COMPUTER ORGANIZATION AND ARCHITECTURE

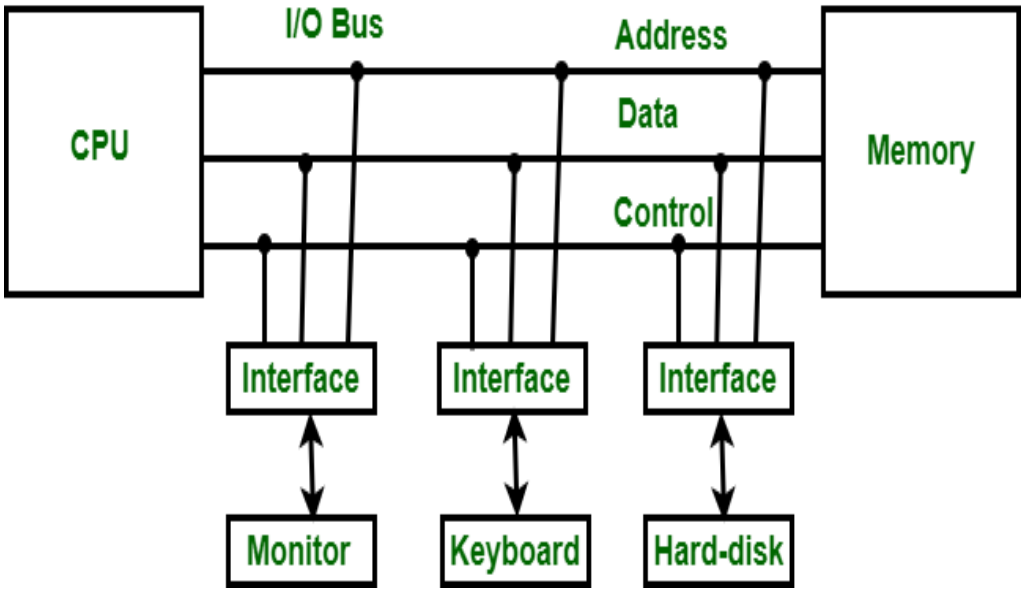
I/O Organization

Input-Output Interface

An **Input-Output Interface** facilitates communication between the **internal storage devices** (like memory) and **external peripheral devices** (like input/output devices). These interfaces are essential in allowing a computer to interact with the external world and exchange information.

The **Input-Output (I/O) Interface** is a method used to transfer data between the internal storage (memory) and external I/O devices. Since the **CPU** and **peripherals** operate at different speeds and formats, **special communication links** are used to manage this interaction.

To handle these differences, **interface units** (special hardware components) are placed between the CPU and peripherals. These units **supervise**, **control**, and **synchronize** all I/O data transfers, ensuring smooth and efficient communication.



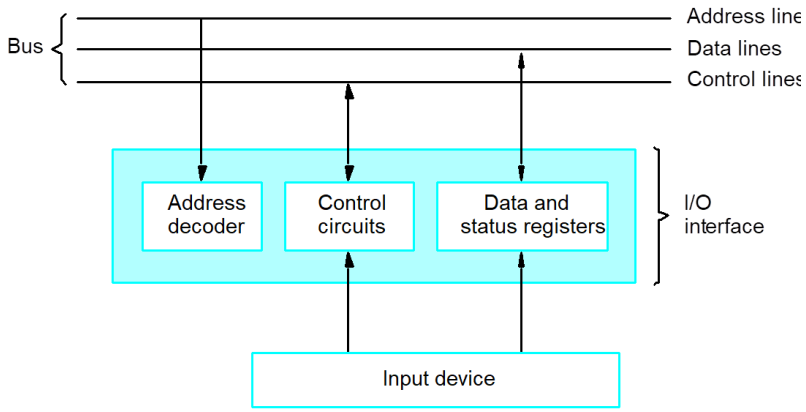


Figure 4.2. I/O interface for an input device.

I/O Devices

Peripheral devices are hardware components that are **external** to the computer's central processing unit (CPU) and memory but are crucial for **expanding the functionality** of the system. These devices provide input, output, or both capabilities to the computer system.

- **Input Devices**: These are devices that **send data into** the computer. Examples include:
    - **Keyboard, Mouse**, **Microphone**, **Scanner**
- **Output Devices**: These are devices that **receive data from** the computer and provide results or feedback to the user. Examples include:
    - **Monitor, Printer, Speaker**
- **Input/Output Devices**: Some devices are capable of **both input and output** operations, such as:
    - **External hard drives**: Can both send data to the computer (input) and store data (output).
    - **Touchscreens**: Act as both input (touch gestures) and output (displaying visuals).
- **Storage Devices**
    - **SSD, HDD, USB Flash Drive**

Functions of Peripheral Devices

- **Extend system functionality**: They help the system interact with the outside world by providing input and output.
- **Enhance user experience**: Devices like monitors and speakers enhance the interaction with the system by providing a graphical and audio interface.
- **Data storage and transfer**: Storage peripherals allow large data volumes to be stored and transferred outside the primary memory.

