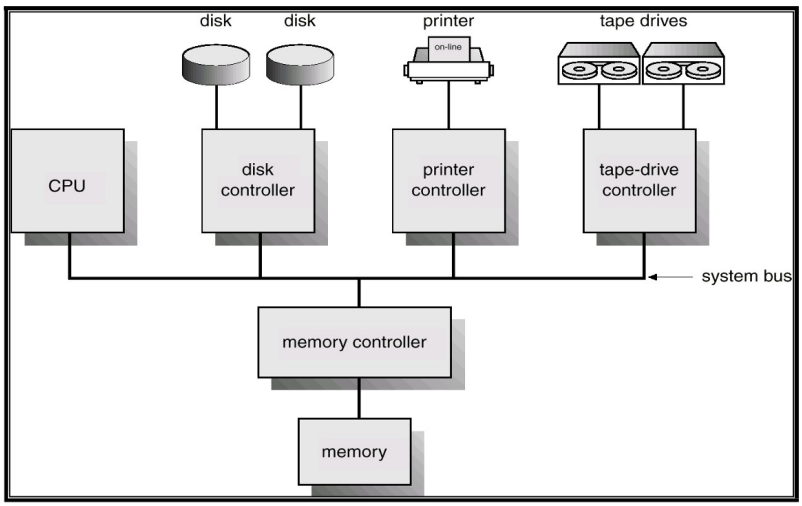
Computer System Architecture

Computer Architecture

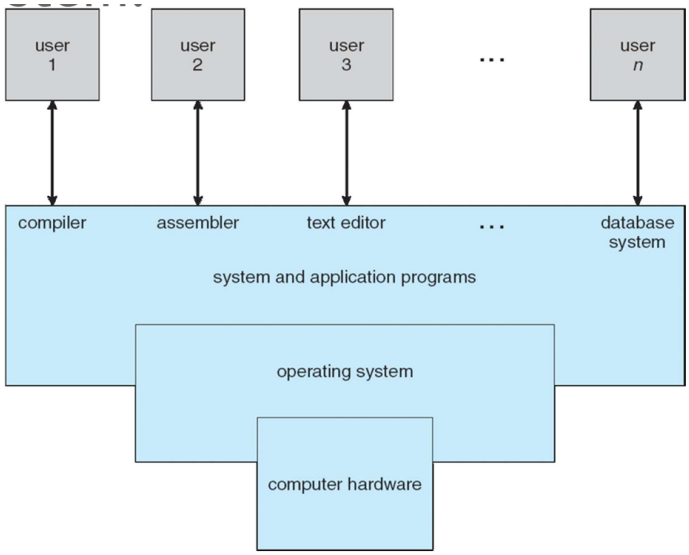
A computer's architecture consists of several components working together to perform various tasks. The core components include the CPU (Central Processing Unit), device controllers, and shared memory. These components are interconnected through a common bus, which allows them to communicate and share resources efficiently.



Key Components:

1. **CPU (Central Processing Unit):**
   - **Role:** The CPU is the brain of the computer, responsible for executing instructions, performing calculations, and managing data flow.
   - **Interaction:** The CPU communicates with other components, such as memory and I/O devices, through the common bus.
2. **Device Controllers:**
   - **Role:** Device controllers manage specific hardware devices like disk drives, audio devices, and video displays.
   - **Local Buffers:** Each device controller has its own local buffer, a small storage area used to temporarily hold data being transferred to or from the device.
   - **Example:** The disk drive controller manages the operations of a hard drive, ensuring that data is read from or written to the disk.
3. **Shared Memory:**
   - **Role:** Shared memory is a common area that the CPU and device controllers use to store and retrieve data.

- **Access:** Both the CPU and device controllers access this memory via the common bus, allowing them to share information.



Major Computer Components

1. CPU (Central Processing Unit):

The CPU is the main component that processes instructions, performs calculations, and manages data. It has several key parts:

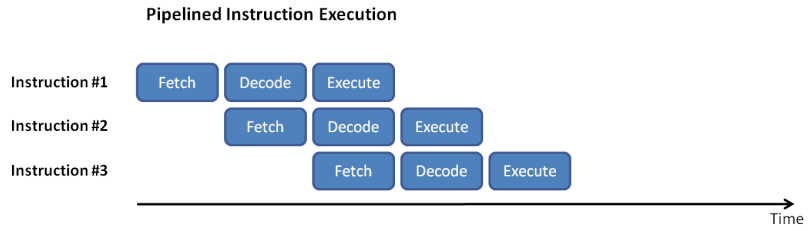
Registers:

- **Program Counter (PC):** Keeps track of the next instruction to be executed.
- **Instruction Register (IR):** Holds the most recent instruction fetched from memory.
- **Program Status Word (PSW):** Stores condition codes, interrupt status, and the mode of operation (supervisor/user).
- **General-Purpose Registers:** Temporary storage for data and addresses during processing.

