Pipelining

PIPELINING AND SPEEDUP OF A PIPELINE PROCESSING

PROFESSOR SAMIR ABOU EL-SEOUD

Introduction

- An **instruction pipeline** is a technique used in the design of computers to increase their instruction **throughput** (the number of instructions that can be executed in a unit of time). Pipelining does not reduce the time to complete an instruction, but increases the number of instructions that can be processed at once
- Each instruction is split into a sequence of dependent steps. The first step is always to fetch the instruction from memory; the final step is usually writing the results of the instruction to processor registers or to memory. Pipelining seeks to let the processor work on as many instructions as there are dependent steps, just as an **assembly line** builds many vehicles at once, rather than waiting until one vehicle has passed through the line before admitting the next one

The speedup of a pipeline processing

- Let's assume we have n tasks to be performed. If a non-pipeline unite performs the same operation as a pipeline unit and it takes a time equal to $\mathbf{t_n}$ to complete each task; hence, the time required to perform the n tasks will be $\mathbf{nt_n}$. Here $\mathbf{t_n}$ denotes the amount of time needed to process one piece of data using a non-pipelined arithmetic unit, which is the clock period of this unit. Now consider the case where a **k- segment** (**k-stage**) pipeline with a clock cycle time $\mathbf{t_p}$ is used to execute the same **n** tasks. Here $\mathbf{t_p}$ is the clock period of the **k-stage** pipeline. (Think of the no-pipelined unit as a one-stage pipeline, hence the notation $\mathbf{t_n}$ for its clock period.) If the stages have different minimum clock periods, or stage delays, $\mathbf{t_p}$ is the largest of these periods. The first task **T1** requires a time equal to $\mathbf{kt_p}$ to complete its operation since there are **k** segments in the pipe. The remaining **n-1** tasks emerge from the pipe at the rate of one task per clock cycle and will be completed after a time equal to $\mathbf{(n-1)\ t_p}$. Therefore, to complete **n** tasks using a **k-segment** pipeline requires **k+(n-1).**

Hence, the total time required by the pipeline unit to perform the **n** tasks will be **(k+n-1) t_p**. In other words, the pipelined unit requires **k** time unites, each of duration **t_p** to move the first piece of data through the pipeline. Because additional data enters the pipeline during every cycle, it will output the remaining **n-1** results during the next **n-1** cycles, one per cycle. Thus, the pipelined unit requires **(k+n-1)** cycles, each of **t_p** time, to calculate the same **n** results. Hence, the total time required by a pipelined unit to perform the **n** tasks will be **(k+n-1)t_p** The speedup of a pipeline processing over an equivalent non-pipeline processing may be defined by the ratio:

$$S_n = nt_n/(k+n-1)\ t_p$$

The **steady-state speedup** is the upper limit of the speedup that a pipeline can achieve. Looking at this, it seems that as $\mathbf{n \rightarrow \infty}$ then $\mathbf{S \rightarrow t_n/t_p}$ meaning that the bigger the program is, the more efficient it is (because no matter how fast the pipeline is, it starts empty and ends with a single instruction-the final one-inside. In other words, the starting and ending conditions are less efficient.

So a measure of efficiency, on the other hand, must take account of these start and end conditions. Efficiency is the total number of instructions divided by the pipelined operating time

$$E = n/(n+k-1)$$

But does this not look similar to the speed-up equation? **Yes**! $\mathbf{E = S/(t_n/t_p)}$ and this also the same as **throughput**, which is the number of instructions completed per unit time.

Consider the following snippet of code:

```
FOR i = 1 TO 100 DO
 {
  A[i] ←(B[i].C[i]) + D[i]
 }
```

Assume that each operation, multiplication and addition, requires 10 ns to complete. A non-pipelined uniprocessor takes 20 ns to calculate A[i], and 2,000 ns to execute the code (excluding, the time needed to fetch the operands). A pipelined unit could break this computation into two stages as shown in the figure below. The first stage performs the multiplication and the second stage performs the addition. Note that latches store the output of each stage in the pipeline. Ignore the time delay of the latches for the moment. During the first 10 ns, the first stage calculates B[1].C[1]. In the next 10 ns, stage 2 adds this value to D[1] and stores the result in A[1]. At the same time, stage 1 multiplies B{2] and C[2]. During the following 10 ns, stage 1 forms B[3].C[3] and stage 2 calculates the final value of A[2]. Instead of 2,000 ns, this pipeline executes the code in 1,010 ns.

