• A **pipeline** is a set of data processing elements connected in series, so that the output of one element is the input of the next one.

• The pipeline organization can be demonstrated by this simple example:

  - **Multiply and Add Operation**: \( Ai \times Bi + Ci \)
  - **3 Suboperation Segment**
    1. \( R1 \leftarrow Ai, R2 \leftarrow Bi \) : Input \( Ai \) and \( Bi \)
    2. \( R3 \leftarrow R1 \times R2, R4 \leftarrow Ci \) : Multiply and input \( Ci \)
    3. \( R5 \leftarrow R3 + R4 \) : Add \( Ci \)
Each clock produces new output and moves the data one step down the pipeline. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.
Pipeline Logic:

- A clock drives all the registers in the pipeline. This clock causes the CLC output to be latched in the register which provides input to the next stage, and thus making a start of new computation possible for next stage.
- The maximum clock rate is decided by the time delay of the CLC in the stage and the delay of the staging latch.
Each segment consists of CLC (Si) that performs a suboperation over the data stream flowing through the pipe.

The segments are separated by registers (Ri) that hold the intermediate results between stages.
Speedup $S$ : Nonpipeline / Pipeline

- $S = n \cdot t_n / (k + n - 1) \cdot t_p$
  - $n$ : task number (6)
  - $t_n$ : time to complete each task in nonpipeline
  - $t_p$ : clock cycle time (1 clock cycle)
  - $k$ : segment number (4)

\[
\begin{array}{cccccccccc}
\text{Segment} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
1 & T_1 & T_2 & T_3 & T_4 & T_5 & T_6 & & & \\
2 & T_1 & T_2 & T_3 & T_4 & T_5 & T_6 & & & \\
3 & T_1 & T_2 & T_3 & T_4 & T_5 & T_6 & & & \\
4 & T_1 & T_2 & T_3 & T_4 & T_5 & T_6 & & & \\
\end{array}
\]

- If $n \to \infty$, $S = t_n / t_p$, $k + n - 1 \approx n$
- nonpipeline ($t_n$) = pipeline ($k \cdot t_p$)

\[
S = t_n / t_p = k \cdot t_p / t_p = k
\]
Instruction Pipelining

- Instruction execution is extremely complex and involves several operations which are executed successively.

### The Instruction Cycle

- Fetch instruction
- Decode
- Fetch operand
- Execute instruction

- Calculate operand address (CO)
- Fetch operand (FO)
- Execute instruction (EI)
- Write back operand (WO)

- Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. This is solved without additional hardware but only by letting different parts of the hardware work for different instructions at the same time.
The pipeline organization of a CPU is similar to an assembly line: the work to be done in an instruction is broken into smaller steps (pieces), each of which takes a fraction of the time needed to complete the entire instruction. Each of these steps is a pipe stage (or a pipe segment).

Pipe stages are connected to form a pipe:

Stage 1 → Stage 2 → ... → Stage n

The time required for moving an instruction from one stage to the next: a machine cycle (often this is one clock cycle). The execution of one instruction takes several machine cycles as it passes through the pipeline.

F - Fetch ML Instruction
R - Read Source Registers
A - ALU Operation
M - Memory Access
Acceleration by Pipelining

Two stage pipeline:

- FI: fetch instruction
- EI: execute instruction

Clock cycle → 1 2 3 4 5 6 7 8

Instr. i           FI   EI
Instr. i+1         FI   EI
Instr. i+2         FI   EI
Instr. i+3         FI   EI
Instr. i+4         FI   EI
Instr. i+5         FI   EI
Instr. i+6

We consider that each instruction takes execution time $T_{ex}$.