Connectivity Types

- **Wired**: USB, HDMI, VGA, etc.
- **Wireless**: Bluetooth, Wi-Fi, etc.

Functions of Input-Output Interface

1. **Synchronization**: It synchronizes the operating speed of the CPU with the slower input-output devices to ensure smooth data transfer.

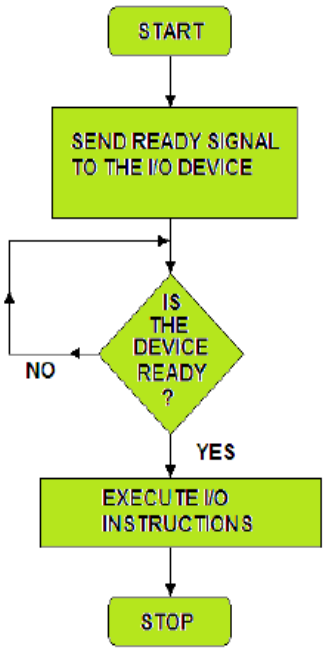
2. **Device Selection**: It selects the appropriate input/output device for communication, based on the type of signal or request.
3. **Control and Timing Signals**: The interface generates necessary **control** and **timing signals** to coordinate the data exchange between the CPU and I/O devices.
4. **Data Buffering**: Temporary **data buffering** is handled using the data bus to store data during transfer, helping in smooth communication.
5. **Error Detection**: It includes mechanisms to **detect errors** in data transmission, improving reliability.
6. **Data Conversion (Serial ↔ Parallel)**: Converts **serial data to parallel** and **parallel to serial** depending on the device requirements.
7. **Analog-Digital Conversion**: Converts **digital signals to analog** and **analog signals to digital**, enabling communication with a variety of devices like sensors or audio systems.

Modes of Data Transfer

In a computer system, data is transferred between the **CPU** and **I/O devices**, but the **memory unit** is always the source or destination of this data. The CPU acts as the processor, but actual data movement happens between **memory** and **peripherals**.

There are **three main modes** of data transfer:

1. **Programmed I/O (PIO)**
2. **Interrupt-Initiated I/O**
3. **Direct Memory Access (DMA)**



1. Programmed I/O (PIO)

Programmed I/O is a method where the **CPU actively controls** data transfer between itself and an I/O device.

How It Works

- The **CPU continuously checks** (polls) the I/O device to see if it's ready.
- Once ready, the CPU executes instructions to perform data transfer.

CPU Involvement

- The CPU is **fully responsible** for every read/write operation.

- It stays in a loop until the I/O operation completes (busy waiting).

Drawbacks

- **Inefficient**: CPU is **occupied** with I/O tasks, even when it could be doing something else.
- **Slow**: Overall system performance drops due to constant CPU intervention.

Alternatives

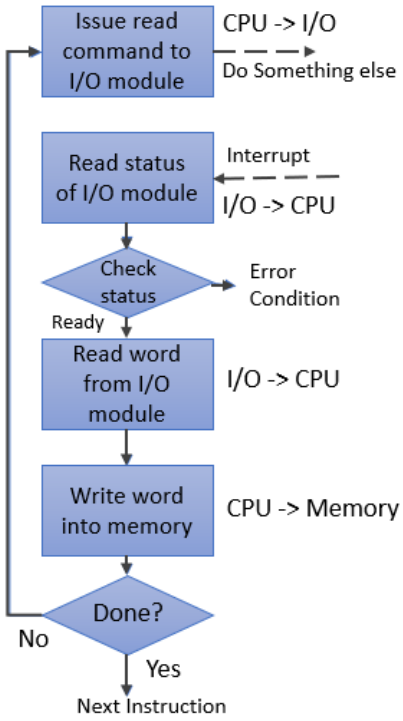
To improve efficiency:

- Use **Interrupt-Initiated I/O**, where the device notifies the CPU only when ready.
- Use **DMA**, where data is transferred directly between memory and device, bypassing the CPU.

---

2. Interrupt-Initiated I/O (also called Hardware-Initiated I/O)

This method allows the **I/O device to send an interrupt signal** to the CPU when it is ready for data transfer. The **CPU continues other tasks** and only responds when interrupted, making it more efficient than Programmed I/O.



Interrupted I/O to Transfer Data from I/O Module to Memory

How It Works

- The CPU continues executing other instructions.
- When the I/O device is ready, it sends an **interrupt signal** to the CPU.
- The CPU **pauses its current task**, handles the I/O request, and then **resumes** its previous task.

