The only problem with SEQ is that it is too slow. The clock must run slowly enough so that signals can propagate through all of the stages within a single cycle. As an example, consider the processing of a `ret` instruction. Starting with an updated program counter at the beginning of the clock cycle, the instruction must be read from the instruction memory, the stack pointer must be read from the register file, the ALU must increment the stack pointer by 8, and the return address must be read from the memory in order to determine the next value for the program counter. All of these must be completed by the end of the clock cycle.

This style of implementation does not make very good use of our hardware units, since each unit is only active for a fraction of the total clock cycle. We will see that we can achieve much better performance by introducing pipelining.

4.4 General Principles of Pipelining

Before attempting to design a pipelined Y86-64 processor, let us consider some general properties and principles of pipelined systems. Such systems are familiar to anyone who has been through the serving line at a cafeteria or run a car through an automated car wash. In a pipelined system, the task to be performed is divided into a series of discrete stages. In a cafeteria, this involves supplying salad, a main dish, dessert, and beverage. In a car wash, this involves spraying water and soap, scrubbing, applying wax, and drying. Rather than having one customer run through the entire sequence from beginning to end before the next can begin, we allow multiple customers to proceed through the system at once. In a traditional cafeteria line, the customers maintain the same order in the pipeline and pass through all stages, even if they do not want some of the courses. In the case of the car wash, a new car is allowed to enter the spraying stage as the preceding car moves from the spraying stage to the scrubbing stage. In general, the cars must move through the system at the same rate to avoid having one car crash into the next.

A key feature of pipelining is that it increases the *throughput* of the system (i.e., the number of customers served per unit time), but it may also slightly increase the *latency* (i.e., the time required to service an individual customer). For example, a customer in a cafeteria who only wants a dessert could pass through a nonpipelined system very quickly, stopping only at the dessert stage. A customer in a pipelined system who attempts to go directly to the dessert stage risks incurring the wrath of other customers.

4.4.1 Computational Pipelines

Shifting our focus to computational pipelines, the "customers" are instructions and the stages perform some portion of the instruction execution. Figure 4.32(a) shows an example of a simple nonpipelined hardware system. It consists of some logic that performs a computation, followed by a register to hold the results of this computation. A clock signal controls the loading of the register at some regular time interval. An example of such a system is the decoder in a compact disk (CD) player. The incoming signals are the bits read from the surface of the CD, and

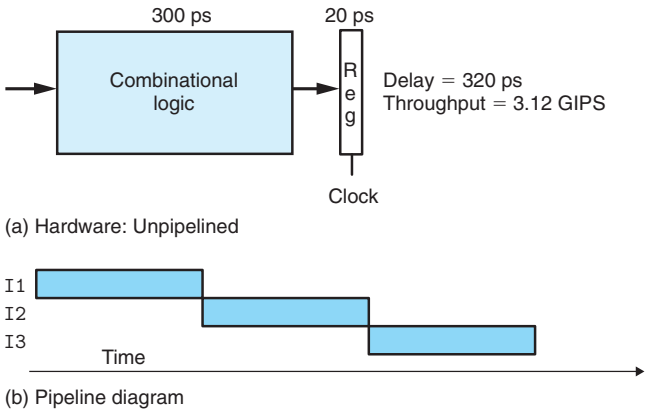


Figure 4.32
Unpipelined computation hardware. On each 320 ps cycle, the system spends 300 ps evaluating a combinational logic function and 20 ps storing the results in an output register.

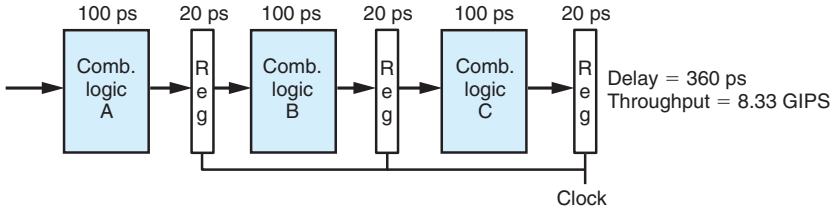
the logic decodes these to generate audio signals. The computational block in the figure is implemented as combinational logic, meaning that the signals will pass through a series of logic gates, with the outputs becoming some function of the inputs after some time delay.

