




Pipelining in Simple Terms

- Pipelining is like an assembly line in a factory. Instead of doing one task at a time, we break a process into smaller steps and work on multiple tasks at once.

Pipelining in Simple Terms

- **Real-World Example: Washing Clothes**
- Imagine you have to wash, dry, and fold 5 sets of clothes.
- **Without Pipelining (One-by-One Processing)**
 1. Wash the first set. 
 2. Dry the first set. 
 3. Fold the first set. 
 4. Repeat for the second set, third set, etc.
- This takes a long time because each step is done one after the other.

Pipelining in Simple Terms

➤ With Pipelining (Overlapping Tasks)

1. Start **Washing** the first set.
 2. While the first set is **Drying**, start **Washing** the second set.
 3. While the first set is **Folding** and the second set is **Drying**, start **Washing** the third set.
- Now, multiple tasks happen at the same time, making the process **faster**!

Pipelining in Simple Terms

- **Key Takeaway** Pipelining increases speed by doing different stages of a task at the same time, just like an assembly line or washing multiple clothes efficiently. 😊

Basic Concepts of Pipelining

- Pipelining is a technique used in computer architecture to improve the throughput of instruction execution.
- It allows multiple instructions to be processed simultaneously by breaking down the execution process into stages.
- Each stage performs a specific task, and the instruction moves through these stages one at a time.

Stages in a Pipeline

➤ In a CPU, instead of completing one instruction at a time, the processor overlaps different stages:

1. Fetch instruction (like picking up clothes to wash).

2. Decode it (like reading the washing instructions).

3. Execute it (like running the washing machine).

4. Write the result back (like putting clean clothes away).

➤ Each stage works in parallel for different instructions, making the computer much faster.



Stages in a Pipeline

➤ A typical instruction pipeline has the following stages:

- 1. Fetch (IF):** Retrieve the instruction from memory.
- 2. Decode (ID):** Interpret the instruction and determine required actions.
- 3. Execute (EX):** Perform operations as per instruction.
- 4. Memory Access (MEM):** Read or write data from/to memory.
- 5. Write Back (WB):** Store the result in registers.

Key Terms in Pipelining

- **Throughput:** Number of instructions processed per unit time.
- **Speedup:** Performance improvement by pipelining, calculated as execution time without pipelining divided by execution time with pipelining.

Key Terms in Pipelining

1. Throughput (Work Done Per Unit Time)

- **Definition:** The number of tasks (e.g., washed clothes) completed in a given time.
- In our example, throughput means how many sets of clothes we finish per hour.

Without Pipelining (One-by-One Processing)

- Suppose each set takes 1 hour (washing + drying + folding).
- For 5 sets, total time = 5 hours.
- Throughput = 5 sets / 5 hours = 1 set per hour.

With Pipelining (Overlapping Tasks)

- Once the first set is in drying, the second set starts washing, and so on.
- After the pipeline is full, we complete 1 set every 20 minutes instead of 1 hour.
- Throughput increases to 3 sets per hour! 🚀

Key Terms in Pipelining

2. Speedup (How Much Faster?)

- **Definition:** How much faster the pipelined process is compared to the non-pipelined process.
- **Formula:**

$$\text{Speedup} = \frac{\text{Time without pipelining}}{\text{Time with pipelining}}$$

Without Pipelining

- Takes 5 hours for 5 sets.

With Pipelining

- Takes around 2 hours for 5 sets.

Speedup Calculation

$$\text{Speedup} = \frac{5 \text{ hours}}{2 \text{ hours}} = 2.5 \times \text{faster}$$

Key Terms in Pipelining

➤ Final Takeaway

- **Throughput** = How many sets we complete per hour.
- **Speedup** = How much faster pipelining makes the process.
- **Pipelining increases throughput and speedup, making everything more efficient!**



Pipeline Hazards

- Issues that can slow down a pipeline. Three main types:
 - - Structural Hazards
 - - Data Hazards
 - - Control Hazards

Pipeline Hazards

- A **pipeline** is like an assembly line in a factory, where multiple instructions are processed in different stages simultaneously to increase speed.
- However, sometimes problems occur that slow down or stop this smooth flow- these problems are called **pipeline hazards**.

Structural Hazards

- **What happens?** The hardware is not enough to handle multiple instructions at the same time.
- **Real-world example:** Imagine a coffee shop with only one coffee machine. If two baristas need to use it at the same time, one of them has to wait. Similarly, if a CPU has only one memory unit but multiple instructions need to access it at once, a delay occurs.