CPU Involvement

- CPU is **not continuously involved**.
- It only **responds when interrupted** by the I/O device.
- This frees the CPU from **unnecessary waiting**.

Advantages

- ✅ **Reduced CPU overhead** – CPU works on other tasks and only responds when needed.
- ✅ **No constant status checking** of I/O devices.
- ✅ **Improves system performance** by reducing CPU idle time.

Disadvantages

- ❌ **Complex to implement**, especially in low-level programming.
- ❌ **Interrupt handling routines** are required, which may introduce delays.

Useful for systems where I/O events occur **occasionally** and unpredictably (e.g., keyboard input, mouse clicks).

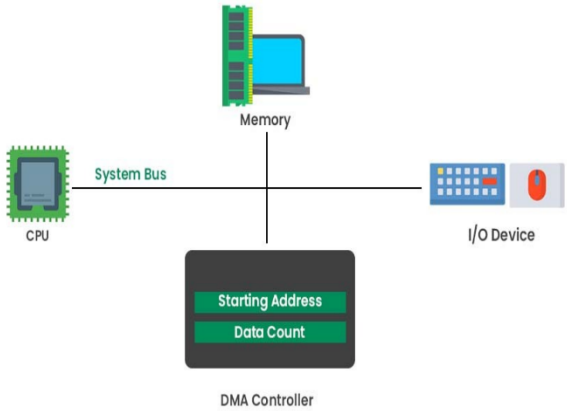
---

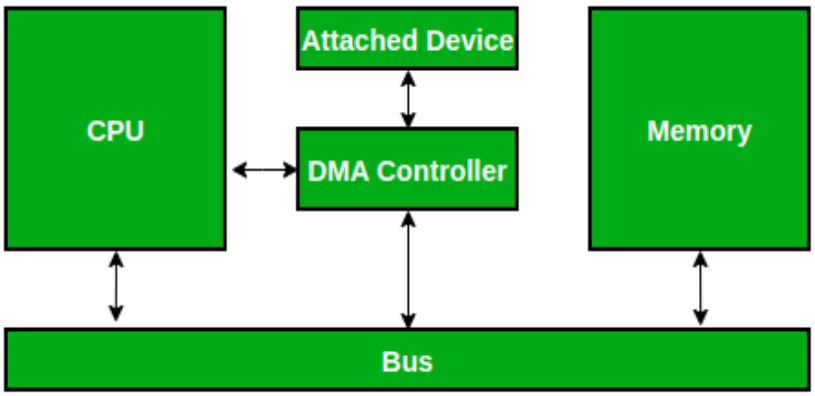
3. Direct Memory Access (DMA)

Direct Memory Access (DMA) is a feature or method used in computer systems that allows certain hardware components to **access the system's main memory (RAM)** without requiring the central processing unit (CPU) for each data transfer. DMA is generally used to improve data transfer efficiency between peripheral devices and memory.

🔹 How DMA Works

1. **Initialization**:
   CPU sets up the DMA controller (source, destination, size, direction).
2. **Peripheral Request**:
   A peripheral device (e.g., disk, sound card) requests DMA.
3. **Bus Control**:
   DMA controller takes control of the system bus (after CPU grants permission).
4. **Data Transfer**:
   DMA transfers data between memory and the device directly.
5. **Parallel CPU Operation**:
   CPU continues executing instructions while transfer occurs.
6. **Completion Notification**:
   DMA notifies the device (and optionally the CPU) when transfer is complete.





🔹 **Advantages of DMA**

- ✅ **Reduces CPU overhead** – CPU is free for other tasks.
- ✅ **Improves system performance** – Enables multitasking.
- ✅ **Faster data transfers** – Transfers data at hardware speed.
- ✅ **Efficient resource usage** – Multiple devices can work in parallel.
- ✅ **Supports large-volume transfers** – Ideal for disk I/O, multimedia, networking.
- ✅ **Enables parallel processing** – CPU and DMA work independently.

