Notes on: 8085 Assembly Language Programming_from_0

1.) Memory classifications

Memory is a fundamental component of any computer system, including systems built around the 8085 microprocessor. It's where the 8085 stores the programs it needs to execute and the data it needs to process. Just like how we organize information in different ways (notebooks, sticky notes, long-term archives), computer memory is also categorized based on its characteristics and purpose. Understanding these classifications is crucial for designing efficient 8085-based systems and writing effective assembly language programs.

Let's explore the key ways memory is classified:

1. Classification by Volatility

This classification distinguishes memory based on whether it retains its contents when power is removed.

• Volatile Memory:
• Definition: This type of memory requires continuous power to maintain the stored information. If the power supply is interrupted, all data is lost.
• Analogy: Think of it like a blackboard where you write temporary notes. If you erase it or turn off the lights, the notes are gone.
• Relevance to 8085: The primary working memory (RAM) used by the 8085 for storing program variables and temporary data is volatile. When you power down an 8085 development board, the contents of its RAM are cleared.

• Non-Volatile Memory:
• Definition: This type of memory retains its contents even when the power is turned off.
• Analogy: Like a book or a stone tablet where information is permanently etched.
• Relevance to 8085: Non-volatile memory (ROM) is used to store the 8085's initial boot-up instructions, firmware, or fixed programs that don't change. When you turn on an 8085 system, the program that runs first is typically loaded from non-volatile memory.

2. Classification by Access Method

This classification describes how data is read from or written to memory locations.

• Random Access Memory (RAM):
• Definition: Any memory location can be accessed directly and in approximately the same amount of time, regardless of its physical location.
• Analogy: Imagine a large array of individual mailboxes, and you can go directly to any mailbox by its number without checking others first.
• Relevance to 8085: This is the primary characteristic of the main memory (RAM) that the 8085 uses. Instructions like LDA (Load Accumulator Direct) or STA (Store Accumulator Direct) can instantly access any specified 16-bit memory address provided in the instruction, making it very efficient for processing.

• Sequential Access Memory:
• Definition: Data must be accessed in a specific order, one after another, from the beginning until the desired location is reached.
• Analogy: Like a cassette tape where you have to fast-forward or rewind to find a particular song.
• Relevance to 8085: While not used for the 8085's main working memory, it's good to know this distinction for understanding data storage in general.

- Direct Access Memory:
- Definition: Individual blocks or records have unique addresses. Accessing a block involves moving to its general vicinity and then sequentially searching within that block. It's a mix of random and sequential.
- Analogy: Finding a specific song on a CD. You can jump to a track (random access to a block) and then play it sequentially from there.
- Relevance to 8085: Similar to sequential access, not directly applicable to the 8085's primary memory operations but part of broader memory concepts.

3. Classification by Function or Purpose

This categorizes memory based on what kind of information it typically stores within an 8085 system.

- Program Memory:
- Definition: This portion of memory is dedicated to storing the actual machine code instructions that the 8085 microprocessor executes.
- Relevance to 8085: When you write an 8085 assembly program, compile it, and load it onto the system, the resulting machine code resides in program memory (often ROM or a designated part of RAM). The 8085's Program Counter (PC) register always points to the next instruction in program memory to be fetched.

- Data Memory:
- Definition: This memory space is used to store the data that the program operates on. This includes variables, constants, intermediate results, and input/output buffers.
- Relevance to 8085: Instructions like MOV M,A (Move Accumulator to Memory), LDA (Load Accumulator Direct), or STA (Store Accumulator Direct) directly interact with data memory. The 8085's general-purpose registers (B, C, D, E, H, L) often hold temporary data, but larger datasets or variables that exceed register capacity are stored in data memory (usually RAM).

- Stack Memory:
- Definition: A special region of data memory organized as a Last-In, First-Out (LIFO) structure. It's primarily used for temporary storage during subroutine calls, interrupts, and for saving register contents.
- Analogy: Imagine a stack of plates. You can only add (PUSH) a new plate to the top, and you can only remove (POP) the top plate.
- Relevance to 8085: The 8085 uses its Stack Pointer (SP) register to manage the stack. When a CALL instruction is executed, the return address is PUSHed onto the stack. When a RET (Return) instruction is executed, the return address is POPed from the stack. PUSH and POP instructions are also used by programmers to save and restore the contents of register pairs (like BC, DE, HL) during complex operations or before calling subroutines.

- I/O Memory (Memory-Mapped I/O):
- Definition: In systems using memory-mapped I/O, peripheral devices (like keyboards, displays, sensors) are assigned specific memory addresses. The 8085 interacts with these devices as if they were memory locations.
- Relevance to 8085: The 8085 can use its standard memory access instructions (LDA, STA, MOV M) to read from or write to I/O devices mapped into the memory space. This simplifies the instruction set but consumes part of the available 64KB address space. The 8085 also supports Port-Mapped I/O using specific IN and OUT instructions, which access a separate 256-port address space.

4. Classification by Location/Usage (Context for future topics)

- Internal Memory (Registers): These are small, very fast memory locations located directly within the 8085 CPU itself. They are used for temporary storage during processing.
- Relevance to 8085: The 8085 has registers like the Accumulator (A), B, C, D, E, H, L, Program Counter (PC), Stack Pointer (SP), etc. These are the fastest access points for data.

- External Memory (Main Memory, Auxiliary Memory): This refers to memory components located outside the CPU.

- Main Memory: The primary storage accessible directly by the 8085, typically RAM and ROM chips. It's relatively fast but more expensive than auxiliary memory.
- Auxiliary Memory: Larger, slower, and non-volatile storage used for long-term data archival and program storage when not actively running.

Understanding these classifications helps you appreciate the different roles memory plays in an 8085 system. When you write 8085 assembly code, you're constantly deciding whether to store data in a register for speed, in data memory for general use, or on the stack for temporary function calls, all while knowing that your program instructions reside in program memory (often non-volatile ROM for permanent applications). These foundational classifications lay the groundwork for understanding more complex memory concepts like memory hierarchy, cache, and virtual memory, which you will explore later.

