Basic concepts of Pipelining

Pipelining: Why?

To improve the performance of a CPU we have two options:

1) Improve the hardware by introducing faster circuits.
2) Arrange the hardware such that more than one operation can be performed at the same time.

Since, there is a limit on the speed of hardware and the cost of faster circuits is quite high, we have to adopt the 2nd option.

Pipelining : Pipelining is a process of arrangement of hardware elements of the CPU such that its overall performance is increased. Simultaneous execution of more than one instruction takes place in a pipelined processor.

Concept of Pipelining:

Suppose you wanted to make an automobile from scratch. You might gather up the raw materials, form the metal into recognizable shapes, cast some of the metal into an engine block, connect up fuel lines, wires, etc., to eventually (one would hope) make a workable automobile. To do this, you would need many skills - all the skills of the artisans that make autos, and management skills in addition to being an electrician and a metallurgist. This would not be an efficient way to make a car, but would definitely provide many challenges.

That is the way a multi cycle datapath works - it is designed to do everything - input, output, and computation (recall the fetch-decode-execute sequence). We need to ask ourselves if this is really the best way to compute efficiently, especially when we consider the complexity of control for large (CISC) systems or even smaller RISC processors.

Fortunately, our analogy with car-making is not so far-fetched, and can actually help us arrive at a more efficient processor design. Consider the modern way of making cars - on an *assembly line*. Here, there is an orderly flow of parts down a conveyor belt, and the parts are processed by different stations (also called *segments* of the assembly line). Each segment does one thing, over and over. The segments are coordinated to exploit the *sequentiality* inherent in the automobile assembly process. The work gets done more smoothly (because of the orderly flow of input parts and output results), more efficiently (because each assembler at each segment of the pipeline does his or her task at what one hopes is maximum efficiency), and more reliably because there is greater consistency in one task being done repetitively (provided the assembly line is designed correctly).

A similar analogy exists for computers. Instead of a multicycle datapath with its complex control system that walks, talks, cries, and computes - let us suppose that we could build an *assembly line for computing*. Such objects actually exist, and they are called *pipeline processors*. They have sequentially-arranged stages or segments, each of which perform a specific task in a fixed amount of time. Data flows through these pipelines like cars through an assembly line.

Hence, Pipelining is a technique of decomposing a sequential process into suboperations, 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 name "pipeline" implies a flow of information analogous to an industrial assembly line. It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time. The overlapping of computation is made possible by associating a register with each segment in the pipeline. The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

Example of pipeline processing:

Suppose that we want to perform the combined multiply and add operations with a stream of numbers.

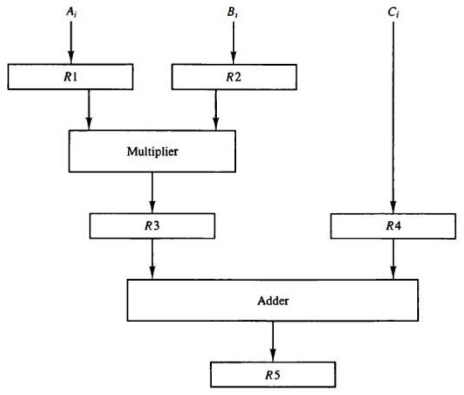
$$A_i * B_i + C_i \qquad \text{for } i = 1, 2, 3, \ldots, 7$$

Each suboperation is to be implemented in a segment within a pipeline.

Soln: The sub operations 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

Figure Example of pipeline processing.



The five registers are loaded with new data every clock pulse. The effect of each clock is shown in following Table. The first clock pulse transfers A1 and 81 into R 1 and R2. The second dock pulse transfers the product of R 1 and R2 into R3 and C1 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 A, and B, into R1 and R2, transfers the product of R1 and R2 into R3, transfers C, into R4, and places the sum of R3 and R4 into RS. It takes three clock pulses to fill up the pipe and retrieve the first output from RS. From there on, each dock 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 Pipeline 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	—	—	—	—	$A_7 * B_7 + C_7$

Pipelining-Speedup:

General Considerations:

Any operation that can be decomposed into a sequence of suboperations of about the same complexity can be implemented by a pipeline processor. The technique is efficient for those applications that need to repeat the same task many times with different sets of data. The general structure of a four-segment pipeline is illustrated in Fig. 9-3.

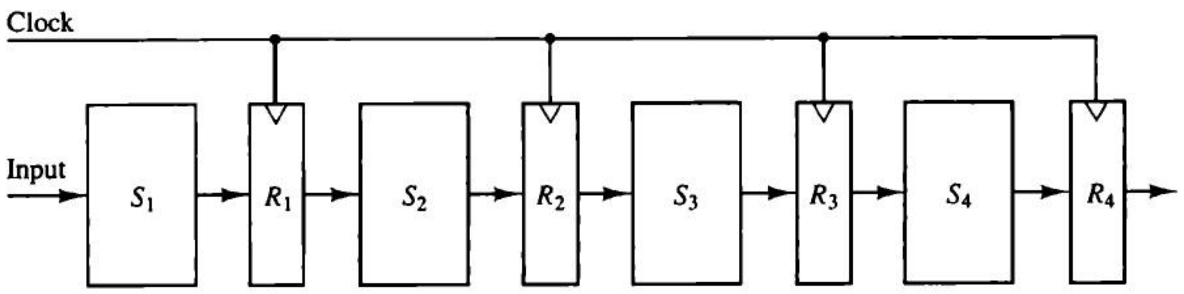


Figure 9-3 Four-segment pipeline.

