Unit 6

Pipelining

The Smartphone Analogy

In this analogy:

- **The Smartphone:** The Computer System.
- **Mobile Apps:** Different programs ($P_1, P_2, P_3 \ldots$).
- **App Tasks:** The instructions within an app ($I_1, I_2, I_3 \ldots$).
- **The "Work":** The 4 stages of an instruction ($F + D + E + S$).

- **Core 1** handles I_1 (Loading a post).
- **Core 2** handles I_2 (Playing a song).
- **Core 3** handles I_3 (Checking GPS).
- **Core 4** handles I_4 (Syncing email).

Sanjeev Thapa, Er. DevOps, SRE, CKA, RHCSA, RHCE, RHCSA-Openstack, MTCNA, MTCTCE, UBSRS, HEv6, Research Evangelist

Pipelining

1. Serial Processing

- In **serial processing**, **only one instruction is processed at a time**.
- An instruction must **fully complete all its steps** before the next instruction starts.
- No overlap occurs between instructions.
- Simple to design but **slow** and **inefficient**.

Instruction Phases

Each instruction consists of:

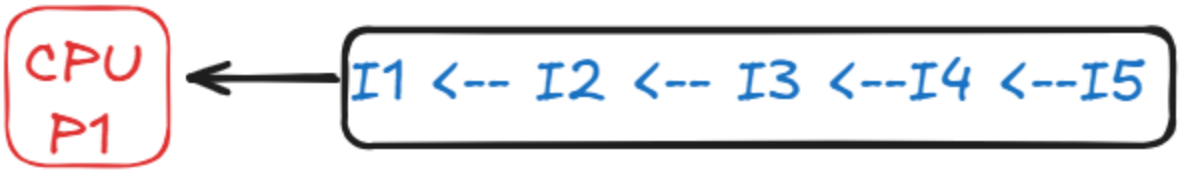
1. **Opcode Fetch (OF)**
2. **Decode (D)**
3. **Execute (E)**
4. **Store (S)**

Time per phase = **1 µs**
Total time per instruction = **4 µs**

Example (Serial Execution of I1–I5)

Instruction Sequence

I1 → I2 → I3 → I4 → I5



I1 (OF + D + E + S) + I2 (OF + D + E + S) + I3 (OF + D + E + S) + I4 (OF + D + E + S) + I5 (OF + D + E + S)

Time Calculation: Time for 1 instruction = 4 µs Number of instructions = 5

Total Time = 5 × 4 = 20 µs

Execution Table (Serial)

Time (µs)	Operation
0–4	I1 (OF + D + E + S)
4–8	I2 (OF + D + E + S)
8–12	I3 (OF + D + E + S)
12–16	I4 (OF + D + E + S)
16–20	I5 (OF + D + E + S)

Key Points

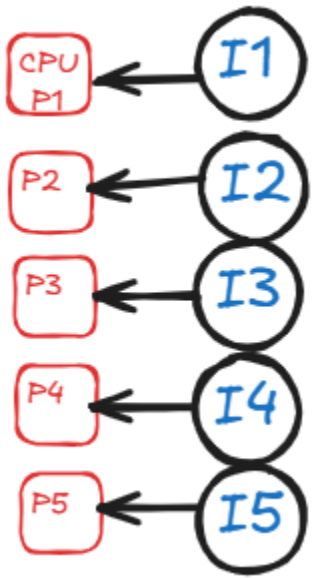
✓ Very easy to understand
❌ Wastes CPU resources
❌ Slow execution speed

---

2. Parallel Processing

- In **parallel processing**, **multiple instructions are executed simultaneously**.
- Requires:
  - Multiple processors or
  - Multiple execution units
- Improves **throughput** and **performance**.
- Used in **multi-core CPUs**, **supercomputers**, **GPUs**.

