate with memory and I/O devices.
• It uses the system clock to synchronize all activities, ensuring that each step of microprocessor operation occurs in the correct sequence and within precise time durations. This unit's architectural design is fundamental to the reliable and orderly operation of the entire system.

• Instruction Register and Decoder:
• When an instruction (an 8-bit opcode) is fetched from memory, it is loaded into the Instruction Register. This register temporarily holds the current instruction to be executed.
• The Instruction Decoder then takes the 8-bit binary code from the Instruction Register and interprets its meaning. This process determines what operation the CPU must perform (e.g., **add, move, jump**) and identifies which internal resources (like specific registers or the ALU) are required for that operation.
• Based on the decoded instruction, the Timing and Control Unit generates the precise sequence of control signals needed to execute the instruction. This unit is essential for translating the high-level language of the program into the low-level actions the microprocessor can understand and perform.

3. Internal Data Flow and Bus Structure

The various functional blocks within the 8085 communicate with each other through a network of internal buses. These pathways are fundamental to how data and addresses move around the chip, enabling complex operations.

• Internal Data Bus:
• The 8085 features an 8-bit internal data bus. This bus serves as the primary pathway for transferring data between the Accumulator, other general-purpose registers, the Instruction Register, and the ALU.
• All 8-bit data manipulations, reads, and writes within the CPU occur over this bus. Its 8-bit width is congruent with the 8-bit data processing capability of the 8085. The choice of an 8-bit internal bus directly impacts the types of data operations and memory interactions the processor can perform efficiently.

• Internal Address Paths:
• Although the 8085 processes 8-bit data, it uses a 16-bit address for accessing memory and I/O devices. This necessitates internal pathways and registers capable of handling 16-bit addresses.
• Addresses are typically generated by the Program Counter or Stack Pointer, and these 16-bit values must be precisely routed to the external address bus.
• The internal architecture includes dedicated logic and pathways to manage, store, and distribute these 16-bit addresses, ensuring that memory and I/O locations can be correctly identified.

• Control Signals:
• In addition to data and address lines, a multitude of internal control signals, generated by the Timing and Control Unit, are distributed across the chip. These signals are not for data transfer but for managing operations.
• These signals activate specific components (e.g., enabling a register to output its content or load new data), control the direction of data flow (e.g., read from ALU to Accumulator), and synchronize various parts of the microprocessor. They are the invisible directors of the entire operation.

• Multiplexing (Architectural Implication):
• A significant architectural design decision in the 8085 to reduce the number of physical pins (for cost and packaging efficiency) is the multiplexing of the lower 8-bits of the address bus (A0-A7) with the 8-bit data bus (D0-D7). These are combined into a set of pins labeled AD0-AD7.
• This means that these physical pins carry address information during the early part of a machine cycle (T1 state) and then carry data information during a later part (T2 and T3 states).
• Internally, this architectural choice demands additional demultiplexing logic and latches within the chip (and externally in the system) to separate the address from the data. This adds to the internal complexity but offers the benefit of a smaller pin count, an important consideration for microprocessors of that era.

4. The Register Set (Architectural Role and Categorization)

The 8085's register set is a defining feature of its architecture, providing essential on-chip storage for data and addresses. These registers are not just passive storage units; their specific roles are integral to the microprocessor's operation and programming paradigm.

• Accumulator (A Register):
• This is an 8-bit register, the most important and versatile general-purpose register in the 8085 architecture.
• Most arithmetic and logical operations are performed using the Accumulator. Typically, one operand for an ALU operation is implicitly assumed to be in the Accumulator, and the 8-bit result of the operation is almost always stored back into the Accumulator.
• It also acts as a primary buffer for data transfer between the CPU and external memory or I/O devices. Its central role makes it a focal point for data processing and I/O operations.

• General-Purpose Registers (B, C, D, E, H, L):
• The 8085 features six 8-bit general-purpose registers: B, C, D, E, H, and L.
• These registers can be used individually to store any 8-bit data value as temporary working storage.
• Architecturally, they have the special capability of being combined to form three 16-bit register pairs: (B, C), (D, E), and (H, L). When paired, they are primarily used to store 16-bit memory addresses or larger data values, enabling the manipulation of memory locations beyond just 8-bit data.
• The (H, L) register pair holds particular architectural significance as it frequently acts as a data

pointer, holding the 16-bit memory address of data that is being accessed or manipulated by certain instructions.

• Program Counter (PC):
• This is a 16-bit special-purpose register. Its fundamental architectural purpose is to store the 16-bit memory address of the next instruction opcode to be fetched for execution.
• After an instruction's opcode is fetched from memory, the PC is automatically incremented by the length of the instruction (1, 2, or 3 bytes) to point to the next instruction in the sequential program flow.
• It enables sequential program execution and facilitates jumps or calls to different parts of the program by allowing new 16-bit addresses to be loaded into the PC, thus altering the flow of execution. Without the PC, the CPU would have no mechanism to track its position in the program.

• Stack Pointer (SP):
• This is also a 16-bit special-purpose register.
• Its architectural role is to store the 16-bit memory address of the current top of the stack. The stack is a dedicated area in RAM used for temporary data storage, particularly during subroutine calls, interrupt service routines, and for saving the contents of registers.
• The stack operates on a Last-In, First-Out (LIFO) principle, where the SP always points to the most recently stored item. Its presence is critical for managing program control flow, allowing the CPU to store return addresses and temporary data efficiently during function calls and context switches.

• Flags Register (or Flag Flip-Flops):
• This is an 8-bit register, though only 5 of its bits are actively used as 'flags' or status indicators.
• These flags are architecturally designed to reflect the status (outcome) of the most recent arithmetic or logical operation performed by the ALU. They are individual flip-flops that are set (1) or reset (0) based on the operation's result.
• The commonly used flags are:
• Sign Flag (S): Set if the most significant bit (bit 7) of the result is 1, indicating a negative result (in 2's complement arithmetic).
• Zero Flag (Z): Set if the result of the operation is 0.
• Auxiliary Carry Flag (AC): Set if there is a carry from bit 3 to bit 4 during an 8-bit operation, primarily useful for Binary Coded Decimal (BCD) arithmetic.
• Parity Flag (P): Set if the result has an even number of 1s (even parity); reset if odd parity.
• Carry Flag (CY): Set if there is a carry out of the most significant bit (bit 7) or a borrow into the operation.
• These flags are indispensable for conditional program execution, allowing the program to make decisions and alter its flow based on the outcome of previous operations (e.g., **jump if zero, jump if carry**).

5. Memory and I/O Interaction (Architectural Provision)

The 8085 architecture inherently defines how it interacts with external memory and Input/Output (I/O) devices, laying the groundwork for system design.

• Addressable Memory Space:
• With its 16-bit address bus, the 8085 can generate 2^16 unique memory addresses. This translates to a total addressable physical memory space of 65,536 bytes, or 64 Kilobytes (KB).
• This memory space is typically allocated for both ROM (Read-Only Memory, for permanent programs like firmware) and RAM (Random Access Memory, for temporary data and user programs).
• The architectural design provides the necessary 16 address lines and internal control logic to manage this 64KB memory space effectively, allowing the CPU to read from and write to any location within this range.

• I/O Addressing:
• The 8085 architecture supports two distinct methods for I/O addressing, offering flexibility to system designers:
• I/O-mapped I/O: This approach provides a separate address space specifically for I/O devices. Using 8 address lines for I/O, it allows for 2^8 = 256 unique I/O port addresses. Communication uses dedicated I/O instructions (IN and OUT) and specific I/O read/write control signals (IOR, IOW).
• Memory-mapped I/O: In this scheme, I/O devices are treated as if they are memory locations. They

occupy a portion of the 64KB memory address space. Standard memory instructions (e.g., MOV, LDA) are used to communicate with these devices.
• The internal architecture includes provisions for generating the necessary control signals and routing addresses for both approaches.

6. Control and Status Signals (Internal Perspective)

While external pin details are a separate topic, the internal architecture *relies* on various control and status signals generated and consumed by the Timing and Control Unit to function correctly and coordinate activities.

• Read/Write Control: Internally, signals such as MEMR (Memory Read), MEMW (Memory Write), IOR (I/O Read), and IOW (I/O Write) are generated. These signals are fundamental to controlling the direction and timing of data transfers to and from memory and I/O devices, orchestrating bus transactions.
• Clock and Reset: The internal architecture requires a stable clock signal to synchronize all operations, providing the timing pulses that drive the CPU. Similarly, an internal reset signal is essential to bring the microprocessor to a known, initial state, typically clearing registers and setting the Program Counter to a default start address (e.g., 0000H) upon power-up or system reset.
• Interrupt Signals: The architecture incorporates internal logic to detect and respond to various interrupt requests. While the detailed interrupt structure and handling mechanisms are complex topics, the presence of internal interrupt processing mechanisms is a fundamental architectural feature, enabling the CPU to react asynchronously to external events and handle urgent tasks.
• Status Signals: Internally, the 8085 generates signals like S0 and S1, which together indicate the status of the current machine cycle (e.g., opcode fetch, memory read, memory write). The IO/M signal differentiates between memory and I/O operations. These status signals are crucial for internal sequencing and can also be used externally by support chips to understand the CPU's current activity.

Summary of Key Points:

• The 8085 is an 8-bit microprocessor that uses a 16-bit address bus, allowing it to address 64KB of physical memory.
• Its core architecture comprises an Arithmetic Logic Unit (ALU) for computations, a Register Array for fast data and address storage, and a Timing and Control Unit for overall coordination.
• The Instruction Register and Instruction Decoder are vital for fetching and interpreting program instructions.
• Internal communication is facilitated by an 8-bit internal data bus and 16-bit internal address pathways.
• A key architectural design choice is the multiplexing of the lower 8 bits of the address bus with the 8-bit data bus (AD0-AD7) to reduce pin count, necessitating internal demultiplexing logic.
• The register set includes the 8-bit Accumulator, six 8-bit general-purpose registers (B, C, D, E, H, L) that can form three 16-bit pairs, a 16-bit Program Counter (PC) for sequential instruction fetching, a 16-bit Stack Pointer (SP) for stack management, and an 8-bit Flags Register to reflect the status of ALU operations.
• The architecture provides a 64KB memory address space and a separate 256-port I/O address space, supporting both I/O-mapped and memory-mapped I/O schemes.
• Internal control signals (like read/write controls, clock, reset) and status signals (like S0, S1, IO/M) manage data flow, synchronize operations, and indicate the CPU's current state.

This comprehensive architectural overview lays the foundation for understanding the detailed functional blocks, pin descriptions, and operational timings that will be covered in subsequent topics. The design choices made in the 8085's architecture directly influence its programming model, performance characteristics, and how it efficiently interfaces with the outside world.

4.) 8085 Pin Description

The 8085 Microprocessor is a 40-pin integrated circuit, serving as the central processing unit (CPU) in many embedded systems and early microcomputer designs. Its pins are the interface through which it communicates with the outside world – memory, input/output (I/O) devices, and other peripherals. Understanding the function of each pin is absolutely fundamental for a computer engineering student, as it dictates how an 8085-based system is designed, interfaced, and debugged. This detailed description will break down each pin's purpose, behavior, and the underlying reasoning for its existence, building from basic concepts to a comprehensive understanding.

The 8085 is packaged in a 40-pin Dual In-line Package (DIP). Pins are numbered counter-clockwise, starting from pin 1, which is usually marked with a dot or a notch on the chip.

Let's categorize the pins based on their primary function:

1. Address Bus Pins
2. Data Bus Pins (Multiplexed with Lower Address)
3. Control and Status Signals
4. Power Supply and Frequency Pins
5. Interrupt Signals
6. DMA and Bus Control Signals
7. Reset Signals
8. Serial I/O Signals

Let's explore each category and its associated pins in detail:

1. Address Bus Pins

The address bus is unidirectional (output from the 8085) and is used by the microprocessor to specify the memory location or I/O port it wants to communicate with. The 8085 has a 16-bit address bus, enabling it to access 2^16 = 65,536 (64 KB) unique memory or I/O locations. These 16 bits are split into two groups:

- A15-A8 (Higher Order Address Bus)
- Pins: 28-21 (A8 to A15)
- Purpose: These 8 pins constitute the higher 8 bits of the 16-bit address bus. They are solely dedicated to outputting the high-order address byte (A8-A15).
- Unidirectionality: These are output-only pins, meaning the 8085 sends address information out through them to select memory locations or I/O ports.
- Stability: Unlike the lower address/data bus, these lines are stable and hold the high-order address throughout the entire machine cycle (after the T1 state), making them easier to interface with directly.
- Reasoning: By having dedicated higher address lines, external devices can decode part of the address continuously without waiting for demultiplexing. This also simplifies the design of larger memory systems.

- AD7-AD0 (Lower Order Address/Data Bus)
- Pins: 12-19 (AD0 to AD7)
- Purpose: These 8 pins serve a dual purpose, meaning they are multiplexed. During the first clock cycle (T1 state) of a machine cycle, they carry the lower 8 bits of the address (A0-A7). In subsequent clock cycles (T2, T3, etc.), they function as the data bus, carrying 8-bit data during memory or I/O read/write operations.
- Bidirectionality (as Data Bus): When acting as the data bus, these pins can both send data from the 8085 to memory/I/O or receive data from memory/I/O into the 8085.
- Reasoning for Multiplexing: The primary reason for multiplexing the lower address and data bus is to reduce the total number of pins on the integrated circuit. This reduces the chip's package size, manufacturing cost, and complexity. While beneficial for the chip, it necessitates external demultiplexing hardware (e.g., a latch) to separate the address from the data for external memory and I/O devices.

2. Data Bus Pins

The data bus is bidirectional and carries 8-bit data between the 8085 and external memory or I/O devices.

• AD7-AD0 (Lower Order Address/Data Bus)
• Pins: 12-19 (AD0 to AD7)
• Purpose: As explained above, after the first clock cycle (T1) of a machine cycle, these pins switch from carrying the lower 8 bits of the address (A0-A7) to carrying the 8-bit data (D0-D7) for actual data transfers.
• Bidirectionality: These pins can act as inputs to receive data from memory/I/O or as outputs to send data to memory/I/O, depending on whether it's a read or write operation.
• Reasoning: An 8-bit data bus means the 8085 is an 8-bit microprocessor, capable of processing data in 8-bit chunks. The multiplexing helps keep the pin count low.

3. Control and Status Signals

These signals are vital for coordinating the flow of data and instructions, indicating the type of operation currently being performed by the 8085, and synchronizing with external devices.

• ALE (Address Latch Enable)
• Pin: 30
• Type: Output signal
• Purpose: ALE goes high (logic 1) during the T1 state of a machine cycle, precisely when the AD7-AD0 lines are carrying the lower 8 bits of the address (A0-A7). It then goes low for the rest of the machine cycle.
• Reasoning: ALE acts as a strobe signal for external latches. When ALE is high, an external latch (e.g., a 74LS373) is enabled to capture the address present on AD7-AD0. This effectively demultiplexes the address from the data, providing a stable 16-bit address (A0-A15) for memory and I/O devices.

• RD- (Read)
• Pin: 31
• Type: Active low output signal
• Purpose: When RD- is active low, it indicates that the 8085 is performing a memory read or I/O read operation. This signal is used to enable data output from the selected memory or I/O device onto the data bus (AD7-AD0).
• Reasoning: It acts as a control signal to tell external devices to place data on the data bus for the 8085 to read.

• WR- (Write)
• Pin: 32
• Type: Active low output signal
• Purpose: When WR- is active low, it indicates that the 8085 is performing a memory write or I/O write operation. This signal is used to enable data input into the selected memory or I/O device from the data bus (AD7-AD0).
• Reasoning: It acts as a control signal to tell external devices to latch data from the data bus, indicating that the 8085 is outputting data to be stored.

• IO/M- (I/O or Memory Select)
• Pin: 34
• Type: Output signal
• Purpose: This signal differentiates between memory operations and I/O operations. When IO/M- is high (logic 1), it indicates an I/O operation (I/O read or I/O write). When it is low (logic 0), it indicates a memory operation (memory read or memory write).
• Reasoning: Since the 8085 uses the same RD- and WR- signals for both memory and I/O accesses, the IO/M- signal is crucial. External decoding logic uses IO/M- along with RD- or WR- to generate specific control signals like MEMR-, MEMW-, IOR-, IOW-, allowing memory and I/O devices to respond only to their intended operations.

• S1 and S0 (Status Signals)
• Pins: 33 (S1), 29 (S0)
• Type: Output signals
• Purpose: These two pins provide information about the type of machine cycle the 8085 is currently

executing. Together, they indicate the current status of the CPU.
- States:
- S1 = 0, S0 = 0: HALT (CPU is in a halt state)
- S1 = 0, S0 = 1: WRITE (Memory or I/O write operation)
- S1 = 1, S0 = 0: READ (Memory or I/O read operation)
- S1 = 1, S0 = 1: OPCODE FETCH (Fetching an instruction from memory)
- Reasoning: These status signals can be used by external logic to monitor the 8085's operations, or for more complex bus arbitration and control scenarios, although for basic interfacing, IO/M-, RD-, and WR- are often sufficient.

- READY
- Pin: 35
- Type: Input signal
- Purpose: This signal is used by slower memory or I/O devices to inform the 8085 that they are not yet ready for data transfer. If the READY signal is low (inactive) during a read or write cycle, the 8085 enters a wait state (T-wait state) and extends the machine cycle. It will remain in wait states until READY goes high (active).
- Reasoning: The 8085 operates at a certain clock speed. If a peripheral cannot respond fast enough within the allotted time, data corruption or loss could occur. READY ensures synchronization, allowing the 8085 to pause and wait for slower devices, thus preventing timing conflicts.

4. Power Supply and Frequency Pins

These pins are essential for providing the electrical power and the fundamental timing signals for the microprocessor's operation.

- VCC
- Pin: 40
- Type: Power input
- Purpose: This is the +5V DC power supply input for the 8085.
- Reasoning: All digital integrated circuits require a stable power source to function.

- VSS
- Pin: 20
- Type: Ground
- Purpose: This is the ground reference (0V) for the 8085.
- Reasoning: Provides the common reference voltage for all internal circuitry and signals.

- X1, X2 (Crystal Inputs)
- Pins: 1 (X1), 2 (X2)
- Type: Input pins
- Purpose: These pins are used to connect an external crystal oscillator or an RC (Resistor-Capacitor) network. The 8085 has an internal oscillator circuit that uses these inputs. The internal operating frequency of the 8085 is half of the frequency applied to X1/X2. For example, if a 6 MHz crystal is connected, the internal clock frequency will be 3 MHz.
- Reasoning: The clock signal is the heartbeat of the microprocessor, synchronizing all its internal operations. A crystal provides a highly stable and precise clock frequency, crucial for reliable timing.

- CLK OUT (Clock Out)
- Pin: 37
- Type: Output signal
- Purpose: This pin provides an output clock signal at the internal operating frequency of the 8085 (f_osc / 2).
- Reasoning: This clock signal can be used to synchronize other peripheral devices and components in the 8085-based system, ensuring that all parts operate in harmony with the CPU's timing.

5. Interrupt Signals

Interrupts are mechanisms that allow external devices to temporarily suspend the normal execution of the CPU's program and force it to execute a special routine (Interrupt Service Routine - ISR) to handle

an event. The 8085 has a sophisticated interrupt structure with five hardware interrupt pins.

- TRAP
- Pin: 6
- Type: Input signal
- Purpose: TRAP is a non-maskable, highest priority interrupt. It is both edge and level triggered (it responds to a positive edge and then must remain high until sampled, then fall back low). Once acknowledged by the CPU, it remains active until the CPU handles it.
- Reasoning: Non-maskable means it cannot be disabled by software. This makes TRAP suitable for critical events that require immediate attention, such as power failure, memory parity errors, or other system emergencies where continued normal operation is not feasible. It has a dedicated vector address, meaning the CPU automatically jumps to a fixed memory location (0024H) upon a TRAP interrupt, eliminating the need for an external device to provide an instruction.

- RST 7.5, RST 6.5, RST 5.5 (Restart Interrupts)
- Pins: 7 (RST 7.5), 8 (RST 6.5), 9 (RST 5.5)
- Type: Input signals
- Purpose: These are maskable, vectored interrupts, meaning they can be enabled or disabled by software, and they also have dedicated vector addresses:
- RST 7.5: Vector address 003CH
- RST 6.5: Vector address 0034H
- RST 5.5: Vector address 002CH
- Triggering:
- RST 7.5 is positive edge-triggered only. Once the edge is detected, the flag is set even if the input goes low later. This ensures short pulses are not missed.
- RST 6.5 and RST 5.5 are level-triggered. The input must be held high until sampled by the CPU.
- Priorities: Among these, RST 7.5 has the highest priority, followed by RST 6.5, then RST 5.5.
- Reasoning: These provide structured and prioritized ways for specific external events to trigger predefined interrupt service routines. The ability to mask them allows a programmer to temporarily ignore less critical interrupts during specific code execution phases.

- INTR (Interrupt Request)
- Pin: 10
- Type: Input signal
- Purpose: This is a maskable, general-purpose interrupt request. It is level-triggered, meaning it must be held high until sampled by the CPU. It has the lowest priority among all interrupts.
- Reasoning: Unlike the vectored interrupts, INTR does not have a dedicated vector address. When an INTR is acknowledged, the 8085 expects the interrupting device to provide the instruction (usually an RST or CALL instruction) for the CPU to execute. This allows for greater flexibility and the use of external programmable interrupt controllers (like the 8259) to manage multiple interrupting devices, assigning them priorities and specific service routines.

- INTA- (Interrupt Acknowledge)
- Pin: 11
- Type: Active low output signal
- Purpose: This signal is sent by the 8085 in response to an INTR request when the interrupt is enabled and processed. When INTA- goes active low, it tells the external interrupting device to place its RST or CALL instruction on the data bus (AD7-AD0) so the 8085 can fetch it.
- Reasoning: INTA- completes the handshaking process for the INTR interrupt. It confirms to the external device that its request has been accepted and that it should now provide the vector information required to service its specific need.

6. DMA and Bus Control Signals

These signals are used for Direct Memory Access (DMA) operations, allowing high-speed data transfer between peripherals and memory without continuous CPU intervention, and for controlling bus access.

- HOLD
- Pin: 39
- Type: Input signal

• Purpose: This input is used by a device (typically a DMA controller) to request control of the system bus (address, data, and control lines). When HOLD is active high, the 8085 relinquishes control of its address bus (A8-A15, AD0-AD7), data bus (AD0-AD7), and control signals (RD-, WR-, IO/M-).
• Reasoning: DMA is crucial for applications requiring high-speed data transfer (e.g., disk controllers, graphics controllers). By giving up the bus, the 8085 allows another master (the DMA controller) to directly access memory, significantly speeding up data movement as the CPU is not involved in each byte transfer.

• HLDA (Hold Acknowledge)
• Pin: 38
• Type: Output signal
• Purpose: This active high signal is output by the 8085 to acknowledge a HOLD request. When HLDA goes high, it indicates that the 8085 has placed its address, data, and control lines into a high-impedance state (effectively disconnecting them from the bus) and that the requesting device can now take control of the system bus.
• Reasoning: HLDA provides feedback to the DMA controller, confirming that the bus is now available for its use. This handshaking ensures that only one device controls the bus at a time, preventing bus contention.

7. Reset Signals

Reset signals are used to bring the microprocessor to a known, initial state, typically at power-up or when a system restart is required.

• RESET IN-
• Pin: 36
• Type: Active low input signal
• Purpose: When RESET IN- is held low, the 8085 undergoes a reset operation. This sets the Program Counter (PC) to 0000H, clears all interrupt enable flip-flops (masking interrupts), clears the HOLD and HLDA flip-flops, and disables the serial I/O (SOD and SID). The CPU then starts fetching instructions from memory location 0000H when RESET IN- goes high again.
• Reasoning: Essential for initializing the system. It ensures that the microprocessor always starts execution from a predefined address (0000H) and in a predictable state, which is critical for system reliability and program execution flow.