In contemporary logic design, we measure circuit delays in units of *picoseconds* (abbreviated "ps"), or 10^{-12} seconds. In this example, we assume the combinational logic requires 300 ps, while the loading of the register requires 20 ps. Figure 4.32 shows a form of timing diagram known as a *pipeline diagram*. In this diagram, time flows from left to right. A series of instructions (here named `I1`, `I2`, and `I3`) are written from top to bottom. The solid rectangles indicate the times during which these instructions are executed. In this implementation, we must complete one instruction before beginning the next. Hence, the boxes do not overlap one another vertically. The following formula gives the maximum rate at which we could operate the system:

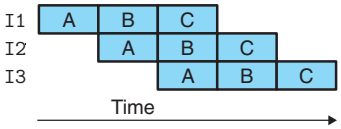
$$\textit{Throughput} = \frac{1 \text{ instruction}}{(20 + 300) \text{ picoseconds}} \cdot \frac{1{,}000 \text{ picoseconds}}{1 \text{ nanosecond}} \approx 3.12 \text{ GIPS}$$

We express throughput in units of giga-instructions per second (abbreviated GIPS), or billions of instructions per second. The total time required to perform a single instruction from beginning to end is known as the *latency*. In this system, the latency is 320 ps, the reciprocal of the throughput.

Suppose we could divide the computation performed by our system into three stages, A, B, and C, where each requires 100 ps, as illustrated in Figure 4.33. Then we could put *pipeline registers* between the stages so that each instruction moves through the system in three steps, requiring three complete clock cycles from beginning to end. As the pipeline diagram in Figure 4.33 illustrates, we could allow `I2` to enter stage A as soon as `I1` moves from A to B, and so on. In steady state, all three stages would be active, with one instruction leaving and a new one entering the system every clock cycle. We can see this during the third clock cycle in the pipeline diagram where `I1` is in stage C, `I2` is in stage B, and `I3` is in stage A. In



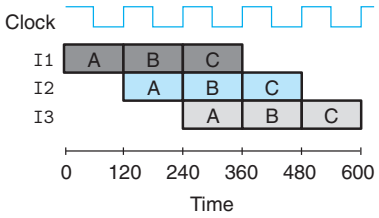
(a) Hardware: Three-stage pipeline



(b) Pipeline diagram

Figure 4.33 **Three-stage pipelined computation hardware.** The computation is split into stages A, B, and C. On each 120 ps cycle, each instruction progresses through one stage.

Figure 4.34
Three-stage pipeline timing. The rising edge of the clock signal controls the movement of instructions from one pipeline stage to the next.



this system, we could cycle the clocks every $100 + 20 = 120$ picoseconds, giving a throughput of around 8.33 GIPS. Since processing a single instruction requires 3 clock cycles, the latency of this pipeline is $3 \times 120 = 360$ ps. We have increased the throughput of the system by a factor of $8.33/3.12 = 2.67$ at the expense of some added hardware and a slight increase in the latency ($360/320 = 1.12$). The increased latency is due to the time overhead of the added pipeline registers.

4.4.2 A Detailed Look at Pipeline Operation

To better understand how pipelining works, let us look in some detail at the timing and operation of pipeline computations. Figure 4.34 shows the pipeline diagram for the three-stage pipeline we have already looked at (Figure 4.33). The transfer of the instructions between pipeline stages is controlled by a clock signal, as shown above the pipeline diagram. Every 120 ps, this signal rises from 0 to 1, initiating the next set of pipeline stage evaluations.

Figure 4.35 traces the circuit activity between times 240 and 360, as instruction `I1` (shown in dark gray) propagates through stage C, `I2` (shown in blue)