The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit S; that performs a suboperation over the data stream flowing through the pipe. The segments are separated by registers R; that hold the intermediate results between the stages. Information flows between adjacent stages under the control of a common clock applied to all the registers simultaneously. We task define a task as the total operation performed going through all the segments in the pipeline

Space-time diagram for pipeline:

The behavior of a pipeline can be illustrated with a space-time diagram. This is a diagram that shows the segment utilization as a function of time. The space-time cliagram of a four-segment pipeline is demonstrated in Fig. 9-4. The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number.

Figure 9-4 Space-time diagram for pipeline.

	1	2	3	4	5	6	7	8	9	→ Clock cycles
Segment: 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	

The diagram shows six tasks T_1 through T_6 executed in four segments. Initially, task T_1 is handled by segment 1. After the first clock, segment 2 is busy with T_1, while segment 1 is busy with task T_2. Continuing in this manner, the first task T_1 is completed after the fourth clock cycle. From then on, the pipe completes a task every clock cycle. No matter how many segments there are in the system, once the pipeline is full, it takes only one clock period to obtain an output.

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 diagram of Fig. 9-4 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 diagram.

Next consider a nonpipeline 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 nonpipeline processing is defined by the ratio

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

As the number of tasks increases, n becomes much larger than $k - 1$, 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 speedup 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 $(k + n - 1)t_p = (4 + 99) \times 20 = 2060$ ns to complete. Assuming that $t_n = kt_p = 4 \times 20 = 80$ ns, a nonpipeline system requires $nkt_p = 100 \times 80 = 8000$ 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 $60/20 = 3$.

To duplicate the theoretical speed advantage of a pipeline process by means of multiple functional units, it is necessary to construct k identical units that will be operating in parallel. The implication is that a k-segment pipeline processor can be expected to equal the performance of k copies of an equivalent nonpipeline circuit under equal operating conditions. This is illustrated in Fig. 9-5, where four identical circuits are connected in parallel. Each P circuit performs the same task of an equivalent pipeline circuit. Instead of operating with the input data in sequence as in a pipeline, the parallel circuits acceptfour input data items simultaneously and perform four tasks at the same time. As far as the speed of operation is concerned, this is equivalent to a four segment pipeline. Note that the four-unit circuit of Fig. 9-5 constitutes a single-instruction multiple-data (SIMD) organization since the same instruction is used to operate on multiple data in parallel.

There are various reasons why the pipeline cannot operate at its maximum theoretical rate. Different segments may take different times to complete their suboperation. The clock cycle must be chosen to equal the time delay of the segment with the maximum propagation time. This causes all other segments to waste time while waiting for the next clock. Moreover, it is not always

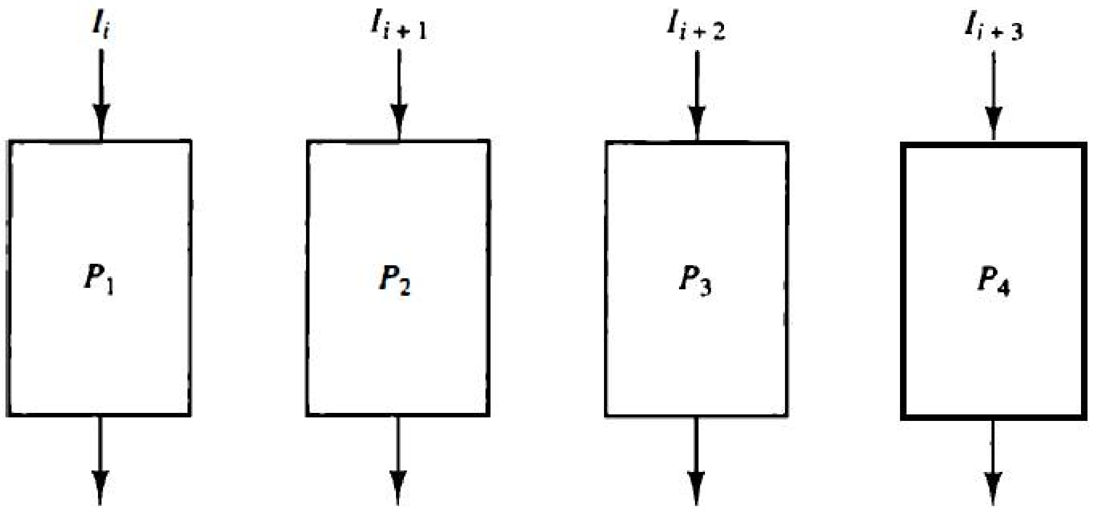


Figure 9-5 Multiple functional units in parallel.

correct to assume that a nonpipe circuit has the same time delay as that of an equivalent pipeline circuit. Many of the intermediate registers will not be needed in a single-unit circuit, which can usually be constructed entirely as a combinational circuit. Nevertheless, the pipeline technique provides a faster operation over a purely serial sequence even though the maximum theoretical speed is never fully achieved.

Home work: Explain with a suitable example - "Dependencies" and Data Hazard in a pipelined processor.

(Try to do it.. any problem for this homework please let me know.. I will explain.)