• RESET OUT
• Pin: 3
• Type: Output signal
• Purpose: This output signal goes high when RESET IN- is active (low) and remains high as long as the 8085 is in its reset state.
• Reasoning: This signal provides a synchronized reset signal to other peripheral devices in the system. When the 8085 is reset, it's often necessary to reset other components as well to ensure the entire system starts up coherently.

8. Serial I/O Signals

The 8085 includes basic built-in hardware for serial communication, simplifying interaction with serial devices without requiring an external UART (Universal Asynchronous Receiver/Transmitter) for simple applications.

• SID (Serial Input Data)
• Pin: 5
• Type: Input signal
• Purpose: This pin is used to receive serial data, one bit at a time, into the 8085. The state of this pin can be read into the accumulator's D7 bit using the RIM (Read Interrupt Mask) instruction.
• Reasoning: Provides a very basic, software-controlled serial input capability. While not a full-fledged UART, it's useful for simple debugging, data logging, or communicating with single-wire serial devices where precise timing is handled by software.

• SOD (Serial Output Data)
• Pin: 4
• Type: Output signal
• Purpose: This pin is used to transmit serial data, one bit at a time, from the 8085. The state of this pin can be set (to 0 or 1) from the accumulator's D7 bit using the SIM (Set Interrupt Mask) instruction.
• Reasoning: Provides a basic, software-controlled serial output capability. Similar to SID, it allows for simple serial communication, often used for debugging messages, controlling external devices with a serial interface, or generating custom serial protocols.

Summary of Key Points:

• The 8085 is a 40-pin microprocessor, with its pins categorized into functionally distinct groups.
• It features a 16-bit address bus (A0-A15) for addressing 64KB of memory/I/O, and an 8-bit data bus (D0-D7) for data transfer.
• A key design choice is the multiplexing of the lower 8-bit address (A0-A7) and the 8-bit data bus (D0-D7) onto the AD7-AD0 pins to reduce pin count. The ALE signal is crucial for externally demultiplexing these lines.
• Control signals like RD-, WR-, and IO/M- manage memory and I/O read/write operations, while S0/S1 provide status information about machine cycles.
• The READY input allows the 8085 to synchronize with slower peripherals by entering wait states.
• Power is supplied via VCC (+5V) and VSS (GND), and the internal clock is generated using an external crystal connected to X1/X2, with CLK OUT providing a system clock.
• A comprehensive interrupt system includes the non-maskable TRAP, vectored and maskable RST 7.5/6.5/5.5, and the general-purpose INTR/INTA- for flexible event handling.
• HOLD and HLDA pins facilitate Direct Memory Access (DMA), allowing external controllers to take control of the system bus for high-speed data transfers.
• RESET IN- and RESET OUT ensure proper system initialization on power-up or manual reset.
• Built-in serial I/O capabilities are provided by the SID (Serial Input Data) and SOD (Serial Output Data) pins, controlled via software instructions (RIM/SIM).
• Understanding the function and interaction of these pins is paramount for designing, troubleshooting, and programming 8085-based systems, forming the bedrock of microprocessor interfacing.

5.) Functional Blocks of 8085 (ALU, Registers, Timing & Control Unit)

The 8085 Microprocessor, an 8-bit central processing unit, is built upon several fundamental functional blocks that work in concert to execute instructions and manage data. Understanding these blocks - the Arithmetic and Logic Unit (ALU), the various Registers, and the Timing and Control Unit - is crucial to comprehending how the processor operates at a fundamental level. While we've already discussed its overall architecture and pin descriptions, let's now dive deeper into the internal workings of these key components.

1. Arithmetic and Logic Unit (ALU)
The ALU is essentially the computational powerhouse of the 8085. It is responsible for performing all arithmetic operations (like addition, subtraction, increment, decrement) and logical operations (like AND, OR, XOR, complement, compare).

• Functionality:
• It takes data from specific internal registers, primarily the Accumulator and temporary data registers, as inputs.
• It performs the requested operation on this data.
• The result of the operation is typically stored back into the Accumulator.
• In addition to the result, the ALU also generates status flags based on the characteristics of the result.

• Key Operations Performed:
• Arithmetic Operations:
• Addition: Adds two 8-bit numbers. For example, ADD B (Adds content of B register to Accumulator, result in Accumulator).
• Subtraction: Subtracts one 8-bit number from another. For example, SUB C (Subtracts content of C from Accumulator, result in Accumulator).
• Increment/Decrement: Increases or decreases the value of an 8-bit register or memory location by one.
• Note: The 8085 does not have direct hardware support for multiplication or division. These operations are typically implemented through sequences of additions/subtractions in software.
• Logical Operations:
• AND (ANA), OR (ORA), XOR (XRA): Perform bit-wise logical operations between the Accumulator and another register/memory location.
• Complement (CMA): Inverts all bits of the Accumulator (1s become 0s, 0s become 1s).
• Compare (CMP): Compares the Accumulator with another register/memory location by subtracting them but discarding the result; only the flags are affected.
• Rotate Operations:
• RAL (Rotate Accumulator Left): Shifts bits left, MSB moves to Carry Flag and LSB.
• RAR (Rotate Accumulator Right): Shifts bits right, LSB moves to Carry Flag and MSB.

• Flags Register:
• The Flags Register is an 8-bit register, but only 5 bits are actively used to indicate the status of the most recent ALU operation. These flags are crucial for conditional branching and multi-byte arithmetic. The unused bits are **don't care** bits.
• Each flag is a single bit that is set (1) or reset (0) depending on the outcome:
• 1- Sign Flag (S): Set if the most significant bit (D7) of the result is 1, indicating a negative number in signed arithmetic. Reset if D7 is 0 (positive).
• Reasoning: Allows the processor to determine the sign of the result, essential for signed number operations and conditional jumps based on sign.
• 2- Zero Flag (Z): Set if the result of the operation is 0. Reset if the result is non-zero.
• Reasoning: Highly useful for loop control, checking for equality, and determining if a register has been cleared.
• 3- Auxiliary Carry Flag (AC): Set if there is a carry out from bit D3 to bit D4 during an arithmetic operation. Reset otherwise.
• Reasoning: Primarily used in BCD (Binary Coded Decimal) arithmetic. It helps in adjusting the results of BCD additions to maintain correct BCD representation.
• 4- Parity Flag (P): Set if the result contains an even number of 1s (even parity). Reset if the result contains an odd number of 1s (odd parity).
• Reasoning: Can be used for basic error checking in data transmission or manipulation, though less frequently relied upon for general computation compared to other flags.
• 5- Carry Flag (CY): Set if there is a carry out from the most significant bit (D7) of the result for addition, or a borrow into D7 for subtraction. Reset otherwise.
• Reasoning: Absolutely vital for multi-byte arithmetic, allowing the carry from one byte operation to be propagated to the next. Also indicates overflow in unsigned arithmetic.
• Analogy: Think of the flags register as a series of warning lights on a car's dashboard. Each light (flag) illuminates (sets) to indicate a specific condition (like low fuel or high temperature) about the engine's (ALU's) recent activity.

2. Registers
Registers are small, high-speed storage locations within the CPU itself. They temporarily hold data, addresses, or control information needed during instruction execution. The 8085 has several types of registers, each with a specific role.

• General Purpose Registers (B, C, D, E, H, L):
• These are six 8-bit registers (B, C, D, E, H, L) that can be used for temporary data storage during program execution.
• They can be used individually for 8-bit operations.
• Reasoning for pairing: While the 8085 is an 8-bit processor, it needs to handle 16-bit memory addresses and sometimes 16-bit data. These registers can be paired up to form three 16-bit register

pairs:
- BC pair (B is the most significant byte, C is the least significant byte)
- DE pair (D is MSB, E is LSB)
- HL pair (H is MSB, L is LSB)
- The HL pair is particularly significant as it often acts as a memory pointer, directly accessing memory locations through instructions like MOV A, M (Move content of memory pointed by HL into A).

- Accumulator (A):
- This is a special 8-bit register, arguably the most important data register in the 8085.
- It serves as a primary operand for almost all arithmetic and logical operations performed by the ALU (e.g., in ADD B, the Accumulator holds one operand and the result).
- It is the destination for the results of most ALU operations.
- It also acts as the interface for most input/output (I/O) operations, transferring data between the CPU and peripheral devices.
- Reasoning: Centralizing data flow through the Accumulator simplifies the instruction set design and hardware, making operations efficient.

- Program Counter (PC):
- This is a 16-bit register.
- Its primary function is to store the memory address of the next instruction byte to be fetched from memory.
- During the instruction fetch cycle, the CPU sends the address from the PC to the memory. After fetching an instruction byte, the PC is automatically incremented to point to the next byte of the instruction or the next instruction.
- Reasoning: Enables sequential execution of a program. Its content can also be explicitly modified by jump, call, or return instructions, allowing for control flow changes (loops, subroutines, conditional execution).

- Stack Pointer (SP):
- This is also a 16-bit register.
- It holds the 16-bit memory address of the **top** of the stack.
- The stack is a Last-In, First-Out (LIFO) data structure located in the main memory (RAM), used for temporary storage of data, particularly during subroutine calls (to save return addresses) and interrupt service routines (to save CPU context).
- When data is PUSHed onto the stack, the SP is decremented first (by two bytes for 16-bit data), then the data is stored.
- When data is POPed from the stack, the data is retrieved first, then the SP is incremented (by two bytes).
- Reasoning: Provides a flexible and efficient mechanism for saving and restoring the processor's state, managing subroutine calls and returns, and handling local variables.

- Instruction Register (IR):
- This is an 8-bit register.
- After an instruction's opcode is fetched from memory, it is temporarily stored in the Instruction Register.
- The content of the IR is then passed to the Instruction Decoder for interpretation.
- Reasoning: It acts as a buffer between the memory and the instruction decoder, allowing the fetch and decode stages of the instruction cycle to proceed somewhat independently.

3. Timing and Control Unit
The Timing and Control Unit is the central nervous system of the 8085. It generates the necessary timing and control signals required for the internal and external operations of the microprocessor. It orchestrates all activities, ensuring that every operation occurs at the correct time and in the correct sequence.

- Overall Purpose:
- Synchronizes all internal registers and external peripherals.
- Generates control signals for memory and I/O devices.
- Manages the instruction fetch, decode, and execute cycles.
- Responds to external signals like interrupts and DMA requests.

• Key Components/Functions:
• Clock Generator:
• The 8085 requires an external clock source (crystal or RC network connected to X1 and X2 pins) to generate its internal clock signal.
• It also provides an external clock output (CLK OUT) for synchronizing other peripheral devices in the system.
• Reasoning: All operations within the microprocessor are sequential and time-dependent. A stable and precise clock signal is essential for synchronizing these operations.
• Instruction Decoder:
• This component receives the 8-bit opcode from the Instruction Register.
• It interprets (decodes) the opcode to understand which instruction is to be executed.
• Based on the decoded instruction, it generates a sequence of micro-operations (elementary steps) required to perform that instruction.
• Analogy: Like a language translator that takes a command in one language (the opcode) and translates it into a series of detailed actions for other units to perform.
• Control Signal Generation:
• Based on the decoded instruction, the current machine cycle (e.g., opcode fetch, memory read/write, I/O read/write), and the state of the processor, this unit generates various control signals. These signals are crucial for managing data flow and communication with external devices.
• Read (RD) and Write (WR) signals: Active low signals used to control memory and I/O read/write operations. When RD is active, data is read; when WR is active, data is written.
• I/O or Memory (IO/M): This signal differentiates between memory and I/O operations. When high, it indicates an I/O operation; when low, it indicates a memory operation. Combined with RD/WR, it generates specific control signals like MEMR (Memory Read), MEMW (Memory Write), IOR (I/O Read), IOW (I/O Write).
• Status Signals (S0, S1): These two signals provide information about the type of machine cycle the processor is currently executing (e.g., Opcode Fetch, Memory Read, Memory Write, I/O Read, I/O Write).
• Address Latch Enable (ALE): This signal is used to demultiplex the lower 8 bits of the address/data bus (AD0-AD7). It goes high during the first clock period of a machine cycle, indicating that AD0-AD7 contain the low byte of the address. External latches use ALE to capture this address byte.
• Reasoning: The 8085 uses a multiplexed address/data bus (AD0-AD7) to reduce the number of pins. ALE allows external circuitry to separate the address information from the data information at the correct time.
• READY: An input signal. If a slower peripheral device cannot respond quickly, it can pull the READY pin low, causing the 8085 to enter wait states until the READY pin goes high again.
• Reasoning: Essential for synchronizing the fast CPU with potentially slower memory or I/O devices, preventing data corruption due to speed mismatches.
• Interrupt Control Signals (INTR, RST 7.5/6.5/5.5, TRAP): These are inputs that allow external devices to interrupt the CPU's normal execution flow to request service. The control unit manages the recognition and prioritization of these interrupts.
• DMA Control Signals (HOLD, HLDA): These signals facilitate Direct Memory Access (DMA), allowing other devices to temporarily take control of the system buses to directly access memory. (The control unit handles the handshaking process).
• Reset Signals (RESET IN, RESET OUT): RESET IN is an input to initialize the CPU, and RESET OUT is an output signal used to reset other connected devices when the CPU is reset.
• Timing and Synchronization:
• The unit ensures that all internal operations (like fetching instruction, reading/writing data, performing ALU operations) and external interactions happen at precisely the right clock edges and durations.
• It coordinates the sequence of machine cycles (e.g., Opcode Fetch followed by Memory Read/Write, etc.) that constitute the execution of a single instruction.
• Reasoning: Without meticulous timing and synchronization, the various parts of the microprocessor would operate asynchronously, leading to chaotic and incorrect results. It's the **conductor** ensuring every instrument (functional block) plays in harmony.

Summary of Key Points:
• The ALU is the computational core, executing arithmetic and logical operations, and setting status

flags based on results.
• The Flags Register, part of the ALU's output, provides critical information (Sign, Zero, Auxiliary Carry, Parity, Carry) about the last operation, enabling conditional execution and multi-byte arithmetic.
• Registers provide fast, temporary storage:
• General Purpose Registers (B, C, D, E, H, L) store 8-bit data, can form 16-bit pairs for addressing.
• The Accumulator (A) is the primary operand and result storage for most ALU and I/O operations.
• The Program Counter (PC) keeps track of the next instruction to be fetched.
• The Stack Pointer (SP) manages the stack in RAM for temporary data storage and subroutine/interrupt handling.
• The Instruction Register (IR) temporarily holds the fetched opcode for decoding.
• The Timing and Control Unit acts as the orchestrator, generating clock signals, decoding instructions, and producing all necessary control signals (RD, WR, IO/M, ALE, READY, etc.) to manage data flow, synchronize operations, and communicate with memory and I/O devices.

6.) Memory Organization and Mapping

Memory Organization and Mapping in 8085 Microprocessor Systems

This section delves into how memory is structured and accessed by the 8085 microprocessor, a fundamental aspect of understanding how any microprocessor-based system functions. We will explore the organization of memory chips and the critical process of memory mapping, which allocates specific address ranges to these chips.

1- Introduction to Memory in 8085 Systems

Memory serves as the storage medium for both instructions (the program code) and data that the 8085 microprocessor needs to operate. Without memory, a microprocessor cannot execute any task. The 8085 is an 8-bit microprocessor, meaning it processes data in 8-bit chunks (bytes). Consequently, its memory is typically organized as a collection of individually addressable bytes.

• The 8085 interacts with memory using its bus system. Although the detailed system bus structure is a future topic, it is essential to understand that:
• It uses 16 address lines (A0-A15) to select a unique memory location.
• It uses 8 data lines (D0-D7) to transfer data to or from the selected memory location.
• It uses control signals (like RD for read, WR for write) to specify the type of operation.
• Maximum Addressable Memory: With 16 address lines, the 8085 can generate 2^16 unique addresses. This translates to 65,536 distinct memory locations, or 64 Kilobytes (KB). This 64KB range (from address 0000H to FFFFH in hexadecimal) is the entire address space available to the 8085. All memory components (RAM, ROM, and potentially I/O devices via memory-mapped I/O, a future topic) must fit within this 64KB address space.

2- Memory Organization

Memory organization refers to the physical and logical structure of the memory components within a computer system. It describes how individual memory chips are designed and how they appear to the microprocessor.

• Basic Memory Unit: For the 8085, the smallest addressable unit of memory is typically one byte (8 bits). Each byte stored in memory has a unique 16-bit address associated with it.
• Types of Memory Used with 8085:
• RAM (Random Access Memory):
• Volatile: Data is lost when power is removed.
• Read/Write Capability: The 8085 can both read data from and write data to RAM.
• Purpose: Used for storing temporary data, variables, program stack, and intermediate results during program execution.
• Examples: Static RAM (SRAM) for faster, smaller caches (not typically directly addressable by 8085 in this context, but good to know), Dynamic RAM (DRAM) for larger main memory (often needs refresh

circuitry which complicates 8085 interfacing, so SRAM is more common for directly interfaced smaller systems).

• ROM (Read Only Memory):

• Non-Volatile: Data persists even when power is removed.

• Read-Only (primarily): The 8085 can only read data from ROM; it cannot write to it during normal operation.

• Purpose: Used for storing permanent programs like the boot-up code, operating system kernel (in more complex systems), lookup tables, or fixed firmware. The reset vector (0000H) typically points to the start of ROM.

• Examples: PROM (Programmable ROM), EPROM (Erasable PROM), EEPROM (Electrically Erasable PROM), and Flash Memory.

• Memory Chip Structure:

• Each individual memory chip (e.g., a 2KB ROM chip or an 8KB RAM chip) has its own internal organization.

• It will have a specific number of data lines (usually 8 for 8085 compatibility) and a specific number of internal address lines.

• For example, a 2KB (2048-byte) memory chip requires 11 internal address lines because 2^11 = 2048. These lines are labeled A0 to A10.

• A 'Chip Select' (CS) or 'Chip Enable' (CE) pin is present on most memory chips. This pin, when active (usually low), enables the chip to respond to the address and control signals. If inactive, the chip effectively disconnects itself from the data bus, preventing conflicts.

3- Memory Mapping

Memory mapping is the process of assigning a specific range of addresses within the microprocessor's total address space (the 64KB for 8085) to a particular physical memory chip or a block of memory. It defines where each memory component **lives** in the overall system memory.

• Analogy: Imagine the 8085's 64KB address space as a very large apartment building with 65,536 apartments, each having a unique number from 0 to 65535. Each physical memory chip (a RAM chip or a ROM chip) is like a section of this building, say, a block of apartments from 1000-1999 for one chip, and 4000-4999 for another. Memory mapping is the process of deciding which chips get which ranges of apartment numbers.

• Purpose of Memory Mapping:

• Avoid Address Conflicts: Ensures that no two physical memory chips respond to the same address at the same time. If they did, it would lead to bus conflicts and data corruption.

• Efficient Use of Address Space: Allows different sized memory chips to be placed at appropriate locations within the 64KB address space.

• System Design Flexibility: Enables designers to combine different types and sizes of RAM and ROM chips to meet the system's specific requirements.

• The Role of Address Decoding:

• Address decoding is the hardware mechanism that implements memory mapping. It is a logic circuit that monitors the microprocessor's address lines (A0-A15) and generates the appropriate 'Chip Select' (CS) or 'Chip Enable' (CE) signal for a particular memory chip only when an address within that chip's assigned range appears on the address bus.

• High-order address lines are primarily used for chip selection. For example, if a system has several 2KB chips, the lowest 11 address lines (A0-A10) select a location *within* a 2KB chip. The remaining higher-order address lines (A11-A15) are used by the address decoder to decide *which* 2KB chip should be active.

• Low-order address lines are directly connected to the memory chip's internal address pins. They determine the specific location (byte) within the selected chip.

4- Address Decoding Techniques

The way address decoding logic is implemented significantly impacts the efficiency and complexity of the memory map.

• Full Decoding:

• Principle: In full decoding, all relevant higher-order address lines are used by the decoding logic to generate the Chip Select signal for each memory chip.

• Outcome: Every single address within the 8085's 64KB address space maps to either a unique physical memory location or to no physical memory at all (an unassigned address).
• Advantages:
• No address conflicts: Each location has a truly unique address.
• No **mirror mapping** or **fold-back memory** (explained below).
• Maximizes the efficient use of the address space.
• Disadvantages:
• More complex decoding logic: Requires more gates or a dedicated decoder IC (like a 74LS138 3-to-8 line decoder) to ensure precise address range selection.
• Higher hardware cost and potentially larger board space.
• Example: If you have a 4KB ROM at 0000H-0FFFH and an 8KB RAM at 4000H-5FFFH, full decoding ensures that only addresses within 0000H-0FFFH activate the ROM, and only addresses within 4000H-5FFFH activate the RAM, and no other addresses activate either.

• Partial Decoding:
• Principle: Only a subset of the necessary high-order address lines are used in the decoding logic to generate the Chip Select signal. Some higher-order address lines are ignored or **don't care.**
• Outcome: Simpler hardware but introduces **mirror mapping** or **fold-back memory.**
• Advantages:
• Simpler decoding logic: Requires fewer gates, lower cost, and less board space.
• Often sufficient for small systems that do not need to utilize the full 64KB address space.
• Disadvantages:
• Mirror Mapping (Fold-back Memory): This is the main drawback. Because some high-order address lines are ignored by the decoder, a single physical memory location will appear at multiple different logical addresses within the microprocessor's address space.
• Example of Mirror Mapping: Suppose you have a 2KB RAM chip (requiring A0-A10 internally) and you decide to use only address line A15 to generate its chip select (e.g., CS is active when A15=0). The 2KB RAM will occupy addresses 0000H-07FFH. However, since A11, A12, A13, A14 are ignored, this same 2KB chip will also respond to addresses 0800H-0FFFH, 1000H-17FFH, and so on, up to 7800H-7FFFH. Any address with A15=0 will activate the chip. So, writing to 0000H is the same as writing to 0800H, 1000H, etc., as they all map to the same physical location within the chip.
• Increased difficulty in debugging due to multiple addresses referring to the same location.
• Inefficient use of the address space.
• When used: In small, embedded systems where the total memory requirement is much less than the 64KB address space, and the simplicity of the hardware outweighs the minor inconvenience of mirror addresses. It's often chosen for cost-effectiveness.

5- Practical Example - 8085 Memory Map Illustration

Let's design a simple memory map for an 8085 system.
Assume the system requires:
• 4KB ROM for the operating system/firmware.
• 8KB RAM for data, stack, and variables.

The total memory needed is 4KB + 8KB = 12KB, which is well within the 8085's 64KB address space.

• Step 1: Determine internal address lines for each chip.
• 4KB ROM: 4KB = 4 * 1024 bytes = 4096 bytes. Requires 12 internal address lines (A0-A11), since 2^12 = 4096.
• 8KB RAM: 8KB = 8 * 1024 bytes = 8192 bytes. Requires 13 internal address lines (A0-A12), since 2^13 = 8192.

• Step 2: Assign Address Ranges.
• ROM: Typically, ROM containing the boot code starts at address 0000H because the 8085's reset vector is 0000H.
• Start Address: 0000H
• End Address: 0000H + 4096 - 1 = 0FFFH
• Address Range for ROM: 0000H - 0FFFH.
• In binary, for 0000H to 0FFFH: The highest four bits (A15 A14 A13 A12) are all 0s (0000). The lower

12 bits (A0-A11) vary from 000000000000 to 111111111111.
• RAM: Let's assign RAM starting at address 4000H to leave space after ROM and for potential future expansion.
• Start Address: 4000H
• End Address: 4000H + 8192 - 1 = 5FFFH
• Address Range for RAM: 4000H - 5FFFH.
• In binary, for 4000H to 5FFFH: The highest three bits (A15 A14 A13) are 010. The next bit (A12) varies (0 to 1). The lower 12 bits (A0-A11) vary. So, 4000H is 0100 0000 0000 0000b, and 5FFFH is 0101 1111 1111 1111b. Notice how A15=0, A14=1, A13=0 consistently for this range.