2.) Memory Hierarchy

Memory Hierarchy

The Memory Hierarchy is a fundamental concept in computer organization and architecture that addresses the challenge of providing both fast and large memory for a computer system. Modern processors operate at very high speeds, but it is economically impractical to build a single, very large memory system that can match these speeds. The memory hierarchy solves this problem by organizing different types of memory into a tiered structure based on their speed, cost, and capacity. This creates the illusion of a single, large, fast, and inexpensive memory to the Central Processing Unit (CPU).

1. The Need for Memory Hierarchy

Computer systems require memory for various purposes:
- To store instructions (programs) that the CPU will execute.
- To store data that the program operates on.
- To provide temporary storage during calculations and operations.

The ideal memory would possess three main characteristics:
- Extremely fast access time (to keep up with the CPU's processing speed).
- Very large capacity (to store extensive programs and large amounts of data).
- Very low cost (to be economically viable for large systems).

However, these three properties are inherently contradictory. Fast memories are typically expensive and have smaller capacities. Conversely, large, inexpensive memories are usually slower to access. The memory hierarchy is a clever engineering compromise, designed to optimize the overall performance of the computer system by balancing these conflicting requirements.

Analogy: Consider a student working on a complex assignment.
- CPU Registers are like the notes, calculator, and pen right on your desk (extremely fast to access, very small capacity).
- Main Memory (RAM) is like your textbooks and notebooks on a bookshelf in your room (slower than your desk, larger capacity, still relatively quick access).
- Auxiliary Memory (Hard Drive) is like the university library or external storage (slowest access, massive capacity, very inexpensive per unit of storage).

You always try to keep the most relevant and frequently used information on your desk, then on your bookshelf, and only go to the library when absolutely necessary for less frequently accessed or very large resources.

2. Levels of Memory Hierarchy (General Overview)

The memory hierarchy typically consists of several distinct levels. Each level has different characteristics concerning speed, capacity, and cost. As you move from the top level (closest to the CPU) to the bottom level (farthest from the CPU) in the hierarchy:

- Speed decreases (meaning access time increases).
- Capacity increases (more data can be stored).
- Cost per bit of storage decreases.
- The frequency of access by the CPU to that level decreases.

The typical levels, ordered from fastest and smallest to slowest and largest, are:

- CPU Registers
- Cache Memory (a high-speed buffer between CPU and main memory, primarily in modern CPUs)
- Main Memory (RAM, ROM)
- Auxiliary/Secondary Storage (Hard Disk Drives, Solid State Drives, etc.)

3. Memory Hierarchy in the Context of 8085 Microprocessor

While modern microprocessors have complex multi-level cache systems and virtual memory, the 8085 microprocessor operates with a simpler, yet effective, memory hierarchy. Its primary memory interaction involves two main levels and one auxiliary level.

1. CPU Registers:

- These are the fastest memory elements, built directly into the 8085 microprocessor chip itself.
- They hold data, addresses, and control information that the CPU is actively processing or will process in the immediate future.
- Examples in the 8085 include the Accumulator (A), General Purpose Registers (B, C, D, E, H, L), Program Counter (PC), Stack Pointer (SP), and Flag Register.
- Access to registers occurs within a single CPU clock cycle, providing immediate data for processing and instruction execution without any delay.
- For instance, when an 8085 assembly instruction like MOV A, B (Move the content of register B to register A) is executed, the data transfer happens directly and instantly between these internal registers.

2. Main Memory:

- This is the primary storage area connected externally to the 8085 microprocessor.
- It holds the actual program instructions (machine code) and data that the 8085 is currently executing or needs to access frequently.
- The 8085 uses its 16-bit address bus to access up to 64KB (2^16 bytes) of physical memory locations. It performs read (fetching data/instructions) and write (storing data) operations to this memory.
- Main memory typically consists of RAM (Random Access Memory) for volatile, read/write storage of programs and data, and ROM (Read-Only Memory) for non-volatile storage of boot-up instructions or permanent data.
- When an 8085 instruction like LDA 2000H (Load Accumulator Direct from memory location 2000H) is executed, the CPU sends the address 2000H to the external memory, and the content at that location is fetched into the Accumulator. This access is much slower than register access (taking several clock cycles) but is still relatively fast compared to auxiliary storage.

3. Auxiliary/Secondary Storage:

- For the 8085 and similar older systems, this level refers to external, non-volatile storage devices like magnetic tapes or floppy disks that were common during its era.
- These memories are not directly accessible by the 8085 CPU using its address bus for instruction execution or data manipulation. Instead, data and programs must first be loaded from auxiliary storage into the main memory before the 8085 can work with them.
- Accessing auxiliary storage is significantly slower (often taking milliseconds), but it offers very large storage capacity at a much lower cost per bit and retains data even when power is off.
- This level is used for long-term storage of programs, operating system components, and large datasets that are not currently in active use by the 8085.

4. Characteristics of Each Level

- CPU Registers:

- Speed: Extremely Fast (typically single clock cycle access).
- Capacity: Very Small (a few bytes to tens of bytes, e.g., 8-bit or 16-bit registers).
- Cost: Very High (part of the CPU chip itself).
- Volatility: Volatile (data is lost when power is removed).

- Main Memory (RAM/ROM for 8085 context):
- Speed: Fast (tens of clock cycles for access, significantly slower than registers but much faster than auxiliary storage).
- Capacity: Medium (typically 64KB directly for an 8085 system, but can be expanded with memory banking).
- Cost: Medium.
- Volatility: RAM is volatile, ROM is non-volatile.

- Auxiliary Storage (e.g., Floppy Disk, Hard Drive):
- Speed: Very Slow (milliseconds for access).
- Capacity: Very Large (megabytes to gigabytes and beyond).
- Cost: Very Low per bit.
- Volatility: Non-volatile (data persists without power).

5. Principle of Locality

The effectiveness of the memory hierarchy in bridging the speed gap between CPU and slower memory levels relies heavily on a fundamental concept called the **principle of locality.** This principle states that during any given time interval, a program tends to access a relatively small portion of its entire address space, and these accesses are not random. There are two main types of locality:

- Temporal Locality: If a particular memory location is accessed (read or written), it is highly likely that the same location will be accessed again in the near future.
- Example in 8085: Instructions within a loop are repeatedly executed, meaning their addresses are accessed multiple times. Similarly, frequently used data variables, like a counter or an intermediate result, are held in registers or main memory and accessed many times within a short period.