Example (Parallel Processing of I1–I5)



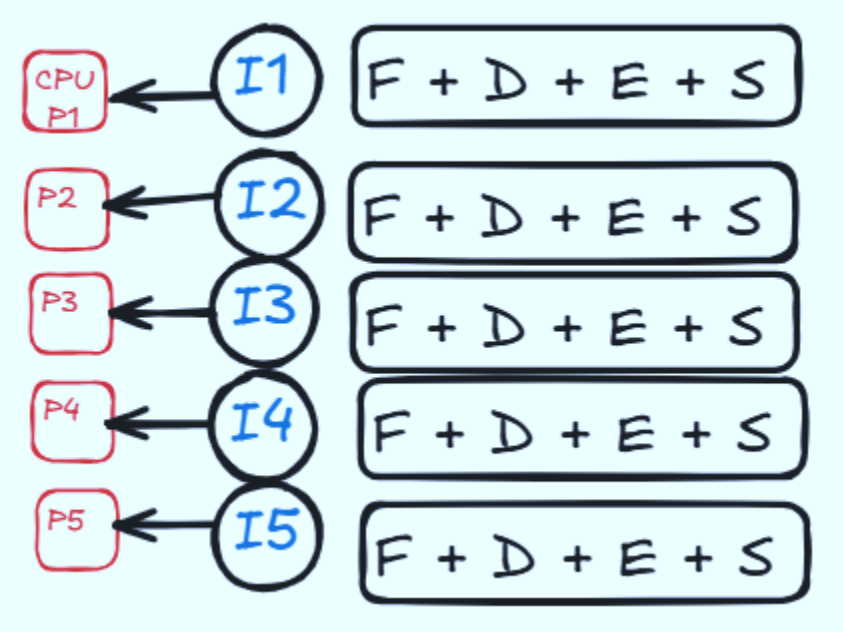
Assume **5 processors** available.

- Each instruction takes **4 µs**
- All instructions run at the same time

Total Time = 4 µs

Execution View

Processor	Instruction	Time
P1	I1	0–4 µs
P2	I2	0–4 µs
P3	I3	0–4 µs
P4	I4	0–4 µs
P5	I5	0–4 µs