• Step 3: Conceptual Address Decoding Logic (Full Decoding Example).
• ROM (0000H - 0FFFH):
• To select the ROM chip, the address decoder needs to check that all high-order address lines A15, A14, A13, and A12 are LOW (0).
• CS_ROM will be active (e.g., LOW) when A15=0, A14=0, A13=0, and A12=0. This could be implemented with a 4-input NAND gate whose inputs are A15', A14', A13', A12' (where ' indicates inversion).
• The remaining address lines A0-A11 from the 8085 are connected directly to the ROM chip's A0-A11 pins.
• RAM (4000H - 5FFFH):
• To select the RAM chip, the address decoder needs to check that A15=0, A14=1, and A13=0. (Note: A12 for RAM is part of its internal address lines, so it's not used for chip selection in this case as 8KB needs A0-A12. This is valid full decoding if using a 74LS138 for instance which has 3 selection inputs and enable pins).
• CS_RAM will be active (e.g., LOW) when A15=0, A14=1, and A13=0. This could be implemented with a 3-input NAND gate whose inputs are A15', A14, A13'.
• The remaining address lines A0-A12 from the 8085 are connected directly to the RAM chip's A0-A12 pins.

This systematic assignment ensures that when the 8085 places an address like 0123H on its address bus, only the ROM chip is enabled. If it places 4567H, only the RAM chip is enabled. If it places an address like 1000H (which is not in either range), neither chip is enabled.

6- Importance for 8085 Programming

Understanding memory organization and mapping is crucial for anyone programming the 8085 microprocessor, particularly in assembly language:

• Code Placement: You need to know where your program code (instructions) will reside in memory, especially where the program starts (often 0000H for bootloaders in ROM).
• Data Storage: You must know the address ranges allocated for RAM to correctly store and retrieve variables, arrays, and other data structures. Instructions like LDA (Load Accumulator Direct) and STA (Store Accumulator Direct) require explicit 16-bit memory addresses.
• Stack Operations: The stack (used for temporary data storage, subroutine calls, and interrupt handling) is typically located in RAM, usually configured to grow downwards from a high address. Knowing its mapped location is vital for initializing the Stack Pointer (SP) register.
• Debugging: If your program behaves unexpectedly, knowing the memory map helps in debugging. For instance, attempting to write to a ROM address will fail, or accessing an unmapped address might lead to a system crash.
• Peripheral Interaction: While I/O mapping is a future topic, many peripherals can be memory-mapped, meaning they also occupy specific address ranges, which must be known for proper interaction.

7- Summary of Key Points

• The 8085 can address 64KB (65,536 bytes) of memory using its 16 address lines (0000H to FFFFH).
• Memory organization defines how physical memory chips (RAM and ROM) are structured and their characteristics (volatile/non-volatile, read/write).
• Each byte in memory has a unique 16-bit address.

• Memory mapping is the process of assigning specific address ranges within the 64KB address space to different physical memory chips.
• Address decoding is the hardware logic that generates Chip Select (CS) signals to activate the correct memory chip based on the address placed by the microprocessor.
• Full decoding uses all high-order address lines for unique address assignment, preventing conflicts and mirror mapping.
• Partial decoding uses only a subset of high-order address lines, simplifying hardware but leading to **mirror mapping** where physical locations can be accessed by multiple logical addresses.
• Understanding the memory map is essential for 8085 programming to correctly place code, store data, manage the stack, and interact with hardware.

7.) Memory Interfacing Techniques

Memory Interfacing Techniques

Welcome to the topic of Memory Interfacing Techniques, a critical aspect of computer organization and architecture, especially when working with microprocessors like the 8085. After understanding the 8085's architecture and how memory is organized, the next logical step is to learn how to physically connect and control memory chips so the microprocessor can read from and write to them.

1. What is Memory Interfacing?
Memory interfacing refers to the process of connecting memory chips (RAM and ROM) to the microprocessor unit (MPU) in such a way that the MPU can access specific memory locations. This involves ensuring that the MPU's address, data, and control signals are correctly connected to the corresponding pins of the memory chips, and most importantly, that each memory location has a unique address.

2. Why is Memory Interfacing Needed?
The 8085 microprocessor has a 16-bit address bus, allowing it to access 2^16 = 65,536 (64 KB) unique memory locations. However, individual memory chips typically have smaller capacities (e.g., 2KB, 4KB, 8KB, 32KB). To utilize the full address space or combine multiple memory chips, we need a method to:
• Select the correct memory chip when its address range is accessed.
• Map the MPU's address lines to the memory chip's address lines.
• Ensure proper data flow between the MPU and memory.
• Control read/write operations.

3. Core Requirements for Memory Interfacing
To interface memory chips with the 8085, four main groups of connections are essential:

a. Address Lines Connection
• The 8085 has 16 address lines (A0-A15).
• Each memory chip also has a certain number of address lines based on its capacity. For example, a 2KB (2048 bytes) memory chip needs 11 address lines (A0-A10) because 2^11 = 2048. An 8KB (8192 bytes) chip needs 13 address lines (A0-A12) because 2^13 = 8192.
• The lower-order address lines of the 8085 (A0 to A_n, where A_n is the highest address line needed by the memory chip) are directly connected to the corresponding address lines of the memory chip.
• The higher-order address lines of the 8085 (A_n+1 to A15) are used for **address decoding,** which determines which specific memory chip is being accessed.

b. Data Lines Connection
• The 8085 has an 8-bit data bus (D0-D7).
• Most memory chips used with the 8085 are also 8-bit wide (e.g., 2Kx8, 8Kx8).
• The 8085's data lines (AD0-AD7, after demultiplexing from the address/data bus) are directly connected to the data pins (D0-D7) of the memory chip.

c. Control Signals Connection

• The 8085 generates control signals that dictate the type of operation (read or write) and whether it's a memory or I/O operation.

• IO/M# (Input/Output or Memory bar): This signal is low (0) for memory operations and high (1) for I/O operations. For memory interfacing, it must be low.

• RD# (Read bar): This signal goes low (0) when the 8085 wants to read data from memory or I/O.

• WR# (Write bar): This signal goes low (0) when the 8085 wants to write data to memory or I/O.

• Memory chips have corresponding control pins:

• OE# (Output Enable bar): Connected to the 8085's RD# signal. When low, data flows from memory to the data bus.

• WE# (Write Enable bar): Connected to the 8085's WR# signal. When low, data from the data bus is written into memory.

• For ROM, only RD# (to OE#) is relevant as ROM is read-only.

d. Chip Select (CS# or CE#)

• This is the most crucial control signal for memory interfacing. Each memory chip has a Chip Select (CS#) or Chip Enable (CE#) pin.

• This pin must be activated (usually active low) for the memory chip to respond to any read or write request. If it's not active, the chip remains in a high-impedance state, effectively disconnected from the data bus.

• The CS# signal is generated by **address decoding logic,** which uses the higher-order address lines of the 8085 to determine if the currently requested address falls within the range assigned to that particular memory chip.

4. The Challenge: Address Uniqueness and Decoding

The core challenge is that the 8085 has 16 address lines, but a single memory chip only uses a subset of these for its internal addressing. The remaining higher-order address lines must be used to select which chip is active. This process is called address decoding.

5. Address Decoding Techniques

Address decoding ensures that each memory location has a unique address and prevents **address contention** where multiple chips might respond to the same address. There are three main techniques:

a. No Decoding / Straight Decoding

• This is the simplest form and is used when only one memory chip (or very few, but small ones) is connected, and its capacity is significantly less than the MPU's address space.

• The higher-order address lines of the 8085 are left unconnected or tied to a fixed logic level (e.g., ground) to generate the Chip Select directly.

• Example: If we connect a 2KB ROM (11 address lines: A0-A10) directly to an 8085. We might connect A15 or a combination of higher address lines (A11-A15) directly to the Chip Select of the ROM. If A15 is connected to CS# (active low), the ROM will respond whenever A15 is low.

• Limitation: This creates **shadow memory** or **mirror mapping.** For instance, if a 2KB ROM is placed at addresses 0000H-07FFH by connecting A11-A15 to CS# through an inverter. If only A15 is used, the ROM would respond for all addresses 0000H-7FFFH (where A15 is 0) repeating the 2KB content multiple times. This wastes a lot of address space and is inefficient. It is only suitable for very small systems where address space is not a concern or for specific testing.

b. Partial Decoding

• In partial decoding, only a few of the higher-order address lines are used to generate the Chip Select signal.

• This technique is simple, cost-effective (requires fewer logic gates), and common in small to medium embedded systems where the full 64KB address space is not required, or only a few memory chips are used.

• Example: Consider connecting an 8KB RAM (needs A0-A12) and a 4KB ROM (needs A0-A11) to an 8085.

• For the 8KB RAM, we might use A15 and A14 to generate its CS#. If we connect A15-low AND A14-high to the CS# of RAM, the RAM would respond to addresses in the range 4000H-5FFFH. (Since A15=0, A14=1). The remaining higher address lines (A13) are left unconnected or ignored by the decoder.

• For the 4KB ROM, we might use A15-low, A14-low, and A13-high for its CS#. The ROM would respond to addresses in the range 2000H-2FFFH.
• Advantage: Simpler logic, fewer gates, lower cost.
• Disadvantage: Still suffers from some level of **mirror mapping.** Since not all higher address lines are decoded, multiple physical addresses will map to the same logical memory location within a chip. For example, if A13 is ignored, any change in A13 would still point to the same physical location within the 8KB RAM. This can be problematic for debugging and future expansion but might be acceptable in simple, fixed-function systems.

c. Full Decoding / Absolute Decoding
• This is the most robust and preferred method for larger systems or when optimal utilization of the address space is required.
• In full decoding, all unused higher-order address lines of the MPU (i.e., those not directly connected to the memory chip's address pins) are used by the decoding logic to generate the Chip Select signal.
• This ensures that each memory location has a unique address, preventing any mirror mapping.
• Decoding Logic Components:
• Logic Gates (AND, NAND, OR, NOR): For simple decoding involving a few address lines, basic logic gates can be combined.
• Decoders (e.g., 74LS138 - 3-to-8 line decoder): These are specialized ICs that take a small number of address lines as input and generate an active low output for each unique combination, effectively selecting one of eight possible chips or blocks of memory.
• Comparators: Used to compare a range of address lines with a predefined address range.

• Example using a 74LS138 Decoder:
• Suppose we want to connect two 8KB RAM chips and two 4KB ROM chips to an 8085.
• An 8KB chip uses A0-A12. A 4KB chip uses A0-A11.
• The 8085's address lines A13, A14, A15 will be used for decoding.
• We can connect A13, A14, A15 to the inputs (A, B, C) of a 74LS138 decoder.
• The 74LS138 has 8 active-low outputs (Y0# to Y7#).
• We also need to ensure the 74LS138 is enabled only during memory operations. This can be done by connecting the 8085's IO/M# signal (inverted if necessary) to one of the 74LS138's enable pins.
• Each output (Y0# to Y7#) can then be connected to the CS# pin of a specific memory chip or a block of memory, assigning it a unique 8KB or 4KB block of the 64KB address space.
• For instance, if A15-A13 are 000, Y0# is active. This can select a RAM chip for addresses 0000H-1FFFH. If A15-A13 are 001, Y1# is active, selecting another memory block for 2000H-3FFFH, and so on.
• Advantage: No mirror mapping, efficient use of address space, easier system expansion, and debugging.
• Disadvantage: More complex decoding logic, higher component count, and cost.

6. Memory Read and Write Cycle Operations
While specific timing diagrams are for future topics, understanding the control signal sequence is key:
• Memory Read:
1. The 8085 places the 16-bit address on the address bus (A0-A15, and AD0-AD7 multiplexed).
2. It asserts IO/M# low (memory operation).
3. The address decoding logic uses the higher address lines (A_n+1 to A15) to activate the Chip Select (CS#) of the target memory chip.
4. The 8085 asserts RD# low.
5. The selected memory chip, with its CS# and OE# (connected to RD#) active, places the data from the addressed location onto the data bus (D0-D7).
6. The 8085 reads the data and then de-asserts RD# and IO/M#.

• Memory Write:
1. The 8085 places the 16-bit address on the address bus.
2. It places the 8-bit data to be written onto the data bus.
3. It asserts IO/M# low.
4. The address decoding logic activates the Chip Select (CS#) of the target memory chip.
5. The 8085 asserts WR# low.
6. The selected memory chip, with its CS# and WE# (connected to WR#) active, latches the data from

the data bus into the addressed location.
7. The 8085 de-asserts WR# and IO/M#.

Summary of Key Points:
• Memory interfacing is about connecting memory chips to the 8085, requiring proper connection of address, data, and control lines.
• The 8085's 16-bit address bus allows access to 64KB of memory.
• Lower-order address lines of the 8085 directly connect to memory chip's address pins.
• Higher-order address lines are used for **address decoding** to select the correct memory chip via its Chip Select (CS#).
• Control signals (IO/M#, RD#, WR#) from the 8085 map to memory chip's control pins (OE#, WE#) to manage read/write operations.
• Address decoding ensures each memory location has a unique address.
• Decoding techniques include:
• No Decoding: Simplest but leads to extensive mirror mapping and wastes address space.
• Partial Decoding: Uses only some higher-order address lines, cost-effective, but still has mirror mapping. Suitable for smaller, fixed systems.
• Full/Absolute Decoding: Uses all higher-order address lines, requires more logic (gates, decoders like 74LS138), but provides unique addresses for every location, making it efficient and robust for larger systems.

8.) I/O Addressing and Interfacing

I/O Addressing and Interfacing

In the realm of microprocessor systems, the CPU not only needs to communicate with memory but also with various peripheral devices that allow it to interact with the outside world. These are known as Input/Output (I/O) devices. This topic delves into how the 8085 Microprocessor manages to communicate with these devices, specifically focusing on how they are addressed and how they are physically connected, or **interfaced,** to the system bus.

1. The Role of I/O Devices and the Need for Interfacing

Microprocessors are the brains of a computer system, but they are useless without the ability to receive input and produce output.
• Input Devices: Provide data to the microprocessor (e.g., keyboard, sensor, switch).
• Output Devices: Receive data from the microprocessor (e.g., display, LED, motor).
• Interfacing: The process of connecting these I/O devices to the microprocessor system. It's necessary because I/O devices often have different operating speeds, voltage levels, data formats, and control requirements compared to the microprocessor. Interfacing circuits bridge these differences, making devices compatible with the system bus.

2. I/O Addressing Schemes in 8085

The 8085 Microprocessor, like many others, uses addresses to select specific I/O devices, much like it addresses memory locations. There are two primary schemes for I/O addressing: Isolated I/O and Memory-Mapped I/O. The choice of scheme affects how the I/O devices are accessed and what instructions are used.

2.1. Isolated I/O (I/O-Mapped I/O)

• Concept: In this scheme, I/O devices have their own separate address space, distinct from the memory address space. This means an I/O device at address 05H is completely different from a memory location at 05H.
• 8085 Implementation:
• Dedicated Instructions: The 8085 uses special instructions, IN (Input) and OUT (Output), for accessing I/O ports.

• Control Signal: The IO/M (Input/Output or Memory) control signal is crucial here. When IO/M is low (0), the 8085 indicates an I/O operation. When it's high (1), it indicates a memory operation. This signal, in conjunction with the RD (Read) and WR (Write) signals, generates specific I/O Read (IOR) and I/O Write (IOW) control pulses.
• IOR = IO/M (low) AND RD (low)
• IOW = IO/M (low) AND WR (low)
• Address Lines: For Isolated I/O, the 8085 uses only the lower 8-bit address lines (AD0-AD7) to specify the I/O port address. This means the I/O address space is from 00H to FFH, allowing for 256 unique I/O port addresses. The contents of A8-A15 are ignored or typically replicated from AD0-AD7 for I/O operations (though not used for decoding).
• Advantages:
• Clear Distinction: Separate address spaces make it easy to distinguish between memory and I/O operations.
• Dedicated Instructions: IN/OUT instructions are specifically designed for I/O, simplifying programming logic.
• Smaller Address Decoders: Since only 8 address lines are used, the decoding logic for I/O ports can be simpler.
• Disadvantages:
• Limited I/O Ports: Only 256 unique I/O ports are available.
• Limited Instruction Set: Only IN and OUT instructions can be used. Data manipulation instructions (like MOV, ADD, SUB) cannot directly operate on I/O data. Data must first be moved to the Accumulator, then processed, and then moved back to an I/O port.

2.2. Memory-Mapped I/O

• Concept: In this scheme, I/O devices are treated as if they are memory locations. They share the same address space as memory. An I/O device might reside at memory address E000H, just like a RAM chip.
• 8085 Implementation:
• Standard Memory Instructions: Any instruction that can operate on memory locations (e.g., MOV, LDA, STA, ADD M, SUB M) can be used to access I/O devices.
• Control Signal: The IO/M signal is high (1) during a memory-mapped I/O access, just as it would be for a regular memory access. The system uses the MEMR (Memory Read) and MEMW (Memory Write) control pulses.
• MEMR = IO/M (high) AND RD (low)
• MEMW = IO/M (high) AND WR (low)
• Address Lines: All 16 address lines (A0-A15) are used to specify the I/O device address, just as they are for memory.
• Advantages:
• Larger I/O Space: I/O devices can be placed anywhere within the microprocessor's 64KB memory address space, limited only by the amount of physical memory installed.
• Rich Instruction Set: All memory-related instructions can be used to manipulate I/O data, offering greater flexibility and potentially more efficient code.
• Disadvantages:
• Reduced Memory Space: A portion of the available memory address space is sacrificed for I/O devices, reducing the total available memory for programs and data.
• Complex Decoding: All 16 address lines must be decoded, which can lead to more complex address decoding hardware.
• No Clear Distinction: It can be harder to distinguish between memory and I/O operations purely by looking at the address, requiring careful system design and documentation.

2.3. Comparison: Isolated I/O vs. Memory-Mapped I/O

• Address Space: Isolated uses 256 separate I/O addresses; Memory-mapped uses a portion of the 64KB memory space.
• Instructions: Isolated uses IN/OUT; Memory-mapped uses all memory-related instructions.
• Control Signals: Isolated uses IOR/IOW (IO/M=0); Memory-mapped uses MEMR/MEMW (IO/M=1).
• Address Lines Used for Decoding: Isolated uses A0-A7; Memory-mapped uses A0-A15.
• Flexibility: Memory-mapped generally offers more flexibility in data manipulation; Isolated is simpler

for basic port control.

3. I/O Interfacing Principles

Interfacing an I/O device to the 8085 involves several key components and considerations to ensure proper communication.

3.1. Need for Interfacing Components

- Address Decoding: To select the correct I/O device when its unique address appears on the address bus.
- Data Bus Buffering: To isolate the data bus from the I/O device and provide sufficient current drive.
- Latching/Buffering: To hold output data stable for output devices or to buffer input data for input devices.
- Control Signal Generation: To provide the correct read/write pulses to the I/O device at the right time.

3.2. Essential Components for Interfacing

- Address Decoders (e.g., 74LS138 3-to-8 line decoder):
- Purpose: To generate a unique **chip select** (CS) signal for a specific I/O device when its address is placed on the address bus.
- How it works: The higher-order address lines (A8-A15 for isolated I/O, or A0-A15 for memory-mapped I/O) are fed into the decoder. When the specific address matches, one of the decoder's output lines goes low, enabling the desired I/O chip. For the 8085, AD0-AD7 are multiplexed and must be demultiplexed for use as address lines A0-A7.
- Tri-state Buffers/Transceivers (e.g., 74LS245 bi-directional transceiver, 74LS244 for input, 74LS373/374 for output):
- Purpose: To connect the I/O device's data lines to the 8085's data bus (D0-D7). Tri-state buffers allow their outputs to be in one of three states: high, low, or high-impedance (effectively disconnected).
- Data Flow Control: For input, the buffer allows data to flow from the device to the data bus only when enabled for a read operation. For output, a latch/register is used to hold the data, and a transceiver can be used to control the direction of data flow if the port is bi-directional.
- Latches/Flip-flops (e.g., 74LS373 8-bit latch):
- Purpose: For output ports, the data from the microprocessor's data bus must be held stable for the output device, as the data bus is only active for a short duration. Latches capture and hold the data until the next write operation.
- Gating: The write control signal (IOW or MEMW) is typically used as the enable signal for the latch, ensuring data is captured only during a write operation.
- Buffers (e.g., 74LS244 8-bit buffer):
- Purpose: For input ports, buffers are used to isolate the input device from the data bus and to provide necessary current amplification if the input device cannot drive the bus directly. They also prevent noise or glitches from the input device from affecting the data bus.
- Gating: The read control signal (IOR or MEMR) is typically used to enable the buffer, allowing data to pass from the input device to the data bus only during a read operation.
- Control Signal Logic: Combinational logic (AND, OR gates) is often used to combine the 8085's primary control signals (IO/M, RD, WR) with the address decoder's chip select output to generate the precise I/O Read (IOR) or I/O Write (IOW) pulse needed by the I/O device.

4. Example: Simple I/O Port Interfacing with 8085 (Isolated I/O)

Let's consider interfacing a simple 8-bit output port and an 8-bit input port using the Isolated I/O scheme, as it clearly demonstrates the use of IN/OUT instructions and the IO/M signal.

4.1. Interfacing an 8-bit Output Port

- Goal: To turn on/off 8 LEDs connected to an output port. Let's assign it an I/O address of 00H.
- Components:
- 8085 Microprocessor.
- 74LS373 8-bit transparent latch: To hold the 8-bit data for the LEDs.

• Address Decoder (e.g., a simple NAND gate or 74LS138): To generate the chip select for address 00H.
• LEDs with current-limiting resistors.
• Operation:
• Address Decoding: Since we use address 00H, all A0-A7 lines would need to be 0 for a basic decoding. However, a common approach is to use a specific higher address line for the chip select of the port (e.g., A7=0, A6=0, ... A0=0). A simple way to get a unique address is to use a portion of the address lines. For a port at 00H, we would usually gate A7-A1, or more practically, use a decoder like the 74LS138 where some address lines are permanently set or used to select the decoder itself. For a simple example, let's assume a simplified decoder that activates when A0-A7 are all zero, outputting a CHIP_SELECT_PORT0.
• Control Signal: The IOW signal is generated by combining IO/M (low) and WR (low). This signal acts as the G (Gate) or LE (Latch Enable) for the 74LS373.
• Data Transfer: When the 8085 executes an OUT 00H instruction, it places 00H on AD0-AD7 (as address). The data to be outputted is placed on D0-D7 (after the address is latched). The IO/M goes low, and WR goes low, creating the IOW pulse.
• Latch Operation: The CHIP_SELECT_PORT0 (active low) from the address decoder is ANDed with IOW (active low) to generate the final latch enable signal. When this signal goes low, the 74LS373 latch captures the data present on the data bus (D0-D7) and outputs it to the LEDs. The LEDs will then display the output.

4.2. Interfacing an 8-bit Input Port

• Goal: To read the status of 8 switches connected to an input port. Let's assign it an I/O address of 01H.
• Components:
• 8085 Microprocessor.
• 74LS244 8-bit tri-state buffer: To allow switch data onto the data bus only when read.
• Address Decoder: Similar to the output port, to generate the chip select for address 01H.
• 8 switches, typically with pull-up or pull-down resistors.
• Operation:
• Address Decoding: The address decoder activates when address 01H is present, generating a CHIP_SELECT_PORT1 (active low).
• Control Signal: The IOR signal is generated by combining IO/M (low) and RD (low). This signal acts as the G (Output Enable) for the 74LS244.
• Data Transfer: When the 8085 executes an IN 01H instruction, it places 01H on AD0-AD7 (as address). The IO/M goes low, and RD goes low, creating the IOR pulse.
• Buffer Operation: The CHIP_SELECT_PORT1 (active low) from the address decoder is ANDed with IOR (active low) to generate the final enable signal for the 74LS244 buffer. When this signal goes low, the 74LS244 enables its outputs, placing the current status of the 8 switches onto the 8085's data bus (D0-D7). The 8085 then reads this data into its Accumulator.

5. Summary of Key Points

• I/O devices enable microprocessor interaction with the outside world.
• Interfacing bridges compatibility gaps between the microprocessor and I/O devices.
• Two main I/O addressing schemes exist: Isolated I/O and Memory-Mapped I/O.
• Isolated I/O uses a separate 256-port address space, IN/OUT instructions, and the IO/M signal (low). Only A0-A7 are used for port addressing.
• Memory-Mapped I/O treats I/O devices as memory locations, using memory-related instructions and the IO/M signal (high). All A0-A15 lines are used for addressing.
• Interfacing involves address decoding (to select the device), data bus buffering (to manage data flow), and latches/buffers (to hold output data or buffer input data).
• The 8085's control signals (IO/M, RD, WR) are crucial for generating the appropriate I/O Read (IOR) or I/O Write (IOW) pulses for device control.