- Spatial Locality: If a particular memory location is accessed, it is highly likely that nearby or adjacent memory locations will be accessed soon.
- Example in 8085: Instructions in an 8085 program are usually stored and executed sequentially. When the Program Counter (PC) fetches an instruction, it often increments to fetch the next instruction from the adjacent memory location. Similarly, when processing a block of data, such as elements of an array, contiguous memory locations are accessed in sequence.

By leveraging the principle of locality, the memory hierarchy ensures that frequently used instructions and data are kept in the faster, smaller memory levels (like registers and main memory for the 8085), minimizing the need to access the much slower, larger auxiliary storage. This design greatly optimizes overall system performance by reducing the average memory access time.

3.) Various types of Main memories(RAM, ROM, PROM, EPROM, EEPROM, Associative Memory)

Main memories are critical components in any computer system, including those built around the 8085 microprocessor. They are directly accessible by the CPU and hold the programs and data that the CPU is actively working on. In the context of 8085 assembly language programming, you will typically be loading your assembled machine code instructions and any data your program uses into main memory (RAM) for the 8085 to fetch and execute. Other types of main memory, like ROM, are essential for storing the system's basic operating instructions.

Here, we explore various types of main memories:

1. RAM (Random Access Memory)

RAM is a type of volatile main memory, meaning it loses its stored information once the power is turned off. It is called **Random Access** because any memory location can be accessed directly in roughly the same amount of time, regardless of its physical location. RAM is primarily used for read and write operations.

• Role in 8085: When you write an 8085 assembly program, you compile or assemble it into machine code. This machine code, along with any variables or data your program uses, is loaded into RAM. The 8085 CPU then fetches instructions from RAM, executes them, and stores results back into RAM. This allows the 8085 to run dynamic programs and handle changing data.

• Types of RAM:
• SRAM (Static RAM):
• Faster and more expensive.
• Uses latches or flip-flops to store each bit.
• Does not need to be refreshed periodically, hence **static**.
• Often used for smaller, faster memory needs like CPU caches (though cache is a future topic, SRAM itself is a type of RAM).
• DRAM (Dynamic RAM):
• Slower and less expensive.
• Uses capacitors to store each bit. A capacitor naturally discharges, so DRAM requires periodic **refresh** cycles to maintain the data.
• This refresh operation makes it slower but allows for much higher density and lower cost per bit, making it suitable for the main memory of most computers.

• Practical Example: Imagine an 8085-based training kit. The area where you load your assembly program and observe its execution, and where variables are modified, is typically SRAM. When you turn off the kit, your program and data are lost from this RAM.

2. ROM (Read-Only Memory)

ROM is a type of non-volatile main memory, meaning it retains its stored information even when the power is turned off. As the name suggests, it is primarily used for read operations; writing to it is either impossible or requires special procedures.

• Role in 8085: In an 8085-based system, ROM is crucial for storing the **bootstrap** program or firmware. This is a small program that runs immediately after the 8085 powers on. It might initialize the system, load a larger operating system (if applicable), or provide a basic monitor program for interacting with the 8085. This way, the system knows what to do even before any other software is loaded.

• Mask ROM: This is the most basic type of ROM. Its data is physically embedded into the chip during the manufacturing process by masking. It is very cost-effective for high-volume production but cannot be changed once manufactured.

• Practical Example: The firmware that allows your 8085 training kit to display text on a simple LCD screen or respond to keyboard inputs when it first powers on is stored in a ROM chip.

3. PROM (Programmable Read-Only Memory)

PROM is a type of ROM that can be programmed once by the user, rather than being factory-programmed like Mask ROM. It consists of an array of fuses (or anti-fuses) that can be **blown** (or created) using a special device called a PROM programmer. Once a fuse is blown, the connection is permanently altered, programming a '0' or '1' into that bit.

• Role in 8085: PROM is useful for prototyping or small production runs of 8085-based embedded systems where custom firmware is needed, but the cost of mask ROM is too high, or the design might still undergo minor changes. Once programmed, it acts just like a Mask ROM.

• Practical Example: A small company developing a specialized controller using an 8085 might use a PROM to store their finalized control program. After programming, this chip is then integrated into the product.

4. EPROM (Erasable Programmable Read-Only Memory)

EPROM is a type of PROM that can be erased and reprogrammed multiple times. Data is stored by trapping electrons in an insulated gate structure within a transistor. To erase an EPROM, the chip is exposed to strong ultraviolet (UV) light, which provides enough energy to release the trapped electrons, clearing the memory content. The chip usually has a quartz window on top to allow UV light exposure.

• Role in 8085: EPROMs are extremely valuable during the development phase of an 8085 assembly language program or embedded system firmware. If you find a bug in your 8085 program, or if you need to update its functionality, you can erase the EPROM with UV light and then reprogram it with the updated code using an EPROM programmer.

• Practical Example: An engineer developing firmware for an 8085-controlled robot arm might write an assembly program, test it, find errors, then remove the EPROM from the circuit, place it under a UV eraser lamp for 20-30 minutes, and finally reprogram it with the corrected code.

5. EEPROM (Electrically Erasable Programmable Read-Only Memory)

EEPROM is an advancement over EPROM, offering electrical erasure and reprogramming. Unlike EPROM, it does not require UV light or removal from the circuit. Individual bytes or blocks of data can be erased and rewritten electrically, typically using higher voltages than normal operation. This allows for in-circuit programming and data modification.

• Role in 8085: EEPROMs are ideal for storing configuration parameters, user settings, or calibration data in an 8085-based system that needs to be non-volatile but also updated occasionally by the system itself (without needing external equipment). The 8085 program can contain routines to read from and write to the EEPROM.

• Practical Example: An 8085-based thermostat might store user-defined temperature settings or scheduling information in an EEPROM. The 8085 can read these settings on power-up, and the user can change them via a keypad, with the 8085 writing the new settings back to the EEPROM.

