

PAG U₃

Principles of Linear Pipelining

In **pipelining**, a complex **task** is divided into a sequence of **subtasks** to improve processing efficiency. Each subtask handles a portion of the overall job and can be executed in parallel with others, but under certain constraints.

Precedence Relation

For a set of subtasks $\{T_1, T_2, \dots, T_k\}$ that make up a given task T , a **precedence relation** specifies the execution order. This means a subtask T_j cannot begin until some earlier subtask T_i (where $i < j$) is completed. This dependency ensures the correctness of task execution.

Precedence Graph

The dependencies between all subtasks are visually and logically represented using a **precedence graph**. This graph outlines which subtasks depend on the completion of others, and how the pipeline must handle these dependencies.

Conditions for Pipelined Processing

For a **pipeline** to efficiently process successive subtasks, two main conditions must be satisfied:

- **Subtasks have linear precedence order**: The subtasks should follow a straight, ordered sequence without complex branching or loops.
- **Each subtask takes nearly the same time to complete**: Uniform time across subtasks avoids delays and maximizes pipeline utilization.

Clock Period (r) for the Pipeline

In a linear pipeline, data moves through **stages** connected by **latches**. Each stage includes some **circuitry** and is separated from the next by a latch.

Let:

- t_i = time delay of the **circuitry** in stage S_i
- t_l = time delay of the **latch**

Then, the **clock period** r of the pipeline is the maximum time taken by any stage, including latch delay:

$$r = \max\{t_i\} + t_l$$

This ensures that the clock is slow enough to accommodate the slowest stage in the pipeline.

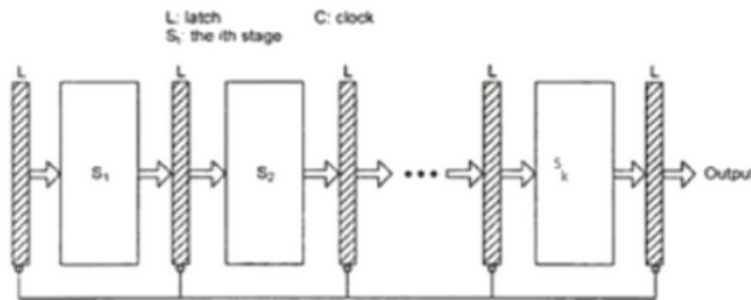
Clock Frequency (f)

The clock frequency f of a pipeline processor is the reciprocal of the clock period r :

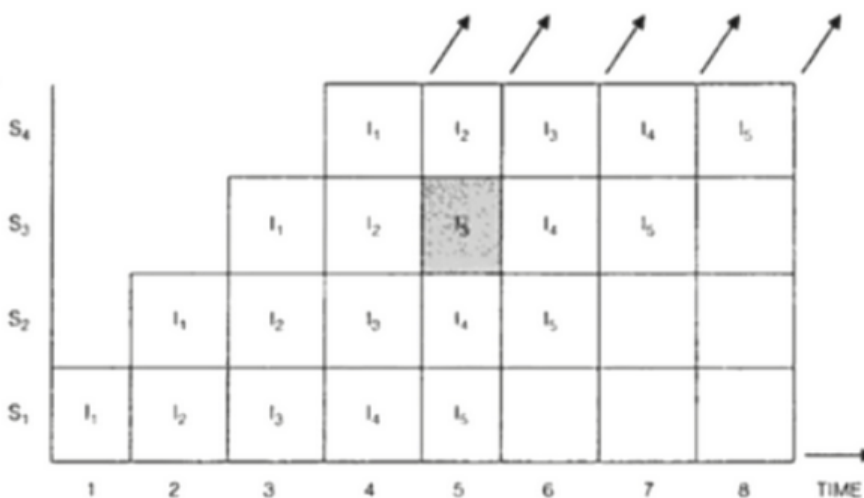
$$f = \frac{1}{r}$$

A higher clock frequency means faster task processing, assuming the pipeline is optimally utilized.

Basic Linear Pipeline



- L : latches, interface between different stages of pipeline
- S_1, S_2 , etc. : pipeline stages
- It consists of cascade of processing stages.
- **Stages : Pure combinational circuits performing arithmetic or logic operations over the data** flowing through the pipe.
- Stages are separated by **high speed interface latches**.
- **Latches** : Fast Registers holding intermediate results between stages
- **Information Flow** are under the **control of common clock** applied to all latches



Floating Point Addition with Aligning and Normalizing

Case I: Right Shift Needed ($u = -1$)

Step 1: Given Values

We are given two numbers:

- $A = 4.5 \times 10^2$
- $B = 7.2 \times 10^0$

Step 2: Aligning the Exponents

- Exponent of A: 2
- Exponent of B: 0

To align B with A's exponent of 2, we need to **increase the exponent of B** by 2:

$$B = 7.2 \times 10^0 = 0.072 \times 10^2$$

Now, both A and B have the same exponent of 10^2 , and we can proceed to add their mantissas.