9.) Data Transfer Schemes (Programmed, Interrupt-Driven, DMA)

Data Transfer Schemes in the 8085 Microprocessor

When a microprocessor like the 8085 needs to interact with external peripheral devices (like keyboards, displays, printers, or storage drives), it must exchange data with them. This data exchange involves reading data from an input device or writing data to an output device. The efficiency and speed of this data transfer are crucial for the overall performance of the system. To manage this interaction, several data transfer schemes are employed, each with its own advantages and disadvantages regarding CPU involvement, data rate, and hardware complexity.

The primary goal of these schemes is to move data between the microprocessor's internal registers or memory and the I/O devices. The 8085, with its specific pins and instruction set, supports these different methodologies.

1. Programmed I/O (Polled I/O)

This is the simplest and most direct method of data transfer, where the microprocessor directly controls the entire I/O operation.

• Explanation: In Programmed I/O, the CPU is entirely responsible for initiating, executing, and terminating the data transfer. It does this by continuously checking the status of the I/O device to see if it's ready to send or receive data. This continuous checking is often referred to as **polling.**

• Mechanism:
1. The CPU sends a command to the I/O device to perform an operation (e.g., **prepare to send data**).
2. The CPU then enters a loop, repeatedly reading the status register of the I/O device. This register contains flags indicating the device's current state (e.g., **buffer full** for input, **buffer empty** for output).
3. When the status flag indicates the device is ready (e.g., data is available from an input device), the CPU exits the loop.
4. The CPU then reads the data from the device's data register into one of its internal registers or writes data from an internal register to the device's data register.
5. This process is repeated for every byte of data to be transferred.

• 8085 Context: The 8085 uses its IN and OUT instructions for Programmed I/O.
• IN port_address: Reads data from the specified I/O port into the Accumulator.
• OUT port_address: Writes data from the Accumulator to the specified I/O port.
• Typically, one port address is assigned to the device's status register and another to its data register. The 8085 program would repeatedly execute an IN instruction to read the status, check a specific bit, and then execute another IN/OUT instruction for data transfer once ready.

• Example/Analogy: Imagine you are waiting for a very important email. With programmed I/O, you would constantly open your email client, refresh it, and check if the email has arrived, every few seconds, until it does.

• Advantages:
1. Simplicity: Easy to implement in both hardware and software. No special hardware (like interrupt controllers) is required beyond the basic I/O ports.
2. Low overhead for simple, infrequent transfers: For very short bursts of data or infrequent operations, the simplicity can be beneficial.

• Disadvantages:
1. CPU Busy-Waiting: The most significant drawback is that the CPU spends most of its time **busy-waiting** (polling) for the device to become ready. During this time, the CPU cannot perform any other useful computations, leading to a significant waste of processing power.
2. Inefficiency: Very inefficient for slow I/O devices, as the CPU might poll thousands of times while waiting for a single byte.
3. Low Throughput: The maximum data transfer rate is limited by how fast the CPU can poll and process data, and it monopolizes the CPU for the duration of the transfer.

2. Interrupt-Driven I/O

This scheme addresses the inefficiency of Programmed I/O by allowing the I/O device to signal the CPU only when it needs attention.

• Explanation: Instead of the CPU constantly checking the device, the I/O device itself notifies the CPU when it's ready for data transfer or when an event occurs. This notification is called an **interrupt.**

• Mechanism:
1. The CPU initiates an I/O operation by sending a command to the device.
2. The CPU then continues with other tasks or processes. It does not wait for the I/O device.
3. When the I/O device completes its operation (e.g., a character is received, or a print job is done) or requires service, it generates an interrupt signal on a specific pin of the CPU.
4. The CPU, upon detecting the interrupt, suspends its current task.
5. It saves the current state of its internal registers (context) onto the stack.
6. It then branches to a special routine called an Interrupt Service Routine (ISR) associated with that specific interrupt.
7. The ISR handles the data transfer or whatever service the device requires.
8. Once the ISR completes, the CPU restores its saved context from the stack and resumes the task it was performing before the interrupt occurred.

• 8085 Context: The 8085 Microprocessor has dedicated hardware interrupt pins (TRAP, RST7.5, RST6.5, RST5.5, INTR) to receive interrupt requests. When an interrupt occurs, the 8085 stops its current execution, acknowledges the interrupt (via the INTA signal for INTR), and jumps to a predefined memory location (or vector) associated with that interrupt, where the ISR begins. The ISR then uses IN/OUT instructions to transfer data with the device, similar to Programmed I/O, but only when needed.

• Example/Analogy: Continuing with the email analogy, with interrupt-driven I/O, you would configure your email client to notify you with a pop-up or sound when a new email arrives. You can do other work, and only when the notification appears do you switch to checking the email.

• Advantages:
1. CPU Efficiency: The CPU is free to perform other tasks while the I/O device prepares for data transfer. It only gets involved when explicitly requested by the device.
2. Responsiveness: Devices can be serviced more promptly as soon as they are ready, leading to better real-time behavior.
3. Higher Throughput: Generally offers higher throughput than Programmed I/O because CPU cycles are not wasted on polling.

• Disadvantages:
1. CPU Overhead: There's still significant CPU overhead for each interrupt. This includes saving the CPU's context, executing the ISR, and restoring the context. For very high data rates (byte by byte), this overhead can become substantial.
2. Hardware Complexity: Requires more complex hardware (interrupt controller, interrupt logic) and software (ISR design, interrupt handling).
3. Latency: A small delay (interrupt latency) occurs between the device requesting an interrupt and the CPU actually starting the ISR.

3. Direct Memory Access (DMA)

DMA is the most advanced and efficient method for high-speed data transfer, especially for large blocks of data. It minimizes CPU involvement in the actual data transfer process.

• Explanation: With DMA, the I/O device transfers data directly to or from the system's main memory without requiring the CPU to fetch or store each byte. A dedicated hardware component, called a DMA controller (DMAC), manages this transfer.

• Mechanism:
1. The CPU initiates the DMA transfer by programming the DMA controller with information such as:
   • The starting memory address for the transfer.

• The starting I/O device address.
• The number of bytes (or words) to transfer.
• The direction of transfer (read from device to memory, or write from memory to device).
2. After programming the DMAC, the CPU instructs the DMAC to start the transfer and then goes on to execute other tasks.
3. The DMAC requests control of the system bus (address, data, and control buses) from the CPU.
4. The CPU temporarily releases control of the buses and enters a **hold** or idle state, effectively becoming a spectator.
5. The DMAC then directly communicates with the memory and the I/O device, generating the necessary addresses and control signals to transfer data blocks directly between them.
6. Once the entire block transfer is complete, the DMAC signals the CPU (usually via an interrupt) that the transfer is finished.
7. The CPU then regains control of the system buses and resumes its normal operation.

• 8085 Context: The 8085 Microprocessor has two specific pins to facilitate DMA:
• HOLD: This is an input pin. When an external DMA controller wants to take control of the system bus, it asserts a high signal on the HOLD pin.
• HLDA (Hold Acknowledge): This is an output pin. When the 8085 receives a HOLD request, it finishes its current instruction execution, tri-states its address, data, and some control lines (like RD, WR, IO/M), and then asserts HLDA high to acknowledge that it has relinquished control of the buses. The DMA controller can then take over the buses. When the DMA controller is done, it de-asserts HOLD, and the 8085 then de-asserts HLDA and regains bus control.

• Example/Analogy: Imagine you're moving a large library of books. With DMA, you hire a moving company (DMAC). You tell them where the books are (source address), where they need to go (destination address), and how many boxes there are (count). Then you go about your day (CPU doing other tasks). The moving company directly loads and unloads the boxes. Once they're done, they notify you, and you can then interact with your books in their new location.

• Advantages:
1. Very High Data Rates: Achieves the fastest possible data transfer rates, as it bypasses the CPU entirely for the actual data movement.
2. CPU Free: The CPU is free to execute other programs during the data transfer, except for the short periods when the DMA controller **steals** bus cycles (cycle stealing DMA) or when the CPU is entirely put on hold (burst mode DMA). This significantly improves overall system performance for bulk transfers.
3. Efficient for Block Transfers: Ideal for moving large blocks of data between memory and high-speed peripherals (e.g., hard drives, network cards).

• Disadvantages:
1. Complexity: Requires a specialized hardware component (DMA controller), making the system more complex and costly.
2. Initial Setup Overhead: The CPU still needs to spend some time initially programming the DMA controller, but this is a one-time setup cost per transfer, amortized over a large block of data.
3. Bus Contention: DMA introduces temporary bus contention where the DMAC competes with the CPU for bus access, potentially causing slight delays for the CPU's current operations.

Summary of Key Points:

• Data transfer schemes dictate how the CPU and I/O devices exchange information.
• Programmed I/O involves the CPU continuously polling the device status, leading to wasted CPU cycles but offering simplicity. It uses IN/OUT instructions in 8085.
• Interrupt-Driven I/O allows the I/O device to signal the CPU when it needs attention, freeing the CPU for other tasks but incurring context-switching overhead for each byte. The 8085 uses its hardware interrupt pins.
• Direct Memory Access (DMA) enables I/O devices to transfer large blocks of data directly to/from memory without CPU involvement for each byte, leveraging a DMA controller and the 8085's HOLD/HLDA pins for maximum efficiency and throughput.

10.) 8085 Instruction Set Classification

Understanding the 8085 Instruction Set Classification is fundamental to programming the 8085 microprocessor. An instruction set is the complete collection of commands that a specific microprocessor can understand and execute. Each instruction tells the microprocessor to perform a very specific, elementary operation. The 8085, like any microprocessor, processes data, performs calculations, and controls the flow of its program by executing these instructions one after another.

Why classify these instructions? Because the 8085 has a rich set of 246 different instructions, each represented by a unique 8-bit binary code (opcode). Grouping them into categories makes it much easier to comprehend their purpose, remember them, and effectively use them for developing assembly language programs. This classification helps in systematically understanding the capabilities of the microprocessor and organizing programming logic.

The 8085 instruction set is broadly classified into five main categories based on the type of operation they perform. This categorization helps to logically organize the diverse functions the microprocessor can execute.

1. Data Transfer (Copy) Instructions

• Purpose: These instructions are responsible for moving (copying) data between various locations within the microprocessor system. This includes transferring data between internal registers, between a register and a memory location, between a register and an I/O port, or loading data directly into a register.
• Nature of Operation: It's crucial to understand that **data transfer** in this context implies copying. The content of the source location remains unchanged, while the content of the destination location is overwritten with the new data. These instructions do not modify the data itself; they simply relocate it.
• Reasoning: These instructions are foundational for any program as data needs to be moved into the ALU for processing, stored in memory for later use, or sent to/from external devices. They enable the microprocessor to access and manipulate data stored in different parts of its architecture.
• Examples:
• MOV R1, R2: Copies the 8-bit content of register R2 to register R1. For example, MOV B, C copies the value from register C to register B.
• MOV R, M: Copies the 8-bit content of the memory location pointed to by the HL register pair to register R. For example, MOV A, M copies data from memory to the Accumulator.
• MVI R, data: Moves an 8-bit immediate data byte directly into register R. For example, MVI A, 45H loads the hexadecimal value 45 into the Accumulator.
• MVI M, data: Moves an 8-bit immediate data byte directly into the memory location pointed to by the HL register pair.
• LDA address: Loads the 8-bit content of the memory location specified by the 16-bit address into the Accumulator. For example, LDA 2050H loads the data at memory address 2050H into the Accumulator.
• STA address: Stores the 8-bit content of the Accumulator into the memory location specified by the 16-bit address. For example, STA 3080H stores the Accumulator's content at address 3080H.
• LXI Rp, data16: Loads a 16-bit immediate data into the specified register pair (BC, DE, or HL). For example, LXI H, 4000H loads 4000H into the HL pair (H=40H, L=00H). This is often used to set up a memory pointer.
• SHLD address: Stores the 16-bit content of the HL register pair into two consecutive memory locations starting from the specified 16-bit address (L-register content at 'address', H-register content at 'address+1').
• LHLD address: Loads the 16-bit content from two consecutive memory locations into the HL register pair (L-register from 'address', H-register from 'address+1').
• PUSH Rp: Pushes the 16-bit content of a register pair (BC, DE, HL, or PSW - Program Status Word which includes A and Flag registers) onto the stack. This is a special type of data transfer involving the Stack Pointer.
• POP Rp: Pops the 16-bit content from the top of the stack into a specified register pair.
• IN port-address: Reads an 8-bit data byte from the specified 8-bit I/O port address and places it into the Accumulator.
• OUT port-address: Writes the 8-bit content of the Accumulator to the specified 8-bit I/O port

address.

2. Arithmetic Instructions

• Purpose: These instructions are used to perform mathematical operations such as addition, subtraction, increment, and decrement on 8-bit or 16-bit data.
• Nature of Operation: These operations modify the data by performing calculations. The results of these operations often affect the CPU's Flag Register, indicating conditions like carry, zero, sign, parity, and auxiliary carry. The Accumulator typically acts as one of the operands and stores the result for 8-bit operations.
• Reasoning: Arithmetic capabilities are essential for any computation, data manipulation, address calculations, and loop control within a program. The ALU (Arithmetic Logic Unit) inside the 8085 is responsible for executing these instructions.
• Examples:
• ADD R: Adds the 8-bit content of register R to the content of the Accumulator. The result is stored in the Accumulator. For example, ADD B adds B's content to A's.
• ADD M: Adds the 8-bit content of the memory location pointed to by HL to the Accumulator. Result in Accumulator.
• ADI data: Adds an 8-bit immediate data byte to the content of the Accumulator. Result in Accumulator. For example, ADI 0FH adds 0FH to A.
• ADC R: Adds the 8-bit content of register R and the Carry flag to the Accumulator. Result in Accumulator. This is useful for multi-byte additions.
• SUB R: Subtracts the 8-bit content of register R from the content of the Accumulator. The result is stored in the Accumulator. For example, SUB C subtracts C's content from A's.
• SUI data: Subtracts an 8-bit immediate data byte from the content of the Accumulator. Result in Accumulator.
• INR R: Increments the 8-bit content of register R by 1. For example, INR D increments register D.
• DCR R: Decrements the 8-bit content of register R by 1. For example, DCR L decrements register L.
• INR M: Increments the 8-bit content of the memory location pointed to by HL by 1.
• DCR M: Decrements the 8-bit content of the memory location pointed to by HL by 1.
• DAA (Decimal Adjust Accumulator): Adjusts the Accumulator content after an addition operation to obtain a valid BCD (Binary Coded Decimal) result.
• DAD Rp: Adds the 16-bit content of the specified register pair (BC, DE, or SP) to the content of the HL register pair. The result is stored in HL. This is a 16-bit addition, primarily used for address calculations.

3. Logical Instructions

• Purpose: These instructions perform bit-wise logical operations, such as AND, OR, XOR, complement, and rotations, on data. They are also used for comparing data values without affecting the operands.
• Nature of Operation: Logical instructions manipulate individual bits within a byte. Like arithmetic instructions, they often affect the Flag Register based on the result of the operation. Comparisons are crucial for decision-making in programs.
• Reasoning: Logical operations are vital for tasks like masking specific bits, setting/clearing bits, checking the status of individual bits, and implementing complex conditional logic.
• Examples:
• ANA R: Performs a bit-wise logical AND operation between the content of register R and the Accumulator. Result stored in Accumulator. For example, ANA B ANDs B with A.
• ANI data: Performs a bit-wise logical AND operation between an 8-bit immediate data byte and the Accumulator. Result stored in Accumulator.
• ORA R: Performs a bit-wise logical OR operation between the content of register R and the Accumulator. Result stored in Accumulator.
• ORI data: Performs a bit-wise logical OR operation between an 8-bit immediate data byte and the Accumulator. Result stored in Accumulator.
• XRA R: Performs a bit-wise logical XOR (Exclusive OR) operation between the content of register R and the Accumulator. Result stored in Accumulator.
• XRI data: Performs a bit-wise logical XOR operation between an 8-bit immediate data byte and the Accumulator. Result stored in Accumulator.

• CMP R: Compares the 8-bit content of register R with the Accumulator. This is effectively A - R, but the result is NOT stored. Only the flags are affected (Zero flag set if A=R, Carry flag set if A<R).
• CPI data: Compares an 8-bit immediate data byte with the Accumulator. Flags are affected.
• RLC (Rotate Left Accumulator): Rotates the bits of the Accumulator one position to the left. The most significant bit (D7) moves to the Carry flag and also to the least significant bit (D0).
• RRC (Rotate Right Accumulator): Rotates the bits of the Accumulator one position to the right. The least significant bit (D0) moves to the Carry flag and also to the most significant bit (D7).
• RAL (Rotate Left Accumulator through Carry): Rotates the bits of the Accumulator one position to the left. The most significant bit (D7) moves to the Carry flag, and the original Carry flag value moves to the least significant bit (D0).
• RAR (Rotate Right Accumulator through Carry): Rotates the bits of the Accumulator one position to the right. The least significant bit (D0) moves to the Carry flag, and the original Carry flag value moves to the most significant bit (D7).
• CMA (Complement Accumulator): Complements (flips all bits, 1's complement) the content of the Accumulator.
• CMC (Complement Carry): Complements the Carry flag.
• STC (Set Carry): Sets the Carry flag to 1.

4. Branching (Jump, Call, Return) Instructions

• Purpose: These instructions allow the program to alter its normal sequential flow of execution. Instead of executing the next instruction in memory, they can transfer control to a different part of the program.
• Nature of Operation: Branching instructions directly manipulate the Program Counter (PC), which holds the address of the next instruction to be executed. By loading a new address into the PC, the program **jumps** to that new location. They can be unconditional (always jump) or conditional (jump only if a specific flag condition is met).
• Reasoning: Program flow control is essential for implementing loops, conditional statements (if-then-else), and subroutines. Without branching, programs would be strictly linear and severely limited in functionality.
• Examples:
• JMP address: Unconditional Jump. Transfers program control to the 16-bit address specified. The PC is loaded with 'address'. For example, JMP 5000H immediately executes the instruction at 5000H.
• JZ address: Jump if Zero. Transfers program control to 'address' only if the Zero flag (Z) is set (i.e., the result of a previous operation was zero).
• JNZ address: Jump if Not Zero. Transfers program control to 'address' only if the Zero flag (Z) is reset (i.e., the result of a previous operation was non-zero).
• JC address: Jump if Carry. Transfers program control to 'address' only if the Carry flag (CY) is set.
• JNC address: Jump if Not Carry. Transfers program control to 'address' only if the Carry flag (CY) is reset.
• JP address: Jump if Plus (Sign flag is 0).
• JM address: Jump if Minus (Sign flag is 1).
• JPE address: Jump if Parity Even.
• JPO address: Jump if Parity Odd.
• CALL address: Unconditional Call. Used to invoke a subroutine. It first saves the address of the next instruction (return address) onto the stack and then transfers program control to the 16-bit 'address' of the subroutine.
• CC address: Call if Carry. Calls a subroutine only if the Carry flag is set.
• CNC address: Call if Not Carry.
• CZ address: Call if Zero.
• CNZ address: Call if Not Zero.
• RET: Unconditional Return. Used at the end of a subroutine. It retrieves the return address from the top of the stack and loads it into the PC, returning control to the instruction immediately after the CALL.
• RC: Return if Carry. Returns from a subroutine only if the Carry flag is set.
• RST n (Restart): A special type of call instruction, typically used for hardware interrupts, but also as a software instruction. It saves the current PC onto the stack and then transfers control to a fixed memory location (00H, 08H, 10H, etc., depending on 'n').

5. Stack, I/O, and Machine Control Instructions

• Purpose: This category groups instructions that don't fit neatly into the previous four. They handle specific operations related to the stack, input/output with peripheral devices, and overall control of the microprocessor's state.
• Nature of Operation: These instructions provide control over the microprocessor's operational mode (like enabling/disabling interrupts), allow direct interaction with external hardware via I/O ports, and manage the stack, which is a crucial area of memory for temporary data storage and subroutine management.
• Reasoning: These instructions provide the necessary mechanisms for the 8085 to interact with its external environment, manage temporary data efficiently, and control its internal operational state.
• Examples:
• PUSH Rp: (As mentioned in Data Transfer, but critical for Stack operations) Pushes a register pair onto the stack.
• POP Rp: (As mentioned in Data Transfer, but critical for Stack operations) Pops a register pair from the stack.
• SPHL: Copies the content of the HL register pair to the Stack Pointer (SP).
• XTHL: Exchanges the content of the top of the stack with the HL register pair.
• IN port-address: (As mentioned in Data Transfer, but critical for I/O operations) Reads data from an I/O port into the Accumulator.
• OUT port-address: (As mentioned in Data Transfer, but critical for I/O operations) Writes data from the Accumulator to an I/O port.
• EI (Enable Interrupts): Enables the maskable interrupts of the 8085.
• DI (Disable Interrupts): Disables the maskable interrupts of the 8085.
• HLT (Halt): Stops the processor's execution. The processor enters a low-power state and waits for an interrupt or a reset to resume operation.
• NOP (No Operation): Performs no operation other than incrementing the Program Counter. It effectively does nothing, often used for timing delays or padding.
• SIM (Set Interrupt Mask): Used to set the interrupt mask bits for RST7.5, RST6.5, RST5.5 interrupts and also for serial output data (SOD).
• RIM (Read Interrupt Mask): Used to read the current status of interrupt masks, pending interrupts, and serial input data (SID).

Summary of Key Points:

The 8085 instruction set is systematically classified into five categories to simplify understanding and programming:
1. Data Transfer (Copy) Instructions: Move data between registers, memory, and I/O devices without altering the source data.
2. Arithmetic Instructions: Perform mathematical operations like addition, subtraction, increment, and decrement, often affecting status flags.
3. Logical Instructions: Execute bitwise logical operations (AND, OR, XOR, complement, rotate) and comparisons, also affecting status flags.
4. Branching Instructions: Alter the program's execution flow using jumps, calls to subroutines, and returns, either unconditionally or conditionally based on flag status.
5. Stack, I/O, and Machine Control Instructions: Manage stack operations, handle data transfer with external I/O devices, and control the overall state of the microprocessor (e.g., halt, interrupt enable/disable).

This classification provides a structured framework for learning and applying the 8085's capabilities, laying the groundwork for more advanced assembly language programming.

11.) 8085 Addressing Modes

8085 ADDRESSING MODES

Addressing modes are the different ways in which a microprocessor can specify the operand for an

instruction. An operand is the data on which the instruction operates. Understanding addressing modes is crucial because it dictates how efficiently and flexibly data can be accessed and manipulated by the CPU. The 8085 microprocessor supports five fundamental addressing modes, each designed for specific scenarios of data access, allowing programmers to choose the most suitable method for their needs. These modes determine where the CPU should look to find the data or the memory location that an instruction needs to process.

The necessity of different addressing modes arises from the varied ways data is used in programs: some data might be constant, some might be in CPU registers for fast access, some in specific memory locations, and some might need to be accessed indirectly through pointers. Each mode has implications for instruction size, execution speed, and programming flexibility.

The 8085 Microprocessor primarily supports the following 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 of these in detail.

1. IMPLICIT OR IMPLIED ADDRESSING MODE

This is the simplest addressing mode where the operand is not explicitly specified in the instruction. Instead, the instruction itself implicitly operates on a fixed or specific register. The operand is **implied** by the opcode of the instruction. This means the CPU inherently knows which register to operate on as part of the instruction's definition.