6. Associative Memory (Content-Addressable Memory - CAM)

Associative Memory, also known as Content-Addressable Memory (CAM), is a special type of memory that is accessed based on its *content* rather than its *address*. Instead of providing an address to retrieve data, you provide data (a search key), and the memory returns the address (or addresses) where that data is stored, or an indication if it's present.

• How it works: Each memory word in an associative memory has a comparison logic circuit associated with it. When a search key is presented, all words in the memory are compared simultaneously with the key. Matching words signal their presence. This parallel search capability makes CAM extremely fast for lookup operations.

• Role in 8085 (Specialized Context): While not typically used as general-purpose main memory for storing 8085 instructions or program data, associative memory finds its place in specialized applications where very fast pattern matching or data lookup is critical. For an 8085 system, if there's a need to quickly determine if a specific sensor reading or command code has been previously encountered without iterating through a list, a small associative memory unit could be employed. It's more of a specialized lookup hardware block than a primary program or data storage for the 8085.

• Practical Example: Imagine an 8085-controlled security system that needs to quickly identify known threat patterns or authorized access codes. A small associative memory could be used to store these patterns. When a new input comes, it's simultaneously compared against all stored patterns, providing a very rapid match/no-match result.

These various memory types illustrate the diverse needs in computer systems, from volatile workspace (RAM) to permanent instruction storage (ROM, PROM, EPROM, EEPROM), and specialized fast lookup (Associative Memory). Understanding their characteristics is fundamental to designing and programming efficient 8085-based systems.

4.) Various types of Auxiliary memories(Magnetic tape, Floppy disk, Hard Disks, Flash Memory)

Auxiliary memories, also known as secondary storage, provide non-volatile and high-capacity storage for data and programs that cannot fit into main memory or need to be stored permanently even when the computer is turned off. Unlike the main memory (RAM), which is fast but volatile, auxiliary memory is slower but offers much larger storage at a lower cost per bit, and its contents are retained without power. In the context of an 8085 microprocessor system, auxiliary memory acts as a crucial repository for the operating system, user applications, and large datasets that the 8085 can load into its main memory (RAM) for processing. The 8085 interacts with these devices through dedicated I/O controllers and ports, using IN and OUT instructions to manage data transfer.

Let's explore various types of auxiliary memories:

1. Magnetic Tape
Magnetic tape is one of the oldest forms of auxiliary storage, relying on magnetic recording on a thin strip of plastic film.

• Basic Concept: Data is stored sequentially along the length of the tape. Think of it like a music cassette tape; you have to fast-forward or rewind to find a specific song.
• Working Principle: A magnetic read/write head moves across the tape as the tape passes under it. Small magnetic domains on the tape are polarized in one direction or another to represent binary 0s and 1s. To read, the head detects these magnetic patterns.
• Characteristics:
• Sequential Access: Data must be read or written in order. This makes it very slow for retrieving specific pieces of data directly.
• High Capacity: Historically, tapes offered very high storage capacity at a very low cost per bit compared to other storage types.
• Durability: Good for long-term archiving due to its stability.
• Use Cases: Primarily used for backups, archiving large amounts of data, and disaster recovery. Even today, modern tape drives are used in data centers for these purposes due to their cost-effectiveness and capacity.
• 8085 Context: An 8085 system would communicate with a tape drive controller via its I/O ports. The controller would handle the complex mechanics of the tape drive, like winding and rewinding, and converting serial tape data into parallel bytes for the 8085 data bus. Programs or data could be loaded from tape into RAM for execution.

2. Floppy Disk
The floppy disk, once a ubiquitous form of removable storage, is a flexible magnetic disk encased in a protective jacket.

• Basic Concept: A thin, flexible plastic disk coated with a magnetic material rotates inside its jacket, allowing a read/write head to access data on its surface.
• Working Principle: When inserted into a floppy drive, the disk spins. A read/write head, positioned by an actuator arm, moves across the disk's surface. Data is organized into concentric circles called tracks, and each track is divided into sectors. The head magnetizes specific areas of the disk to store bits.
• Characteristics:
• Direct Access (but slow): Unlike tape, you can directly jump to a specific track and sector, making it a direct access storage device. However, access speed is relatively slow due to mechanical movement.

• Removable: Easily transportable, allowing data transfer between systems.
• Low Capacity: Typically 1.44 MB for the most common 3.5-inch floppy, very small by modern standards.
• Vulnerable: Susceptible to damage from dust, magnetic fields, and physical bending.
• Use Cases: Historically used for distributing software, transferring small files between computers, and booting operating systems on older PCs.
• 8085 Context: An 8085 system could use a floppy disk drive controller to load operating system components, utility programs, or small data files into its RAM. The 8085 would send commands to the controller (e.g., read sector X) and then receive data from it.

3. Hard Disks (HDD - Hard Disk Drive)
Hard disks are the primary mass storage devices in most traditional computers, providing much higher capacity and speed than floppy disks.

• Basic Concept: Composed of one or more rigid, rapidly rotating platters coated with magnetic material, sealed within an enclosure.
• Working Principle:
• Platters: Multiple metallic platters are stacked on a central spindle, rotating at very high speeds (e.g., 5400 to 15000 RPM).
• Read/Write Heads: Each platter surface has its own read/write head, mounted on an actuator arm. The heads float on a cushion of air created by the spinning platters, never touching the surface.
• Tracks, Sectors, Cylinders: Data is organized into concentric tracks. Each track is divided into sectors. A cylinder is the set of all tracks at the same radial position on all platters.
• Random Access: The actuator arm quickly moves the heads to the desired track, and the platter rotation brings the desired sector under the head. This allows for fast, random access to any piece of data.
• Characteristics:
• High Capacity: Ranging from hundreds of gigabytes to several terabytes.
• High Speed (relative to tape/floppy): Much faster data transfer rates and lower access times due to higher rotation speeds and faster head movement.
• Non-volatile: Data persists without power.
• Mechanical: Contains moving parts, making them susceptible to physical shock and wear over time.
• Use Cases: The primary storage for operating systems, applications, and user data in desktop computers, laptops, and servers.
• 8085 Context: For an 8085-based system requiring significant storage, a hard disk could be interfaced. The 8085 would interact with a sophisticated hard disk controller. This controller abstracts the complex physical operations (head positioning, timing, error correction) and presents a simpler interface to the 8085, allowing it to request data blocks by logical block addresses. Data would be transferred in blocks between the hard disk controller and the 8085's main memory.