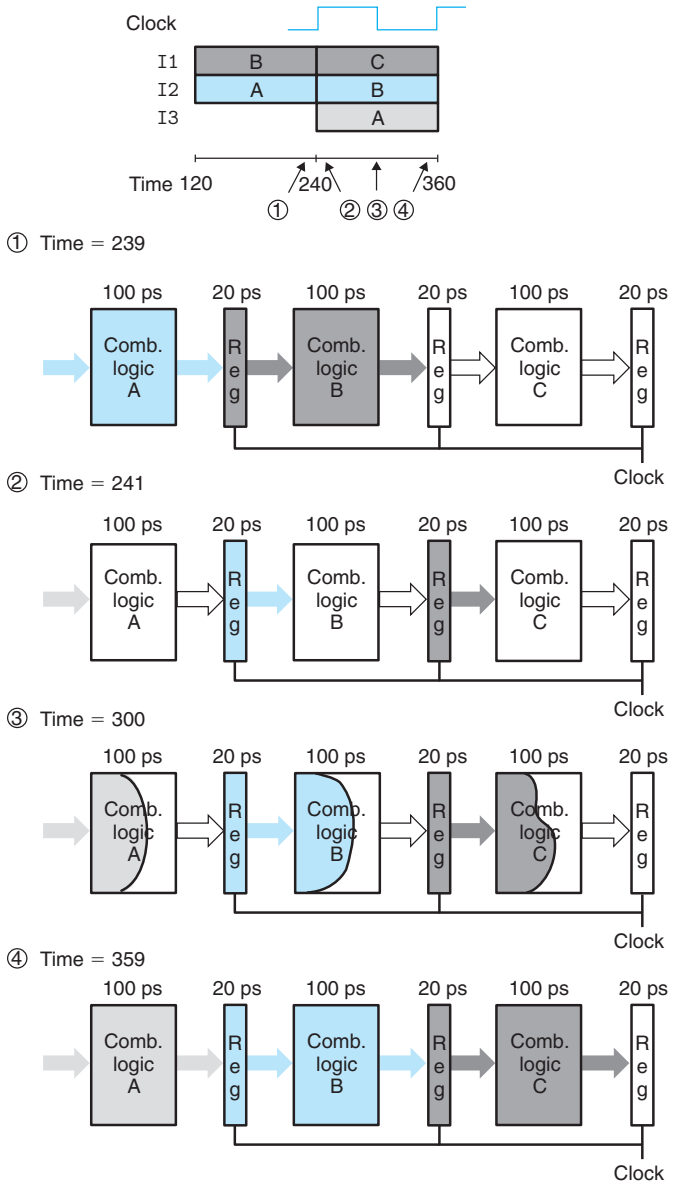


Figure 4.35 **One clock cycle of pipeline operation.** Just before the clock rises at time 240 (point 1), instructions `I1` (shown in dark gray) and `I2` (shown in blue) have completed stages B and A. After the clock rises, these instructions begin propagating through stages C and B, while instruction `I3` (shown in light gray) begins propagating through stage A (points 2 and 3). Just before the clock rises again, the results for the instructions have propagated to the inputs of the pipeline registers (point 4).

propagates through stage B, and I3 (shown in light gray) propagates through stage A. Just before the rising clock at time 240 (point 1), the values computed in stage A for instruction I2 have reached the input of the first pipeline register, but its state and output remain set to those computed during stage A for instruction I1. The values computed in stage B for instruction I1 have reached the input of the second pipeline register. As the clock rises, these inputs are loaded into the pipeline registers, becoming the register outputs (point 2). In addition, the input to stage A is set to initiate the computation of instruction I3. The signals then propagate through the combinational logic for the different stages (point 3). As the curved wave fronts in the diagram at point 3 suggest, signals can propagate through different sections at different rates. Before time 360, the result values reach the inputs of the pipeline registers (point 4). When the clock rises at time 360, each of the instructions will have progressed through one pipeline stage.

We can see from this detailed view of pipeline operation that slowing down the clock would not change the pipeline behavior. The signals propagate to the pipeline register inputs, but no change in the register states will occur until the clock rises. On the other hand, we could have disastrous effects if the clock were run too fast. The values would not have time to propagate through the combinational logic, and so the register inputs would not yet be valid when the clock rises.

As with our discussion of the timing for the SEQ processor (Section 4.3.3), we see that the simple mechanism of having clocked registers between blocks of combinational logic suffices to control the flow of instructions in the pipeline. As the clock rises and falls repeatedly, the different instructions flow through the stages of the pipeline without interfering with one another.

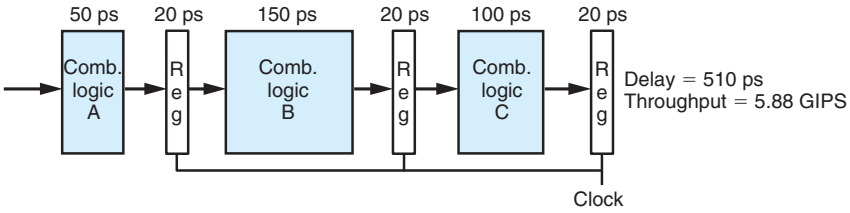
4.4.3 Limitations of Pipelining

The example of Figure 4.33 shows an ideal pipelined system in which we are able to divide the computation into three independent stages, each requiring one-third of the time required by the original logic. Unfortunately, other factors often arise that diminish the effectiveness of pipelining.