- Description: Instructions using this mode do not require any additional operand information. They are typically one-byte instructions. The CPU automatically understands which register or memory location to use.
- Reasoning: This mode is used for operations that always affect the same, predetermined internal register. It simplifies the instruction format and makes instructions very compact and fast, as the CPU doesn't need to fetch any operand address or data from memory after fetching the opcode.
- Example Instructions:
- CMA (Complement Accumulator): This instruction complements the contents of the Accumulator (A register) byte by byte. No operand is specified because it is always the Accumulator that is complemented. The CPU directly fetches the content of the A register, performs the bitwise NOT operation, and stores the result back into the A register.
- RAL (Rotate Accumulator Left): This instruction rotates the bits of the Accumulator one bit to the left through the Carry flag. Again, the Accumulator is the implied operand.
- STC (Set Carry Flag): Sets the Carry flag to 1. The operand here is implicitly the Carry flag within the Flag register.
- DAA (Decimal Adjust Accumulator): Adjusts the content of the Accumulator to form a BCD number after an addition. The Accumulator is the implied operand.
- Operation: When an instruction like CMA is fetched, the 8085's control unit immediately knows to perform the complement operation on the Accumulator. No further memory access is needed for the operand. This makes these instructions very efficient in terms of execution cycles and memory footprint (1 byte).

2. REGISTER ADDRESSING MODE

In this mode, the operand is located in one of the 8085's general-purpose registers or special-purpose registers. The instruction specifies the register(s) containing the data.

- Description: The instruction itself contains the name or code of the register where the operand is stored. The 8085's general-purpose registers include B, C, D, E, H, L, and the Accumulator (A).
- Reasoning: This mode allows for very fast data access and manipulation because the data is already within the CPU's internal high-speed storage. It's ideal for computations and data transfers

between registers, avoiding slower memory access cycles.

• Example Instructions:

• MOV A, B (Move data from register B to register A): Here, 'B' is the source operand and 'A' is the destination operand. Both are registers. The instruction fetches the content of register B and copies it into register A. The original content of register B remains unchanged.

• ADD C (Add content of register C to Accumulator): The source operand is register C, and the implied destination operand is the Accumulator. The 8085 reads the value from C, adds it to the value in A, and stores the result back in A. The content of C remains unchanged.

• INR D (Increment content of register D): The operand is register D. Its content is incremented by 1.

• DCR E (Decrement content of register E): The operand is register E. Its content is decremented by 1.

• Operation: For MOV A, B, the opcode tells the CPU to transfer data between specific internal registers. This is a very fast operation, typically completing within one machine cycle (after opcode fetch), as it doesn't involve external memory access for the operand. Most register addressing instructions are 1-byte long.

3. DIRECT ADDRESSING MODE

In direct addressing mode, the instruction contains the 16-bit memory address of the operand. The CPU directly accesses this specified memory location to fetch or store the data.

• Description: The full 16-bit address of the memory location where the operand resides is explicitly provided as part of the instruction itself.

• Reasoning: This mode is useful when you need to access a specific, fixed memory location. It provides a straightforward way to load data from or store data into a known memory address, such as specific I/O ports (memory-mapped I/O) or global variables.

• Example Instructions:

• LDA 2000H (Load Accumulator Direct with content of memory location 2000H): Here, '2000H' is the 16-bit address. The instruction tells the 8085 to go to memory location 2000H, fetch the 8-bit data stored there, and load it into the Accumulator.

• STA 3050H (Store Accumulator Direct to memory location 3050H): The instruction takes the 8-bit content of the Accumulator and stores it into memory location 3050H.

• LHLD 4000H (Load H and L registers Direct with contents of memory locations 4000H and 4001H): This is a 3-byte instruction. It fetches the data from 4000H into L register and data from 4001H into H register.

• SHLD 5000H (Store H and L registers Direct to memory locations 5000H and 5001H): Stores the content of L into 5000H and H into 5001H.

• Operation: Instructions in direct addressing mode are typically 3-byte instructions: the first byte is the opcode, and the next two bytes provide the 16-bit memory address (low-order byte first, then high-order byte). After fetching the opcode, the CPU fetches the two address bytes, reconstructs the full 16-bit address, and then performs a memory read or write operation at that address. This involves more memory accesses (for the address bytes and then for the data itself) compared to implied or register modes, making it slower.

4. REGISTER INDIRECT ADDRESSING MODE

In register indirect addressing, the instruction specifies a register pair that holds the 16-bit memory address of the operand. The content of this register pair acts as a pointer to the memory location where the actual data resides.

• Description: Instead of providing the explicit memory address within the instruction, a register pair (BC, DE, or HL) is used to hold the 16-bit address. The Mnemonic 'M' in 8085 instructions often denotes the memory location addressed by the HL register pair.

• Reasoning: This mode is highly flexible and powerful. It allows for dynamic memory access, such as working with arrays, implementing pointers, and accessing data structures where the address is not fixed at compile time but calculated or changed during program execution. It also allows 1-byte instructions to access memory, making code more compact than direct addressing for repeated memory operations.

• Example Instructions:

• MOV A, M (Move data from memory pointed by HL to Accumulator): Here, 'M' represents the

memory location whose address is currently stored in the HL register pair. The CPU takes the 16-bit address from HL, accesses that memory location, and copies its 8-bit content into the Accumulator.

• MOV M, B (Move data from register B to memory pointed by HL): The 8-bit content of register B is stored into the memory location whose address is in the HL pair.

• ADD M (Add content of memory pointed by HL to Accumulator): The 8-bit content from the memory location addressed by HL is added to the Accumulator, and the result is stored in the Accumulator.

• LDAX B (Load Accumulator Indirect using BC pair): The Accumulator is loaded with the content of the memory location whose address is specified by the BC register pair.

• STAX D (Store Accumulator Indirect using DE pair): The content of the Accumulator is stored into the memory location whose address is specified by the DE register pair.

• Operation: Instructions like MOV A, M are 1-byte instructions. After fetching the opcode, the CPU reads the 16-bit address from the specified register pair (e.g., HL). It then uses this address to perform a memory read or write operation. This mode is efficient for sequential memory access, as the register pair can be incremented or decremented to point to the next memory location without needing to modify the instruction itself.

5. IMMEDIATE ADDRESSING MODE

In immediate addressing mode, the operand data itself is included as part of the instruction. The instruction explicitly contains the actual 8-bit or 16-bit data value that needs to be processed.

• Description: The data to be operated upon is provided directly after the opcode in the instruction stream. It's not an address, but the actual value.

• Reasoning: This mode is used to load constant values into registers or memory locations. It's efficient for initializing registers with known values or performing operations with fixed constants.

• Example Instructions:

• MVI A, 05H (Move Immediate data 05H to Accumulator): '05H' is the 8-bit immediate data. The instruction fetches this data and loads it into the Accumulator.

• MVI B, FFH (Move Immediate data FFH to register B): The 8-bit immediate data FFH is loaded into register B.

• ADI 10H (Add Immediate data 10H to Accumulator): The 8-bit immediate data 10H is added to the content of the Accumulator, and the result is stored in the Accumulator.

• LXI H, 2000H (Load eXtended Immediate data 2000H into HL pair): Here, '2000H' is the 16-bit immediate data. The high-order byte (20H) is loaded into register H, and the low-order byte (00H) is loaded into register L.

• Operation: Instructions in immediate addressing mode are either 2-byte (for 8-bit data) or 3-byte (for 16-bit data) instructions. For MVI A, 05H, the first byte is the opcode, and the second byte is the immediate data (05H). The CPU fetches the opcode, then fetches the data byte, and then loads it into the specified register. For LXI H, 2000H, the first byte is the opcode, the second byte is 00H (low-order data), and the third byte is 20H (high-order data). The CPU fetches these two data bytes and loads them into L and H respectively.

SUMMARY OF KEY POINTS:

• Addressing modes define how the 8085 Microprocessor finds the operand for an instruction.

• There are five main addressing modes: Implicit, Register, Direct, Register Indirect, and Immediate.

• Implicit Addressing: Operand is fixed and known to the instruction itself (e.g., CMA operates on Accumulator). These are 1-byte instructions and very fast.

• Register Addressing: Operand is in a CPU register specified by the instruction (e.g., MOV A, B). These are 1-byte instructions and very fast.

• Direct Addressing: The instruction contains the exact 16-bit memory address of the operand (e.g., LDA 2000H). These are 3-byte instructions, slower due to more memory fetches.

• Register Indirect Addressing: A register pair (BC, DE, or HL) holds the 16-bit memory address of the operand (e.g., MOV A, M where M refers to HL content). These are typically 1-byte instructions, offering flexibility for dynamic memory access.

• Immediate Addressing: The actual data value (8-bit or 16-bit) is part of the instruction itself (e.g., MVI A, 05H or LXI H, 2000H). These are 2-byte or 3-byte instructions, used for loading constant values.

• The choice of addressing mode impacts instruction size, execution speed, and programming flexibility, allowing the programmer to optimize code for different scenarios.

12.) Assembly Language Programming Basics

Assembly Language Programming Basics

Assembly language serves as a crucial bridge between high-level programming languages and the raw machine code that a microprocessor, like the 8085, directly executes. Understanding assembly language is fundamental for grasping how a computer truly operates at its lowest level, how hardware interacts with software, and for developing highly optimized or hardware-specific applications.

1. What is Assembly Language?

Assembly language is a low-level programming language that uses mnemonic codes and symbols to represent machine code instructions. Instead of using binary (0s and 1s), which is difficult for humans to read and write, assembly language provides a more human-readable representation. Each assembly language instruction typically corresponds to one machine language instruction.

• Relation to Machine Code: Machine code is the native language of the microprocessor, consisting of binary instructions (opcodes and operands). Assembly language provides symbolic representations (mnemonics) for these binary codes, making programming less error-prone and more manageable than directly writing binary.
• Relation to High-Level Languages: High-level languages (like C, Java, Python) are far more abstract, using statements that compile into many machine code instructions. Assembly language offers direct control over the microprocessor's hardware components, such as registers, memory, and I/O ports.
• The Assembler: A special program called an Assembler translates assembly language code into machine code. For the 8085, an assembler takes a .ASM source file and converts it into an object file (e.g., .OBJ or .HEX), which can then be loaded into the microprocessor's memory for execution.
• Why use Assembly Language:
• Direct Hardware Control: Essential for embedded systems, device drivers, and operating system kernels where direct manipulation of hardware resources is required.
• Performance Optimization: Assembly code can be meticulously optimized for speed and memory usage, critical in resource-constrained environments or for time-sensitive tasks.
• Understanding Microprocessor Architecture: It provides deep insights into how the CPU's registers, ALU, and control unit work together to execute instructions.

2. Basic Program Structure (8085 Context)

An assembly language program for the 8085 is a sequence of instructions and directives, organized to perform a specific task.

• Structure Elements:
• Labels: Symbolic names given to memory locations (either instructions or data). They make code more readable and allow jumping to specific points in the program without knowing their exact memory address. Labels are terminated with a colon (e.g., START:).
• Opcodes (Operation Codes): Mnemonics representing the actual operation the microprocessor should perform (e.g., MOV for move, ADD for add, JMP for jump).
• Operands: The data or memory addresses on which the opcode operates. An instruction can have zero, one, or two operands (e.g., MOV A, B; JMP LOOP; NOP).
• Comments: Explanatory notes ignored by the assembler. They are crucial for code understanding and maintenance. In 8085 assembly, comments usually start with a semicolon (;).

Example Line:
LOOP: MOV A, M ; Move content of memory pointed by HL to Accumulator

3. Instruction Cycle and Execution Flow (Recap and Detail)

The 8085 microprocessor executes instructions in a continuous cycle, which involves fetching, decoding, and executing each instruction.

• Fetch: The Program Counter (PC) holds the address of the next instruction to be fetched. The CPU places this address on the address bus, reads the opcode from memory, and stores it in the Instruction Register (IR). The PC is then incremented to point to the next byte (which could be another opcode or an operand).
• Decode: The instruction in the IR is decoded by the instruction decoder and the timing and control unit. This unit determines what operation needs to be performed and what resources (registers, ALU, memory) are required.
• Execute: Based on the decoded instruction, the control unit generates the necessary control signals to perform the operation. This might involve:
• Reading data from registers or memory.
• Performing an arithmetic or logical operation in the ALU.
• Writing data to registers or memory.
• Updating the Program Counter for jumps or calls.
• Modifying the Flag Register based on the result of an operation.

The execution of one instruction prepares the system for fetching the next instruction, leading to a sequential flow unless explicitly altered by a jump, call, or return instruction.

4. Working with Data

Assembly language provides instructions to define, load, store, and manipulate data within the 8085's registers and memory.

• Data Representation: Data can be represented in binary, hexadecimal, or decimal form within the assembly code, though the microprocessor internally works with binary. For example, MVI A, 05H means move hexadecimal 05 into accumulator.
• Storing Data in Registers:
• MVI R, Data: Move Immediate 8-bit Data to Register R. (e.g., MVI B, 45H - loads 45H into register B).
• LXI Rp, Data16: Load Register Pair Immediate 16-bit Data. (e.g., LXI H, 2000H - loads 2000H into the HL register pair).
• MOV Rd, Rs: Move data from source register Rs to destination register Rd. (e.g., MOV A, B - copies content of B to A).
• Storing Data in Memory:
• STA address: Store Accumulator Direct. (e.g., STA 3050H - stores the content of A into memory location 3050H).
• LDA address: Load Accumulator Direct. (e.g., LDA 3050H - loads the content of memory location 3050H into A).
• MOV M, R: Move data from register R to memory location pointed by HL. (e.g., MOV M, B - copies B's content to the memory location whose address is in HL).
• MOV R, M: Move data from memory location pointed by HL to register R. (e.g., MOV A, M - copies content of memory at HL to A).
• SHLD address: Store HL Direct. (e.g., SHLD 4000H - stores L register content at 4000H and H register content at 4001H).
• LHLD address: Load HL Direct. (e.g., LHLD 4000H - loads content of 4000H into L and 4001H into H).
• Basic Arithmetic Operations:
• ADD R: Add Register R to Accumulator. (e.g., ADD B - A = A + B). Affects flags.
• ADI Data: Add Immediate Data to Accumulator. (e.g., ADI 05H - A = A + 05H). Affects flags.
• SUB R: Subtract Register R from Accumulator. (e.g., SUB C - A = A - C). Affects flags.
• SUI Data: Subtract Immediate Data from Accumulator. (e.g., SUI 10H - A = A - 10H). Affects flags.
• INR R: Increment Register R by 1. (e.g., INR B - B = B + 1). Affects all flags except Carry.
• DCR R: Decrement Register R by 1. (e.g., DCR C - C = C - 1). Affects all flags except Carry.
• INX Rp: Increment Register Pair by 1. (e.g., INX H - HL = HL + 1). Does NOT affect any flags.
• DCX Rp: Decrement Register Pair by 1. (e.g., DCX D - DE = DE - 1). Does NOT affect any flags.
• Logical Operations (briefly):
• ANA R / ANI Data: Logical AND with Accumulator.
• ORA R / ORI Data: Logical OR with Accumulator.
• XRA R / XRI Data: Logical XOR with Accumulator.

- CMA: Complement Accumulator (bitwise NOT).

These operations are essential for bit manipulation and testing.

5. Program Flow Control

Typically, instructions are executed sequentially. However, to implement decision-making and looping, control flow instructions are used. The 8085 uses its Flag Register to make conditional decisions.

- Sequential Execution: After each instruction, the PC is automatically incremented to point to the next instruction in memory.
- Unconditional Jumps:
- JMP address: Jump unconditionally to the specified 16-bit address. (e.g., JMP START). The PC is loaded with the new address, altering the flow.
- Conditional Jumps: These instructions check the status of specific flags in the Flag Register (Zero, Carry, Sign, Parity) and jump only if the condition is met. If the condition is false, execution continues to the next instruction.
- JZ address: Jump if Zero flag is set (Z=1).
- JNZ address: Jump if Zero flag is reset (Z=0).
- JC address: Jump if Carry flag is set (C=1).
- JNC address: Jump if Carry flag is reset (C=0).
- JP address: Jump if Positive (Sign flag S=0).
- JM address: Jump if Minus (Sign flag S=1).
- JPE address: Jump if Parity Even (Parity flag P=1).
- JPO address: Jump if Parity Odd (Parity flag P=0).

Example of a Loop:

```
MVI C, 0AH ; Initialize counter C with 10 (0A in hex)
LOOP: NOP ; Do nothing for demonstration
DCR C ; Decrement counter
JNZ LOOP ; If C is not zero, jump back to LOOP
```

6. Assembler Directives (Pseudo-operations)

Directives are instructions for the assembler, not for the microprocessor. They guide the assembler in organizing the program, allocating memory, and defining data.

- ORG address (Origin): Sets the starting address for the subsequent code or data. The assembler will place the following machine code or data starting from this address.

Example: ORG 2000H ; Program starts at memory address 2000H

- END: Marks the end of the source program. Any text after this directive is ignored by the assembler. Every assembly program must have an END directive.
- EQU value (Equate): Assigns a symbolic name to a constant value. This improves readability and maintainability.

```
Example: COUNT EQU 10H ; Define COUNT as hexadecimal 10
MVI B, COUNT ; Move 10H into B
```

- DB value(s) (Define Byte): Reserves one or more memory bytes and initializes them with the specified 8-bit value(s).

```
Example: DATA1 DB 05H, 10H, 15H ; Defines 3 bytes with values 5, 10, 15
MESSAGE DB 'HELLO' ; Defines 5 bytes for ASCII characters
```

- DW value(s) (Define Word): Reserves one or more memory words (16-bit) and initializes them with the specified 16-bit value(s). A word typically means two bytes, stored in little-endian format (low byte first, then high byte).

Example: ADDRESS DW 2050H ; Defines a word with value 2050H (50H at current loc, 20H at next loc)

- DS count (Define Storage): Reserves a specified number of uninitialized memory bytes. Useful for creating buffers or temporary storage areas.

Example: BUFFER DS 20H ; Reserves 20 (32 decimal) bytes for a buffer

7. Writing a Simple 8085 Assembly Program (Example)

Let's write a program to add two 8-bit numbers stored in memory and store the result in another memory location.

```
ORG 2000H ; Program starts at 2000H

START: LXI H, 2050H ; Initialize HL with address 2050H (first number)
MOV A, M ; Move content of 2050H into Accumulator (A = num1)
INX H ; Increment HL to point to 2051H (second number)
ADD M ; Add content of 2051H to Accumulator (A = num1 + num2)
INX H ; Increment HL to point to 2052H (result location)
MOV M, A ; Store the result from Accumulator into 2052H
HLT ; Halt the processor

; Data section (can be defined anywhere, but good practice to separate)
ORG 2050H ; Data starts at 2050H
NUM1 DB 0AH ; First number = 10 (decimal)
NUM2 DB 05H ; Second number = 5 (decimal)
RESULT DB 00H ; Space for result (will be 0FH or 15 decimal)

END ; End of assembly program
```

Explanation:

- The program starts at 2000H.
- LXI H, 2050H loads the 16-bit address 2050H into the HL register pair. HL now points to NUM1.
- MOV A, M copies the byte at the memory location pointed to by HL (which is 0AH from NUM1) into the Accumulator.
- INX H increments HL to 2051H, pointing to NUM2.
- ADD M adds the byte at the memory location pointed to by HL (which is 05H from NUM2) to the Accumulator. A now holds 0AH + 05H = 0FH. The flags (like Zero, Carry) would be updated based on this addition.
- INX H increments HL to 2052H, pointing to RESULT.
- MOV M, A copies the content of the Accumulator (0FH) into the memory location pointed to by HL (2052H, which is RESULT).
- HLT stops the microprocessor execution.
- The ORG 2050H and DB directives define the data values in memory starting from address 2050H.

8. Debugging Basics (Conceptual)

Debugging is the process of finding and fixing errors in a program. In assembly language, this often requires a detailed understanding of the microprocessor's state.

- Syntax Errors: These are caught by the assembler during the translation phase (e.g., misspelled opcodes, incorrect operand types). The assembler will generate error messages.
- Logic Errors: The program assembles correctly but doesn't produce the desired output. These are harder to find.
- Tracing: Stepping through the program instruction by instruction, observing the changes in register contents, flag states, and memory locations. This is typically done using an emulator, simulator, or an in-circuit debugger.
- Breakpoints: Halting execution at specific points to examine the system state.
- Examining Memory and Registers: Checking if data is loaded, processed, and stored as expected.

Summary of Key Points:

- Assembly language uses mnemonics for machine instructions, making low-level programming more manageable than direct binary.
- An Assembler translates assembly code into executable machine code.
- 8085 assembly programs consist of labels, opcodes, operands, and comments.
- The microprocessor executes instructions via a Fetch-Decode-Execute cycle, with the Program Counter driving sequential flow.

• Data is manipulated using instructions to move, load, store, and perform arithmetic/logical operations on registers and memory.
• Program flow control is managed by unconditional (JMP) and conditional (JZ, JNC, etc.) jump instructions, which rely on the 8085's Flag Register.
• Assembler Directives (like ORG, END, DB, DW, DS, EQU) are essential for organizing code, allocating memory, and defining data for the assembler.
• Writing assembly programs requires careful planning of register usage, memory addresses, and instruction sequences.
• Debugging involves identifying and correcting syntax and logic errors, often through tracing and examining the microprocessor's internal state.

13.) Stack and Subroutines

The Stack and Subroutines in the 8085 Microprocessor

The 8085 microprocessor, like most modern processors, utilizes a dedicated memory area known as the 'stack' to manage temporary data storage and facilitate the execution of subroutoutines. Understanding the stack's operation and how subroutines interact with it is fundamental for efficient assembly language programming and grasping the processor's internal control flow.

1. Introduction to the Stack

The stack is a special area of RAM (Random Access Memory) used for temporary data storage. It operates on a Last-In, First-Out (LIFO) principle, similar to a stack of plates: the last plate placed on top is the first one removed. In the 8085, the stack typically grows downwards in memory, meaning new data is stored at successively lower memory addresses.

• Why do we need a Stack?
The stack provides a structured way to temporarily save the contents of registers or specific memory locations. This is crucial for:
• Preserving the state of the CPU before executing a subroutine.
• Storing return addresses for subroutines.
• Passing parameters between routines.
• Handling interrupts (though that's a future topic, the stack's role is similar).

• The Stack Pointer (SP) Register
The 8085 microprocessor has a dedicated 16-bit register called the Stack Pointer (SP). This register always holds the 16-bit memory address of the topmost element of the stack. Because the stack grows downwards, the SP is decremented when data is pushed onto the stack and incremented when data is popped off.

• Stack Memory
The stack is not a separate physical memory unit; it's simply a designated portion of the general-purpose RAM. The programmer must initialize the SP with a starting 16-bit address. For example, if the user decides to use memory locations from CFFFH down to C000H as the stack, the SP would initially be loaded with the address CFFFH (or C000H depending on convention, but for 8085 it points to the **next available** lower address after a push). A common practice is to initialize SP to an address just *after* the highest memory location intended for the stack, so the first PUSH operation effectively stores data starting from the highest available stack address. For example, if the stack is from CFFFH down, SP might be initialized to D000H, and the first PUSH will decrement SP to CFFE-CFFF. More accurately, if SP is loaded with CFFFH, the first PUSH operation will decrement SP to CFFE H and store data at CFFE H and CFFD H.

2. 8085 Stack Operations - PUSH

The PUSH instruction is used to store the content of a register pair onto the stack. Since the 8085 is an

8-bit processor, but its registers are often handled in pairs (BC, DE, HL), the stack operations typically involve 16-bit data.

- Instruction Format: PUSH Rp
- Rp refers to a valid register pair: B (BC), D (DE), H (HL).
- An exception is PUSH PSW, which pushes the contents of the Accumulator (A) and the Flag Register (PSW stands for Program Status Word, which is essentially A and F combined for stack operations).

- Internal Mechanism of PUSH Rp:

A PUSH instruction always stores 16 bits (two bytes) of data onto the stack. The order of storage is crucial:
1. The Stack Pointer (SP) is decremented by 1.
2. The content of the high-order register of the specified pair (e.g., register B for PUSH B, register D for PUSH D, register H for PUSH H, or register A for PUSH PSW) is copied to the memory location pointed to by the new SP.
3. The Stack Pointer (SP) is again decremented by 1.
4. The content of the low-order register of the specified pair (e.g., register C for PUSH B, register E for PUSH D, register L for PUSH H, or the Flag Register for PUSH PSW) is copied to the memory location pointed to by the new SP.