Data Hazards

- **What happens?** One instruction needs the result of a previous instruction, but the result is not yet ready.
- **Real-world example:** Suppose you are cooking pasta:
 - Step 1: Boil water.
 - Step 2: Add pasta.
 - Step 3: Drain pasta.
- You cannot drain the pasta before it is cooked. If step 3 starts before step 2 finishes, an error occurs.
- In CPUs, if one instruction depends on the result of a previous instruction that is still being processed, it causes a data hazard.

Control Hazards

- **What happens?** The processor guesses which instruction to execute next but gets it wrong, leading to wasted work.
- **Real-world example:** Imagine you are driving and come to a T-junction. You guess that your destination is on the right, but after driving for a while, you realize it's on the left. You now have to go back and take the correct route.
- Similarly, when the CPU guesses the wrong next instruction (due to branch prediction), it has to discard the incorrect instructions and start over, causing delays.

Summary

- **Structural Hazard** → Not enough resources (e.g., one coffee machine for two baristas).
- **Data Hazard** → An instruction depends on a previous result (e.g., cannot drain pasta before cooking).
- **Control Hazard** → Wrong branch prediction (e.g., taking the wrong turn and going back).
- These hazards slow down CPU performance, so modern processors use techniques like **pipelining optimization, forwarding, and branch prediction** to reduce delays. 🚀

Parallel Processors

- Parallel processors refer to **multiple processing units (CPUs) working together** to perform tasks faster by dividing the work among them. Instead of executing tasks one by one (sequential processing), parallel processing allows multiple tasks to be handled **simultaneously**.
- Parallel processors are widely used in **high-performance computing, AI, gaming, and data centers** to process large amounts of data quickly.

Why Use Parallel Processing?

- **Speed Improvement** – Tasks get completed faster.
- **Efficient Resource Utilization** – Multiple CPUs or cores work together efficiently.
- **Better Performance for Large Tasks** – Useful in scientific simulations, AI, and large-scale computations.
- **Handling Multiple Users** – Used in servers and cloud computing to manage multiple requests.

Types of Parallel Processing

1.Bit-Level Parallelism

- 1.Operates on **multiple bits simultaneously** within a CPU register.
- 2.Example: Increasing the size of data buses from 8-bit to 16-bit speeds up operations.

2.Instruction-Level Parallelism (ILP)

- 1.Executes **multiple instructions simultaneously** using pipelining and superscalar execution.
- 2.Example: Modern CPUs execute different parts of multiple instructions at the same time..

Types of Parallel Processing

3. Data-Level Parallelism (DLP)

3. The same operation is performed on **multiple data elements** at once.
4. Example: **Vector processing** in graphics rendering and AI computations.

4. Task-Level Parallelism (TLP)

1. Different **tasks or processes** run in parallel on multiple processors.
2. Example: Running multiple apps (browser, video player, and game) on a multi-core CPU.

Types of Parallel Processing

1. **SISD (Single Instruction, Single Data)** Traditional single-core processors.
2. **SIMD (Single Instruction, Multiple Data)** One instruction operates on multiple data points at once.
3. **MISD (Multiple Instruction, Single Data)** Multiple instructions operate on the same data stream (rarely used).
4. **MIMD (Multiple Instruction, Multiple Data)** Each processor executes different instructions on different data.

Concurrent Access to Memory

➤ Real World Example

➤ **Single Counter (No Concurrency):**
Customers form a single queue, processed one at a time.

1. Multiple Counters (Concurrent Access):
Many customers are processed at different counters.

Concurrent Access to Memory

- In modern computing, multiple processes or threads often need to access the same memory simultaneously.
- This is known as **concurrent access to memory**.
- While this improves performance and resource utilization, it also introduces challenges such as **race conditions, deadlocks, and data inconsistency**.

Key Issues in Concurrent Memory Access

Race Conditions

- A race condition occurs when multiple threads try to read or write shared memory simultaneously, and the final outcome depends on the order of execution.
- Example: Two threads updating a shared counter at the same time might result in incorrect values.

Key Issues in Concurrent Memory Access

➤ Data Inconsistency

- When multiple processes modify the same data without synchronization, inconsistent or corrupted data may be stored.
- Example: A bank account balance update might be incorrect if two simultaneous transactions read and write conflicting values.

Key Issues in Concurrent Memory Access

➤ Deadlocks

- A deadlock happens when two or more processes wait indefinitely for resources held by each other.
- Example: Two trains approaching a single-track bridge from opposite directions, both waiting for the other to move.

