Pipelining

- Pipeline processing is an implementation technique where arithmetic sub-operations or the phases of a computer instruction cycle overlap in execution.
- Pipelining is a technique of decomposing a sequential process into sub-operations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments.
- A pipeline can be visualized as a collection of processing segments through which binary information flows.
- Each segment performs partial processing dictated by the way the task is partitioned.
- The result obtained from the computation in each segment is transferred to the next segment in the pipeline.
- The final result is obtained after the data have passed through all segments.

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

$$A_i \times B_i + C_i \; ; for \; i = 1,2,3 \ldots .7$$

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 Figure.

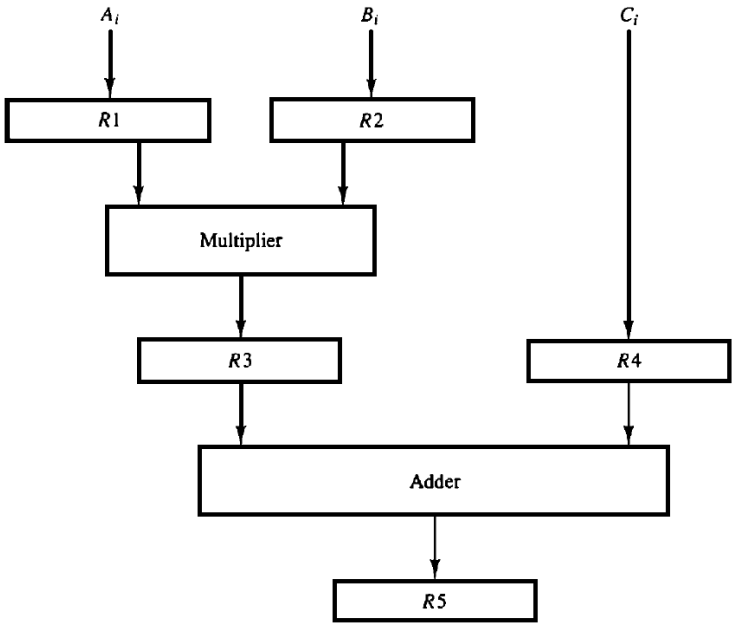

Figure: Example of Pipeline Processing

R1 through RS 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:

$$R1 \leftarrow A_i, R2 \leftarrow B_i \qquad \text{Input } A_i \text{ and Bi}$$

$$R3 \leftarrow R1 \times R2, R4 \leftarrow C_i \qquad \text{Multiply and Input } C_i$$

$$R5 \leftarrow R3 + R4 \qquad \text{Add } C_i \text{ to Product}$$

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table. The first clock pulse transfers Al and *B,* into R1 and *R2.* The second clock pulse transfers the product of *R*1 and *R2* into R3 and C_1 into *R4.* The same clock pulse transfers *A2* and *B2* into *R* 1 and *R2.* The third clock pulse operates on all three segments simultaneously. It places *A3* and B3 into *R* 1 and *R2,* transfers the product of R 1 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.

Table: Content of Registers in Pipelining Example

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	--	--	--	--	$A_7 * B_7 + C_7$

Now consider the case where a k-segment pipeline with a clock cycle time t_p is used to execute *n* tasks. The first task T_1 requires a time equal to 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 they will be completed after a time equal to $(n - 1)t_p$. Therefore, to complete *n* tasks using a k-segment pipeline requires *k* + (*n* - 1) clock cycles. For example, the table shows four segments and six tasks. The time required to complete all the operations is 4 + (6 - 1) = 9 clock cycles, as indicated in the table.

Next consider a non pipeline unit that performs the same operation and takes a time equal to t_n to complete each task. The total time required for *n* tasks is nt_n. The speedup of a pipeline processing over an equivalent non pipeline processing is defined by the ratio

		Clock Cycles								
Segment		1	2	3	4	5	6	7	8	9
	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

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

As the number of tasks increases, *n* becomes much larger than *k* - I, and *k* + *n* - 1 approaches the value of *n.* Under this condition, the speedup becomes

$$S = \frac{t_n}{t_p}$$

If we assume that the time it takes to process a task is the same in the pipeline and nonpipeline circuits, we will have $t_n = kt_p$. Including this assumption, the speedup reduces to

$$S = \frac{kt_p}{t_p} = k$$

This shows that the theoretical maximum speed-up that a pipeline can provide is *k,* where *k* is the number of segments in the pipeline.

To clarify the meaning of the speedup ratio, consider the following numerical example.

- Let the time it takes to process a suboperation in each segment be equal to t_p = 20 ns.
- Assume that the pipeline has *k* = 4 segments and executes *n* = 100 tasks in sequence.
- The pipeline system will take $(k + n - l)t_p$ = (4 + 99) x 20 = 2060 ns to complete.
- Assuming that $t_n = kt_p$ = 4 x 20 = 80 ns,
- A nonpipeline system requires nk_t, = 100 x 80 = 8000 ns to complete the 100 tasks.
- The speedup ratio is equal to *8000/2060* = 3.88.
- As the number of tasks increases, the speedup will approach 4, which is equal to the number of segments in the pipeline. If we assume that t_n = 60 ns, the speedup becomes *60/20* = 3.