Example

- The pipeline organization will be demonstrated by means of a simple example. Suppose we want to perform the combined multiply and add operations with a stream of numbers:

$$A_i * B_i + C_i \quad \text{for } i=1, 2, 3, \ldots,7$$

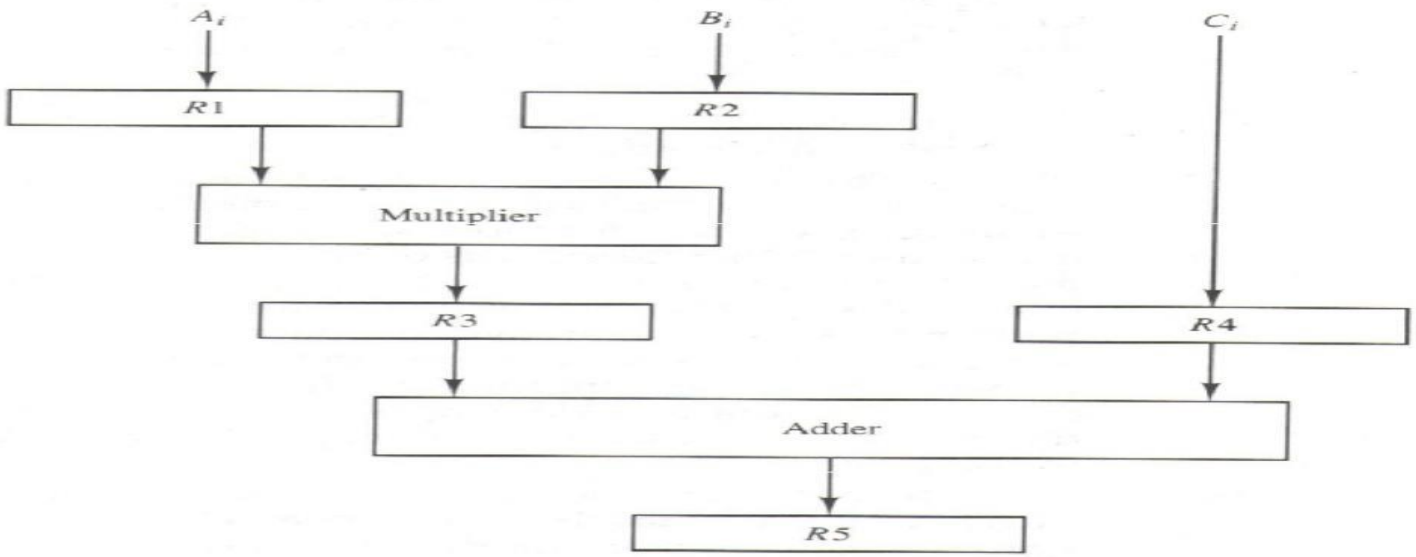
$$A_i * B_i + C_i \qquad \text{for } i = 1, 2, 3, \ldots, 7$$

Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig. below $R1$ through $R5$ are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The suboperations performed in each segment of the pipeline are as follows:

$R1 \leftarrow A_i, \quad R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in table below The first clock pulse transfers A_1 and B_1 into $R1$ and

Figure Example of pipeline processing.



Pipeline's speedup

- As we defined before, a pipeline's speedup, $\mathbf{S_n}$, is the amount of time needed to process **n** pieces of data using a non-pipelined arithmetic unit, divided by the time needed to process the same data using **k-stage** pipeline.