Execution time for the 7 instructions, with pipelining:

$\frac{T_{ex}}{2} \times 8 = 4 \times T_{ex}$
Four-segment CPU pipeline:

» 1) FI : Instruction Fetch
» 2) DA : Decode Instruction & calculate EA
» 3) FO : Operand Fetch
» 4) EX : Execution
Six stage pipeline

- FI: fetch instruction
- DI: decode instruction
- CO: calculate operand address
- FO: fetch operand
- EI: execute instruction
- WO: write operand

Clock cycle → 1 2 3 4 5 6 7 8 9 10 11 12

Instr. i  
FI DI CO FO EI WO

Instr. i+1  
FI DI CO FO EI WO

Instr. i+2  
FI DI CO FO EI WO

Instr. i+3  
FI DI CO FO EI WO

Instr. i+4  
FI DI CO FO EI WO

Instr. i+5  
FI DI CO FO EI WO

Instr. i+6  
FI DI CO FO EI WO

Execution time for the 7 instructions, with pipelining:

$$(T_{ex}/6) \times 12 = 2 \times T_{ex}$$
• After a certain time (N-1 cycles) all the N stages of the pipeline are working: the pipeline is filled. Now, theoretically, the pipeline works providing maximal parallelism (N instructions are active simultaneously).

• Apparently a greater number of stages always provides better performance. However:
  - a greater number of stages increases the overhead in moving information between stages and synchronization between stages.
  - with the number of stages the complexity of the CPU grows.
  - it is difficult to keep a large pipeline at maximum rate because of pipeline hazards.

**80486 and Pentium:** five-stage pipeline for integer instr.

**PowerPC:**
- eight-stage pipeline for FP instr.
- four-stage pipeline for integer instr.
- six-stage pipeline for FP instr.
Pipeline and Parallel Processors:

Example: \[ R = \frac{(X \times Y) + Z}{M} \]

A vector machine program would be like that:

- LOAD X
- MULT Y
- ADD Z
- DIV M
- STR R

(Pipeline as a plow with jagged saw)
Steady-state Analysis of Pipelines:

Let \( n \) = # of stages.

\( m \) = # of tasks run on the pipeline

- **Efficiency (e):** is the ratio of the energy used in doing the work and the total energy supplied.

\[
    e = \frac{m \cdot n}{(m+n-1) \cdot n} = \frac{m}{m+n-1}
\]

When \( n \gg m \), then \( e \) tends to be \( m/n \).

When \( m \gg n \), then \( e \) tends to be \( 1 \).

When \( n=m \), then \( e \) is \( 0.5 \).
Steady-state Analysis of Pipelines:

Let $n =$ # of stages.
$m =$ # of tasks run on the pipeline

**Speedup ($S$):** is the ratio of the time taken by the nonpipelined architecture to the time taken by the pipeline.

$$S = \frac{(n.t).m}{(m+n-1).t} = \frac{m.n}{(m+n-1)}$$

When $n >> m$, then $S$ tends to be $m$.
When $m >> n$, then $S$ tends to be $n$.
When $n = m$, then $S$ is $n/m$
Steady-state Analysis of Pipelines:
Let \( n \) = # of stages.
\( m \) = # of tasks run on the pipeline

**Throughput (Q):** is defined as the number of tasks completed per unit amount of time.

\[
Q = \frac{m}{(m+n-1).t} = \frac{e}{t}
\]

When \( n \gg m \), then \( Q \) tends to be \( m/(n.t) \)
When \( m \gg n \), then \( Q \) tends to be \( 1/t \).
When \( n=m \), then \( Q \) is \( 0.5/t \)
Processor Design: Pipeline
Summary

- Instructions are executed by the CPU as a sequence of steps. Instruction execution can be substantially accelerated by *instruction pipelining*.
- A pipeline is organized as a succession of N stages. At a certain moment N instructions can be active inside the pipeline.
- Keeping a pipeline at its maximal rate is prevented by *pipeline hazards*. *Structural hazards* are due to resource conflicts. *Data hazards* are produced by data dependencies between instructions. *Control hazards* are produced as consequence of branch instructions.
- With conditional branch we have a penalty even if the branch has *not* been taken. This is because we have to wait until the branch condition is available.
- Branch instructions represent a major problem in assuring an optimal flow through the pipeline. Several approaches have been taken for reducing branch penalties.