Monogram on pipelining

By Mr. Vishal Jaiswal

Pipelining:

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. 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 overlapping of computation is made possible by associating a register with each segment in the pipeline.** The register holds the data and the combinational circuit performs the suboperation in the particular segment. The output of the combinational circuit in a given segment is applied to the input register of the next segment. A clock is applied to all registers after enough time has elapsed to perform all segment activity.

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

Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit. 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.

Figure 9-2 Example of pipeline processing.

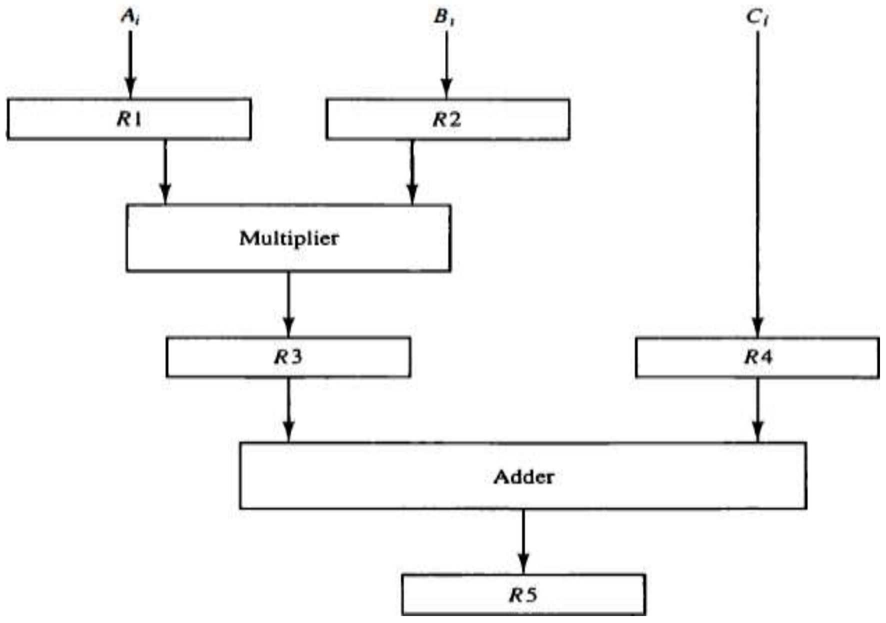


TABLE 9-1 Content of Registers in Pipeline Example

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	—	—	—	—	$A_7 * B_7 + C_7$

Four Segment Pipeline: Consider the case where a k-segment pipeline with a clock cycle time t_p is used to execute n tasks. The first task T1 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.

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

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

Figure 9-4 Space-time diagram for pipeline.

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

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 non pipeline circuits, we will have $t_p = kt_n$. 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.

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 non pipeline 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.

Question: Determine the number of dock cycles that it takes to process 200 tasks in a six-segment pipeline.

k = 6 segments
n = 200 tasks (k + n – 1) = 6 + 200 – 1 = 205 cycles

Question: A nonpipeline system takes 50 ns to process a task. The same task can be processed in a six-segment pipeline with a dock cycle of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speedup that can be achieved?

$t_n = 50$ ns
$k = 6$
$t_p = 10$ ns
$n = 100$

$$S = \frac{nt_n}{(k+n-1)t_p} = \frac{100 \times 50}{(6-99) \times 10} = 4.76$$

$$S_{max} = \frac{t_n}{t_p} = \frac{50}{10} = 5$$

Question: The pipeline has the following propagation times: 40 ns for the operands to be read from memory into registers R1 and R2, 45 ns for the signal to propagate

through the multiplier, 5 ns for the transfer into R3, and 15 ns to add the two numbers into R5.
a. What is the minimum dock cycle time that can be used?
b. A non pipeline system can perform the same operation by removing R3 and R4. How long will it take to multiply and add the operands without using the pipeline?
c. Calculate the speedup of the pipeline for 10 tasks and again for 100 tasks.
d. What is the maximum speedup that can be achieved?

(a) $t_p = 45 + 5 = 50$ ns k = 3
(b) $t_n = 40 + 45 + 15 = 100$ ns

(c) $$S = \frac{nt_n}{(k+n-1)\,t_p} = \frac{10\times 100}{(3+9)50} = 1.67$$ for n = 10

$$= \frac{100\times 100}{(3+99)50} = 1.96$$ for n = 100

(d) $$S_{max} = \frac{t_n}{t_p} = \frac{100}{50} = 2$$

Instruction Pipeline:

The computer needs to process each instruction with the following sequence of steps.