Step 3: Add the Mantissas

Now that both numbers have the same exponent, we can add the mantissas:

$$\begin{aligned} A &= 4.5, & B &= 0.072 \\ A + B &= 4.5 + 0.072 = 4.572 \end{aligned}$$

So the result is:

$$\text{Result} = 4.572 \times 10^2$$

Step 4: Normalize the Result

We need to normalize the result. Since 4.572 is too large for the normalized form (we want the mantissa in the range 0.xxx), we **right shift** by 1 place :

$$4.572 \times 10^2 = 0.4572 \times 10^2$$

So, we have the normalized form:

$$d = 0.4572 \times 10^2$$

Step 5: Adjust the Exponent

The original exponent was $r = 2$, and we performed a **right shift** by 1, so the new exponent is:

$$s = r - (-1) = 2 - (-1) = 3$$

Thus, the final result is:

$$0.4572 \times 10^3$$

Case 2: Left Shift Needed ($u = 2$)

Step 1: Given Values

We are given two numbers:

- $A = 6.3 \times 10^2$
- $B = 7.5 \times 10^2$

Step 2: Add the Mantissas

Since both numbers already have the same exponent of 10^2 , we can directly add their mantissas:

$$\begin{aligned} A &= 6.3, \quad B = 7.5 \\ A + B &= 6.3 + 7.5 = 13.8 \end{aligned}$$

So the result is:

$$\text{Result} = 13.8 \times 10^2$$

Step 3: Normalize the Result

We need to normalize the result. Since 13.8 is too large for the normalized form (we want the mantissa in the range $0.xxx$), we **left shift** by 1 place:

$$13.8 \times 10^2 = 1.38 \times 10^3$$

So, we have the normalized form:

$$d = 1.38 \times 10^3$$

Step 4: Adjust the Exponent

The original exponent was $r = 2$, and we performed a **left shift** by 1, so the new exponent is:

$$s = r - 1 = 2 - 1 = 3$$

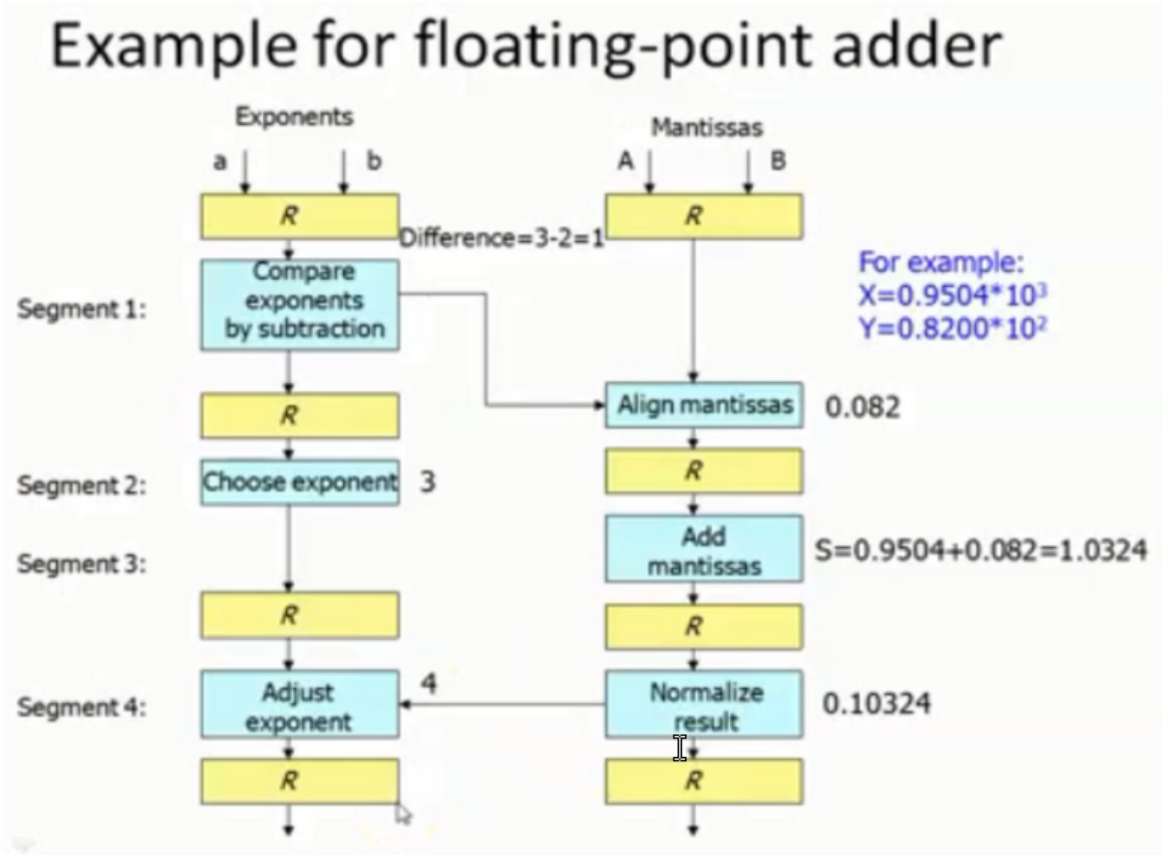
Thus, the final result is:

$$1.38 \times 10^3$$

Summary

Situation	Raw c	Shift	u	Final Form
Too large	$c \geq 1.0$	Right	< 0	$0.xxx \times 10^s$

Situation	Raw c	Shift	u	Final Form
Too small	$c < 1.0$	Left	> 0	$0.xxx \times 10^s$



Pipelining: Key Metrics and Comparison Table

Notations

- CT = Clock Time or Clock Period
- T_{wp} = Total time without pipelining
- T_p = Total time with pipelining
- n = Number of tasks
- k = Number of pipeline stages
- η = Efficiency or Utilization
- CPI = Cycles Per Instruction
- Throughput = Number of tasks completed per unit time
- Latency = Time taken for a single task to pass through the system
- Speedup = Ratio of performance improvement when using pipelining

Comparison Table

Metric	Without Pipelining	With Pipelining	Common (Both Cases)
Clock Time (CT)	$\sum$ of all delays	$\max\{S_i\} + t_l$ (longest stage + buffer)	$\frac{1}{x \text{ GHz}}$ ns
Total Time	$T_{wp} = n \times CT$	$T_p = (k + n - 1) \times CT$	—
Throughput	$\frac{1}{CT}$	$\frac{1}{CT}$	—
Latency	CT	$k \times CT$	—
Efficiency / Utilization	—	—	$\eta = \frac{n}{k+n-1}$
Speedup	—	—	$\frac{T_{wp}}{T_p}$ or $\eta \times k$
Performance	—	—	$CT \times CPI$

Types of Pipelining Based on Levels of Processing

This classification is based on **what kind of operations** are being pipelined. The main types are:

I. Arithmetic Pipelining

Description:

- Used to **speed up arithmetic operations**, especially complex ones like floating-point addition or multiplication.
- The arithmetic operation is divided into multiple **stages**, and each stage is executed in parallel with others for different inputs.

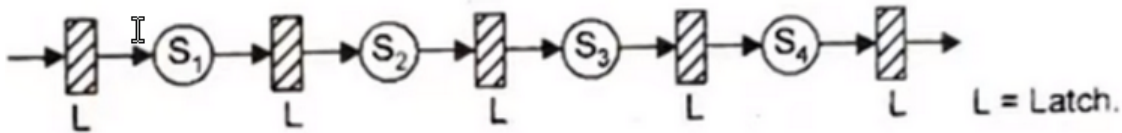
Typical Stages:

1. Operand Fetch
2. Alignment (for floating point)
3. Operation (Addition/Multiplication)
4. Normalization
5. Rounding

Example:

Floating-point addition:

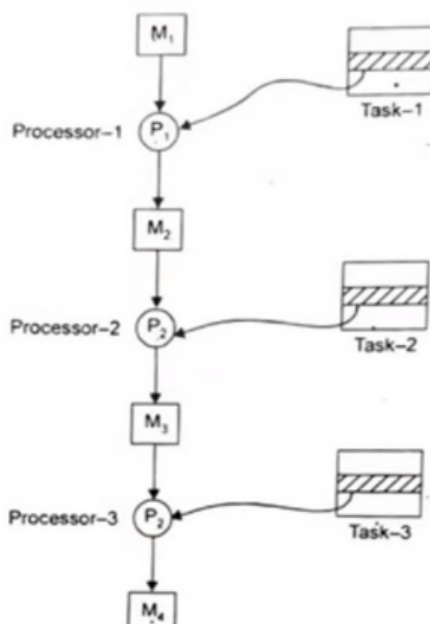
- Align exponents
- Add mantissas
- Normalize result
- Round result



2. Processor Pipelining

Description:

- In processor pipeline processing, the same data stream is processed by a cascade of processors. Each processor performs a specific task. The data stream passes the first processor with results stored in a memory block which is also accessible by second processor.
- The second processor processes this result and passes it to third and so on.
- This pipeline processor is not much popular. There is no practical example found for processor pipeline.