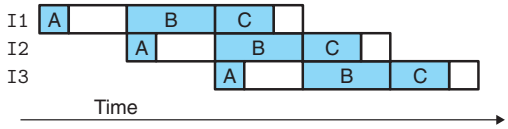
Nonuniform Partitioning

Figure 4.36 shows a system in which we divide the computation into three stages as before, but the delays through the stages range from 50 to 150 ps. The sum of the delays through all of the stages remains 300 ps. However, the rate at which we can operate the clock is limited by the delay of the slowest stage. As the pipeline diagram in this figure shows, stage A will be idle (shown as a white box) for 100 ps every clock cycle, while stage C will be idle for 50 ps every clock cycle. Only stage B will be continuously active. We must set the clock cycle to $150 + 20 = 170$ picoseconds, giving a throughput of 5.88 GIPS. In addition, the latency would increase to 510 ps due to the slower clock rate.

Devising a partitioning of the system computation into a series of stages having uniform delays can be a major challenge for hardware designers. Often,



(a) Hardware: Three-stage pipeline, nonuniform stage delays



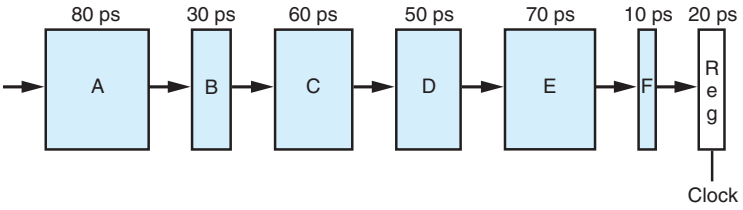
(b) Pipeline diagram

Figure 4.36 Limitations of pipelining due to nonuniform stage delays. The system throughput is limited by the speed of the slowest stage.

some of the hardware units in a processor, such as the ALU and the memories, cannot be subdivided into multiple units with shorter delay. This makes it difficult to create a set of balanced stages. We will not concern ourselves with this level of detail in designing our pipelined Y86-64 processor, but it is important to appreciate the importance of timing optimization in actual system design.

Practice Problem 4.28 (solution page 525)

Suppose we analyze the combinational logic of Figure 4.32 and determine that it can be separated into a sequence of six blocks, named A to F, having delays of 80, 30, 60, 50, 70, and 10 ps, respectively, illustrated as follows:



We can create pipelined versions of this design by inserting pipeline registers between pairs of these blocks. Different combinations of pipeline depth (how many stages) and maximum throughput arise, depending on where we insert the pipeline registers. Assume that a pipeline register has a delay of 20 ps.

A. Inserting a single register gives a two-stage pipeline. Where should the register be inserted to maximize throughput? What would be the throughput and latency?

B. Where should two registers be inserted to maximize the throughput of a three-stage pipeline? What would be the throughput and latency?

C. Where should three registers be inserted to maximize the throughput of a 4-stage pipeline? What would be the throughput and latency?

D. What is the minimum number of stages that would yield a design with the maximum achievable throughput? Describe this design, its throughput, and its latency.

Diminishing Returns of Deep Pipelining

Figure 4.37 illustrates another limitation of pipelining. In this example, we have divided the computation into six stages, each requiring 50 ps. Inserting a pipeline register between each pair of stages yields a six-stage pipeline. The minimum clock period for this system is $50 + 20 = 70$ picoseconds, giving a throughput of 14.29 GIPS. Thus, in doubling the number of pipeline stages, we improve the performance by a factor of $14.29/8.33 = 1.71$. Even though we have cut the time required for each computation block by a factor of 2, we do not get a doubling of the throughput, due to the delay through the pipeline registers. This delay becomes a limiting factor in the throughput of the pipeline. In our new design, this delay consumes 28.6% of the total clock period.

Modern processors employ very deep pipelines (15 or more stages) in an attempt to maximize the processor clock rate. The processor architects divide the instruction execution into a large number of very simple steps so that each stage can have a very small delay. The circuit designers carefully design the pipeline registers to minimize their delay. The chip designers must also carefully design the clock distribution network to ensure that the clock changes at the exact same time across the entire chip. All of these factors contribute to the challenge of designing high-speed microprocessors.

Practice Problem 4.29 (solution page 526)

Suppose we could take the system of Figure 4.32 and divide it into an arbitrary number of pipeline stages k, each having a delay of $300/k$, and with each pipeline register having a delay of 20 ps.

