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. The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit S_i that performs a suboperation over the data stream flowing through the pipe. The segments are separated by registers R_i 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 define a *task* as the total operation performed going through all the segments in the pipeline.

task

space-time diagram

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 diagram 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

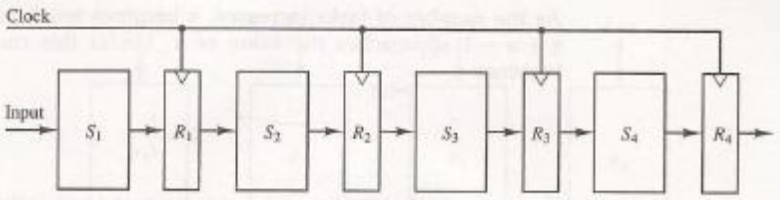

Figure 9-3 Four-segment pipeline.

segment number. 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

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

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 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 accept four 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

9-1. In certain scientific computations it is necessary to perform the arithmetic operation $(A_i + B_i)(C_i + D_i)$ with a stream of numbers. Specify a pipeline configuration to carry out this task. List the contents of all registers in the pipeline for $i = 1$ through 6.

9-2. Draw a space-time diagram for a six-segment pipeline showing the time it takes to process eight tasks.

9-3. Determine the number of clock cycles that it takes to process 200 tasks in a six-segment pipeline.

9-4. A nonpipeline system takes 50 ns to process a task. The same task can be processed in a six-segment pipeline with a clock cycle of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speedup that can be achieved?

9-5. The pipeline of Fig. 9-2 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 clock cycle time that can be used?

b. A nonpipeline 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?

9-6. It is necessary to design a pipeline for a fixed-point multiplier that multiplies two 8-bit binary integers. Each segment consists of a number of AND gates and a binary adder similar to an array multiplier as shown in Fig. 10-10.

a. How many AND gates are there in each segment, and what size of adder is needed?

b. How many segments are there in the pipeline?

c. If the propagation delay in each segment is 30 ns, what is the average time that it takes to multiply two fixed-point numbers in the pipeline?

9-7. The time delay of the four segments in the pipeline of Fig. 9-6 are as follows: $t_1 = 50$ ns, $t_2 = 30$ ns, $t_3 = 95$ ns, and $t_4 = 45$ ns. The interface registers delay time $t_r = 5$ ns.

a. How long would it take to add 100 pairs of numbers in the pipeline?

b. How can we reduce the total time to about one-half of the time calculated in part (a)?

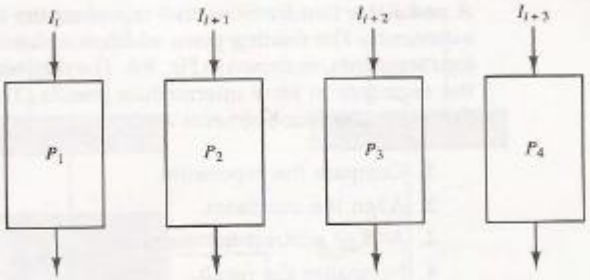

Figure 9-5 Multiple functional units in parallel.

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.

There are two areas of computer design where the pipeline organization is applicable. An *arithmetic pipeline* divides an arithmetic operation into sub-operations for execution in the pipeline segments. An *instruction pipeline* operates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle. The two types of pipelines are explained in the following sections.