4. Flash Memory
Flash memory is a type of non-volatile solid-state memory that uses electronic components rather than magnetic surfaces or moving parts.

• Basic Concept: Stores information in an array of memory cells made from floating-gate transistors. These transistors can trap or release electrons, representing binary 0s or 1s.
• Working Principle: When an electrical voltage is applied, electrons are either forced into the **floating gate** (to store a 1) or removed from it (to store a 0). Once set, the charge remains trapped for a long time, even without power, making it non-volatile. Erasure is typically done in blocks, hence the term **flash** (quick erasure).
• Types:
• NAND Flash: Most common for high-capacity storage like SSDs, USB drives, and SD cards. Offers higher density and lower cost per bit.
• NOR Flash: Used for storing code (like BIOS or firmware) because it allows individual bytes to be read and modified more easily.
• Characteristics:
• Solid-State: No moving parts, making it highly durable, shock-resistant, and silent.
• High Speed: Generally much faster than HDDs for both reading and writing, especially for random access.

• Compact and Low Power: Enables very small form factors and is ideal for portable devices.
• Non-volatile: Data is retained without power.
• Limited Write Cycles: Each memory cell can only be erased and rewritten a finite number of times (though modern flash controllers use wear-leveling techniques to extend lifespan).
• Use Cases: USB flash drives, Solid State Drives (SSDs - replacing HDDs in many modern systems), SD cards, memory cards in cameras and phones, embedded systems for storing firmware and configuration data.
• 8085 Context: Flash memory (especially smaller capacity NOR flash) could serve as a direct replacement for EPROM or EEPROM in an 8085 system to store the bootstrap loader, firmware, or frequently accessed data logs. For larger capacities (NAND flash), it would require a flash controller that abstracts the complex management (like wear-leveling and bad block management) and provides a block-oriented interface, similar to how an HDD controller works. The 8085 would communicate with this controller via I/O ports or memory-mapped I/O to read and write data. Its speed and reliability make it an excellent choice for critical non-volatile storage in embedded 8085 applications.

These auxiliary memory types illustrate the evolution of storage technology, moving from slow, sequential access mechanical devices to fast, solid-state electronic ones. Each has played, or continues to play, a vital role in computer architecture, providing the necessary persistence and capacity for our digital world.

5.) Cache Memory

Cache Memory

Welcome to the topic of Cache Memory. In the vast landscape of computer architecture, cache memory plays a critical role in enhancing system performance by acting as a fast intermediary between the central processing unit (CPU) and the slower main memory. Understanding cache is fundamental for anyone studying computer organization, especially when considering how processors interact with memory.

• Why Cache Memory is Needed - The Speed Gap
The journey of a program involves the CPU constantly fetching instructions and data from memory. CPUs have become incredibly fast, executing millions or billions of operations per second. However, main memory (like dynamic RAM, DRAM) has not kept pace with this speed. Accessing data from main memory takes significantly longer than the time a CPU needs to process it. This speed mismatch causes the CPU to often sit idle, waiting for data, which slows down the entire system.
• Imagine a chef (CPU) who cooks incredibly fast but has to walk to a large pantry (main memory) far away for every single ingredient. A lot of time is wasted just walking back and forth.

• What is Cache Memory?
Cache memory is a small, very fast type of RAM (usually static RAM, SRAM) that stores copies of data from the most frequently used main memory locations. It is placed physically closer to the CPU than main memory.
• In our chef analogy, the cache is like a small, easily accessible rack right next to the chef, holding ingredients frequently used in recipes.

• How Cache Memory Works - The Principle of Locality
Cache memory relies heavily on a phenomenon called **locality of reference.** This principle states that computer programs tend to access data and instructions that are spatially or temporally close to previously accessed data and instructions.
• Temporal Locality: If a particular data item is accessed, it is likely to be accessed again in the near future. (e.g., a loop counter, a frequently called function).
• Spatial Locality: If a particular data item is accessed, data items near it in memory are likely to be accessed soon. (e.g., elements of an array, sequential instructions in a program).
• When the CPU needs data, it first checks the cache.
• If the data is found in the cache, it's called a **cache hit.** This is very fast.

- If the data is not found in the cache, it's called a **cache miss.** The CPU then has to fetch the data from the slower main memory. When data is fetched from main memory due to a miss, a copy of that data, along with some surrounding data (a **cache line** or **block**), is also brought into the cache, anticipating future use based on spatial locality.

- Cache Structure and Organization

Cache memory is divided into blocks or lines. Each block from main memory can be mapped to a specific location in the cache. How this mapping occurs is crucial for cache efficiency.

- 1- Direct Mapped Cache
- This is the simplest mapping technique. Each block from main memory has only one possible location where it can be stored in the cache.
- The main memory address is divided into three parts:
- Tag: Identifies which main memory block is currently in the cache line.
- Index: Determines which cache line the main memory block can go into.
- Block Offset: Specifies the position of the desired word within that cache block.
- Advantage: Simple to implement and fast to check.
- Disadvantage: High probability of **conflict misses** if two frequently used main memory blocks map to the same cache line, even if other cache lines are empty.

- 2- Fully Associative Cache
- In this organization, any block from main memory can be placed in any available cache line.
- The main memory address is divided into two parts:
- Tag: Identifies the main memory block.
- Block Offset: Specifies the position of the desired word within that cache block.
- Advantage: Most flexible, minimizes conflict misses, and has the lowest miss rate for a given cache size.
- Disadvantage: Complex and expensive to implement because it requires simultaneously searching all cache tags to find a match. This makes it impractical for large caches.

- 3- Set-Associative Cache
- This is a hybrid approach that combines aspects of both direct-mapped and fully associative caches. The cache is divided into **sets,** and each set contains a small number of cache lines (e.g., 2-way, 4-way, 8-way).
- A main memory block can map to any cache line within a specific set.
- The main memory address is divided into three parts:
- Tag: Identifies which main memory block is in the set.
- Set Index: Determines which set the main memory block can go into.
- Block Offset: Specifies the position of the desired word within that cache block.
- Advantage: Offers a good balance between flexibility (lower conflict misses) and implementation complexity (cost). Most modern caches use this organization.