🔁 **Programmed I/O** vs **Interrupt-Initiated I/O** vs **DMA**

Feature	Programmed I/O (PIO)	Interrupt-Driven I/O	Direct Memory Access (DMA)
Definition	Data transfer is controlled by the CPU. The CPU continually checks the I/O device's status.	The I/O device sends an **interrupt** to the CPU when it is ready for data transfer.	Data is transferred **directly** between I/O devices and memory, **without CPU involvement** in the transfer.
CPU Involvement	The CPU is **actively** involved in the entire transfer.	The CPU is **interrupted** only when the device is ready, allowing the CPU to perform other tasks in between.	The CPU configures DMA but is **not involved** in the data transfer itself.
Efficiency	**Low Efficiency:** The CPU is busy checking the device status, which can waste processing time.	**Moderate Efficiency:** The CPU can multitask and only handles I/O when required.	**High Efficiency:** Data transfer is managed by DMA, reducing CPU involvement and allowing parallel processing.
Speed	**Slow:** CPU must handle each transfer step, making it inefficient for high-speed transfers.	**Faster than PIO:** The CPU can handle other tasks while waiting for the interrupt, allowing for faster transfers overall.	**Fastest:** DMA allows for direct data transfer at the maximum speed of the system bus.
Complexity	**Simple to Implement:** No need for special	**More Complex than PIO:** Requires *interrupt*	**Complex to Implement:** Requires *DMA controller*

Feature	Programmed I/O (PIO)	Interrupt-Driven I/O	Direct Memory Access (DMA)
	hardware, just CPU instructions.	handling mechanisms and routines.	and proper coordination with the CPU and memory.
Overhead	**High Overhead:** CPU is constantly checking the I/O device, leading to inefficiency.	**Lower Overhead:** The CPU only intervenes when necessary, reducing its workload.	**Minimal Overhead:** DMA offloads the data transfer work from the CPU, improving overall system performance.
Use Case	Suitable for low-**speed devices** where CPU monitoring is acceptable.	Suitable for devices with sporadic data transfer requirements (e.g., keyboard, mouse).	Ideal for **high-speed, large volume data transfers** (e.g., disk I/O, network cards).
Example	Reading/writing small files from a disk (slow transfer rate).	Data transfer from a sensor to the CPU when the sensor is ready.	Transferring large chunks of data between memory and peripherals like a hard disk or network adapter.

Concept of Interrupt

An **interrupt** is a mechanism used by hardware or software to temporarily halt the normal execution of a program and redirect the CPU's attention to a specific task or event. When an interrupt occurs, the CPU stops executing its current instructions and jumps to a special routine, known as an **interrupt service routine (ISR)** or **interrupt handler**, to handle the event. After the interrupt is serviced, the CPU resumes executing the interrupted program from where it left off.

Key Points:

- **Hardware Interrupts**: Generated by external devices (e.g., keyboard, mouse).
- **Software Interrupts**: Triggered by software to request system services.

Advantages:

- **Efficient CPU usage**: CPU performs other tasks until an interrupt occurs.
- **Real-time processing**: Useful for time-sensitive events.
- **Reduces polling**: Devices signal when they need attention, instead of constant checking.

Disadvantages:

- **Complexity**: Managing interrupts requires careful handling of priorities and states.
- **Overhead**: Interrupts can introduce small delays in processing.

Interrupts enable **efficient**, **real-time communication** between the CPU and peripherals, improving **multitasking** and **responsiveness**.

Hardware Interrupt

A **hardware interrupt** is generated by an external device or hardware component to signal the CPU that it needs attention. It interrupts the CPU's current operation to handle an event or perform a task.

Key Characteristics:

- **Source**: Generated by external devices like keyboards, mouse, timers, or sensors.
- **Purpose**: To notify the CPU of an event (e.g., data is ready to be read).
- **Maskable**: Some hardware interrupts can be disabled (maskable), while others are non-maskable and must be serviced immediately (e.g., power failure).

Example: A keyboard sends an interrupt to notify the CPU that a key has been pressed.

Software Interrupt

A **software interrupt** is triggered by a program or software to request system-level services or to perform a specific task.

Key Characteristics:

- **Source**: Initiated by software (programs, OS, or system calls).
- **Purpose**: Used for system calls, error handling, or to request services from the operating system.
- **Controlled**: Software interrupts are typically programmable and can be executed by a software instruction.

Example: A program invokes a software interrupt to request the operating system to perform I/O operations, like reading a file.

Concept of USB (Universal Serial Bus)

USB (Universal Serial Bus) is a widely used interface standard for connecting peripheral devices to computers, enabling **communication**, **data transfer** and **power supply**. It was introduced to simplify and standardize the process of connecting various devices (such as keyboards, mice, storage devices, printers, and cameras) to a computer, replacing older, more complex interfaces like serial and parallel ports.

Features of USB:

1. **Plug and Play**: Devices can be connected or disconnected without restarting the computer.
2. **Data Transfer Speeds**: USB supports multiple speeds for efficient data transfer.
3. **Power Supply**: Provides power to peripherals like phones and external drives.
4. **Backward Compatibility**: New USB versions are compatible with older versions.
5. **Bidirectional Communication**: Allows both sending and receiving data.
6. **Multi-Device Support**: USB hubs can connect multiple devices to one port.