Speedup

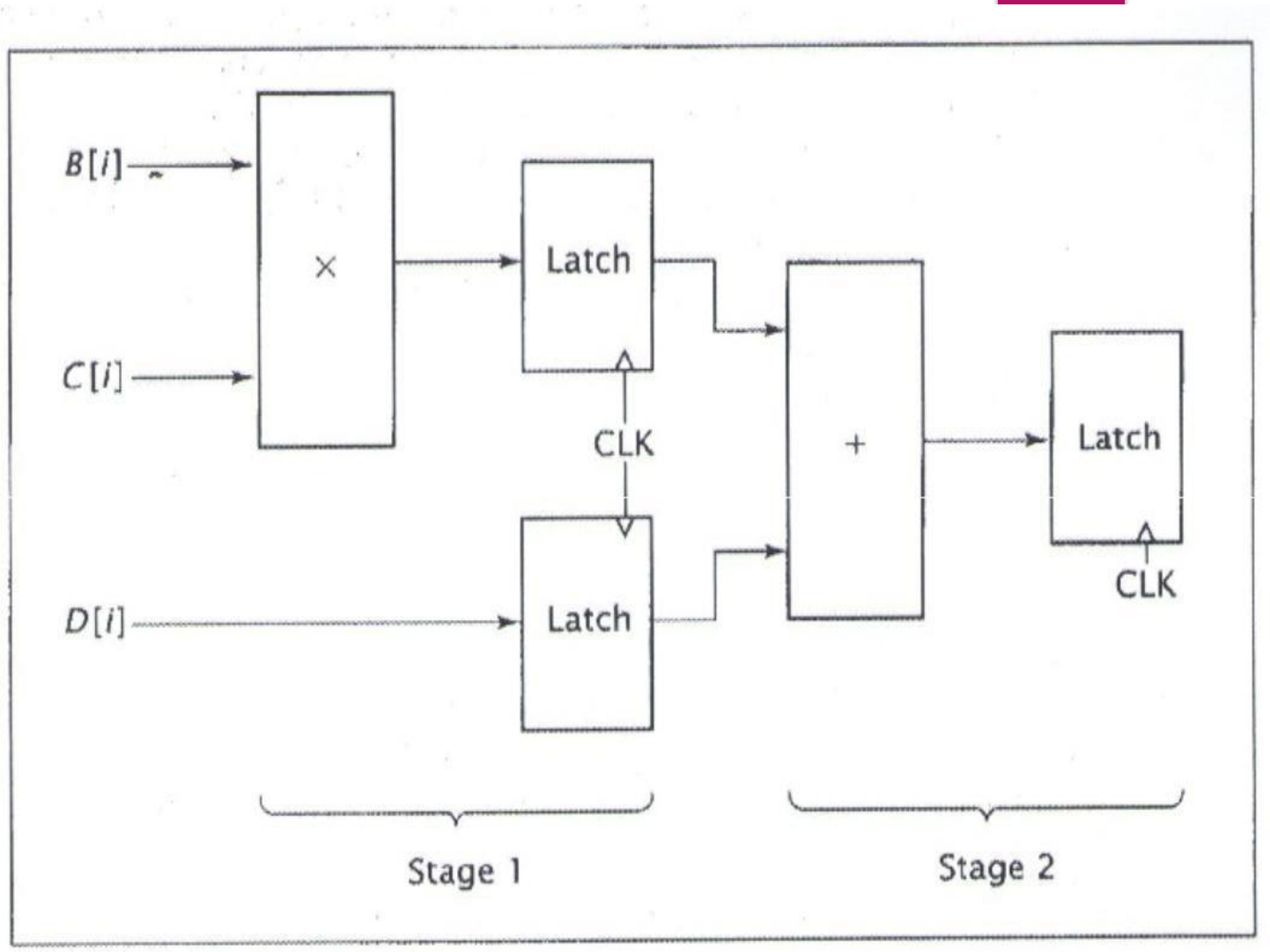
In our example, since a non-pipeline unit requires tn time to calculate one result, it requires ntn time to calculate n results. On the other hand, the pipelined unit requires k time unit, each of duration tp, to move the first piece of data through the pipeline. Because additional data enters the pipeline during every cycle, it will output the remaining (n-1) results during the next (n-1) cycles, one per cycle. Thus, the pipelined unit requires (k+n-1) cycles, each of time tp, to calculate the same n results. Now with

n=100, tn = 20 ns, k= 2, tp = 10 ns

The speedup can be calculated to be

$$S_n = n\, t_n/(k+n-1)t_p$$

= 100 . 20 ns / (2+100 -1). 100 ns = 1.98



The steady-state speedup of Pipeline

- The steady-state speedup is the upper limit of the speedup that a pipeline can achieve. As we mentioned above, as n approaches infinity, the steady-state speedup is given by: $\mathbf{S_\infty = t_n/t_p}$
- In reality, the latches require some amount of time to store their values. This is part of the overhead introduced by pipelines.
- If each latch needs 2 ns to load data, then each stage requires 12 ns, 10 to calculate its value and 2 to store its result. The pipeline would now require 1,212 ns to calculate A[1..100], which is still faster than the non-pipelined case. The actual speedup is

$$\mathbf{S_{100} = nt_n/(k+n-1)t_p = [100x20\ ns]/[(2+100-1)x12\ ns] = 1.65}$$

- However, if only one value, A[1], is calculated, the speedup is less than one:

$$\mathbf{S_1 = [1x20\ ns]/[(2+1-1)x12\ ns] = 0.83}$$

- The pipeline is actually slower than the non-pipelined arithmetic unit, due to the delays of the latches at the end of each stage. This is basic pipelining. Pipelining is also used to speed up the fetching, decoding, and execution of instructions in CPUs.