- Cache Write Policies

When the CPU writes data, there are two primary ways the cache handles updating main memory:

- 1- Write-Through
- When the CPU writes data, it is written to both the cache and main memory simultaneously.
- Advantage: Main memory is always up-to-date, simpler to implement.
- Disadvantage: Slower writes because every write operation must wait for the slower main memory.

- 2- Write-Back
- When the CPU writes data, it is written only to the cache. A **dirty bit** is set for that cache line. The corresponding main memory block is updated only when the modified cache line is evicted (replaced) from the cache.
- Advantage: Faster writes because the CPU doesn't always wait for main memory.
- Disadvantage: More complex to implement (requires managing dirty bits) and main memory can be inconsistent with the cache for short periods.

- Cache Performance Metrics

The effectiveness of a cache is measured by:
- Hit Rate: The percentage of memory accesses found in the cache.
- Miss Rate: The percentage of memory accesses not found in the cache (1 - Hit Rate).
- Average Memory Access Time (AMAT): A measure of how long it takes, on average, to access memory.
- AMAT = (Hit Rate x Cache Access Time) + (Miss Rate x Main Memory Access Time)
- A lower AMAT indicates better performance.

- Cache and the 8085 Microprocessor
- The 8085 is an 8-bit microprocessor introduced in the 1970s. It is a much simpler architecture compared to modern CPUs.
- Critically, the 8085 microprocessor does NOT feature an integrated on-chip cache memory. Its architecture predates the widespread adoption of on-chip caching.
- However, the fundamental problem that cache memory solves – the speed mismatch between a fast CPU and slower memory – was very much present even in 8085-based systems.
- The 8085 accesses external memory directly via its address and data buses. If the external memory connected to the 8085 is slow (e.g., slower DRAM or ROM), the 8085 is designed to insert **wait states.** A wait state is a clock cycle during which the CPU idles, giving the slower memory more time to respond. While this ensures correct operation, it directly reduces the effective speed of the 8085.
- In systems designed around the 8085, system architects sometimes employed external, faster static RAM (SRAM) as a **scratchpad** or **buffer** memory for critical code or frequently accessed data. This external SRAM, while not a true cache automatically managed by the CPU, served a similar *purpose*: providing faster access to commonly used information to reduce the need for wait states when accessing the main, slower memory. This required explicit programming or system design choices, rather than the transparent, automatic management provided by modern caches.
- Understanding cache memory, even when studying the 8085, helps you appreciate the evolution of computer architecture and the continuous effort to optimize performance by overcoming memory bottlenecks. The concepts of locality of reference and the need for faster access to frequently used data are universal, even if the implementation varies significantly across different processor generations.

- Real-World Cache Levels

Modern processors use multiple levels of cache:
- L1 Cache: Smallest, fastest, located directly on the CPU die, usually split into instruction cache and data cache.
- L2 Cache: Larger and slightly slower than L1, typically on the CPU die.
- L3 Cache: Largest and slowest of the caches, often shared among multiple CPU cores.
- The data flows from main memory to L3, then to L2, then to L1, and finally to the CPU core. This hierarchy allows for efficient data access, leveraging speed and capacity at each level.

This concludes our discussion on Cache Memory. The principles discussed here are foundational to understanding high-performance computing and memory management, providing a vital bridge to future topics like Virtual Memory.

6.) Virtual Memory

Virtual Memory

Virtual Memory is a memory management technique implemented by the operating system (OS) and hardware that provides an application with the illusion of having a very large, contiguous, and private memory space. This memory space can be much larger than the actual physical RAM installed in the system.

1- The Core Idea: An Illusion of Large Memory
- Imagine you have a small physical workbench (your physical RAM) but need to work with many large projects (programs and their data). You can't fit everything on the workbench at once.
- Virtual memory provides the illusion that you have an enormous, seemingly endless workbench.

You only bring the parts of the projects you are actively working on to your physical workbench. The rest are stored neatly on a shelf (secondary storage, like a hard disk) and brought to the workbench when needed.

- This **illusion** makes programming easier because developers don't have to worry about the actual physical memory constraints.

2- Why do we need Virtual Memory?

- To overcome physical memory limitations: Modern programs often require more memory than a system's physical RAM can provide. Virtual memory allows these large programs to run.
- For multitasking and multiprogramming: It enables multiple programs to run concurrently without interfering with each other's memory space. Each program believes it has a full, dedicated memory space.
- For memory protection: Each program's memory is isolated. A bug in one program is less likely to crash the entire system or corrupt data of other programs.
- For efficient memory usage: Only the currently active parts of a program need to be loaded into physical RAM. Inactive parts reside on secondary storage.

3- Key Components and Concepts

- Virtual Address Space:
- This is the set of memory addresses that a program uses. It's the memory map the program **sees.** For example, a program might think it's using addresses from 0x00000000 to 0xFFFFFFFF, regardless of the physical RAM available.

- Physical Address Space:
- This is the set of actual addresses available in the physical RAM chips installed in the computer. These are the real, hardware-level addresses.

- Pages:
- The virtual address space of a program is divided into fixed-size blocks called pages. Common page sizes are 4 KB, 8 KB, or 16 KB.

- Frames (or Page Frames):
- The physical memory (RAM) is also divided into blocks of the exact same size as pages. These physical blocks are called frames.

- Page Table:
- This is a crucial data structure, maintained by the operating system for each running program. It maps virtual pages to physical frames.
- Each entry in a page table indicates:
- Whether a virtual page is currently in physical RAM.
- If it is, which physical frame it occupies.
- Access permissions (read-only, read/write, execute).

- Memory Management Unit (MMU):
- This is a dedicated hardware component (often part of the CPU) responsible for translating virtual addresses into physical addresses. It uses the page tables to perform this translation quickly.