Properties of USB:

1. **Data Transfer Rates**:
    - USB 1.1: 12 Mbps
    - USB 2.0: 480 Mbps
    - USB 3.0: 5 Gbps
    - USB 3.1/3.2: Up to 20 Gbps
    - USB 4.0: Up to 40 Gbps
2. **Voltage and Current**: USB provides **5V** power with varying current levels.
3. **Connector Types**:
    - USB-A, USB-B, Micro-USB, USB-C
4. **Power Delivery (PD)**: Fast charging technology supported by USB-C.
5. **Device Classes**: USB devices are classified based on function (e.g., audio, storage).

Types of USB:

1. **USB 1.1/2.0**: Older versions with lower speeds (12 Mbps and 480 Mbps).
2. **USB 3.0/3.1/3.2**: Faster data transfer rates (up to 20 Gbps).
3. **USB 4.0**: Latest version, offering 40 Gbps speeds.

4. **USB-C**: Compact, reversible connector supporting fast transfer and high-power charging.
5. **USB OTG (On-The-Go)**: Allows mobile devices to connect to peripherals directly.

USB simplifies device connections, offering flexibility, speed, and power, making it essential for modern devices.

Memory Segmentation in COA is a memory management technique that **divides memory** into different **segments**, each with a specific purpose. These segments are independently managed, making memory usage more efficient.

Types of Segments:

1. **Code Segment**: Stores program instructions.
2. **Data Segment**: Holds variables and global/static data.
3. **Stack Segment**: Manages function calls, local variables, and return addresses.
4. **Heap Segment**: Stores dynamically allocated memory.

How Segmentation Works:

- **Segment Table**: Keeps track of each segment's **base address** and **size**.
- When the program accesses memory, the CPU uses this table to translate logical addresses to physical addresses.

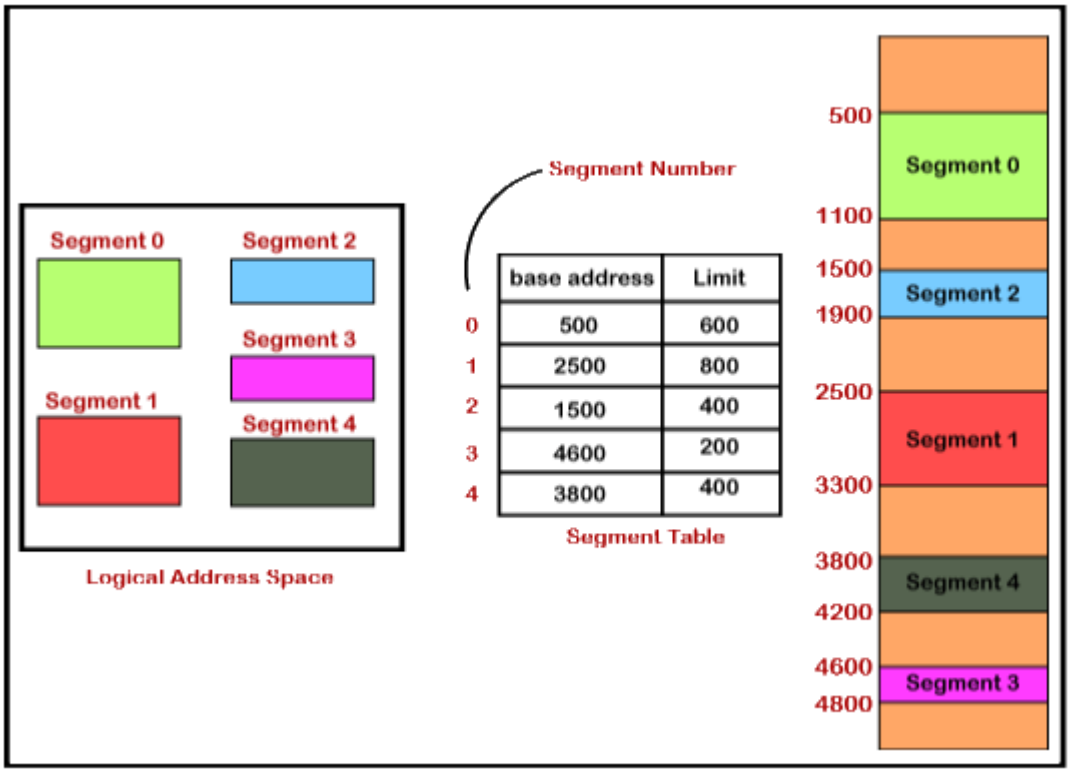
Advantages:

- **Logical Organization**: Easier to manage large programs.
- **Protection**: Different segments can have specific access controls.
- **Efficient Memory Use**: Reduces fragmentation by logically dividing memory.