- Example: PUSH B

Assume:
- SP = 20FFH
- Register B = 11H
- Register C = 22H

After PUSH B:
1. SP becomes 20FEH.
2. Memory location 20FEH gets the value of B (11H).
3. SP becomes 20FDH.
4. Memory location 20FDH gets the value of C (22H).

The stack now holds 11H at 20FEH and 22H at 20FDH, and the SP points to 20FDH. Note that the high-order byte is stored at the higher address, and the low-order byte is stored at the lower address, relative to the pair.

3. 8085 Stack Operations - POP

The POP instruction is used to retrieve 16 bits (two bytes) of data from the stack and load it into a specified register pair. It performs the reverse operation of PUSH.

- Instruction Format: POP Rp
- Rp refers to a valid register pair: B (BC), D (DE), H (HL).
- An exception is POP PSW, which restores the contents of the Flag Register and the Accumulator.

- Internal Mechanism of POP Rp:

1. The content of the memory location pointed to by SP (which contains the low-order byte of the previously pushed pair) is copied to the low-order register of the specified pair (e.g., register C for POP B, register E for POP D, register L for POP H, or the Flag Register for POP PSW).
2. The Stack Pointer (SP) is incremented by 1.
3. The content of the memory location pointed to by SP (which now contains the high-order byte) is copied to the high-order register of the specified pair (e.g., register B for POP B, register D for POP D, register H for POP H, or register A for POP PSW).
4. The Stack Pointer (SP) is again incremented by 1.

- Example: POP B (following the previous PUSH B example)

Assume:
- SP = 20FDH
- Memory location 20FDH = 22H

- Memory location 20FEH = 11H
- Registers B and C contain arbitrary values (e.g., B=AAH, C=BBH) before POP B.

After POP B:
1. Register C gets the value from 20FDH (22H).
2. SP becomes 20FEH.
3. Register B gets the value from 20FEH (11H).
4. SP becomes 20FFH.

Registers B and C now contain 11H and 22H respectively, restoring their values from before the PUSH. The SP is back to its original position (20FFH) before the PUSH B operation.

4. Introduction to Subroutines

A subroutine is a self-contained block of code that performs a specific task. It can be called from different parts of the main program or even from other subroutines. In high-level languages, these are known as functions or procedures.

- Why do we need Subroutines?
- Modularity: Breaks down a complex problem into smaller, manageable, and logically separate tasks, making the code easier to understand, debug, and maintain.
- Reusability: A subroutine can be written once and called multiple times from different locations in the program, eliminating redundant code and saving memory space.
- Code Size Reduction: Instead of repeating the same sequence of instructions, a single subroutine call is used, significantly reducing the overall program size.

- Main Program vs. Subroutine

The 'main program' is the primary sequence of instructions that typically initiates the execution. A 'subroutine' is a separate block of code that the main program temporarily transfers control to. After the subroutine completes its task, control is returned to the exact point in the main program where it was called.

- Transferring Control to and from a Subroutine

This transfer of control requires a mechanism to remember where to return. This is where the stack plays a crucial role. When a subroutine is called, the address of the next instruction in the main program (the 'return address') is saved on the stack. When the subroutine finishes, this return address is retrieved from the stack, allowing the program execution to resume correctly in the main program.

5. 8085 Subroutine Call and Return

The 8085 uses specific instructions for calling and returning from subroutines: CALL and RET.

- CALL Instruction: CALL addr
- 'addr' is the 16-bit starting memory address of the subroutine.
- When the 8085 encounters a CALL instruction, it performs the following steps:

1. It determines the address of the instruction immediately following the CALL instruction in the main program. This is the 'return address'. The PC (Program Counter) already points to the next instruction after the CALL opcode and its 2-byte address. So, the PC's current value is the return address.
2. The Stack Pointer (SP) is decremented by 1.
3. The high-order byte of the return address (from PC) is stored at the memory location pointed to by the new SP.
4. The Stack Pointer (SP) is again decremented by 1.
5. The low-order byte of the return address (from PC) is stored at the memory location pointed to by the new SP.
6. Finally, the Program Counter (PC) is loaded with the 16-bit 'addr' specified in the CALL instruction. This transfers program execution control to the beginning of the subroutine.

- Example: CALL 2050H

Assume:

• Main program instruction: CALL 2050H at address 1000H.
• The instruction after CALL 2050H is at 1003H (since CALL is a 3-byte instruction: opcode at 1000H, low-order address at 1001H, high-order address at 1002H). So, the return address is 1003H.
• SP = 20FFH

After CALL 2050H:
1. SP becomes 20FEH.
2. Memory location 20FEH gets the high-order byte of the return address (10H).
3. SP becomes 20FDH.
4. Memory location 20FDH gets the low-order byte of the return address (03H).
5. PC is loaded with 2050H.

The CPU now executes instructions starting from 2050H. The stack (at 20FDH-20FEH) contains 1003H, and SP points to 20FDH.

• RET Instruction: RET
• This instruction is placed at the end of a subroutine to return control to the calling program.
• When the 8085 encounters a RET instruction, it performs the following steps:
1. The content of the memory location pointed to by SP (which contains the low-order byte of the return address) is loaded into the low-order byte of the Program Counter (PC).
2. The Stack Pointer (SP) is incremented by 1.
3. The content of the memory location pointed to by SP (which now contains the high-order byte of the return address) is loaded into the high-order byte of the Program Counter (PC).
4. The Stack Pointer (SP) is again incremented by 1.

• Example: RET (following the previous CALL example)
Assume:
• RET instruction is at the end of the subroutine (e.g., at 2060H).
• SP = 20FDH
• Memory location 20FDH = 03H (low byte of return address)
• Memory location 20FEH = 10H (high byte of return address)

After RET:
1. The low-order byte of PC gets 03H from 20FDH.
2. SP becomes 20FEH.
3. The high-order byte of PC gets 10H from 20FEH.
4. SP becomes 20FFH.

The PC now contains 1003H, and execution resumes at that address in the main program. The SP is restored to its state before the CALL instruction.

6. Conditional CALL and RET Instructions

Just like there are conditional JUMP instructions, the 8085 also provides conditional CALL and RET instructions. These instructions only execute the call or return operation if a specific flag (Zero, Carry, Parity, Sign) is set or reset.

• Conditional CALLs: CC (Call if Carry), CNC (Call if No Carry), CZ (Call if Zero), CNZ (Call if No Zero), CP (Call if Positive), CM (Call if Minus), JPC (Call if Parity Even), JPO (Call if Parity Odd).
• Example: CZ 2050H - Calls subroutine at 2050H only if the Zero flag is set.
• Conditional RETs: RC (Return if Carry), RNC (Return if No Carry), RZ (Return if Zero), RNZ (Return if No Zero), RP (Return if Positive), RM (Return if Minus), RPE (Return if Parity Even), RPO (Return if Parity Odd).
• Example: RZ - Returns from subroutine only if the Zero flag is set. Otherwise, execution continues with the next instruction in the subroutine.

7. Nested Subroutines

A nested subroutine occurs when one subroutine calls another subroutine. This is perfectly handled by

the stack mechanism. Each CALL instruction pushes a new return address onto the stack. When a RET instruction is encountered, the topmost return address is popped and used.

- Example Scenario:

Main Program:

- ...
- CALL SUB1 (Address of next instruction, e.g., M_RET1, is pushed)
- M_RET1: ...

Subroutine SUB1:

- ...
- CALL SUB2 (Address of next instruction, e.g., S1_RET1, is pushed)
- S1_RET1: ...
- RET (Pops S1_RET1, returns to SUB1)

Subroutine SUB2:

- ...
- RET (Pops M_RET1, returns to Main Program)

- The stack ensures that return addresses are retrieved in the correct LIFO order, allowing control to return sequentially through nested calls. It's critical to ensure that every CALL has a corresponding RET; otherwise, the stack can become unbalanced, leading to incorrect program execution (e.g., trying to return to a non-existent address or overflowing the stack).

8. Parameter Passing to Subroutines

While not directly a stack operation, the stack can be used for passing parameters. In the 8085, parameters are often passed:

- Through registers: Data is loaded into registers (A, B, C, D, E, H, L) before calling the subroutine. The subroutine uses these registers, and results might be returned in them.
- Through pre-defined memory locations: Data is stored in specific memory addresses that both the calling program and the subroutine agree upon.
- On the Stack: Parameters can be PUSHed onto the stack by the calling program before the CALL instruction. The subroutine would then POP these parameters. However, if the subroutine also PUSHes its own return address, care must be taken to retrieve parameters from below the return address on the stack. This is less common for simple 8085 programming than using registers.

9. Stack Overflow and Underflow

Improper stack management can lead to issues:

- Stack Overflow: Occurs if too much data is pushed onto the stack, exceeding the allocated stack memory space. This can overwrite other critical data or code in memory, leading to program crashes or unpredictable behavior. This happens if you have too many PUSH operations without corresponding POPs, or too many nested CALLs without corresponding RETs, for the available stack size.
- Stack Underflow: Occurs if a POP operation is attempted when the stack is empty (or has fewer elements than expected). This will retrieve garbage data and can also lead to incorrect program execution, as the SP points to an invalid or unintended memory location. This happens if you have too many POP operations without corresponding PUSHs.

Careful initialization of the SP and balanced PUSH/POP and CALL/RET operations are essential for robust program design.

Summary of Key Points:

- The stack is a LIFO (Last-In, First-Out) memory region used for temporary data storage and managing subroutine calls.
- The 8085's Stack Pointer (SP) is a 16-bit register that always points to the topmost element of the stack. The stack typically grows downwards in memory.
- PUSH Rp instruction stores a 16-bit register pair (or A and F flags for PUSH PSW) onto the stack, decrementing SP twice. The high-order byte is stored at (SP-1) and the low-order byte at (SP-2) relative

to the SP before the operation.
• POP Rp instruction retrieves a 16-bit register pair (or A and F flags for POP PSW) from the stack, incrementing SP twice. It first retrieves the low-order byte from SP, then the high-order byte from (SP+1).
• Subroutines are reusable blocks of code that improve program modularity and reduce code size.
• The CALL addr instruction saves the 16-bit return address (the address of the instruction immediately following CALL) onto the stack and then loads the PC with the subroutine's starting address.
• The RET instruction retrieves the 16-bit return address from the stack and loads it into the PC, returning control to the calling program.
• Conditional CALL and RET instructions allow for subroutine calls or returns based on the status of the 8085's flag register.
• Nested subroutines are handled efficiently by the stack, as each CALL pushes its return address, and each RET pops the most recent one.
• Proper stack management is crucial to prevent stack overflow (exceeding stack memory) and stack underflow (popping from an empty stack).

14.) 8085 Interrupt Structure

The 8085 Interrupt Structure is a fundamental concept in microprocessor-based systems, enabling the CPU to respond efficiently to external or internal events. Instead of continuously checking for events (polling), which wastes CPU time, interrupts allow a device or a program to signal the CPU when it needs attention. When an interrupt occurs, the CPU temporarily suspends its current task, services the interrupt, and then resumes the original task from where it left off. This mechanism is crucial for real-time systems, efficient I/O operations, and handling unexpected events.

Here's a detailed breakdown of the 8085's interrupt structure:

1. Introduction to Interrupts in 8085

• An interrupt is a signal that causes the microprocessor to temporarily halt its current program execution and execute a special routine called an Interrupt Service Routine (ISR).
• After the ISR completes its task, the microprocessor returns to the original program.
• This provides a much more efficient way to handle asynchronous events compared to polling.

2. Types of Interrupts (Contextual Overview)

• Interrupts can broadly be classified into Hardware Interrupts and Software Interrupts.
• The 8085 supports both. Our focus will be on the structure provided by its hardware pins for external interruption.
• (Note: Detailed discussion on Hardware vs. Software Interrupts is a future topic.)

3. 8085's Hardware Interrupt Pins

The 8085 Microprocessor has five dedicated hardware interrupt pins, along with an interrupt acknowledgment signal:

• TRAP
• RST 7.5 (Restart 7.5)
• RST 6.5 (Restart 6.5)
• RST 5.5 (Restart 5.5)
• INTR (Interrupt Request)
• INTA' (Interrupt Acknowledge - an output signal from 8085)

4. Interrupt Prioritization

The 8085 has a fixed priority scheme for its hardware interrupts. If multiple interrupt requests arrive

simultaneously, the CPU will service the one with the highest priority first. The priority order, from highest to lowest, is:

- TRAP (Highest)
- RST 7.5
- RST 6.5
- RST 5.5
- INTR (Lowest)

5. Vectored vs. Non-Vectored Interrupts

Interrupts can also be categorized by how the CPU finds the starting address of the Interrupt Service Routine (ISR):

- Vectored Interrupts: The microprocessor automatically knows the starting address of the ISR once the interrupt is acknowledged. These addresses are fixed in the memory map.
- Non-Vectored Interrupts: The microprocessor does not automatically know the ISR address. An external device must provide an instruction (typically a CALL or RST instruction) to guide the CPU to the ISR.

Classification for 8085 interrupts:
- Vectored: TRAP, RST 7.5, RST 6.5, RST 5.5
- Non-Vectored: INTR

6. Interrupt Enable/Disable Mechanism

The 8085 provides instructions to control when interrupts are processed:

- EI (Enable Interrupts): This instruction sets the Interrupt Enable Flip-Flop, allowing maskable interrupts to be recognized. It should be used after the system is initialized or after a critical section of code.
- DI (Disable Interrupts): This instruction resets the Interrupt Enable Flip-Flop, preventing maskable interrupts from being recognized. It is often used before entering a critical section of code where an interrupt could cause data corruption or timing issues.
- All maskable interrupts (RST 7.5, RST 6.5, RST 5.5, INTR) are affected by the EI/DI instructions. TRAP is non-maskable and is unaffected by EI/DI.
- Additionally, RST 7.5, 6.5, and 5.5 can be individually masked or unmasked using the SIM instruction.

7. Detailed Explanation of Each Hardware Interrupt

a. TRAP

- Nature: Non-maskable, highest priority.
- Triggering: It is both edge-sensitive and level-sensitive. A rising edge on the TRAP pin sets an internal flip-flop, and the interrupt remains active as long as the TRAP pin is high. Once acknowledged, the interrupt is not re-recognized until the pin goes low and then high again. This ensures that a spurious noise pulse doesn't re-trigger it and also that a sustained high signal is recognized reliably.
- Purpose: Reserved for critical events like power failure, emergency shutdown, or memory parity errors, where immediate attention is required and the interrupt should not be ignored.
- Vector Address: 0024H. The CPU automatically jumps to this address to fetch the first instruction of the ISR.
- Unaffected by EI/DI instructions and cannot be masked by any software instruction.

b. RST 7.5

- Nature: Maskable, second highest priority.
- Triggering: It is edge-sensitive (rising edge). A rising edge on the RST 7.5 pin sets an internal flip-flop. Even if the signal goes low immediately, the interrupt request is latched and held until serviced. This latching ensures short pulses are not missed.

• Masking: Can be enabled/disabled globally by EI/DI. Can be individually masked or unmasked using the SIM (Set Interrupt Mask) instruction.
• Resetting: The internal flip-flop (latch) that stores the RST 7.5 request can be explicitly reset using the SIM instruction.
• Vector Address: 003CH. The CPU automatically jumps to this address.

c. RST 6.5

• Nature: Maskable, third highest priority.
• Triggering: It is level-sensitive (high level). The interrupt request is active only as long as the RST 6.5 pin is held high. If the pin goes low before the CPU acknowledges it, the request is lost.
• Masking: Can be enabled/disabled globally by EI/DI. Can be individually masked or unmasked using the SIM instruction.
• Vector Address: 0034H. The CPU automatically jumps to this address.

d. RST 5.5

• Nature: Maskable, fourth highest priority.
• Triggering: It is level-sensitive (high level). Similar to RST 6.5, the interrupt request is active only when the RST 5.5 pin is held high.
• Masking: Can be enabled/disabled globally by EI/DI. Can be individually masked or unmasked using the SIM instruction.
• Vector Address: 002CH. The CPU automatically jumps to this address.

e. INTR

• Nature: Maskable, lowest priority.
• Triggering: It is level-sensitive (high level). The INTR pin must be held high until the 8085 acknowledges it.
• Non-Vectored: Unlike the other four, the INTR interrupt does not have a fixed vector address.
• Acknowledgment: When the 8085 recognizes an INTR request (and it's enabled and has highest priority among pending maskable interrupts), it completes the current instruction, then sends out an INTA' (Interrupt Acknowledge) signal.
• Instruction Fetch: Upon receiving INTA', the external interrupting device must place an instruction on the data bus. This instruction is typically an RST n (Restart n, where n is 0-7) or a CALL instruction, which directs the CPU to the ISR's starting address.
• RST n instructions have fixed vector addresses:
• RST 0 -> 0000H
• RST 1 -> 0008H
• RST 2 -> 0010H
• RST 3 -> 0018H
• RST 4 -> 0020H
• RST 5 -> 0028H
• RST 6 -> 0030H
• RST 7 -> 0038H
• A CALL instruction allows jumping to any 16-bit address specified by the external device.
• Masking: Can be enabled/disabled globally by EI/DI. It cannot be individually masked using SIM.
• This non-vectored nature of INTR provides flexibility, allowing multiple external devices to share the INTR line using an external interrupt controller (e.g., 8259, which is a future topic).

8. The SIM (Set Interrupt Mask) Instruction

The SIM instruction is a single-byte instruction used to:
• Set (mask) or reset (unmask) the RST 7.5, RST 6.5, and RST 5.5 interrupt masks.
• Reset the RST 7.5 internal flip-flop.
• Set the Serial Output Data (SOD) latch. (SID/SOD are future topics, but SIM is responsible for SOD control).

Before executing SIM, the Accumulator must be loaded with an 8-bit pattern defining the desired

operations:

- Bit D7 (SOD): Controls the Serial Output Data pin. If D7=1 and D6=1 (SOE), the SOD pin takes the value of D7.
- Bit D6 (SOE): Serial Output Enable. If SOE=1, the SOD pin is enabled for output.
- Bit D5 (RST 6.5 Mask): If D5=1, RST 6.5 is masked (disabled). If D5=0, RST 6.5 is unmasked (enabled).
- Bit D4 (RST 5.5 Mask): If D4=1, RST 5.5 is masked. If D4=0, RST 5.5 is unmasked.
- Bit D3 (MSE): Mask Set Enable. This bit must be set to 1 (MSE=1) for bits D5, D4, and D2 to take effect for masking/unmasking. If MSE=0, changes to D5, D4, D2 are ignored.
- Bit D2 (RST 7.5 Mask): If D2=1, RST 7.5 is masked. If D2=0, RST 7.5 is unmasked.
- Bit D1 (RST 7.5 Reset): If D1=1, the internal latched flip-flop for RST 7.5 is reset. This clears any pending RST 7.5 request. This bit is edge-sensitive; it only works on the transition from 0 to 1, or when it is 1, it will reset.
- Bit D0 (Unused): Typically set to 0.

Example: To unmask RST 7.5, mask RST 6.5, and unmask RST 5.5, while resetting the RST 7.5 flip-flop:
Accumulator content: X010101X (where X is don't care, assuming D7, D6 not active for SOD).

- D7: X
- D6: X
- D5: 1 (Mask RST 6.5)
- D4: 0 (Unmask RST 5.5)
- D3: 1 (MSE must be 1 for masking bits to take effect)
- D2: 0 (Unmask RST 7.5)
- D1: 1 (Reset RST 7.5 flip-flop)
- D0: X

So, ACC = X010101X, e.g., 00101010B = 2AH.

```
MVI A, 2AH
SIM
```

9. The RIM (Read Interrupt Mask) Instruction

The RIM instruction is a single-byte instruction used to:

- Read the current status of the interrupt masks (for RST 7.5, 6.5, 5.5).
- Read the pending status of the interrupt requests (for RST 7.5, 6.5, 5.5).
- Read the status of the Interrupt Enable Flip-Flop.
- Read the Serial Input Data (SID) pin. (SID/SOD are future topics, but RIM is responsible for SID control).

After executing RIM, the Accumulator is loaded with an 8-bit pattern:

- Bit D7 (SID): Serial Input Data. Contains the value of the SID pin.
- Bit D6 (I7.5): Interrupt Pending for RST 7.5. If 1, RST 7.5 is pending.
- Bit D5 (I6.5): Interrupt Pending for RST 6.5. If 1, RST 6.5 is pending.
- Bit D4 (I5.5): Interrupt Pending for RST 5.5. If 1, RST 5.5 is pending.
- Bit D3 (IE): Interrupt Enable Flag. If 1, interrupts are currently enabled (by EI instruction). If 0, disabled (by DI instruction or implicitly after an interrupt is serviced).
- Bit D2 (M7.5): Mask Status for RST 7.5. If 1, RST 7.5 is masked.
- Bit D1 (M6.5): Mask Status for RST 6.5. If 1, RST 6.5 is masked.
- Bit D0 (M5.5): Mask Status for RST 5.5. If 1, RST 5.5 is masked.

Example: To read the current status:

```
RIM
MOV B, A (Copy Accumulator to B to save the status for analysis)
```

10. General Interrupt Handling Process (Flow)

When an interrupt request is asserted:

- Current Instruction Completion: The 8085 finishes executing its current instruction. It does not

interrupt mid-instruction.
• Interrupt Recognition: The CPU checks if the interrupt is enabled (via EI/DI for maskable interrupts) and if its priority is high enough compared to other pending interrupts.
• Acknowledgment (for INTR): For an INTR, the CPU asserts the INTA' signal.
• Program Counter (PC) Save: The CPU automatically saves the current content of the Program Counter (PC) onto the stack (by pushing it). This is crucial for returning to the main program later.
• Interrupt Disable: The Interrupt Enable Flip-Flop is automatically reset (disabled) by the hardware after acknowledging any maskable interrupt (TRAP also disables subsequent interrupts temporarily, though it's non-maskable). This prevents another interrupt from interrupting the ISR itself, unless explicitly re-enabled.
• Vectoring/Branching:
• For TRAP, RST 7.5, RST 6.5, RST 5.5: The CPU loads the fixed vector address into the PC.
• For INTR: The CPU reads the instruction (RST n or CALL addr) provided by the external device on the data bus and executes it. This instruction then directs the PC to the ISR's address.
• ISR Execution: The Interrupt Service Routine (ISR) starts executing. It performs the necessary tasks to service the interrupt.
• Interrupt Re-enable (Optional): If the ISR needs to allow further interrupts while it is executing (e.g., a high-priority interrupt needs to be serviced during a lower-priority ISR), it can explicitly use the EI instruction.
• Return from ISR: The ISR must end with a RET (Return) instruction. This instruction pops the previously saved PC value from the stack back into the PC, allowing the CPU to resume the main program from where it was interrupted.

Summary of Key Points:

• The 8085 has five hardware interrupt pins: TRAP, RST 7.5, RST 6.5, RST 5.5, and INTR.
• These interrupts have a fixed priority: TRAP > RST 7.5 > RST 6.5 > RST 5.5 > INTR.
• TRAP, RST 7.5, RST 6.5, and RST 5.5 are vectored interrupts with fixed memory locations for their ISRs.
• INTR is a non-vectored interrupt, requiring an external device to supply the ISR address via an instruction during INTA'.
• TRAP is non-maskable and highest priority, useful for critical system failures.
• RST 7.5, 6.5, 5.5, and INTR are maskable.
• The EI and DI instructions globally enable and disable maskable interrupts.
• The SIM instruction is used to individually mask/unmask RST 7.5, 6.5, 5.5, and reset the RST 7.5 latch.
• The RIM instruction is used to read the current mask status, pending interrupt requests, and the global interrupt enable flag.
• When an interrupt occurs, the 8085 saves the current PC on the stack, disables further interrupts, jumps to the ISR, and after the ISR, the RET instruction restores the PC from the stack to resume the main program.

15.) Hardware and Software Interrupts

An interrupt in a microprocessor system is a signal that temporarily stops the normal execution of a program to handle a high-priority event or an external request. Once the event is handled, the microprocessor returns to its original task. This mechanism is crucial for efficient I/O operations and handling asynchronous events without constantly polling devices. The 8085 Microprocessor supports both hardware and software interrupts, each serving distinct purposes.

