Pipeline

Pipelining is a CPU hardware design technique used to enhance overall performance. In a pipelined processor, operations are divided into stages that are executed in parallel. This allows multiple instructions to be processed simultaneously, each in a different stage of execution.

Instead of completing one instruction at a time, the processor begins a new instruction before the previous one finishes, with each instruction progressing through different stages. This approach enables more efficient instruction handling by overlapping the execution steps.

Pipeline Processor

A **pipeline processor** is a type of CPU architecture that improves processing speed by dividing instruction execution into separate stages. Each stage handles a specific part of the instruction, such as fetching, decoding, executing, memory access, and writing results. While one instruction is being executed, others are moving through the earlier stages, allowing multiple instructions to be processed at the same time.

This is similar to an assembly line in a factory, where different workers (stages) handle different tasks on multiple products (instructions) simultaneously. This overlap increases the overall efficiency and throughput of the processor.

For example, consider how cars are built in a factory:

- One worker installs the engine.
- The next adds the wheels.
- Another paints the car.
- The last one performs final checks.

Similarly, in pipelining, different parts of multiple instructions are processed simultaneously at different stages.

Design of a basic Pipeline

- In a pipelined processor, a pipeline has two ends, the input end and the output end. Between these ends, there are multiple stages/segments such that the output of one stage is connected to the input of the next stage and each stage performs a specific operation.
- Interface registers are used to hold the intermediate output between two stages. These interface registers are also called latch or buffer.
- All the stages in the pipeline along with the interface registers are controlled by a common clock.

It consists of a sequence of m data-processing circuits, called stages or segments, which collectively perform a single operation on a stream of data operands passing through them.

Some processing takes place in each stage, but a final result is obtained only after the entire operand set has passed through the entire pipeline.

Components in the Diagram

The components used in the below diagram are given below **:**

- **Data In**: This is the input data that enters the pipeline.
- **Stages (S1, S2, ..., Sm)**: Each stage performs part of the operation. The pipeline is divided into multiple stages (S1, S2, ..., Sm), and each stage handles a specific operation.
- **Registers (R1, R2, ..., Rm)**: These are pipeline registers. They temporarily hold data between stages to ensure smooth transfer and isolation between operations. Example: R1 stores output of Stage S1 before passing it to S2.
- **Computation Units (C1, C2, ..., Cm)**: These perform the actual processing (like arithmetic or logical operations). Each computation unit corresponds to a specific stage.
- **Control Unit**: Manages the timing and control signals for each stage. Ensures that each stage operates in sync and processes the correct data at the right time.

- **Data Out :** The final output after processing is complete across all pipeline stages.

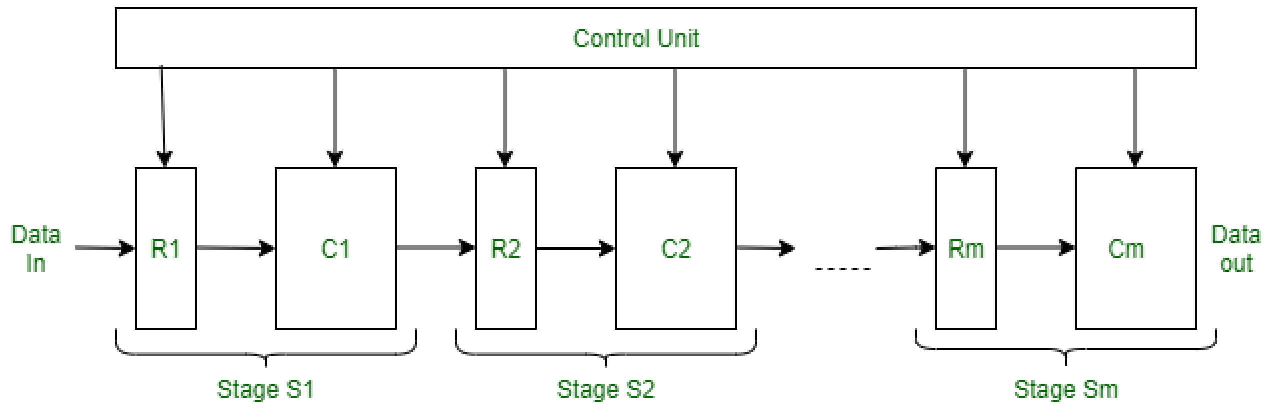


Figure - Structure of a Pipeline Processor

Pipeline Processor

Working

- In **clock cycle 1**, data enters Stage S1 (R1 → C1).
- In **clock cycle 2**, that data moves to Stage S2 (R2 → C2), while new data enters Stage S1.
- This process continues, and each stage is simultaneously processing a different piece of data.
- Eventually, the output appears at the end after passing through all stages.

The working can be explained with the help of the table given below:

Clock Cycle	**Stage S1 (R1→C1)**	**Stage S2 (R2→C2)**	**Stage S3 (R3→C3)**	**...**	**Stage Sm (Rm→Cm)**
1	Data A	-	-	...	...
2	Data B	Data A	-	...	...
3	Data C	Data B	Data A	...	...