Disadvantages:

- **External Fragmentation**: Variable-sized segments can lead to fragmented memory.
- **Complex Management**: Requires tracking segments and handling address translation.
- **Size Limits**: Managing dynamically growing segments can be tricky.

In summary, memory segmentation helps in organizing memory logically, making it easier to manage and allocate, but also introduces challenges like fragmentation and complexity.



Base Address: This is the **starting physical address** of a segment in memory.

Limit: The **limit** defines the **maximum size (or length)** of the segment.

Offset: It is the **distance** (or displacement) from the base of the segment to the desired data/instruction.

Effective Address: Base Address + Offset (computed physical address).

Effective Address

The **Effective Address (EA)** is the address used by the CPU to access a specific memory location.

In COA, the **Effective Address** refers to the final address in memory that a program accesses, which is calculated using different addressing modes (such as direct, indirect, indexed, or base).

How Effective Address is Calculated:

$$Effective\ Address = Base\ Address + Offset$$

Importance in Computer Architecture:

- **Memory Access**: The EA determines where in memory the data is located or where an instruction will be executed.

- **Flexibility**: The effective address allows more flexibility in data access, as it can adapt to different data structures like arrays, lists, or stacks.

In Summary:

The **Effective Address** is the result of the calculation based on the addressing mode, providing the final memory location where data is fetched or stored. It plays a crucial role in translating the program's logical addresses into actual memory locations.

What is a Register?

- A **register** is a small, fast memory unit in the CPU made of **flip-flops**, each storing one bit.
- An **n-bit register** has n flip-flops and can store n bits of binary data.

🔸 **Types of Registers:**

- **Accumulator (AC):** Stores data from memory for processing.
- **General Purpose Registers:** Hold intermediate values during execution.
- **Special Purpose Registers:** Used internally by the system (not user-accessible).

🔹 **Important Special Registers:**

- **MAR (Memory Address Register):** Holds the memory location to access.
- **MBR (Memory Buffer Register):** Stores data/instruction to/from memory.
- **PC (Program Counter):** Points to the next instruction.
- **IR (Instruction Register):** Stores the current instruction.

🔁 **Register Transfer**

- Register transfer is represented as: R2 ← R1
  (Move contents of R1 into R2)

⚙️ **Micro-Operations**

Operations on registers are called **micro-operations**, e.g.:

- **Arithmetic:** Add, Subtract, Increment, Decrement
  Example: R3 ← R1 + R2
  (Add R1 and R2, store result in R3)
- **Logic, Shift, Clear, Load** operations also exist.

RTL (Register Transfer Language)

Register Transfer Language (RTL) is a symbolic notation used to describe operations, **data transfers**, and **manipulations** among registers, memory, and the ALU in a computer system.

It is a symbolic notation used to describe the behavior of a computer system at the register level during the execution of an instruction.

RTL is used to represent the transfer of data between registers and memory, and to specify the operations to be performed in a step-by-step manner.

Symbols	Description	Examples
Capital letters & numerals	Denotes a register	MAR, R2
Parentheses ()	Denotes a part of a register	R2(0-7), R2(L)
Arrow ←	Denotes transfer of information	R2 ← R1
Colon :	Denotes termination of control function	P:
Comma ,	Separates two micro-operations	A ← B, B ← A

Concept of RTL:

- **Registers**: The main storage locations used during execution.
- **Transfer of Data**: Describes the movement of data between registers, memory, and the ALU (Arithmetic Logic Unit).
- **Operations**: Specifies the operations performed on the data, such as addition, subtraction, loading, storing, etc.
- **Control Signals**: Defines the control signals required to operate the registers and other components involved in the execution.

Basic Structure of an RTL Expression:

An RTL expression generally follows the form:

$$destination\ register \leftarrow operation\ (source\ register(s))$$

Where:

- **Destination Register**: The register where the result will be stored.
- **Operation**: The operation to be performed (e.g., addition, load, or store).
- **Source Registers**: The registers or memory locations that supply the data for the operation.

Example of RTL:

Consider the instruction: $ADD\ R1, R2, R3$

This instruction performs the addition of the contents of registers **R2** and **R3**, and stores the result in **R1**.

The **RTL** representation would be: $R1 \leftarrow R2 + R3$

Pipeline:

A **pipeline** is a hardware mechanism in a CPU that allows **multiple instructions** to be processed **simultaneously** by breaking down instruction execution into **separate stages**.

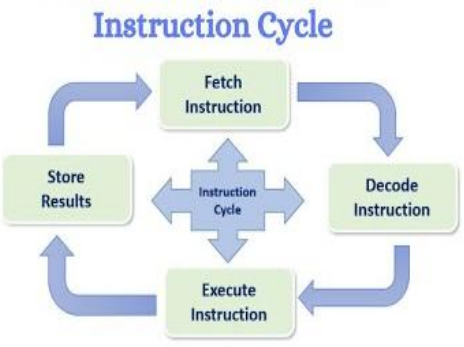
- **Pipelining** is a technique where multiple instructions are executed in **overlapping stages** to improve system **throughput**.
- Think of it as an assembly line — the output of one stage is the input for the next.