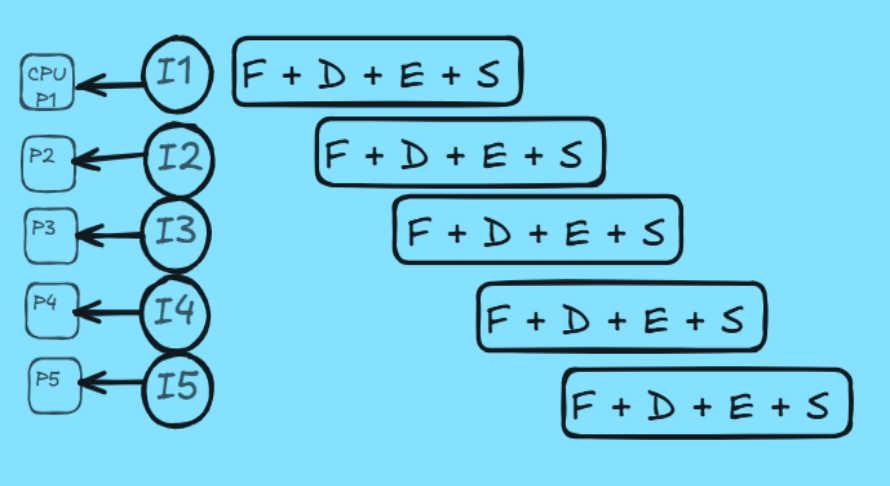
Key Points

✓ Very fast
✓ High performance
✗ Expensive hardware
✗ Complex synchronization

---

3. Pipeline Processing

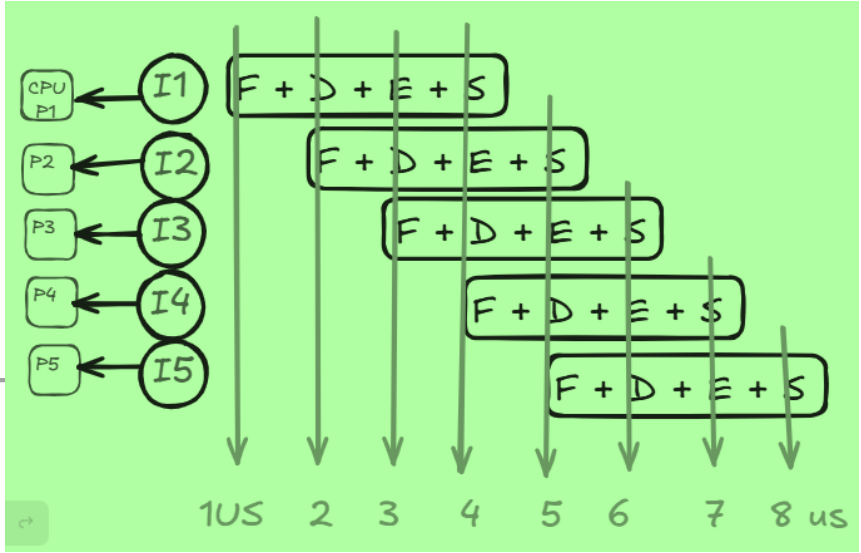
- **Pipeline processing** divides instruction execution into stages.
- Each stage works **simultaneously on different instructions**.
- Similar to an **assembly line**.
- Increases **instruction throughput**, not individual instruction speed.



Pipeline Stages

1. Opcode Fetch (OF)
2. Decode (D)
3. Execute (E)
4. Store (S)

Each stage = **1 µs**



Example (Pipeline Execution of I1–I5)

Pipeline Timing Table

Time (µs)	OF	D	E	S
0–1	I1			
1–2	I2	I1		
2–3	I3	I2	I1	
3–4	I4	I3	I2	I1

Time (µs)	OF	D	E	S
4–5	I5	I4	I3	I2
5–6		I5	I4	I3
6–7			I5	I4
7–8				I5

Total Time	8 µs

Performance Comparison

Method	Total Time for I1–I5
Serial Processing	20 µs
Parallel Processing	4 µs
Pipeline Processing	8 µs

Advantages of Pipeline Processing

- ✓ Better CPU utilization
- ✓ High instruction throughput
- ✓ Faster than serial processing

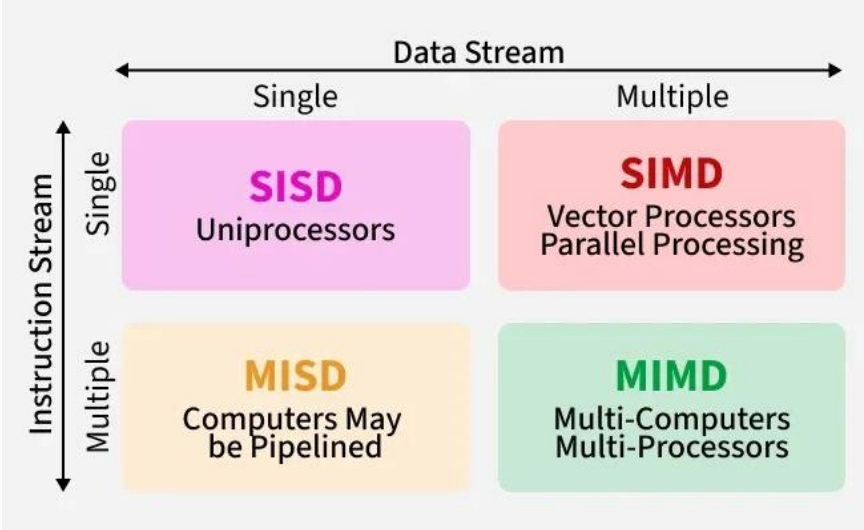
Limitations

- ✗ Pipeline hazards (data, control, structural)
- ✗ Complex control logic

Flynn's Classification of Computers

Flynn's Classification, proposed by Michael J. Flynn in 1966, is a fundamental taxonomy used to categorize computer architectures based on the number of concurrent **Instruction Streams** and **Data Streams**.

As seen in your image, the classification is divided into four main quadrants:



1. SISD (Single Instruction, Single Data)

This represents the traditional **sequential computer** (von Neumann architecture).

- **Mechanism:** A single processor executes one instruction at a time on a single piece of data.
- **Instruction → CPU → Data**
- **Parallelism:** None in the data/instruction flow, though modern versions use **pipelining** to overlap stages of the single instruction stream.
- **Example:** Older single-core CPUs like the Intel Pentium I or II.

Characteristics:

- Simple design
- Low performance
- No parallelism

2. SIMD (Single Instruction, Multiple Data)

This architecture is designed for **Data-Level Parallelism**.