- **Processor Internal Registers:**
  - **Memory Address Register (MAR):** Specifies the address for the next read or write operation.
  - **Memory Buffer Register (MBR):** Holds data being transferred to or from memory.
  - **I/O Address Register & I/O Buffer Register:** Manage data flow to and from I/O devices.
- **User-Visible Registers:**
  - **Data Registers:** Store data being processed.
  - **Address Registers:** Hold memory addresses.
  - **Index and Segment Pointers:** Assist in accessing data structures and memory segments.
  - **Stack Pointer:** Points to the top of the stack, used for managing function calls and returns.

Instruction Set Architecture (ISA):

- **Pipelining:** A technique to execute multiple instructions simultaneously by breaking them into stages like Fetch, Decode, Execute, and Write Back.

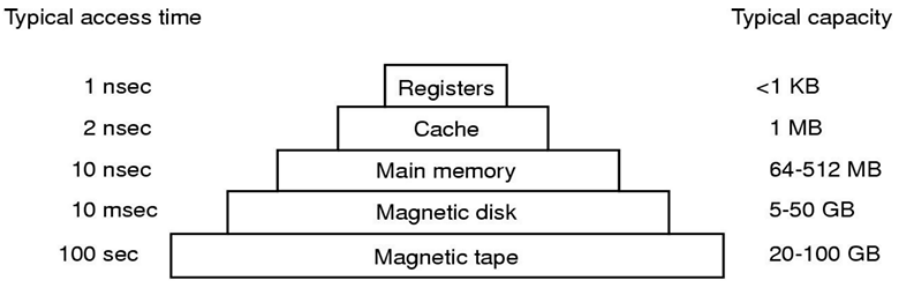


2. Memory Structure

Memory in a computer is organized in a hierarchy based on speed, cost, and volatility:

Storage Hierarchy:

- **Speed:** Faster storage is more expensive but provides quicker access.
- **Cost:** Higher-speed memory costs more.
- **Volatility:** Volatile memory loses its data when power is off, unlike non-volatile memory.



3. I/O Structure

Input/Output (I/O) operations involve data transfer between the computer and external devices:

I/O Process:

- **With Wait:** The CPU waits for the I/O operation to complete, idling in the meantime.
- **Without Wait:** The CPU continues executing other tasks while waiting for I/O completion, often using interrupts or system calls.

Device Controllers:

- **Components:** A device controller is the electronic part that manages a specific device, while the device itself is the mechanical component.
- **Functions:** Controllers manage local buffers, accept commands from the operating system, and handle data transfer. They often include embedded programs to manage these tasks.

Device Driver:

- **Role:** Software that communicates with the device controller, sending commands and receiving responses. Different drivers are needed for different types of controllers.

Device-Status Table:

- **Entries:** Each I/O device has an entry that tracks its type, address, and current state (e.g., functioning, idle, or busy).
- **Queueing:** If a device is busy, new requests are queued until the device is available.

Interrupts

An interrupt is a signal from a hardware device (usually an I/O device) that tells the CPU it needs attention. When an interrupt occurs, the CPU temporarily stops its current tasks, handles the interrupt, and then resumes its previous activities.

Key Points:

- **Interrupt Handling:** When an interrupt occurs, the CPU saves its current state (like the contents of registers and the program counter) so it can resume its previous work later.
- **Interrupt Management:** To avoid losing interrupts while another one is being processed, incoming interrupts are disabled during interrupt handling.