Concept of Instruction Pipelining:

Instruction pipelining is a technique used in computer architecture to improve the throughput of a processor by **executing multiple instructions simultaneously**, but in different stages. Instead of waiting for one instruction to complete before starting the next, instruction pipelining breaks the execution of an instruction into distinct stages, allowing different instructions to be processed at each stage.

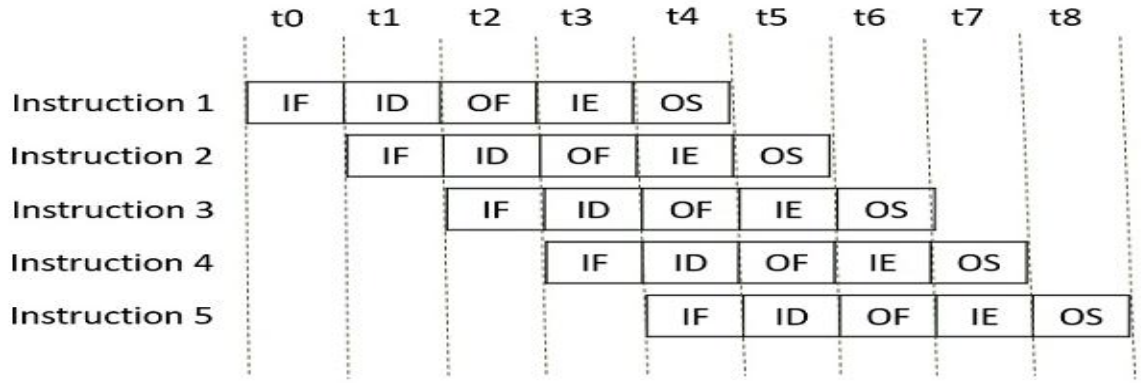
Stages of Instruction Pipelining:

Typically, the stages in an instruction pipeline are:



1. **Fetch (IF)**: The instruction is fetched from memory.
2. **Decode (ID)**: The instruction is decoded to determine the operation and operands.
3. **Execute (EX)**: The operation is executed (e.g., arithmetic or logic operation).
4. **Memory (MEM)**: Access memory if required (e.g., read/write operations).
5. **Write-back (WB)**: The result is written back to the register file.

Each of these stages can handle different instructions concurrently, allowing the CPU to perform more tasks in the same amount of time.

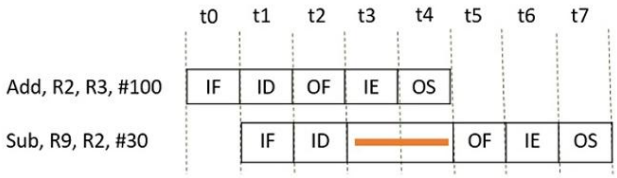


Pipeline Hazards:

Pipelining hazards are situations in a processor pipeline that cause delays in the execution of instructions. These delays arise because different stages of the pipeline rely on one another, and any conflict or dependency between stages can stall the pipeline. There are three primary types of hazards in pipelining:

1. Data Dependency Hazard:

- **Description:** This occurs when an instruction depends on the result of a previous instruction that has not yet completed.
- **Example:**



  - **Instruction 1:** ADD R2, R1, R3 (R2 = R1 + R3)
  - **Instruction 2:** SUB R4, R2, R5 (R4 = R2 - R5)
  - The SUB instruction requires the value of R2 from the ADD instruction, which is only available after the fourth clock cycle. Therefore, the SUB instruction has to stall for two clock cycles (t3 and t4) to avoid generating incorrect results.

2. Memory Delay Hazard:

- **Description:** This occurs when an instruction or data is required from memory but must first be searched in the cache. If the data is not in the cache (cache miss), it will be fetched from main memory, which takes additional clock cycles.
- **Example:** If an instruction needs data from memory and there is a cache miss, it may stall the pipeline for several cycles until the data is fetched from memory.