1. **Fetch the instruction from memory.**
2. **Decode the instruction.**
3. **Calculate the effective address.**
4. **Fetch the operands from memory.**
5. **Execute the instruction.**
6. **Store the result in the proper place.**

Four-Segment Instruction Pipeline

Assume that the decoding of the instruction can be combined with the calculation of the effective address into one segment. Assume further that most of the instructions place the result into a processor register so that the instruction execution and storing of the result can be combined into one segment. This reduces the instruction pipeline into four segments.
The four segments are represented in the diagram with an abbreviated symbol.

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO can proceed at the same time. In the absence of a branch instruction, each segment operates on different instructions. Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.

Pipeline Conflicts or Pipeline Hazard

In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

1. Resource conflicts caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.
2. Data dependency conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
3. Branch difficulties arise from branch and other instructions that change the value of PC .

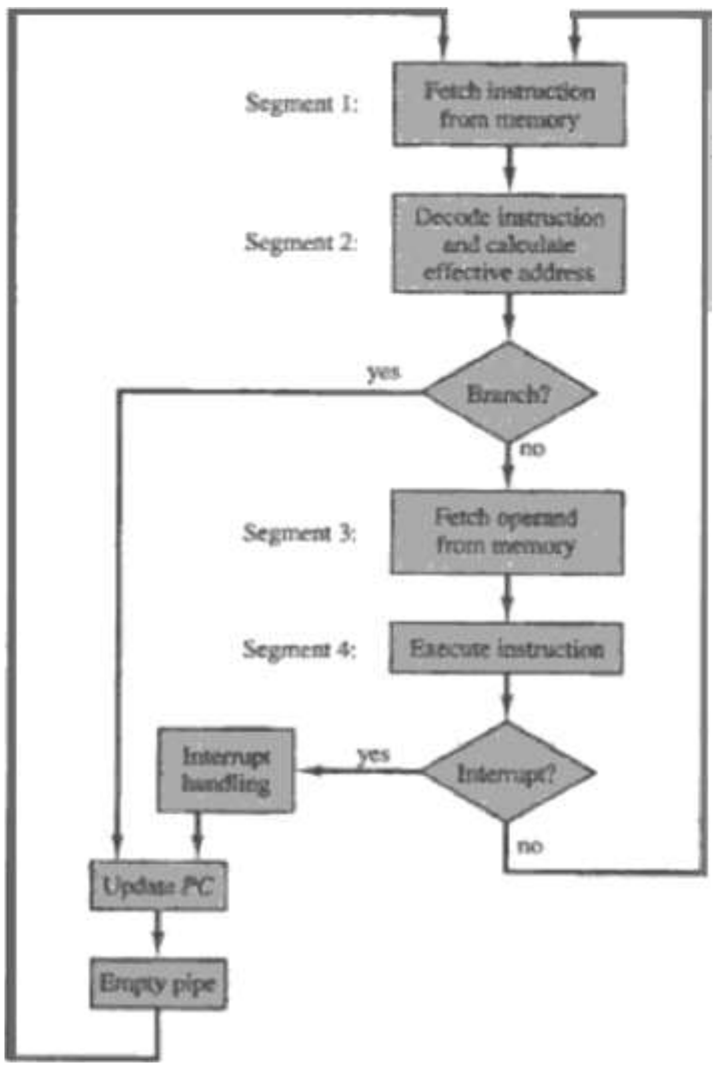
Data dependency: A data dependency occurs when an instruction needs data that are not yet available. For example, an instruction in the FO segment may need to fetch an operand that is being generated at the same time by the previous instruction in segment EX. Therefore, the second instruction must wait for data to become available by the first instruction.

Hardware Interlocks:

An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict.

Operand Forwarding:

Another technique called operand forwarding uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments. For example, instead of transferring an ALU result into a destination register, the hardware checks the destination operand, and if it is needed as a source in the next instruction, it passes the result directly into the ALU input, bypassing the register file.



Step:		1	2	3	4	5	6	7	8	9	IO	I I	I2	I3
I rntruciicn	I	FI	DA	FO	EX									
	2		FI	DA	R3	EX								
(Brnmhj	3			FI	DA	FO	EX							
	4				FI			FI	DA	FO	EX			
	5								Fi	DA	FO	EX		
	6									B	DA	FO	EX	
	7										FI	DA	PO	EX

Figure 9-8 Timing of instruction pipeline.