Goal:

- **Maximize hardware utilization** by ensuring all processor components are busy.

3. Instruction Pipelining

Description:

- Most common type in modern CPUs.
- Each instruction is divided into **sequential stages**, allowing multiple instructions to be processed simultaneously.

Common Stages:

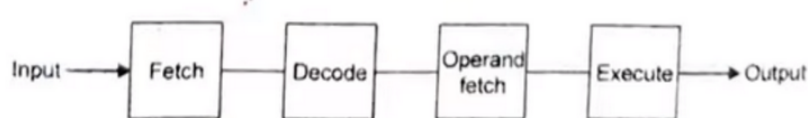
1. **IF** – Instruction Fetch
2. **ID** – Instruction Decode and operand fetch
3. **EX** – Execute

4. MEM – Memory Access

5. WB – Write Back

Benefit:

- After the pipeline is filled, the CPU can **complete one instruction per clock cycle**.



Step	1	2	3	4	5	6	7	8	9
1	FI	DA	FO	EX					
2		FI	DA	FO	EX				
3			FI	DA	FO	EX			
4				FI	DA	FO	EX		
5					FI	DA	FO	EX	

Summary Table

Type	Focus	Example Operations
Arithmetic Pipelining	Speeding up arithmetic computations	Floating-point addition, multiplication
Processor Pipelining	Overlapping different functional units	Memory access while ALU is in use
Instruction Pipelining	Speeding up instruction execution	IF → ID → EX → MEM → WB stages

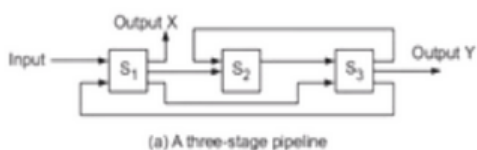
Pipelining based on Pipeline configuration

1. Unifunctional Pipeline

- Definition:** This pipeline is designed to handle **only one specific type of operation** at any given time.
- Key Feature:** All instructions passing through the pipeline perform the **same function** (e.g., only addition or only multiplication).
- Use Case:** Suitable for **dedicated hardware** like a floating-point unit (FPU) designed only for addition.
- Limitation:** Not flexible — can't switch functions without redesigning hardware.

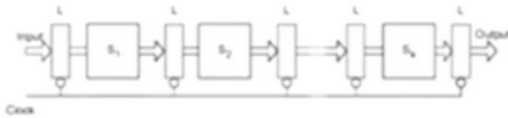
2. Multifunctional Pipeline

- Definition:** Capable of executing **different functions**, such as addition, multiplication, etc., but **only one function at a time**.
- Key Feature:** Function can be selected based on the instruction, but the pipeline stages are **not shared in parallel for different functions**.
- Use Case:** Found in **general-purpose ALUs**, where operations can vary but not concurrently.
- Advantage:** More versatile than unifunction.
- Limitation:** Can't perform multiple types of operations **concurrently**.



3. Static Pipeline

- **Definition:** Pipeline configuration is **fixed at design time**. Each stage has a **predefined role** that does not change.
- **Key Feature:** Simplified control and easier implementation.
- **Use Case:** Common in **RISC architectures**, where the instruction format and execution flow are regular and predictable.
- **Limitation:** **Less adaptable** to different types of operations or workloads.



Example:

Instruction: `ADD R1, R2, R3`

In a **5-stage static pipeline**:

1. **IF** – Fetch instruction
2. **ID** – Decode and register read
3. **EX** – Execute addition
4. **MEM** – No memory used, but stage is traversed
5. **WB** – Write result back to R1

Even if `ADD` doesn't need memory access, it still **goes through the MEM stage**, making it less efficient.

4. Dynamic Pipeline

- **Definition:** Pipeline can **adapt dynamically** to the instructions being executed.
- **Key Feature:** Instructions may enter and move through stages **based on resource availability**, not fixed sequence.
- **Use Case:** Used in **superscalar** and **out-of-order** execution processors.
- **Advantage:** Better utilization of resources and **higher throughput**.
- **Complexity:** Requires sophisticated control logic and hazard management.