- **Mechanism:** One control unit broadcasts a single instruction to multiple processing elements, which then perform that same operation on different data sets simultaneously.

- **Application:** Ideal for tasks involving large arrays or matrices, such as image processing or scientific simulations.

- **Instruction → Multiple Processing Units → Multiple Data**

- **Example:** Graphics Processing Units (GPUs) and Vector Processors.

  Applications:

  - Image processing
  - Matrix operations
  - Scientific computations

3. MISD (Multiple Instruction, Single Data)

This is a rare and largely **theoretical** category.

- **Mechanism:** Multiple processors perform different instructions on the exact same stream of data.
- **Application:** Primarily used for **fault-tolerant systems** where multiple results for the same data are compared to ensure accuracy.
- **Example:** Space Shuttle flight control systems (redundant computing).

  Characteristics:

  - High reliability
  - Complex design

4. MIMD (Multiple Instruction, Multiple Data)

This is the most flexible and widely used architecture for modern **Parallel Processing**.

- **Mechanism:** Multiple autonomous processors execute different instructions on different data sets independently and often asynchronously.
- **Structure:** Can be **Tightly Coupled** (shared memory) or **Loosely Coupled** (distributed memory/multi-computer systems).
- **Example:** Modern multi-core CPUs (like Ryzen or Intel i9) and supercomputers.

  Example:

  - Multi-core CPUs

- Distributed systems
- Cloud computing

6.3 Arithmetic Pipeline

An **Arithmetic Pipeline** is a pipeline designed to perform **arithmetic operations** by dividing them into sub-operations processed in stages.

An **Arithmetic Pipeline** improves computational speed by dividing complex mathematical tasks (like floating-point addition or multiplication) into smaller, concurrent sub-operations.

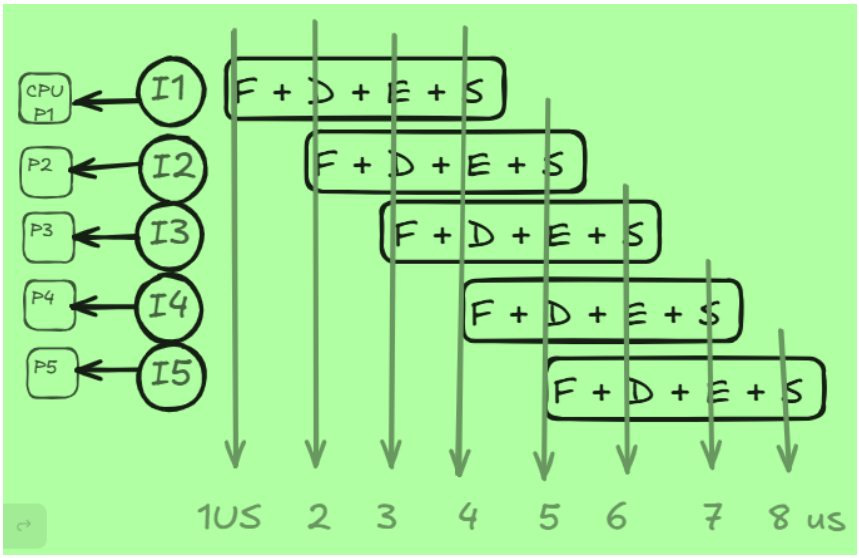
Following the style of your previous instruction examples I1 , I2 …… here is how an arithmetic pipeline handles **Floating-Point Addition**.

Example: Floating-Point Addition Pipeline

Stages

1. Compare exponents
2. Align mantissas
3. Add mantissas
4. Normalize result

Each stage performs one part of the operation.



Example: Floating-Point Addition Pipeline

Suppose we want to add two numbers: $X = 0.9504 \times 10^3$ and $Y = 0.8200 \times 10^2$. This operation is typically divided into **4 stages**:

Stage	Sub-Operation	Description	Process Example
S1	**Compare Exponents**	Subtract exponents to find the difference (k).	$3 - 2 = 1$. The difference is 1.
S2	**Align Mantissa**	Shift the mantissa of the smaller number right by k positions.	Y becomes 0.0820×10^3.
S3	**Add Mantissas**	Add the two aligned mantissas together.	$0.9504 + 0.0820 = 1.0324$.
S4	**Normalize Result**	Shift the result and adjust the exponent for scientific notation.	Result: 0.10324×10^4.