Types of Interrupts:

1. **Hardware Interrupts:**
   - **Definition:** Generated by hardware devices to signal that they need attention from the operating system.
   - **Examples:** A keyboard sends an interrupt when a key is pressed.
   - **Purpose:** These interrupts may indicate that a device has completed a task or needs to transfer data between the device and memory.
2. **Software Interrupts:**
   - **Definition:** Generated by programs to request a system service from the operating system.
   - **Examples:** A program might use a software interrupt to call functions like `printf` in C or `cout` in C++.
   - **Purpose:** These interrupts are used for making system calls, such as requesting file operations or accessing system resources.
3. **Traps:**
   - **Definition:** Generated by the CPU itself to indicate an error or a specific condition that needs attention from the operating system.
   - **Examples:**
     - A division by zero error.
     - An invalid memory access attempt.

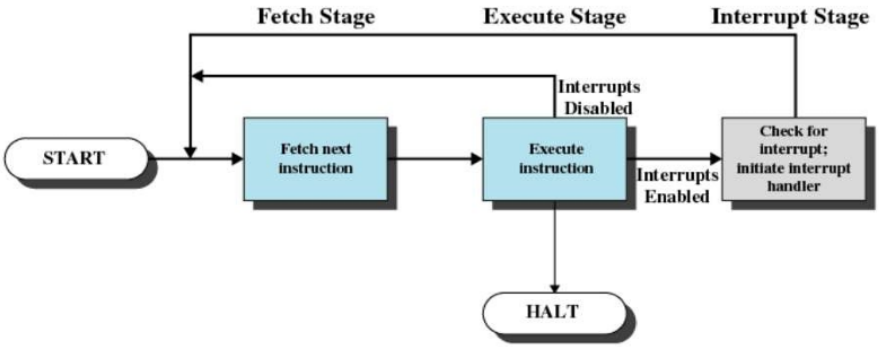
- **Purpose:** Traps are a way for the CPU to signal the operating system about exceptional conditions that require immediate handling.

Interrupt Handling Process

When an interrupt occurs, the CPU suspends its normal sequence of execution and diverts to a routine designed to handle the interrupt, known as an **Interrupt Service Routine (ISR)**. This process ensures that critical tasks triggered by hardware or software can be addressed without interrupting the flow of regular operations for too long.

Steps in the Interrupt Handling Process:

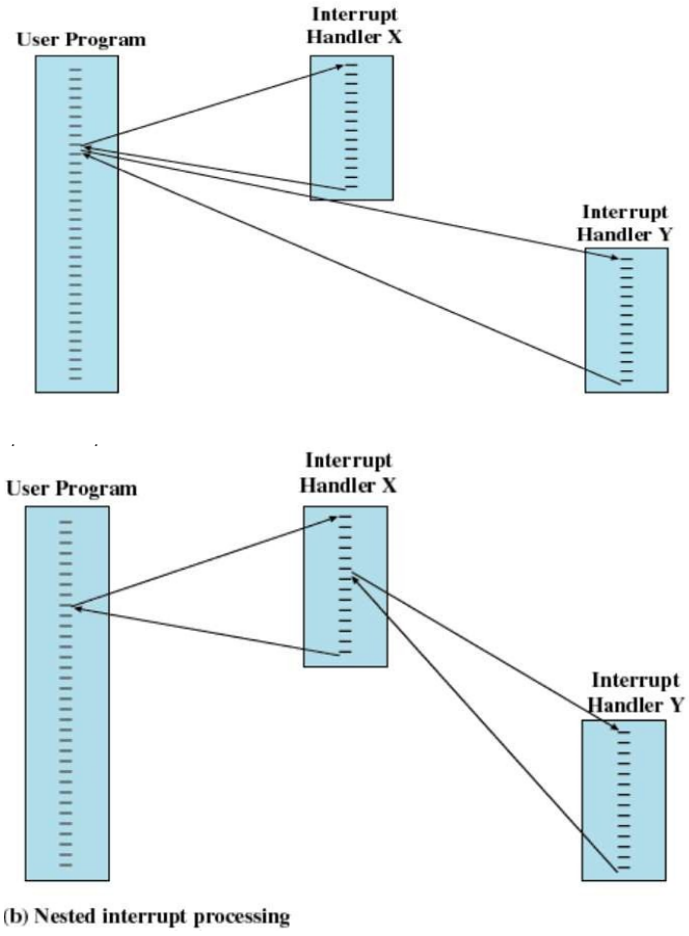
1. **Interrupt Detection:**
   - The CPU detects an interrupt signal from a hardware or software source.
   - The normal execution is temporarily suspended to handle the interrupt.
2. **Saving the Current State:**
   - The CPU saves the address of the currently executing instruction (Program Counter) so that it can resume execution after the interrupt.
   - Other important registers may also be saved to ensure the CPU can restore the exact state after handling the interrupt.
3. **Generic Interrupt Routine Execution:**
   - A generic routine is run to identify the source and nature of the interrupt.
   - The type of interrupt is examined to determine which **Interrupt Service Routine (ISR)** should be called.
4. **Calling the ISR:**
   - The appropriate ISR corresponding to the specific interrupt is executed.
   - ISRs are often stored in a dedicated area of memory, typically the lower part, for quick access.
   - The ISR handles the interrupt, such as completing an I/O operation or responding to an error condition.
5. **Restoring the State:**
   - Once the ISR has finished executing, the saved address of the interrupted instruction is loaded back into the **Program Counter**.
   - Other saved registers are also restored to their previous values.
6. **Resuming Normal Execution:**
   - The CPU resumes normal operation from where it left off, continuing as if the interrupt had never occurred.