• Hardware Interrupts

Hardware interrupts originate from external devices or specific conditions within the system, signaling the microprocessor to suspend its current task. These interrupts are typically asynchronous to the currently executing program. The 8085 has five hardware interrupt pins: TRAP, RST 7.5, RST 6.5, RST 5.5, and INTR.

• Types of Hardware Interrupts based on Characteristics:

1- Maskable vs. Non-Maskable:

• Maskable interrupts can be enabled or disabled by software. This allows the programmer to control when the microprocessor should respond to certain interrupts. In the 8085, the Interrupt Enable (IE) flip-flop, controlled by the EI (Enable Interrupts) and DI (Disable Interrupts) instructions, globally enables or disables all maskable interrupts. Additionally, specific masks for RST 7.5, RST 6.5, and RST 5.5 can be set using the SIM (Set Interrupt Mask) instruction.

• Non-maskable interrupts cannot be disabled by software. They are always active and are reserved for critical events that must be handled immediately, such as power failure or emergency shutdowns. TRAP is the 8085's only non-maskable interrupt.

2- Vectored vs. Non-Vectored:

• Vectored interrupts automatically direct the microprocessor to a fixed, predetermined memory location (called a vector address) where the Interrupt Service Routine (ISR) for that specific interrupt is located. This saves time as the microprocessor doesn't need to determine which device interrupted. TRAP, RST 7.5, RST 6.5, and RST 5.5 are all vectored interrupts in the 8085. Their vector addresses are 0024H, 003CH, 0034H, and 002CH, respectively.

• Non-vectored interrupts do not have a fixed vector address. When such an interrupt occurs, the microprocessor must first determine the source of the interrupt. In the 8085, INTR is a non-vectored interrupt. After receiving an INTR signal, the 8085 enters an interrupt acknowledge cycle, during which the interrupting device must provide an instruction (typically an RST instruction or a CALL instruction) on the data bus to direct the CPU to the appropriate ISR.

3- Edge-Triggered vs. Level-Triggered:

• Edge-triggered interrupts are sensitive to a change in the signal level (e.g., a rising edge from low to high). Once triggered by an edge, the interrupt is registered, even if the signal level changes again. For RST 7.5, the 8085 has an internal D flip-flop that latches the rising edge, ensuring that a brief pulse is not missed and that a continuous high signal doesn't cause repeated interrupts.

• Level-triggered interrupts are sensitive to the sustained level of the signal (e.g., held high). The interrupt remains active as long as the signal is at the specified level. If the signal drops before the microprocessor acknowledges it, the interrupt might be missed. TRAP is special as it's both edge- and level-sensitive; it's triggered on a rising edge and remains active as long as the signal is high. RST 6.5 and RST 5.5 are level-triggered. INTR is also level-triggered.

• 8085 Hardware Interrupt Pins in Detail:

1- TRAP:

• This is the highest priority interrupt and is non-maskable.

• It is unique in its triggering mechanism: it's activated by a rising edge and held high simultaneously. This means it must go from low to high and stay high to be recognized. If it goes low again before the CPU acknowledges it, it won't be recognized. This dual sensitivity prevents false triggering and ensures that critical events (like power failure) are handled.

• Its vector address is 0024H.

• Due to its non-maskable nature, TRAP is reserved for critical situations where the system must respond regardless of the current program state, such as power failure, catastrophic system errors, or emergency shutdowns.

2- RST 7.5:

• This is the second highest priority interrupt and is maskable.

• It is edge-triggered (rising edge). An internal D flip-flop latches the rising edge, meaning a pulse on this pin is registered even if the signal doesn't remain high. This is useful for devices that provide only a momentary pulse.

• The D flip-flop can be reset by a DI instruction, a system reset, or by using the SIM instruction (setting the RST 7.5 mask bit).

• Its vector address is 003CH.

• It is commonly used for time-critical external events where a brief signal needs to be captured reliably.

3- RST 6.5:
- This is the third highest priority interrupt and is maskable.
- It is level-triggered (active high). The interrupt signal must remain high until it is acknowledged by the CPU. If the signal goes low before acknowledgement, the interrupt is lost.
- Its vector address is 0034H.
- Suitable for devices that can hold their interrupt request line high until serviced.

4- RST 5.5:
- This is the fourth highest priority interrupt and is maskable.
- It is also level-triggered (active high), similar to RST 6.5.
- Its vector address is 002CH.
- Used for general-purpose interrupt requests, often from slower peripheral devices.

5- INTR:
- This is the lowest priority interrupt among the hardware interrupts and is maskable.
- It is level-triggered (active high).
- It is a non-vectored interrupt. When INTR goes high and interrupts are enabled, the 8085 completes its current instruction and then sends an INTA (Interrupt Acknowledge) signal. The external device, upon receiving INTA, is responsible for placing an 8-bit instruction (typically an RST n instruction or a CALL instruction followed by the address) on the data bus for the 8085 to execute.
- This flexibility allows multiple devices to share the INTR line via an external interrupt controller (like the 8259A, a future topic), where the controller arbitrates between devices and provides the correct vector.
- Because of its flexibility, INTR is widely used for general-purpose I/O devices where an explicit vector address is not hardwired into the CPU.

- Interrupt Priority in 8085:

The 8085 has a fixed priority scheme for its hardware interrupts, from highest to lowest:
TRAP > RST 7.5 > RST 6.5 > RST 5.5 > INTR.
If multiple interrupts occur simultaneously, the one with higher priority is serviced first.

- Interrupt Control Instructions:
- EI (Enable Interrupts): Sets the Interrupt Enable flip-flop, allowing maskable interrupts to be processed.
- DI (Disable Interrupts): Resets the Interrupt Enable flip-flop, preventing maskable interrupts from being processed.
- SIM (Set Interrupt Mask): This instruction is used to mask (disable) or unmask (enable) the individual RST 7.5, 6.5, and 5.5 interrupts. It also can be used to reset the RST 7.5 flip-flop. The 8-bit accumulator content determines the mask settings.
- RIM (Read Interrupt Mask): This instruction reads the current status of the interrupt masks and the pending interrupt requests for RST 7.5, 6.5, and 5.5 into the accumulator. It also indicates the status of the Interrupt Enable flip-flop.

- Software Interrupts

Software interrupts are generated by executing specific instructions within a program. Unlike hardware interrupts, which are external and asynchronous, software interrupts are synchronous and are explicitly programmed by the developer. They function very similarly to a CALL instruction but are typically single-byte instructions and jump to fixed, predetermined memory locations (vector addresses).

- 8085 Software Interrupt Instructions:

The 8085 uses RST (Restart) instructions from RST 0 to RST 7 as software interrupts. Each RST n instruction is a 1-byte instruction.
- RST 0 (C7H): Jumps to memory location 0000H
- RST 1 (CFH): Jumps to memory location 0008H
- RST 2 (D7H): Jumps to memory location 0010H
- RST 3 (DFH): Jumps to memory location 0018H
- RST 4 (E7H): Jumps to memory location 0020H
- RST 5 (EFH): Jumps to memory location 0028H

• RST 6 (F7H): Jumps to memory location 0030H
• RST 7 (FFH): Jumps to memory location 0038H

• Mechanism of Software Interrupts:
When an RST n instruction is executed, the 8085 performs the following actions:
1- The current content of the Program Counter (PC), which points to the instruction immediately after the RST instruction, is pushed onto the stack. This ensures the microprocessor can return to the main program.
2- The PC is then loaded with the fixed vector address corresponding to RST n (n * 8).
3- The microprocessor then fetches and executes the instruction at this new PC address, which is the beginning of the ISR for that software interrupt.

• Purpose of Software Interrupts:
• System Calls: They can be used to implement system calls in operating systems, providing a standardized way for user programs to request services from the kernel.
• Debugging: They are often used as breakpoints during debugging. By replacing an instruction with an RST instruction, the debugger can gain control when that point in the code is reached.
• Common Subroutines: They provide a quick way to call frequently used short subroutines without needing a 3-byte CALL instruction, saving memory and execution time.
• Error Handling: Can be programmed to jump to specific error handling routines.

• Key Differences and Similarities:

• Differences:
• Origin: Hardware interrupts originate from external devices; software interrupts are initiated by an instruction within the program.
• Asynchronous/Synchronous: Hardware interrupts are asynchronous (can occur at any time); software interrupts are synchronous (occur only when the RST instruction is executed).
• Maskability: Most hardware interrupts are maskable (except TRAP); software interrupts are not maskable (they are simply instructions).
• Trigger: Hardware interrupts are triggered by physical signals (edge/level); software interrupts are triggered by CPU execution of an instruction.
• Priority: Hardware interrupts have a fixed priority scheme; software interrupts don't have a priority among themselves in the same way, as their execution order is determined by their position in the program flow.

• Similarities:
• Both cause the current program execution to be suspended.
• Both save the return address (PC) on the stack.
• Both transfer control to a specific vector address where an ISR is located.
• Both require a RETI (Return from Interrupt) or RET (Return) instruction at the end of their ISR to return control to the main program. (RETI is specifically for hardware interrupts to re-enable interrupts if they were disabled upon entry, while RET is for general subroutines and software interrupts).

• Interrupt Handling Process (Brief Overview):
When an interrupt (hardware or software) occurs and is recognized:
1- The microprocessor completes its current instruction.
2- The content of the Program Counter (PC) is pushed onto the stack.
3- The PC is loaded with the starting address of the corresponding Interrupt Service Routine (ISR). For hardware interrupts, this is a fixed vector address or an address provided by an external device. For software interrupts, it's the fixed vector address for that RST instruction.
4- The CPU starts executing the ISR. An ISR typically saves the state of relevant registers onto the stack to preserve the program's context.
5- After the ISR completes its task, it restores the saved registers (if any).
6- An RETI (Return from Interrupt) instruction for hardware interrupts or a RET (Return) instruction for software interrupts is executed, which pops the return address from the stack back into the PC, and the main program execution resumes from where it was interrupted.

Summary of Key Points:

- Interrupts temporarily halt normal program execution to handle specific events.
- Hardware interrupts are external, asynchronous signals (TRAP, RST 7.5, RST 6.5, RST 5.5, INTR).
- TRAP is non-maskable, highest priority, edge/level sensitive, for critical events.
- RST 7.5, 6.5, 5.5 are maskable, with fixed priorities, triggered by specific signal types (edge for 7.5, level for 6.5, 5.5).
- INTR is maskable, lowest priority, non-vectored, requiring an external device to provide the vector via the INTA signal.
- SIM and RIM instructions control and read hardware interrupt masks and status.
- Software interrupts are internal, synchronous, explicitly programmed instructions (RST 0 to RST 7).
- Both types push the PC onto the stack and jump to a vector address for their Interrupt Service Routine (ISR).
- Hardware interrupts handle asynchronous external events, while software interrupts are used for programmed tasks like system calls or debugging.

16.) Interrupt Service Routines (ISRs)

Interrupt Service Routines (ISRs) are fundamental software components that act as the CPU's immediate response to an interrupt event. When an interrupt occurs, the normal flow of program execution is temporarily suspended, and control is transferred to a specific block of code designed to handle that particular event. This block of code is what we call an Interrupt Service Routine. In the context of the 8085 Microprocessor, ISRs are critical for enabling efficient, event-driven communication between the CPU and peripheral devices, moving beyond the limitations of polling or programmed I/O.

Let's delve into the specifics of ISRs.

1. The Purpose and Role of an ISR

The primary role of an ISR is to quickly and efficiently service the interrupting device or condition. This could involve:
- Reading data from an input device (e.g., a keyboard press).
- Sending data to an output device (e.g., updating a display).
- Clearing a status flag to acknowledge the interrupt.
- Performing some time-critical computation triggered by an external event.
- Handling error conditions (e.g., an arithmetic overflow, although less common for external 8085 interrupts).

An ISR allows the main program to continue executing complex tasks without constantly checking the status of slower peripheral devices. It enables an asynchronous response mechanism, where the CPU only reacts when a device explicitly requests attention.

2. Triggering an ISR in 8085

As you've covered, the 8085 supports both hardware and software interrupts.
- Hardware Interrupts (e.g., TRAP, RST7.5, RST6.5, RST5.5, INTR): When one of these is asserted and recognized by the 8085, the CPU hardware implicitly performs certain actions before transferring control to the ISR.
- Software Interrupts (RST 0-7): These are instructions embedded within the program code. When executed, they directly cause a call to a specific memory location, which is the start of the ISR.

Regardless of the trigger, the common goal is to execute a specialized routine to handle the event.

3. Automatic Actions Before ISR Execution (8085 Specific)

When the 8085 acknowledges an interrupt, a sequence of hardware-level actions takes place before your ISR code even begins. This sequence is crucial for preserving the system state and directing

execution:

- Completion of Current Instruction: The 8085 always finishes executing the current instruction before responding to an interrupt. This ensures the integrity of program flow.
- Disabling Further Interrupts: For all maskable interrupts (RST7.5, RST6.5, RST5.5, INTR), the 8085 automatically disables its interrupt enable flip-flop (IE flip-flop) upon recognizing an interrupt. This is equivalent to executing a DI (Disable Interrupts) instruction. The purpose is to prevent nested interrupts from immediately disrupting the current ISR, allowing it to complete its critical initial tasks. TRAP, being non-maskable, also causes the IE flip-flop to be reset.
- Saving the Program Counter (PC): The most critical piece of context to save is the address of the next instruction in the main program (the current value of the Program Counter). The 8085 pushes the current PC onto the stack. This is implicitly done by the hardware for hardware interrupts (like RST instructions pushing PC) and by the RST instructions themselves for software interrupts (which behave like CALL instructions). The stack pointer (SP) is decremented twice to accommodate the 16-bit PC.
- Loading the ISR Start Address: The PC is then loaded with the starting address of the corresponding ISR. For hardware interrupts, this address is fixed (e.g., 003CH for RST7.5, 0024H for RST4.5 if an external device provides this for INTR). For software interrupts (RST n), the address is 8 * n (e.g., RST 0 goes to 0000H, RST 1 to 0008H).

4. Structure of an ISR (Software Implementation)

Once the CPU jumps to the ISR's starting address, it's the programmer's responsibility to manage the rest. A typical ISR structure in 8085 assembly language follows these steps:

- a. Saving CPU Registers (Context Saving):

The main program uses various registers (A, B, C, D, E, H, L, F - Flag register). If the ISR modifies any of these registers, it will corrupt the main program's state. Therefore, the very first task of an ISR is to save the contents of all registers that it might modify. This is typically done by pushing them onto the stack using PUSH instructions.
Example:
- PUSH PSW (Pushes Accumulator and Flag Register)
- PUSH B (Pushes B and C registers)
- PUSH D (Pushes D and E registers)
- PUSH H (Pushes H and L registers)

This ensures that the main program's environment is preserved.

- b. Servicing the Interrupting Device:

This is the core logic of the ISR. It involves performing the necessary I/O operations or computations to handle the interrupt. This often includes:
- Reading status registers of the peripheral.
- Reading data from input ports (IN instruction).
- Writing data to output ports (OUT instruction).
- Clearing interrupt pending flags on the peripheral (if applicable).
- Performing any necessary data processing.

- c. Re-enabling Interrupts (Conditionally):

The 8085 automatically disables interrupts when an ISR is entered. If the ISR is short and critical, it might remain disabled until the ISR completes. However, for longer ISRs or systems requiring rapid response to other interrupts, it might be necessary to re-enable interrupts within the ISR using the EI (Enable Interrupts) instruction. This allows other, possibly higher-priority, interrupts to be serviced even while the current ISR is running. If EI is used, care must be taken to ensure re-entrancy and proper handling of shared resources. Typically, EI is placed after the initial context saving and before the main service logic.

- d. Restoring CPU Registers (Context Restoring):

Before the ISR finishes, it must restore the CPU's state to what it was before the interrupt occurred. This is done by popping the saved register contents from the stack, in the reverse order of how they were pushed.
Example:
- POP H

- POP D
- POP B
- POP PSW

This restores the registers and flags to their original values for the main program.

- e. Returning from Interrupt:

The final instruction of an ISR for the 8085 is always RET (Return from Subroutine). The RET instruction pops the previously saved Program Counter (PC) value from the stack and loads it back into the PC. This effectively transfers control back to the instruction that was interrupted in the main program.

Unlike some other architectures that have a special RETI or IRET instruction to re-enable interrupts upon return, the 8085's RET instruction only restores the PC. If interrupts were disabled by the automatic hardware action or by a DI instruction at the beginning of the ISR, they will remain disabled upon return unless an EI instruction was explicitly executed within the ISR. Thus, if the system requires interrupts to be enabled after the ISR completes, an EI instruction must be placed just before RET. A common practice is EI followed immediately by RET.

5. ISR Latency and Determinism

- Latency: This refers to the time delay between an interrupt occurring and the start of the ISR's execution. Factors contributing to latency include:
- Completion of the current instruction.
- Hardware interrupt recognition time.
- Context saving (PUSH instructions).
- Any DI instruction that might have blocked the interrupt for some time.
- Determinism: This is the predictability of the latency. A deterministic system guarantees that an ISR will begin execution within a known maximum time frame, which is crucial for real-time applications.

6. Key Considerations for Designing and Implementing ISRs

- Short and Fast: ISRs should be kept as short and efficient as possible. Long ISRs increase interrupt latency for other interrupts, potentially causing data loss or missing real-time deadlines. If a complex task needs to be performed, the ISR should only do the bare minimum (e.g., read data, set a flag) and then signal the main program to complete the more time-consuming processing.
- Atomic Operations: Sections of code that manipulate shared data structures (between the main program and ISR, or between multiple ISRs) should be protected by temporarily disabling interrupts (DI... EI) to prevent race conditions and ensure data integrity. These are called atomic operations.
- Re-entrancy: If an ISR can be interrupted by another interrupt (especially itself, if it's for a high-frequency event), it must be re-entrant. This means it must use local variables (typically on the stack) and save all necessary context, so that multiple invocations don't interfere with each other. For the 8085, this mainly involves careful stack usage and register saving.
- Interrupt Priority: While the 8085 has a fixed priority scheme for its hardware interrupts (TRAP > RST7.5 > RST6.5 > RST5.5 > INTR), designing ISRs requires understanding this hierarchy to ensure critical events are handled first.
- Stack Management: The stack is crucial for context saving and subroutine calls. Ensure there's enough stack space and that PUSH and POP operations are balanced to prevent stack overflow or corruption.

Summary of Key Points:

- ISRs are dedicated code routines executed in response to an interrupt, handling events asynchronously.
- The 8085 automatically saves the PC and disables maskable interrupts upon interrupt recognition.
- A well-structured 8085 ISR must explicitly save CPU registers (PUSH), service the device, optionally re-enable interrupts (EI), restore registers (POP), and return (RET).
- RET in 8085 only restores the PC; it does not automatically re-enable interrupts. An explicit EI instruction is needed if interrupts are to be re-enabled before returning.
- ISRs should be kept short, fast, and protect shared resources to maintain system responsiveness and data integrity.
- Understanding interrupt latency and determinism is vital for real-time system design.

17.) Timing Diagrams (Opcode Fetch, Memory Read/Write, I/O Read/Write)

Timing Diagrams (Opcode Fetch, Memory Read/Write, I/O Read/Write)

Timing diagrams are graphical representations of the signals involved in a microprocessor operation over a period of time. They depict the relationship between various control, address, and data signals, illustrating how the microprocessor interacts with its peripheral devices like memory and I/O ports. Understanding these diagrams is crucial for designing and troubleshooting microprocessor-based systems, as they show the precise sequence of events and the timing constraints. For the 8085 microprocessor, all operations are synchronized by its internal clock.

- Clock Cycles and Machine Cycles

The 8085 operates based on a system clock, typically generated by an external crystal or RC circuit and divided internally. Each pulse of this clock is called a T-state (time state). A sequence of T-states forms a Machine Cycle, which is the time required by the microprocessor to complete one basic operation, such as fetching an instruction byte, reading data from memory, or writing data to an I/O port. An instruction execution can consist of one or more machine cycles.

- Common Signals Involved in Timing Diagrams

To understand timing diagrams, it's essential to recall the functions of several 8085 pins:
- CLK (Clock): The primary timing signal, defining T-states.
- AD0-AD7: Multiplexed Address/Data Bus (lower order address A0-A7 and data D0-D7).
- A8-A15: Higher Order Address Bus (A8-A15).
- ALE (Address Latch Enable): Goes high during T1 to indicate that AD0-AD7 carries the low-order address. Used to latch A0-A7 for memory/I/O.
- IO/M (Input/Output / Memory): High for I/O operations, low for memory operations.
- RD (Read): Active low signal. Indicates a read operation (from memory or I/O).
- WR (Write): Active low signal. Indicates a write operation (to memory or I/O).
- S0, S1 (Status Signals): Provide information about the type of machine cycle being performed (e.g., S1=1, S0=1 for Opcode Fetch; S1=1, S0=0 for Memory Read).
- READY: An input signal from memory/I/O devices. If low, the 8085 enters wait states until READY goes high, allowing slower devices to catch up.

- Opcode Fetch Machine Cycle

The Opcode Fetch cycle is the first machine cycle in the execution of any instruction. Its purpose is to retrieve the instruction byte (opcode) from the memory location pointed to by the Program Counter (PC) and transfer it to the Instruction Register (IR) for decoding. An Opcode Fetch cycle in the 8085 typically consists of four T-states (T1, T2, T3, T4).

- T1 State (Address Output and Latching)
- At the rising edge of T1, the 8085 places the 16-bit address from the Program Counter onto the address buses.
- A15-A8 carry the higher-order address (A15-A8).
- AD7-AD0 carry the lower-order address (A7-A0).
- ALE goes high for approximately one clock period. This high pulse on ALE is used by external latches (like the 74LS373) to capture and hold the lower-order address (A7-A0) from AD7-AD0. This demultiplexes the address bus.
- Simultaneously, IO/M goes low, indicating a memory operation.
- S1 and S0 both go high (S1=1, S0=1), identifying this as an Opcode Fetch cycle.

- T2 State (Memory Read Activation and Data Input Expectation)
- ALE goes low, disabling the address latch, so AD7-AD0 are now available for data transfer.

• RD goes low. This active-low signal informs the memory device that the microprocessor wants to read data from the address specified in T1. The memory controller, upon receiving the address and the RD signal, places the data (opcode) onto the data bus (AD7-AD0).
• If the memory device is slow and cannot provide data within the required time, it can pull the READY pin low. If READY is low at the end of T2, the 8085 will insert wait states (Tw) between T2 and T3 until READY goes high. This ensures data integrity.

• T3 State (Data Read and RD Inactivation)
• The opcode byte from memory is now stable on the AD7-AD0 bus. The 8085 reads this data into its Instruction Register (IR).
• RD goes high, disabling the memory output drivers. This signifies the completion of the read operation from the memory device.
• The data read from memory is now internally available for decoding.

• T4 State (Internal Operation - Decoding)
• This T-state is typically used by the 8085 for internal operations, primarily decoding the fetched opcode. No external bus activity relevant to memory or I/O occurs during T4 of an Opcode Fetch.
• The decoded instruction determines the subsequent machine cycles required (e.g., Memory Read for an operand, Memory Write, etc.).

• Memory Read Machine Cycle

A Memory Read cycle is used to fetch data bytes (operands or data) from memory, subsequent to an opcode fetch. For example, if the instruction is LXI B, 2050H, after fetching LXI B, the 8085 needs to fetch 50H and then 20H from memory. This cycle typically consists of three T-states (T1, T2, T3).

• T1 State (Address Output and Latching)
• The 16-bit memory address (e.g., of the operand) is placed on the address buses (A15-A8 for higher, AD7-AD0 for lower).
• ALE goes high to latch the lower-order address (A7-A0) from AD7-AD0 into an external latch.
• IO/M goes low, indicating a memory operation.
• S1 goes high and S0 goes low (S1=1, S0=0), identifying this as a Memory Read cycle.

• T2 State (Memory Read Activation and Data Input Expectation)
• ALE goes low.
• RD goes low, activating the memory device to place data onto the AD7-AD0 bus.
• If READY is low, wait states are inserted.