Key Issues in Concurrent Memory Access

➤ Cache Coherence Issues

- CPUs use caches to speed up memory access, but when multiple processors access and modify the same memory location, inconsistencies arise.
- Example: Two employees working on the same document offline but seeing different versions when they sync.

Techniques to Handle Concurrent Memory Access

➤ 1. Synchronization Mechanisms

➤ To avoid race conditions and data inconsistencies, we use synchronization techniques:

- **Mutex (Mutual Exclusion)**
- **Semaphores**
- **Atomic Operations**

Techniques to Handle Concurrent Memory Access

- **Mutex (Mutual Exclusion):** A lock that allows only one thread to access a shared resource at a time. Example: A bathroom key in an office — only one person can use it at a time.
- **Semaphores:** Counters that limit access to a resource, allowing a fixed number of processes to use it simultaneously. Example: Parking spaces in a mall — limited spots mean limited access.
- **Atomic Operations:** Operations that complete without interruption, ensuring correct execution. Example: Counting money without anyone interrupting the process.

Techniques to Handle Concurrent Memory Access

2. Memory Consistency Models: Different architectures define how memory operations appear to different threads:

- **Sequential Consistency:** All operations appear in a global sequence.
- **Weak Consistency:** Memory updates are not immediately visible to other threads.

Techniques to Handle Concurrent Memory Access

➤ 3. Hardware Solutions

- **Cache Coherence Protocols:** Ensure all processors see the same memory state.
- **Memory Barriers:** Prevent the CPU from reordering instructions that affect memory access.

Cache Coherency

- **What is Cache Coherency?**
- Cache coherency is a mechanism that ensures **all copies of a shared data item** in different caches of a **multi-core processor system** are up to date.
- This prevents **data inconsistency**, which can occur when multiple processors access and modify the same data stored in separate caches.

Why is Cache Coherency Important?

- In modern computers, multiple processors (cores) are used to improve performance.
- Each processor has its own **cache memory** to store frequently accessed data, reducing the time needed to access the main memory (RAM).
- However, when one processor modifies a data item in its cache, the **other caches may still have an old (stale) version** of the same data.
- If not managed properly, this can cause errors in calculations and incorrect results.


Real-World Example – A Shared Whiteboard in an Office

- Imagine a group of coworkers using a **shared whiteboard** to keep track of a project:
 1. Each person keeps a **personal notepad** (like a cache) to write down updates from the whiteboard.
 2. One person updates the whiteboard with new information.
 3. If others don't **synchronize** their notepads (caches) with the whiteboard, they will still have **old data**.
 4. This can lead to mistakes because some people will work with outdated information.
- Cache coherency works **like a rule** that ensures **everyone updates their notepads** whenever the whiteboard changes so that **everyone always has the latest information**.


How Does Cache Coherency Work?

- To maintain cache coherency, computer systems use **cache coherence protocols**. These protocols ensure that all copies of a shared data item in multiple caches remain consistent. The two most common cache coherence protocols are:
 - 1. Write-Invalidate Protocol
 - 2. Write-Update Protocol

1. Write-Invalidate Protocol

- When a processor modifies a data item, it **invalidates** the copies in all other caches.
- Other processors must then **reload** the latest data from memory before using it.
-  *Example:* If one coworker erases the whiteboard and writes new information, everyone else must **update their notepads** before using the data again.

2. Write-Update Protocol

- When a processor modifies a data item, it **sends the updated data** to all other caches.
- Other caches update their copies **immediately** instead of invalidating them.
-  *Example:* If one coworker updates the whiteboard, they also **announce the change** to everyone, so they update their notepads immediately.

Challenges in Cache Coherency

➤ Even with cache coherence protocols, maintaining consistency is challenging because of:

- 1. Latency (Delay):** Synchronizing multiple caches takes time and can slow down performance.
- 2. Bandwidth Usage:** Frequent updates require more communication between caches, which can consume resources.
- 3. Scalability Issues:** As the number of processors increases, keeping caches coherent becomes more complex.

Conclusion

- **Cache coherency is essential for multi-core processors to function correctly. Without it, different processors might work with outdated or incorrect data, leading to errors. Cache coherence protocols help maintain consistency by either invalidating old data or updating all caches with the latest information.**
- **By using rules similar to a shared whiteboard in an office, cache coherency ensures that all processors always have the most accurate and up-to-date data.** 🚀