Multiple Interrupts

To handle multiple interrupts, the system may use the following methods:

- **Disabling Interrupts:** While one interrupt is being processed, other interrupts are temporarily disabled to prevent conflicts.



(b) Nested interrupt processing

- **Priority Levels:** In systems with multiple interrupt sources, each interrupt is assigned a priority. Higher-priority interrupts can preempt lower-priority ones, ensuring that critical tasks are handled first.

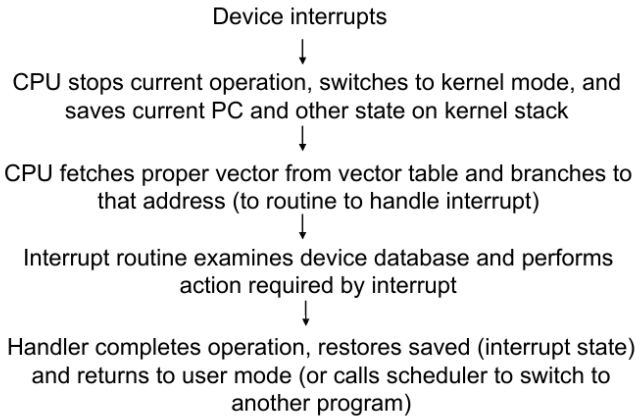
By organizing interrupts through ISRs and prioritization, the CPU can efficiently manage multiple tasks without sacrificing performance.

I/O Interrupts and Methods

I/O Interrupts occur when an I/O device completes its task and needs to inform the CPU. This typically happens after a user program has requested an I/O operation, and the operating system (OS) takes control, interacting with the device controller to manage the request.

I/O Process:

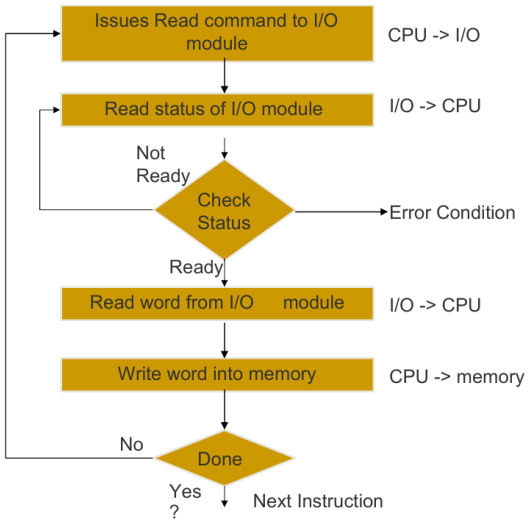
- The **device driver** loads appropriate values into the registers of the device controller.
- The **device controller** checks the command and initiates the requested operation (e.g., read or write).
- Once the I/O operation is complete, the device controller sends an **interrupt** to the CPU to signal completion.