Execution in a pipelined processor

Execution sequence of instructions in a pipelined processor can be visualized using a space-time diagram. For example, consider a processor having 4 stages and let there be 2 instructions to be executed. We can visualize the execution sequence through the following space-time diagrams:

Non overlapped execution:

Stage\Cycle	1	2	3	4	5	6	7	8
S1	I_1				I_2			
S2		I_1				I_2		
S3			I_1				I_2	
S4				I_1				I_2

Total time = 8 cycles

Overlapped execution:

Stage\Cycle	1	2	3	4	5
S1	I_1	I_2			
S2		I_1	I_2		
S3			I_1	I_2	
S4				I_1	I_2

Total time = 5 cycles

Here below is an illustration of the execution of 2 instruction using non-pipeline and pipeline processor. Each instruction cycle is subdivided into a sequence of subcycles or phases, namely:

- 1. Fetch the instruction from memory
- 2. Decode the instruction
- 3. Fetch operand
- 4. Execute the instruction

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction.

A non-pipelined machine would grab and process one instruction, then wait for at instruction to complete before starting on the next one. We use something called a *ervation table* to visualise this:

etch instruction	$Inst_1$				$Inst_2$				$Inst_3$
ecode instruction		$Inst_1$				$Inst_2$			
etch operand			$Inst_1$				$Inst_2$		
xecute instruction				$Inst_1$				$Inst_2$	
lock cycles	1	2	3	4	5	6	7	8	9

The different functional units for handling an instruction are listed on the left de of the table and the clock cycles are shown along the bottom. Inside the table we dicate what is happening in that cycle. The table shown covers nine successive clock cles.

$Inst_1$ is fetched in the first cycle, then decoded, then its operand fetched and nally the function encoded in that instruction is executed. $Inst_2$ then begins its urney.

But think of this reservation table in a different way: if we consider the rows as eing resources and the columns as time slots, it is clear that each resource spends a t of time slots doing nothing. It would be far more efficient if we allowed instruc- ons to overlap, so that resources spend more of the time doing something. Let us try out:

etch instruction	$Inst_1$	$Inst_2$	$Inst_3$	$Inst_4$	$Inst_5$	$Inst_6$	$Inst_7$	$Inst_8$	$Inst_9$
ecode instruction		$Inst_1$	$Inst_2$	$Inst_3$	$Inst_4$	$Inst_5$	$Inst_6$	$Inst_7$	$Inst_8$
etch operand			$Inst_1$	$Inst_2$	$Inst_3$	$Inst_4$	$Inst_5$	$Inst_6$	$Inst_7$
xecute instruction				$Inst_1$	$Inst_2$	$Inst_3$	$Inst_4$	$Inst_5$	$Inst_6$
lock cycles	1	2	3	4	5	6	7	8	9

The most obvious effect is that instead of getting to the start of $Inst_3$ in the nine clock ycles, the overlapping now covers nine instructions: it processes three times faster. I-

Example:

The pipeline organization will be demonstrated by means of a simple example. Suppose we want to perform the combined multiply and add operations with a stream of numbers

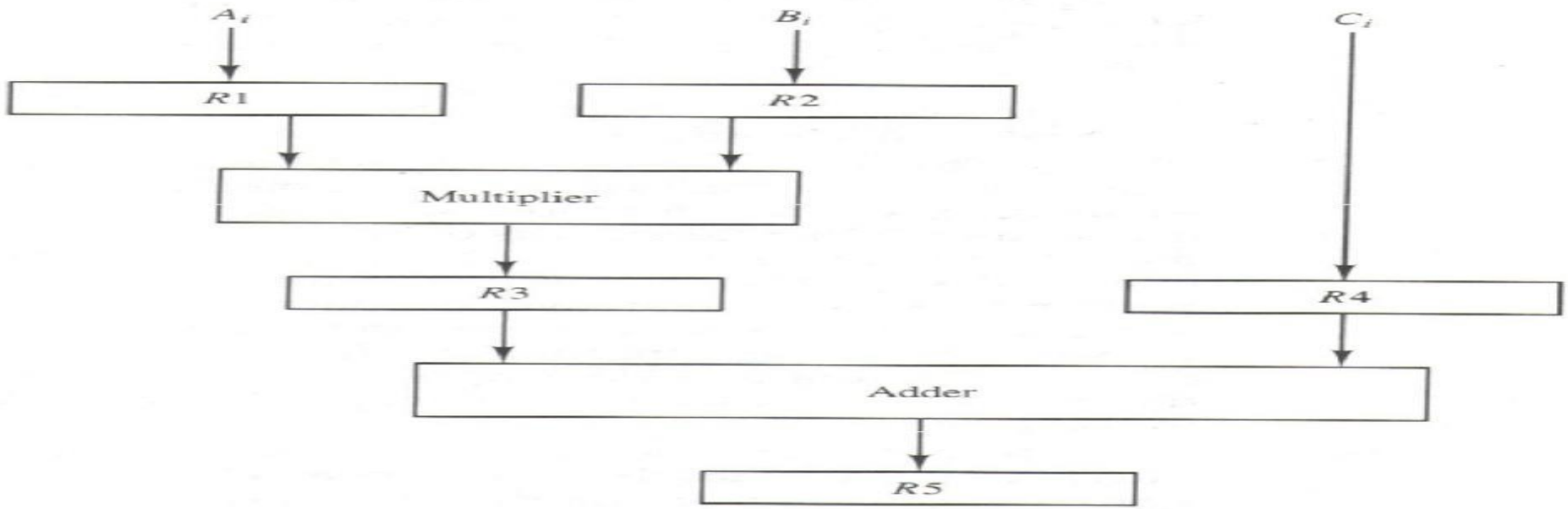
$$A_i * B_i + C_i \qquad \text{for } i = 1, 2, 3, \ldots, 7$$

Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig. 9-2. $R1$ through $R5$ are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The suboperations performed in each segment of the pipeline are as follows:

$R1 \leftarrow A_i, \quad R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table 9-1. The first clock pulse transfers A_1 and B_1 into $R1$ and

Figure 9-2 Example of pipeline processing.



Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

$R2$. The second clock pulse transfers the product of $R1$ and $R2$ into $R3$ and C_1 into $R4$. The same clock pulse transfers A_2 and B_2 into $R1$ and $R2$. The third clock pulse operates on all three segments simultaneously. It places A_3 and B_3 into $R1$ and $R2$, transfers the product of $R1$ and $R2$ into $R3$, transfers C_2 into $R4$, and places the sum of $R3$ and $R4$ into $R5$. It takes three clock pulses to fill up the pipe and retrieve the first output from $R5$. From there on, each clock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

Summary

- In a **non-pipelined** processor, each instruction is executed completely before execution of the next instruction begins.
- In a **pipelined** processor, instruction execution is divided into stages and execution of the next instruction starts as soon as the current instruction has completed the first stage. This increases the rate at which instructions can be executed, improving performance.

Pipeline Characteristics:

1. **Pipelining** is a technique for overlapping the execution of several instructions to reduce the overall execution time of a set of instructions. A pipeline does not speed up an individual computation. A pipeline processor executes instructions in an assembly line manner so multiple tasks can be performed simultaneously. However, at no given time can two of the same tasks be performed, so each task is allotted the same time as the task that takes the longest amount of time. The net effect is that results are output more quickly than in a non- pipelined unit. This increases the **throughput**, the number of results generated per time unit.
2. **Throughput** is also defined to be the rate at which operations get executed (generally expressed as operations/second or operations/cycle). Think of an automated car wash line, where multiple cars are serviced, but only one vehicle at a time can be shampooed, conditioned or dried
3. **Pipeline** in this context refers to a processor organization in which the processor consists of a number of stages, allowing multiple instructions to be executed concurrently.
4. **Pipelining** refers to overlapping operations by moving data or instructions into a conceptual pipe with all stages of the pipe processing simultaneously

5. **Pipelining** can bring significant performance benefits, as each successive action finds all its prerequisites already satisfied, so the action is ready to go immediately.
6. **Pipeline** is filled by the CPU scheduler from a pool of work which is waiting to occur. Each execution unit has a pipeline associated with it, so as to have work pre-planned.
7. **Pipelining** does not reduce the time to complete an instruction, but increases the number of instructions that can be processed at once.
8. **Pipelined** CPUs are significantly more efficient than non-pipelined CPUs, provided the scheduler can keep the pipeline full. It work on an execution unit produces an output that the scheduler had not predicted (i.e. a jump rather than a computation output), then the pipeline stalls, and it has to be completely emptied, while the scheduler reorders work to account for the new outcome.

Thus, the efficiency of a pipelined CPU is entirely dependent on the CPU scheduler's effectiveness at predicting the outcome of each instruction (action). If the workload is such that the predictive scheduler can't do a good job, and frequent pipeline stalls occur, then it will often be the case that a non-pipelined design will perform better on that workload.

The trade-off is thus: the longer the instruction pipeline for an execution unit, the better performance that unit can have, but the harder (and more complex) the work is for the predictive scheduler, and the greater the cost (in terms of performance hit) that a pipeline stall is.