• T3 State (Data Read and RD Inactivation)
• The data byte from memory is stable on the AD7-AD0 bus. The 8085 reads this data into its internal registers (e.g., into register B for LXI B, 2050H).
• RD goes high, disabling the memory output.

• Memory Write Machine Cycle

A Memory Write cycle is used to store data into a memory location. For example, the instruction STA 2050H (Store Accumulator content at memory address 2050H) will involve a Memory Write cycle after fetching the instruction and its operand. This cycle also typically consists of three T-states (T1, T2, T3).

• T1 State (Address Output and Latching)
• The 16-bit memory address (the destination address where data is to be written) is placed on the address buses.
• ALE goes high to latch the lower-order address (A7-A0) from AD7-AD0.
• IO/M goes low, indicating a memory operation.
• S1 goes low and S0 goes high (S1=0, S0=1), identifying this as a Memory Write cycle.
• During this T-state, the data to be written (e.g., the content of the Accumulator) is also placed on the AD7-AD0 bus, but it's important to note that the write strobe (WR) is not active yet.

• T2 State (Data Output and Write Activation)

• ALE goes low.
• The data to be written from the 8085's internal register (e.g., Accumulator) remains on the AD7-AD0 bus.
• WR goes low. This active-low signal informs the memory device that the microprocessor wants to write data. The memory device, using the address latched in T1 and observing the low WR signal, prepares to store the data present on the AD7-AD0 bus.
• If READY is low, wait states are inserted.

• T3 State (Data Write and WR Inactivation)
• The data is now fully written into the specified memory location by the memory device.
• WR goes high, signaling the completion of the write operation and disabling the write strobe.
• The 8085 removes the data from the AD7-AD0 bus.

• I/O Read Machine Cycle

An I/O Read cycle is used to fetch data from an input port. For example, IN 05H (read data from input port 05H). This cycle typically consists of three T-states (T1, T2, T3).

• T1 State (I/O Port Address Output and Latching)
• The 8-bit I/O port address (e.g., 05H) is placed on both the higher-order (A15-A8) and lower-order (AD7-AD0) address buses. This means the I/O address is duplicated across both halves of the address bus.
• ALE goes high to latch the lower-order address (A7-A0, which is the I/O port address) from AD7-AD0.
• IO/M goes high, indicating an I/O operation.
• S1 goes high and S0 goes low (S1=1, S0=0), identifying this as an I/O Read cycle (same status as Memory Read but distinguished by IO/M).

• T2 State (I/O Read Activation and Data Input Expectation)
• ALE goes low.
• RD goes low, activating the I/O input port to place data onto the AD7-AD0 bus.
• If READY is low, wait states are inserted.

• T3 State (Data Read and RD Inactivation)
• The data byte from the I/O port is stable on the AD7-AD0 bus. The 8085 reads this data into its Accumulator.
• RD goes high, disabling the I/O port's output drivers.

• I/O Write Machine Cycle

An I/O Write cycle is used to send data to an output port. For example, OUT 0AH (send Accumulator content to output port 0AH). This cycle also typically consists of three T-states (T1, T2, T3).

• T1 State (I/O Port Address Output and Latching)
• The 8-bit I/O port address (e.g., 0AH) is duplicated and placed on both A15-A8 and AD7-AD0 address buses.
• ALE goes high to latch the lower-order address (A7-A0, the I/O port address) from AD7-AD0.
• IO/M goes high, indicating an I/O operation.
• S1 goes low and S0 goes high (S1=0, S0=1), identifying this as an I/O Write cycle (same status as Memory Write but distinguished by IO/M).
• The data to be written (e.g., from the Accumulator) is placed on AD7-AD0, but WR is not active yet.

• T2 State (Data Output and Write Activation)
• ALE goes low.
• The data from the 8085's internal register (e.g., Accumulator) remains on the AD7-AD0 bus.
• WR goes low. This signals the I/O output port to receive and latch the data present on the AD7-AD0 bus.
• If READY is low, wait states are inserted.

• T3 State (Data Write and WR Inactivation)
• The data is now fully written into the specified I/O port.
• WR goes high, signaling the completion of the write operation.
• The 8085 removes the data from the AD7-AD0 bus.

• Summary of Key Points

• Timing diagrams are essential for understanding the precise, clock-synchronized interactions between the 8085 and external devices.
• All 8085 operations are composed of T-states (clock cycles) grouped into Machine Cycles.
• The Opcode Fetch cycle retrieves an instruction, taking four T-states (T1-T4), with T4 primarily for internal decoding.
• Memory Read and Memory Write cycles, as well as I/O Read and I/O Write cycles, typically take three T-states (T1-T3).
• Key control signals like ALE, IO/M, RD, and WR orchestrate the addressing, data transfer, and read/write operations.
• The AD0-AD7 bus is multiplexed for both lower-order address (A0-A7) and data (D0-D7), with ALE signal managing this.
• S0 and S1 status signals, along with IO/M, specify the type of machine cycle.
• The READY signal allows slower memory or I/O devices to insert wait states, ensuring reliable data transfer.
• I/O operations differ from memory operations by IO/M being high, and the 8-bit I/O address is duplicated on both A8-A15 and AD0-AD7.

18.) System Bus Structure (Address, Data, Control Bus)

The System Bus Structure is the fundamental communication pathway within any microprocessor-based system, including the 8085. It acts as a digital highway that connects the central processing unit (CPU), memory, and input/output (I/O) devices, allowing them to exchange information. Without a robust bus structure, these components would be isolated and unable to work together. In the context of the 8085 Microprocessor, this structure is typically divided into three main functional groups: the Address Bus, the Data Bus, and the Control Bus. Each bus has a specific role in facilitating the coherent operation of the entire system.

Let's delve into each component of the system bus structure:

1. Address Bus

• Purpose: The Address Bus is used by the CPU to specify the memory location or I/O port that it wants to communicate with. Think of it as the postal address written on an envelope; it tells where the data needs to go or where it should be retrieved from.
• Directionality: The Address Bus is unidirectional, meaning information flows only in one direction: from the CPU to memory or I/O devices. The CPU outputs an address, and the memory or I/O device then responds to that specific address.
• Size in 8085: The 8085 Microprocessor has a 16-bit address bus. This means it has 16 individual lines, typically labeled A0 to A15.
• Addressing Capability: A 16-bit address bus allows the 8085 to generate 2 to the power of 16 unique addresses. This translates to 65,536 (64 KB) distinct memory locations or I/O ports that the CPU can directly access. Each unique combination of 0s and 1s on these 16 lines points to a specific location.
• How it Works: When the CPU wants to read data from a specific memory location, say 2000H (hexadecimal), it places the binary equivalent of 2000H onto the 16 lines of the address bus. Similarly, for writing data to memory or communicating with an I/O port, the respective address is placed on the bus. The memory chips or I/O devices connected to the bus constantly monitor these address lines. When they detect their assigned address, they prepare to either provide data (for a read operation) or accept data (for a write operation).

• 8085 Specifics - Multiplexing: A unique aspect of the 8085's address bus is that its lower 8 bits (A0-A7) are time-multiplexed with the data bus (D0-D7). This means the same physical lines (AD0-AD7) are used for two different purposes at different times during a machine cycle.

• During the early part of a machine cycle, these lines carry the lower 8 bits of the address (A0-A7).

• During the later part of the same machine cycle, these same lines carry the 8-bit data (D0-D7).

• The higher 8 bits of the address (A8-A15) are dedicated and non-multiplexed.

• Reasoning for Multiplexing: This multiplexing strategy was employed to reduce the total number of pins on the 8085 chip, which helps in manufacturing cost and package size. However, it necessitates external hardware (an address latch, typically an 8212 IC or similar) to **demultiplex** or separate the address from the data. The Address Latch Enable (ALE) signal, provided by the 8085's Timing and Control Unit, is used to signal when the AD0-AD7 lines carry address information, allowing the external latch to capture and hold this address for the duration of the bus cycle. This ensures that a stable 16-bit address is available for memory and I/O devices, even while the data lines later become active for data transfer.

2. Data Bus

• Purpose: The Data Bus is the pathway used for the actual transfer of data (instructions or operands) between the CPU, memory, and I/O devices. It's like the content of the envelope, carrying the actual message.

• Directionality: The Data Bus is bidirectional, meaning data can flow in both directions: from the CPU to memory/I/O (write operation) or from memory/I/O to the CPU (read operation). This bidirectional nature is essential for the CPU to both store and retrieve information.

• Size in 8085: The 8085 Microprocessor has an 8-bit data bus. This means it can transfer 8 bits (one byte) of data at a time.

• Data Transfer Unit: Each cycle on the data bus transfers a single byte of information. If a larger piece of data (e.g., a 16-bit word) needs to be transferred, it typically requires multiple bus cycles, with the CPU transferring one byte at a time.

• How it Works: During a memory read operation, after the CPU has placed the address on the address bus, the memory device at that address places the requested 8-bit data onto the data bus, which is then read by the CPU. Conversely, during a memory write operation, the CPU places the 8-bit data onto the data bus after selecting the target address, and the memory device stores it.

• 8085 Specifics - Shared with Address Lines: As mentioned, the 8085's data bus uses the same physical lines (AD0-AD7) that carry the lower 8 bits of the address. After the lower address bits are latched externally (using ALE), these AD0-AD7 lines then switch function and become the data bus for the remainder of the cycle. This efficient use of pins is a defining characteristic of the 8085's external interface.

3. Control Bus

• Purpose: The Control Bus is a collection of individual control signals that orchestrate and synchronize the activities of all components connected to the system bus. It's like the traffic controller, signaling when and how data and addresses should be handled. These signals specify the type of operation being performed (e.g., read, write, I/O, memory access), manage the flow of data, and coordinate the timing of events.

• Directionality: Most control signals are output by the CPU (unidirectional), but some are inputs to the CPU (e.g., READY signal from slow peripherals, interrupt requests). Thus, collectively, the control bus can be considered multidirectional.

• Nature: Unlike the address and data buses which are groups of parallel lines, the control bus is a collection of distinct, dedicated signal lines, each with its own specific function.

• Key Control Signals in 8085 (relevant to bus operations):

• RD (Read): This is an active-low output signal from the 8085. When asserted low, it indicates that the CPU is performing a memory read or an I/O read operation. Connected memory or I/O devices then place data onto the data bus.

• WR (Write): This is an active-low output signal from the 8085. When asserted low, it indicates that the CPU is performing a memory write or an I/O write operation. Connected memory or I/O devices then accept data from the data bus.

• IO/M (I/O or Memory): This is an output status signal from the 8085.

• When IO/M is high (1), it indicates that the current operation involves an I/O device.

• When IO/M is low (0), it indicates that the current operation involves memory.
• This signal, in conjunction with RD and WR, helps to differentiate between four types of operations: Memory Read (IO/M=0, RD=0), Memory Write (IO/M=0, WR=0), I/O Read (IO/M=1, RD=0), and I/O Write (IO/M=1, WR=0).
• ALE (Address Latch Enable): As discussed, this is an output signal that goes high during the first clock cycle of a machine cycle, indicating that the AD0-AD7 lines contain the lower 8 bits of the address. It is crucial for demultiplexing the address/data bus.
• S0, S1 (Status Signals): These are output status signals that, along with IO/M, provide more detailed information about the current operation being performed by the 8085 (e.g., Opcode Fetch, Memory Read, Memory Write, I/O Read, I/O Write, Halt, etc.). These signals are useful for external devices to understand the CPU's state.
• READY: This is an input signal to the 8085. If a slow peripheral or memory device cannot respond quickly enough to a read or write request, it can pull the READY line low. The 8085 will then enter a wait state until the READY line goes high, indicating the device is ready. This ensures proper data synchronization between components of varying speeds.
• HOLD and HLDA: While detailed DMA concepts are for future topics, it is worth noting that the HOLD (input) and HLDA (Hold Acknowledge, output) signals are part of the control bus. They are used for direct memory access (DMA) operations, allowing external devices to temporarily take control of the system buses (address, data, control) from the CPU to perform high-speed data transfers, usually between memory and I/O, without CPU intervention.

Interaction and Synchronization:
The three buses work in a tightly synchronized manner, governed by the CPU's internal clock and Timing and Control Unit. For example, during an instruction fetch:
1. The CPU places the 16-bit address of the next instruction on the address bus (A8-A15, and AD0-AD7 momentarily during ALE pulse).
2. It then asserts IO/M low (for memory) and RD low (for read) on the control bus.
3. Memory responds by placing the 8-bit instruction opcode onto the data bus (AD0-AD7, now functioning as data lines).
4. The CPU reads this data from the data bus, increments its Program Counter, and proceeds to decode and execute the instruction.
This sequence is carefully timed, as illustrated in timing diagrams, ensuring that addresses are stable when required and data is available when expected.

Reasoning and Importance:
The system bus structure is vital for several reasons:
• Modular Design: It allows for a modular system design where different components (CPU, memory, I/O) can be connected and interchanged easily, as long as they adhere to the bus interface standards.
• Parallel Communication: By using parallel lines for address and data, multiple bits can be transferred simultaneously, leading to faster data throughput compared to serial communication.
• Resource Sharing: All components share the same bus, allowing the CPU to communicate with any attached memory location or I/O device using a standardized mechanism.
• Control and Synchronization: The control bus is crucial for preventing data collisions, ensuring that only one device is transmitting at a time, and that operations are completed in the correct sequence and within specific timeframes.
• Efficiency and Cost-Effectiveness (8085 Specific): The multiplexing of the address and data lines in the 8085 was a design choice to minimize pin count, reducing manufacturing costs and package size, though it adds complexity to external interfacing with the need for demultiplexing logic.

Summary of Key Points:
• The System Bus is the communication backbone of the 8085 system.
• The Address Bus (16-bit, unidirectional, CPU to Memory/I/O) specifies locations (up to 64KB). Lower 8 bits (AD0-AD7) are multiplexed with data lines.
• The Data Bus (8-bit, bidirectional) carries the actual data (instructions or operands) between components. It shares the AD0-AD7 lines with the lower address bus.
• The Control Bus (collection of discrete signals, multidirectional) synchronizes and regulates operations, defining the type of transfer (read/write, memory/I/O) and managing data flow using signals like RD, WR, IO/M, ALE, and READY.
• All three buses work together in a precisely timed sequence to enable effective communication and

operation of the 8085 microprocessor system.

19.) Clock Generation and Reset Circuitry

Every complex digital system, like our 8085 Microprocessor, relies on two fundamental operations to function reliably: a precise timing reference and a way to start fresh. These are provided by the Clock Generation and Reset Circuitry. Understanding these components is crucial because they form the very foundation upon which the 8085 executes instructions and interacts with the rest of the system. Without a stable clock, the processor would be a chaotic mess, and without a reliable reset, it would never be able to start execution predictably.

Clock Generation

The Rhythmic Heartbeat: What is a Clock Signal?
Imagine an orchestra. For all instruments to play in harmony, they need a conductor to set the tempo. In a microprocessor, the clock signal acts as this conductor. It's a continuous train of square wave pulses, alternating between high and low voltage levels, generated at a very specific and constant frequency. Each pulse, or clock cycle, marks the completion of a basic operation or the initiation of the next step in a sequence.

Why the 8085 Needs a Clock
The 8085, like most digital integrated circuits, is a synchronous device. This means all its internal operations – fetching an instruction, decoding it, reading data from memory, writing data to an I/O port, or updating register contents – are synchronized to the rising or falling edges of the clock signal. This ensures that data is stable when it's read and that different parts of the microprocessor operate in a coordinated fashion, preventing race conditions and ensuring predictable behavior. The timing diagrams you've studied illustrate how operations like opcode fetch or memory read are broken down into machine cycles, and each machine cycle is further divided into T-states, all controlled by the clock.

8085 Clock Pins: X1, X2, CLK OUT
The 8085 provides specific pins to facilitate clock generation:
- X1 and X2: These are the input pins where an external crystal oscillator or an RC network (Resistor-Capacitor) is connected to generate the fundamental frequency. The 8085 has an internal oscillator circuit that uses these pins.
- CLK OUT: This is an output pin that provides a clock signal, typically used to synchronize other peripheral devices in the system with the 8085.

The Crystal Oscillator: Generating Precise Timing
For highly accurate and stable clock frequencies, a crystal oscillator is almost universally used in microprocessor systems, including the 8085.
- Principle: A quartz crystal exhibits the piezoelectric effect. When a voltage is applied across it, it deforms, and when it deforms, it generates a voltage. This property allows it to resonate at a very precise frequency when integrated into an appropriate electronic circuit.
- How it works with X1 and X2: For the 8085, an external quartz crystal (e.g., 6 MHz) is connected between the X1 and X2 pins. Two small capacitors (typically 20-33pF) are also connected from each of X1 and X2 to ground. These capacitors, along with the internal circuitry of the 8085, form a Pierce oscillator configuration. The crystal acts like a highly stable resonant tank circuit, and the 8085's internal amplifier provides the gain needed to sustain oscillations at the crystal's fundamental frequency.
- Example: If you connect a 6 MHz crystal, the internal oscillator will generate an oscillation at 6 MHz.

Internal Clock Generation within 8085
The 8085 does not directly use the frequency generated by the crystal (at X1/X2) for its internal operations. Instead, it has an internal frequency divider circuit.
- Frequency Division: The frequency generated at the X1/X2 pins is internally divided by 2. So, if a 6 MHz crystal is used, the internal operating clock frequency of the 8085 will be 3 MHz (6 MHz / 2). This division is done to achieve better stability and to allow the use of higher frequency crystals which are generally more stable.

• CLK OUT Signal: The CLK OUT pin provides an output clock signal that is identical to the internal operating clock frequency. So, if the internal clock is 3 MHz, CLK OUT will also be 3 MHz. This output is useful for synchronizing other components like memory, I/O devices, or support chips (like the 8255 PPI or 8259 PIC, which you will study later) that need to operate in lockstep with the CPU.

Importance of Clock Stability and Frequency

• Stability: A stable clock frequency ensures that all operations take the expected amount of time, crucial for accurate data processing and timing-dependent protocols. Variations can lead to system crashes or incorrect data.

• Frequency: The clock frequency determines the speed at which the microprocessor operates. A higher frequency means more operations per second, leading to faster execution. However, there are limits imposed by the manufacturing technology and power consumption.

Reset Circuitry

The System's Fresh Start: What is a Reset?

Imagine your computer freezing or encountering an error. What's the first thing you try? Rebooting it. A reset signal serves a similar purpose for a microprocessor. It's a control signal that forces the microprocessor into a known, initial state, regardless of its current operational state.

Why the 8085 Needs a Reset

The 8085 requires a reset for several critical reasons:

• Power-on Initialization: When the power is first applied to the system, the internal state of the 8085 (program counter, registers, flags) is undefined and random. A reset ensures that the microprocessor starts in a predictable state, ready to fetch the first instruction of the operating program.

• Error Recovery: If the system encounters an unrecoverable error, gets stuck in an infinite loop, or if the program logic goes awry, a reset can bring the system back to a known good state, allowing the program to restart cleanly.

• System Restart: For development or testing, a manual reset allows developers to restart the program execution from the beginning without cycling the power.

8085 Reset Pins: RESET IN, RESET OUT

The 8085 provides two dedicated pins for reset functionality:

• RESET IN: This is an input pin used to initiate a reset of the 8085 microprocessor itself.

• RESET OUT: This is an output pin used by the 8085 to signal other peripheral devices in the system that a reset operation is occurring.

RESET IN: Initiating a Reset

• Active-low Nature: The RESET IN pin is an active-low input. This means that a low logic level (approximately 0V) applied to this pin for a sufficient duration will trigger a reset. When the pin is high (approximately +5V), the microprocessor operates normally.

• Minimum Pulse Duration: For a reliable reset, the low pulse applied to RESET IN must last for at least 3 clock periods of the internal operating frequency. If the internal clock is 3 MHz, then 3 periods would be 3 * (1/3MHz) = 1 microsecond. This duration allows the internal circuitry of the 8085 to properly process the reset signal.

• Consequences of an Active RESET IN: When RESET IN is asserted (held low), the 8085 performs the following actions:

• The Program Counter (PC) is cleared to 0000H. This is crucial because it means the 8085 will always start executing instructions from memory location 0000H after a reset.

• All internal registers (B, C, D, E, H, L, Accumulator, Stack Pointer) are cleared to an undefined state in some versions or to zero in others, but their contents are not reliable after a reset unless explicitly initialized by software. However, for a safe start, software should assume they are undefined and initialize them.

• All interrupt enable flags (like INTE flip-flop, RIM/SIM masks) are reset, disabling all maskable interrupts. This prevents any unintended interrupts from occurring during startup.

• The instruction register is cleared.

• The data and address buses are floated to a high-impedance state during the reset pulse.

• The SID (Serial Input Data) pin is automatically reset.

Generating a Reset Signal
Two common circuits are used to generate the necessary reset pulse for the RESET IN pin:

- Power-on Reset (POR) Circuit:
- Purpose: Automatically generates a reset pulse when power is first applied to the system.
- Implementation: A simple RC (Resistor-Capacitor) circuit is often used. When power (+5V) is first turned on, the capacitor is initially discharged, meaning the voltage across it is 0V. This 0V is applied to the RESET IN pin through a resistor. As time progresses, the capacitor begins to charge through the resistor, and its voltage rises. When the capacitor voltage crosses the threshold for a high logic level, the RESET IN pin effectively becomes high, and the reset condition ends. By carefully choosing the R and C values, the duration for which RESET IN is held low can be controlled to meet the minimum required pulse duration.
- Manual Reset Circuit:
- Purpose: Allows a user or developer to manually reset the system at any time.
- Implementation: A momentary push-button switch is typically connected to provide a low pulse to the RESET IN pin when pressed. To ensure a clean low pulse and prevent issues like contact bounce (where the switch rapidly makes and breaks contact causing multiple rapid pulses), a debounce circuit (often another RC circuit or a Schmitt trigger inverter) is usually incorporated. When the button is pressed, it pulls RESET IN to ground (low). When released, a pull-up resistor ensures RESET IN returns to a high state.

RESET OUT: Signaling Peripherals

- Active-high Nature: The RESET OUT pin is an active-high output. When the 8085 itself is being reset (i.e., RESET IN is low), the RESET OUT pin goes high. It remains high as long as the 8085's internal reset process is active.
- Purpose: This output signal is vital for system-wide synchronization. When the 8085 comes out of reset, it then drives RESET OUT high. This high signal is then connected to the reset input pins of other peripheral devices (like memory chips, I/O ports, or other programmable controllers) in the system. This ensures that all other components are also reset and initialized concurrently with the CPU, bringing the entire system into a coherent and predictable starting state. This prevents peripherals from being in an unknown state while the CPU starts executing, which could lead to errors.

The Reset Vector: Where Execution Begins
The clearing of the Program Counter to 0000H after a reset is fundamental. This fixed starting address is known as the **Reset Vector.** It means that immediately after a reset, the 8085 will always fetch its first instruction from memory location 0000H. Therefore, the very first part of any 8085 program, often referred to as the bootloader or initialization code, must reside at memory address 0000H. This code is responsible for setting up the stack pointer, initializing registers, configuring I/O ports, and then jumping to the main application program.

Summary of Key Points:

1. Clock Generation provides a precise, rhythmic timing signal essential for the synchronous operation of the 8085 microprocessor and its peripherals.
2. The 8085 uses X1 and X2 pins for connecting an external crystal oscillator (typically 6 MHz for a 3 MHz internal clock) along with capacitors.
3. The internal circuitry of the 8085 divides the crystal frequency by 2 to generate its operational clock.
4. The CLK OUT pin provides the internal operating frequency (e.g., 3 MHz) for synchronizing other system components.
5. Reset Circuitry initializes the 8085 and the entire system to a known, predictable state, crucial for power-on startup and error recovery.
6. The RESET IN pin is an active-low input. Holding it low for at least 3 clock periods clears the Program Counter to 0000H, disables maskable interrupts, and sets other internal states.
7. Reset signals are typically generated by Power-on Reset (POR) RC circuits and/or manual push-button circuits with debounce.
8. The RESET OUT pin is an active-high output that the 8085 uses to signal and reset other peripheral devices in the system, ensuring system-wide synchronization.
9. After a reset, the 8085 always begins program execution from memory location 0000H, which is known as the Reset Vector.