I/O Methods:

1. Synchronous I/O

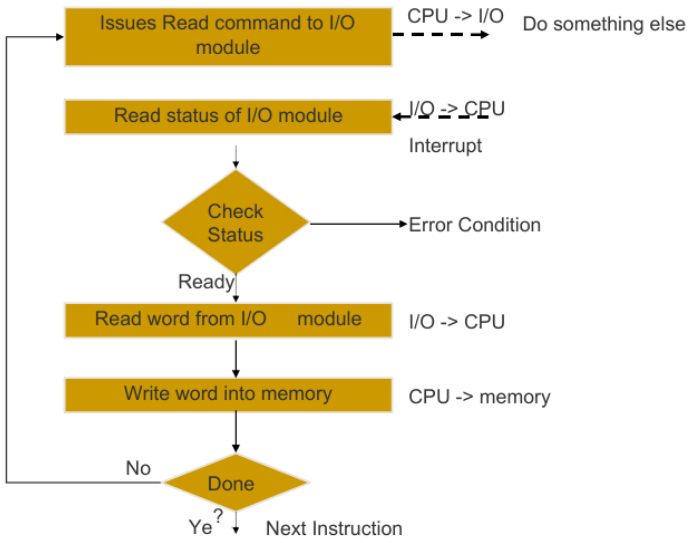
In **Synchronous I/O**, the CPU must wait until the I/O operation completes before resuming execution. This ensures that the I/O task is fully finished before the CPU continues, but it limits the number of concurrent I/O operations.



- **Process:**
  1. The OS issues an I/O command to the device.
  2. The CPU waits for the device to finish the operation.
  3. If the device does not support interrupts, the CPU may enter a **busy-wait loop**, checking the device's status repeatedly (polling).
  4. Once the operation completes, the OS handles the data, and the CPU can proceed with its next task.
- **Drawbacks:**
  - Only one I/O operation can be outstanding at a time, preventing concurrent I/O operations.
  - The CPU remains idle during the wait, which wastes valuable processing cycles.
- **Polling**: The CPU continuously checks the status of the I/O device, waiting for completion.

2. Asynchronous I/O

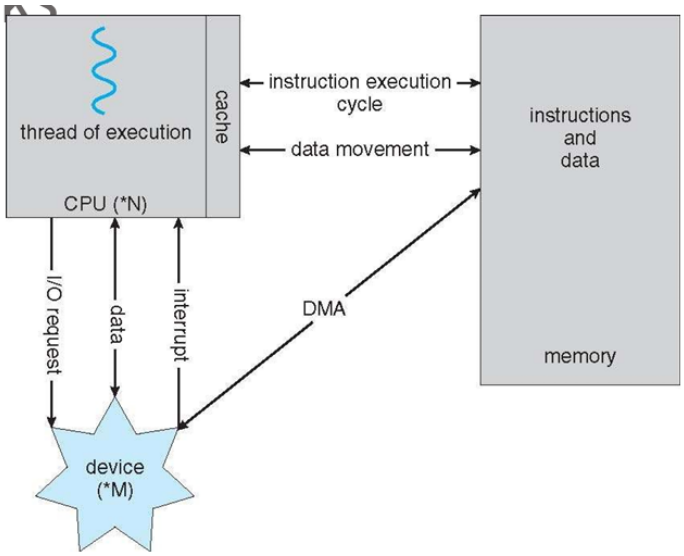
In **Asynchronous I/O**, the CPU initiates the I/O operation and immediately moves on to other tasks. When the I/O operation completes, the device controller sends an interrupt to notify the CPU.



- **Process:**
  1. The OS issues an I/O command to the device.
  2. The CPU continues executing other instructions while the I/O operation takes place.
  3. Once the I/O completes, the device controller sends an **interrupt** to inform the CPU.
  4. The CPU then processes the completed I/O operation.
- **Benefits:**
  - Allows multiple I/O operations to be outstanding simultaneously, improving overall system performance.
  - The CPU doesn't need to wait idly for I/O completion, enabling multitasking.

3. Direct Memory Access (DMA)

DMA is a specialized method used to speed up I/O operations by allowing data transfer between an I/O device and memory without CPU involvement.



- **Process:**
  1. The **DMA module** is configured with the data transfer request, including the device address, memory location, and number of words to transfer.
  2. The **DMA** takes over the task of transferring data directly from the device to main memory.
  3. The **CPU** is free to perform other tasks while the transfer is in progress.
  4. When the transfer is complete, the DMA module sends a single interrupt to notify the CPU.
- **Advantages:**
  - Reduces CPU load by offloading data transfer tasks to the DMA module.
  - Only one interrupt is generated per block of data, rather than one interrupt per byte.
- **Cycle Stealing:**
  - The DMA module may "steal" memory access cycles from the CPU, slightly slowing down the CPU, but it still offers more efficient data transfer compared to other methods.

In modern computers, DMA is commonly used for high-speed devices like disk drives to maximize performance and minimize CPU intervention.