3. Branch Delay Hazard:

- **Description:** This happens in the case of branch instructions. When a branch instruction is encountered, the target instruction address is computed, but previous instructions might have already been fetched. These instructions may need to be discarded, leading to a delay.
- **Example:**
    - **Instruction 1 (I1): Branch (target instruction is Ik)**
    - **Instructions I2, I3, I4 are fetched before the branch target is computed.**
    - **The branch decision is made in cycle t3, and the instructions I2, I3, and I4 are discarded, causing a delay of three cycles (t1, t2, t3).**

Advantages of Pipelining:

- **Improved Throughput:** Pipelining improves the throughput of the system by allowing multiple instructions to be processed concurrently.
- **Concurrent Execution:** Pipelining enables the execution of multiple instructions at the same time, with each instruction in a different stage of execution.
- **Efficient Processor Utilization:** It ensures the processor is never idle, as it is always working on different stages of various instructions.
- **Reduced execution time**: Overall time for program execution is reduced as multiple instructions are being processed concurrently.

Challenges in Instruction Pipelining:

1. **Hazards**: Issues such as data hazards, control hazards, and structural hazards can cause delays and reduce pipeline efficiency.
2. **Pipeline Stalls**: These occur when the pipeline cannot proceed due to a hazard, leading to delays in instruction execution.
3. **Complexity**: Managing the pipeline stages, hazards, and dependencies adds complexity to the CPU design and control logic.

Addressing Modes: Direct and Indirect

Addressing modes define how the operand (data) for an instruction is located or accessed in memory. They are an essential part of an instruction's format, determining how addresses are interpreted.

1. Direct Addressing Mode:

- **Concept:** In direct addressing, the operand (data) is directly specified by the address in the instruction.
- **How it Works:** The instruction itself contains the address of the operand. The CPU directly fetches the operand from that address.
- **Example:** If an instruction is ADD 1000, the value at memory address 1000 is directly used in the operation. No additional calculations are needed to determine the address.
- **Advantages:**
    - Simple and fast as the address is directly given in the instruction.
    - Requires fewer memory accesses, as the operand is directly accessed.
- **Disadvantages:**
    - Limited flexibility, as the operand must always be at a fixed address.
- **Illustration:** $Instruction: ADD\ 1000$

  $Operand: Memory[1000]$

2. Indirect Addressing Mode:

- **Concept:** In indirect addressing, the instruction contains a reference (address) that points to another memory location, where the actual operand resides.
- **How it Works:** The instruction specifies an address that contains the actual address of the operand. This indirection requires an extra memory access to get the operand.
- **Example:** If an instruction is ADD (1000), the value at memory address 1000 is treated as the address where the operand is stored. For example, if memory[1000] contains the address 2000, the operand will be located at memory[2000].
- **Advantages:**
    - More flexible, as the operand's address can change dynamically.
    - Can access large ranges of memory using a small address field.
- **Disadvantages:**
    - Slower than direct addressing, as an additional memory access is needed to fetch the actual operand.
- **Illustration:** $Instruction: ADD\ (1000)$

  $Operand: Memory[Memory[1000]]$

Peripheral Devices vs **CPU** vs **Memory**

Aspect	**Peripheral Devices**	**CPU (Central Processing Unit)**	**Memory (RAM/Storage)**
Definition	External hardware components that expand the system's functionality.	The core component of the computer responsible for executing instructions.	Stores data temporarily (RAM) or permanently (HDD, SSD).
Role in the System	Allow interaction with external devices, enabling input, output, and storage.	Performs computation and controls the overall operation of the system.	Stores data, instructions, and system information used by the CPU.
Examples	Keyboard, Mouse, Monitor, Printer, External Hard Drives	Intel, AMD processors, ARM processors.	RAM (Random Access Memory), SSD, HDD, USB drives.
Type	Hardware components external to the main system.	Primary hardware component for processing data and executing programs.	Can be primary (RAM) or secondary (SSD, HDD) storage.
Data Flow	Send data to the system or receive data from it.	Fetches instructions and data from memory and executes them.	Holds data and instructions temporarily (RAM) or permanently (storage).
Interaction	Interacts with the CPU via I/O interfaces to transfer data.	Processes data and controls the operations of all other components.	Provides data storage and access to the CPU, depending on whether it's RAM or storage.
Speed	Generally slower than the CPU in terms of data transfer rates.	The fastest component responsible for executing operations.	RAM is fast for temporary data, while storage devices (HDD/SSD) are slower but provide permanent storage.
Dependency	Dependent on the CPU and memory for processing and operations.	Relies on memory to store instructions and data for processing.	Dependent on the CPU to access or modify stored data.

Aspect	Peripheral Devices	CPU (Central Processing Unit)	Memory (RAM/Storage)
Power Consumption	Typically consumes less power compared to the CPU.	Consumes significant power during operation, especially in high-performance tasks.	Power consumption varies: RAM uses more during high usage, while storage devices have lower consumption.
Functionality	Extends the system's functionality by providing means for interaction with users and external devices.	Performs computations, runs applications, and controls all system operations.	Stores the operating system, applications, and data for quick access by the CPU (RAM) or long-term storage (HDD/SSD).