Example:

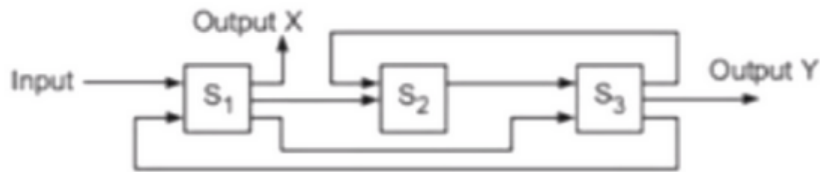
Two instructions:

1. `MUL R1, R2, R3` (3 cycles, uses multiplier)
2. `LOAD R4, 0(R5)` (uses memory)

In a **dynamic pipeline**:

- `MUL` occupies the multiplier for 3 cycles.
- `LOAD` can **proceed independently** to the memory unit.

- Instructions **do not wait unnecessarily** and **can be reordered**.



Pipelining Based on Types of Instruction and Data

1. Scalar Pipelines:

This type of pipeline processes scalar operands of repeated scalar instructions (i.e. processes scalar operands Under the control DO LOOP).

Eg: IBM-360

2. Vector Pipelines:

This type of pipeline processes vector instructions over vector operands.

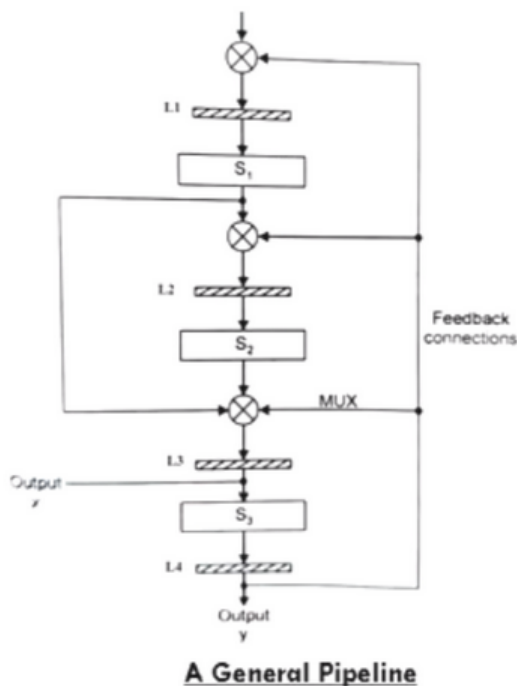
Eg: STAR-100, Cray-I

General Pipeline

In general pipeline, the pipeline may have feed-back connections and feed-forward connections along with streamline (cascade) connections.

So it is nonlinear.

A general pipeline may have multiple paths, parallel usage of multiple stages and non-linear flow of data



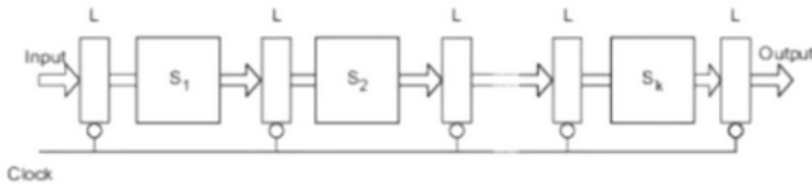
Example: A General Pipeline With 3-stages

- It IS a two function pipeline With function x and function y.
- There are three stages S1, S2, S3.

- The crossed circles are multiplexers (MUX). They are used for selecting multiple connection paths In evaluating different functions.
- The elements L1 to L4 are latches which helps to lust store and forward the Input that they receive. These latches are also used as delays to synchronize the movement of data from one stage of pipeline to another stage and also they help avoid any Intermediate data loss.

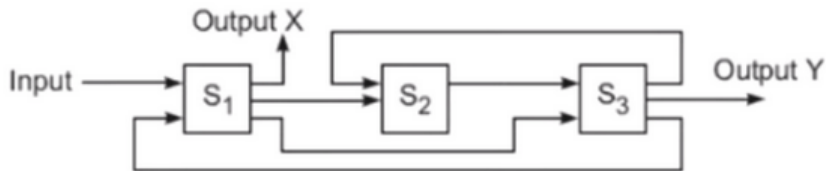
Linear Pipeline

It has streamline (cascade) connections only



Non-linear pipeline

It has feed-back connections and feed-forward connections along with streamline connections.



Reservation table Numericals