4- How Virtual Memory Works (Paging Process)

- Step 1: CPU generates a virtual address when a program tries to access memory.
- Step 2: The MMU takes this virtual address. It splits the address into two parts: a page number and an offset within that page.
- Step 3: The MMU uses the page number to look up the corresponding entry in the process's page table.
- Step 4 (Page Hit): If the page table entry indicates that the virtual page is present in physical RAM (marked as **valid** or **present** bit set), the MMU retrieves the physical frame number. It then combines this physical frame number with the original offset to form the complete physical address. The CPU then accesses the data at this physical RAM location.
- Step 5 (Page Fault): If the page table entry indicates that the virtual page is *not* currently in

physical RAM (e.g., marked as **invalid** or **present** bit clear), a **page fault** occurs.
• The MMU triggers an interrupt, passing control to the operating system's page fault handler.
• The OS identifies the required page on secondary storage (e.g., hard disk, in a dedicated area called **swap space** or **paging file**).
• The OS finds a free page frame in physical RAM.
• If no free frame is available, the OS must choose an existing page in RAM (often using a replacement algorithm like LRU - Least Recently Used) and save its contents back to secondary storage if it has been modified (**swapping out** or **paging out**) to free up a frame.
• The required page is then loaded from secondary storage into the newly freed (or found) physical frame in RAM.
• The page table is updated to reflect the new mapping.
• The instruction that caused the page fault is restarted.

5- Role of Secondary Storage

• Secondary storage (like hard disk drives or SSDs) acts as an extension of main memory in a virtual memory system. It holds pages that are not currently active in RAM.
• This **swap space** or **paging file** is critical for the illusion of a larger address space, allowing programs to exceed physical RAM limits.
• However, accessing data from secondary storage is significantly slower than accessing RAM, leading to performance degradation during frequent page faults.

6- Benefits of Virtual Memory (Recap)

• Provides an address space larger than physical RAM.
• Allows efficient multiprogramming by sharing physical RAM among multiple processes.
• Provides memory protection between processes.
• Simplifies memory management for programmers.

7- Limitations/Drawbacks

• Performance Overhead: Page faults and the need to access slow secondary storage can degrade system performance.
• Complexity: Requires complex hardware (MMU) and sophisticated operating system support.
• Increased Memory Usage: Page tables themselves consume a portion of physical RAM.

8- Virtual Memory in the Context of the 8085 Microprocessor

• The 8085 is an 8-bit microprocessor with a 16-bit address bus. This means it can directly address 2^16 = 65,536 bytes (64 KB) of physical memory.
• Crucially, the 8085 does *not* have a built-in Memory Management Unit (MMU).
• Therefore, the 8085 microprocessor does *not* natively support virtual memory.
• When you write 8085 assembly language programs and specify a memory address (e.g., LDA 2000H to load data from physical address 2000H), the 8085 CPU directly accesses that physical memory location. There is no intermediate translation layer between a virtual address and a physical address. The address used by the program is always a physical address.
• For a computer engineering diploma student, understanding virtual memory in the context of the 8085 involves recognizing its absence and appreciating the simplicity of direct physical addressing in older architectures. It's a foundational concept that explains why modern, more complex processors and operating systems can handle massive applications and multiple tasks so effectively, capabilities that were far beyond the direct 64KB addressing limit of the 8085. It helps in understanding the evolution of memory management techniques in computer architecture.

7.) Summary

The concept of a **summary** in the context of Computer Organization and Architecture, specifically 8085 Assembly Language Programming, refers to concisely capturing essential information. Unlike summarizing a long text document, here it primarily involves understanding and expressing the core purpose, functionality, or results of a piece of assembly code or the state of the 8085 microprocessor.

Why is summarizing crucial in assembly language programming?
1- Understanding Complex Code: Assembly programs can be detailed, with each instruction performing a tiny step. A summary helps grasp the bigger picture without getting lost in individual lines.
2- Documentation: Well-summarized code is easier for others (or your future self) to understand, maintain, and debug.
3- Debugging: When troubleshooting, quickly summarizing what a section of code *should* do helps identify where it's going wrong.
4- Data Analysis: Often, assembly programs are written to process raw data and produce summarized results, like sums, averages, or counts.

In 8085 assembly, a **summary** can take a few forms:

- A brief description of what a subroutine or main program accomplishes.
- A consolidated output or result derived from processing a larger set of data.
- An overview of the microprocessor's state (register values, flag status) after executing a program segment.

Summarizing Program/Routine Functionality

When you write an 8085 assembly language program or a specific subroutine, it's vital to provide a summary of its functionality. This is usually done through comments within the code.

Purpose of summarizing functionality:

- Quick understanding: Anyone reading the code should immediately know what it does.
- Modularity: For larger programs, breaking them into smaller, summarized routines makes development and testing easier.
- Reusability: A well-summarized routine can be easily reused in other programs if its purpose is clear.

How to write effective summaries for assembly code:

- Place a summary at the beginning of the program or before each major subroutine.
- Describe the overall goal of the code block.
- Specify any inputs required (e.g., data in specific registers or memory locations).
- Specify any outputs produced (e.g., result in a register, updated memory).
- Mention any side effects (e.g., certain registers modified).

Example: A subroutine to add two 8-bit numbers.

```
;-- Subroutine: ADD_NUMBERS --
; Purpose: Adds two 8-bit numbers.
; Input: Number 1 in Register B, Number 2 in Register C.
; Output: Sum in Register A.
; Affects: Registers A, Flags (Zero, Carry, Parity, Sign, Auxiliary Carry).
ADD_NUMBERS:
MOV A, B ; Move first number to Accumulator A
ADD C ; Add second number (from C) to A
RET ; Return from subroutine
```

Here, the commented block acts as a summary, providing all essential information without requiring a line-by-line analysis of the assembly instructions.

Data Summarization Techniques in 8085 Assembly

A common application of assembly language programming is to process raw data and produce summarized information. This is **data summarization.** Examples include finding the sum, average, maximum, minimum, or count of specific values within a dataset.

Why summarize data in assembly?

- Efficiency: For resource-constrained systems, assembly allows highly optimized data processing.
- Low-level control: Direct manipulation of memory and registers for specific data analysis tasks.
- Sensor data processing: Microprocessors like 8085 are often used in embedded systems to collect sensor data and summarize it before sending it further or making decisions.