Clock Cycle	**Stage S1 (R1→C1)**	**Stage S2 (R2→C2)**	**Stage S3 (R3→C3)**	**…**	**Stage Sm (Rm→Cm)**
4	Data D	Data C	Data B	…	…
5	Data E	Data D	Data C	…	…
…	…	…	…	…	…

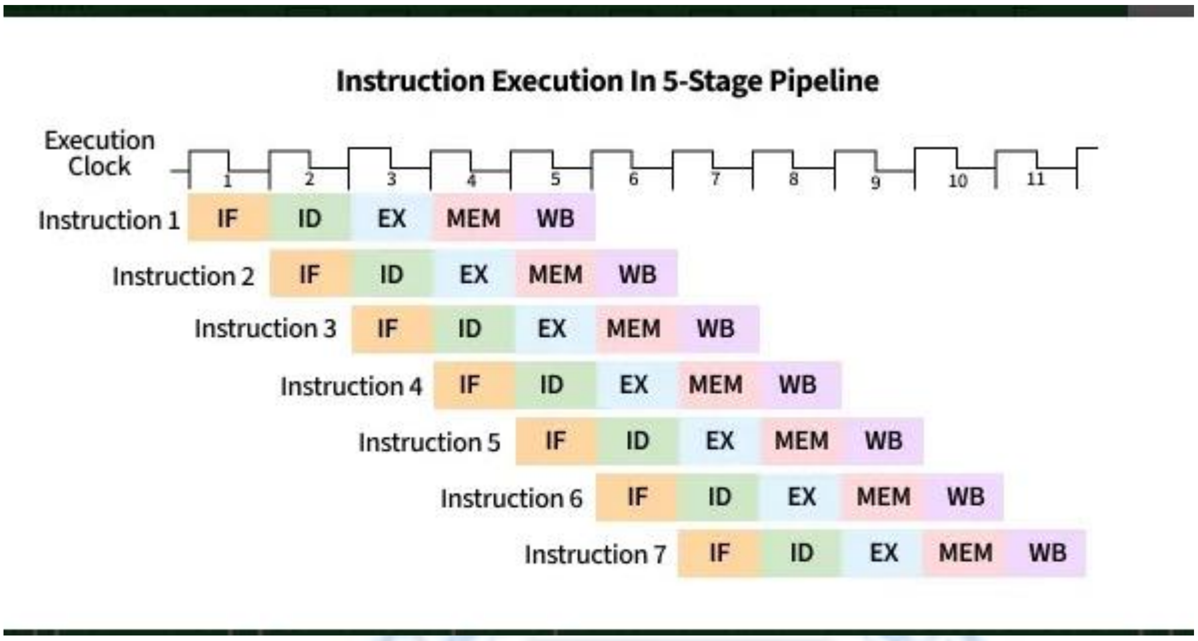
Various Instructions for five stage Pipeline

An instruction in a pipeline refers to a single operation or command that is executed by the processor, which passes through multiple sequential stages (e.g., fetch, decode, execute) as part of a pipelined execution process.

- In a pipelined processor, instructions are divided into smaller steps, and each step is handled by a different part of the CPU simultaneously.
- Each instruction, whether it's a computation, memory access, or branch operation, flows through these stages like an item on an assembly line.
- The purpose of pipelining is to increase instruction throughput that is to complete more instructions in less time by overlapping their execution.

Instructions And Stages For Pipeline

Let us consider following decomposition of instruction execution into five stages. If there are 7 instruction then each instruction passes through each stages and the instruction after another follows the same stage as the previous instruction had passed through which given in the below diagram.


Instruction Execution In 5-Stage Pipeline
Execution Clock
1 2 3 4 5 6 7 8 9 10 11
Instruction 1 IF ID EX MEM WB
Instruction 2 IF ID EX MEM WB
Instruction 3 IF ID EX MEM WB
Instruction 4 IF ID EX MEM WB
Instruction 5 IF ID EX MEM WB
Instruction 6 IF ID EX MEM WB
Instruction 7 IF ID EX MEM WB

5 Stages Pipeline

- Each instruction goes through all 5 stages.
- Instructions are overlapped in the pipeline to improve performance this is the core idea of pipelining.

For the Instruction like ADD T0, T1, T2 add the contents of `T1` and `T2`, and store the result in `T0`

The Stages through which they pass are:

- **Fetch Instruction (IF) :** The processor fetches the instruction from memory usingtheProgramCounter(PC).

  Key actions:
  - PC → Memory
  - Instruction → Instruction Register
  - PC is updated to point to the next instruction.

Fetch `add T0, T1, T2` from address `0x12345`

- **Instruction Decoding (ID) :** The fetched instruction is decoded to understand what it needs to do. The processor also fetches the values of the required registers.

  Key actions:
  - Decode the opcode.

   - Read operands (source registers).
   - Calculate branch address (if needed).

If `T1 = 10` *and* `T2 = 10`*, these values are decoded for the addition.*

- **Execute Instruction (EX) :** Performs the actual computation or calculates the effectivememoryaddress.

  Key actions:

   - ALU operation (e.g., add, subtract).
   - Branch comparison (for conditional jumps).
   - Calculate memory address (for load/store).

ALU computes: 10 `+ 10 = 20`.

- **Memory access/branch completion cycle (MEM) :** Accesses memory if needed(forloadorstoreinstructions).

  Key actions:

   - Read from memory (for load).
   - Write to memory (for store).
   - Otherwise, the instruction just passes through.

No memory access needed, proceed to WB.

- **Write Operand (WO) :** The result of the instruction is written back to the registerfile.

  Key actions:

   - Write result to destination register.
   - Complete the instruction.

Store `15` *in* `T0`.