Advantages

- High throughput
- Efficient for repeated arithmetic tasks

Limitations

- Complex control logic
- Not suitable for irregular operations

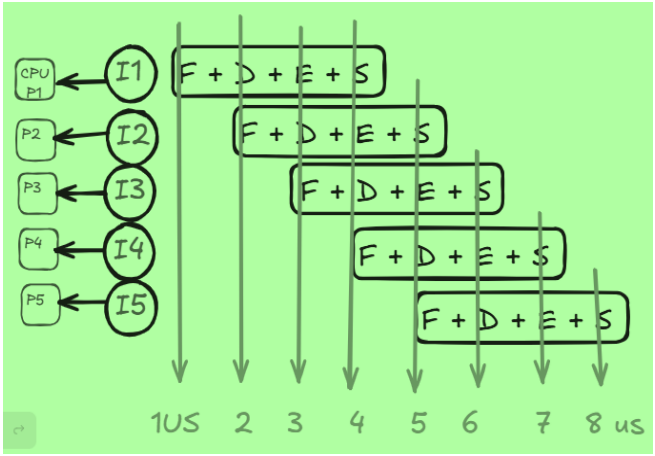
Application Areas

- Scientific computing
- Signal processing
- Weather forecasting

6.4 Instruction Pipeline

Instruction Pipelining improves CPU performance by overlapping the execution of multiple instructions.

Instruction Pipelining is a hardware technique that improves CPU performance by overlapping the execution of multiple instructions. Rather than waiting for one instruction to finish all its steps before starting the next (serial processing), the CPU treats the instruction cycle like an **assembly line**.



How It Works

The instruction cycle is divided into several independent **stages**. Each stage is separated by a **pipeline register** that holds intermediate results for the next clock cycle. This allows the CPU to work on different parts of different instructions simultaneously.

Common 4-Stage Example

Based on your uploaded diagram and general architecture, a standard pipeline might include:

1. **Fetch (F):** Retrieve the instruction from memory.
2. **Decode (D):** Interpret the instruction and identify required operands.
3. **Execute (E):** Perform the actual calculation or operation (e.g., using the ALU).

4. **Store (S):** Write the final result back to memory or a register.

Pipelining Timeline (Example with 5 Instructions)

Assume we have five instructions (I_1 to I_5) and each stage takes **1 microsecond (μs)** to complete.

Time (μs)	Stage 1 (Fetch)	Stage 2 (Decode)	Stage 3 (Execute)	Stage 4 (Store)	Status
1	I_1	—	—	—	I_1 starts
2	I_2	I_1	—	—	Overlap begins
3	I_3	I_2	I_1	—	
4	I_4	I_3	I_2	I_1	I_1 finishes
5	I_5	I_4	I_3	I_2	I_2 finishes
6	—	I_5	I_4	I_3	I_3 finishes
7	—	—	I_5	I_4	I_4 finishes
8	—	—	—	I_5	I_5 finishes

Typical Instruction Pipeline Stages

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute (EX)
4. Memory Access (MEM)
5. Write Back (WB)

Benefits

- Increased instruction throughput
- Better CPU utilization

Limitation

- Performance affected by pipeline hazards

Example

While one instruction is executing, the next is decoded and another is fetched.

6.5 Pipeline Hazards and Their Solutions

Pipeline hazards are conditions that prevent the next instruction from executing in its proper clock cycle.

Types of Pipeline Hazards

1. Structural Hazards

- Occur when hardware resources are insufficient.

Example:

- Single memory used for instruction and data.

Solution:

- Duplicate resources
- Pipeline scheduling

---

2. Data Hazards

- Occur when instructions depend on results of previous instructions.

Types:

- RAW (Read After Write)
- WAR (Write After Read)
- WAW (Write After Write)

Solutions:

- Data forwarding
- Pipeline stalling
- Register renaming

---

3. Control Hazards

- Occur due to branch instructions.

Solutions:

- Branch prediction
- Delayed branching
- Speculative execution