Basic principles for data summarization:

- Initialization: Set up initial values for the summary (e.g., sum = 0, max = first element).
- Iteration: Loop through the dataset, processing each element.
- Accumulation/Comparison: Update the summary value based on each element (e.g., add to sum, compare for max/min).
- Storage: Store the final summarized result in a designated memory location or register.

Example 1: Sum of an array of N 8-bit numbers.
Assume an array of numbers starts at memory address 2001H, and the count N is stored at 2000H.

```
;-- Program: SUM_ARRAY --
; Purpose: Calculates the sum of N 8-bit numbers stored in memory.
; Input: N numbers from 2001H, count N at 2000H.
; Output: Sum stored at memory location 2100H.
; Affects: Registers A, B, C, H, L, Flags.

LXI H, 2000H ; HL points to the count
MOV C, M ; C gets the count (N) of numbers
INR L ; HL now points to the first number (2001H)
MVI B, 00H ; B (Accumulator for sum) initialized to 0

LOOP_SUM:
MOV A, M ; Get the current number into Accumulator A
ADD B ; Add the current number to the running sum in B
MOV B, A ; Update the running sum in B
INR L ; Move to the next number in memory
DCR C ; Decrement the count
JNZ LOOP_SUM ; If count is not zero, loop again

MOV A, B ; Move sum from B to A for STA instruction
STA 2100H ; Store sum at 2100H
HLT ; Halt the processor
```

This program summarizes a list of numbers into a single sum.

Example 2: Finding the maximum of N 8-bit numbers.
Assume numbers from 2001H, count at 2000H.

```
;-- Program: FIND_MAX --
; Purpose: Finds the maximum 8-bit number from an array.
; Input: N numbers from 2001H, count N at 2000H.
; Output: Maximum number stored at memory location 2101H.
; Affects: Registers A, B, C, H, L, Flags.

LXI H, 2000H ; HL points to the count
MOV C, M ; C gets the count (N)
INR L ; HL points to the first number (2001H)

MOV A, M ; A gets the first number (assume it's the max initially)
DCR C ; Decrement count, as first number is processed
JZ STORE_MAX ; If only one number, it's already the max

LOOP_MAX:
INR L ; Move to the next number
```

```
MOV B, M ; Get the current number into B
CMP B ; Compare A (current max) with B (current number)
JC SKIP_UPDATE ; If Carry flag is set (A < B), then B is greater, update A
JMP CONTINUE ; Otherwise, A is greater or equal, keep A as max

SKIP_UPDATE:
MOV A, B ; Update A with the new maximum (from B)

CONTINUE:
DCR C ; Decrement count
JNZ LOOP_MAX ; If count is not zero, loop again

STORE_MAX:
STA 2101H ; Store the final maximum value at 2101H
HLT ; Halt the processor
```

This program summarizes a list of numbers by finding the single largest value.

Summarizing 8085 Microprocessor State

After an 8085 assembly program executes, or at specific breakpoints during debugging, it's crucial to understand the **state** of the microprocessor. Summarizing this state means looking at the values in key components to understand the program's effect.

What constitutes the 8085 microprocessor state?

- Registers: The current values held in general-purpose registers (A, B, C, D, E, H, L), the Stack Pointer (SP), and the Program Counter (PC).
- Flags: The status of the 5 flag bits (Zero, Sign, Parity, Carry, Auxiliary Carry) in the Flag Register. These flags are set or reset based on the results of arithmetic and logical operations, providing a summary of the operation's outcome.
- Memory: The content of specific memory locations that the program accessed or modified.

Why is it important to summarize the state?

- Debugging: If a program isn't working as expected, examining the state (register values, flags) at different points helps pinpoint errors.
- Verification: To confirm that a program has achieved its intended outcome, you check if the final state (e.g., result in a specific memory location, correct flag status) matches expectations.
- Understanding flow: The Program Counter's value shows where the processor is currently executing, and the Stack Pointer's value indicates the state of the stack, summarizing call/return sequences.

How to **read** the summary of the state:

- Using a simulator/debugger: These tools provide a snapshot of all registers, flags, and memory content, making it easy to summarize the processor's condition.
- Manual analysis: By tracing the program's execution manually, you can predict the state changes after each instruction.
- Output devices: For actual hardware, the summarized state might be sent to an output port for display on LEDs or an LCD.

For instance, after executing the **ADD C** instruction in the ADD_NUMBERS subroutine:

- The Accumulator (A) would hold the sum.
- The Carry Flag (CY) would be set if the sum exceeded 255 (FFH).
- The Zero Flag (Z) would be set if the sum was 00H.
- Other flags (S, P, AC) would reflect their respective conditions.

This snapshot of register A and the Flag Register *summarizes* the outcome of the addition operation.

Practical Implications and Best Practices

Summarization, whether of code functionality, data, or microprocessor state, is not just a theoretical concept; it's a fundamental practice for effective 8085 assembly language programming and working with microprocessors in general.

Practical implications:

- Efficient Development: By summarizing the intent of code blocks, developers can focus on implementing one summarized piece at a time.
- Troubleshooting: Clear summaries of expected inputs, outputs, and effects of a routine drastically cut down debugging time.
- System Integration: When different assembly routines interact, their functional summaries ensure they fit together correctly, like modules in a larger system.

Best practices for summarization in 8085 assembly:

- Comment Extensively: Treat comments as your primary tool for summarizing code's purpose, inputs, outputs, and assumptions.
- Use Meaningful Labels: Labels like LOOP_SUM, FIND_MAX, ADD_NUMBERS inherently summarize what a code section does.
- Define Data Structures: Clearly document what data is stored where (e.g., **Memory location 2000H holds the count N, 2100H stores the final sum**).
- Test and Verify: Always test your assembly programs and verify that the summarized data results or the final microprocessor state match your expectations. Use simulation tools to inspect register and memory contents.
- Modular Design: Break down complex problems into smaller, manageable subroutines, each with its own clear summary.

By consistently applying these summarization practices, you will write more robust, understandable, and maintainable 8085 assembly programs, which is a critical skill for any computer engineering diploma student.