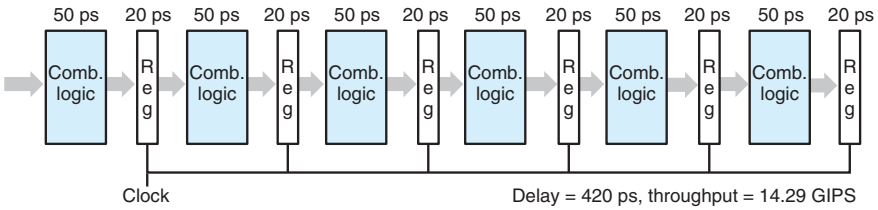


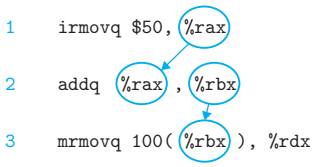
Figure 4.37 Limitations of pipelining due to overhead. As the combinational logic is split into shorter blocks, the delay due to register updating becomes a limiting factor.

A. What would be the latency and the throughput of the system, as functions of k?

B. What would be the ultimate limit on the throughput?

4.4.4 Pipelining a System with Feedback

Up to this point, we have considered only systems in which the objects passing through the pipeline—whether cars, people, or instructions—are completely independent of one another. For a system that executes machine programs such as x86-64 or Y86-64, however, there are potential dependencies between successive instructions. For example, consider the following Y86-64 instruction sequence:



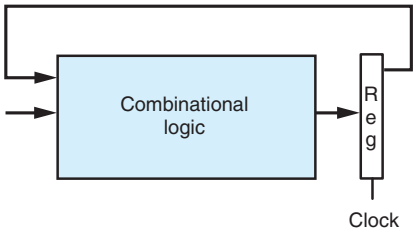
In this three-instruction sequence, there is a *data dependency* between each successive pair of instructions, as indicated by the circled register names and the arrows between them. The `irmovq` instruction (line 1) stores its result in `%rax`, which then must be read by the `addq` instruction (line 2); and this instruction stores its result in `%rbx`, which must then be read by the `mrmovq` instruction (line 3).

Another source of sequential dependencies occurs due to the instruction control flow. Consider the following Y86-64 instruction sequence:

```
1  loop:
2      subq %rdx,%rbx
3      jne targ
4      irmovq $10,%rdx
5      jmp loop
6  targ:
7      halt
```

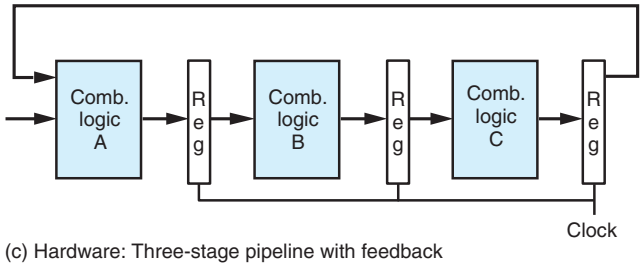
The `jne` instruction (line 3) creates a *control dependency* since the outcome of the conditional test determines whether the next instruction to execute will be the `irmovq` instruction (line 4) or the `halt` instruction (line 7). In our design for SEQ, these dependencies were handled by the feedback paths shown on the right-hand side of Figure 4.22. This feedback brings the updated register values down to the register file and the new PC value down to the PC register.

Figure 4.38 illustrates the perils of introducing pipelining into a system containing feedback paths. In the original system (Figure 4.38(a)), the result of each

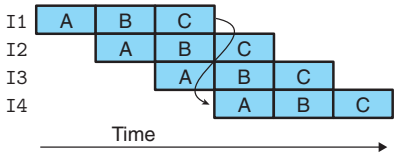


(a) Hardware: Unpipelined with feedback

(b) Pipeline diagram



(c) Hardware: Three-stage pipeline with feedback



(d) Pipeline diagram

Figure 4.38 **Limitations of pipelining due to logical dependencies.** In going from an unpipelined system with feedback (a) to a pipelined one (c), we change its computational behavior, as can be seen by the two pipeline diagrams (b and d).

instruction is fed back around to the next instruction. This is illustrated by the pipeline diagram (Figure 4.38(b)), where the result of `I1` becomes an input to `I2`, and so on. If we attempt to convert this to a three-stage pipeline in the most straightforward manner (Figure 4.38(c)), we change the behavior of the system. As Figure 4.38(c) shows, the result of `I1` becomes an input to `I4`. In attempting to speed up the system via pipelining, we have changed the system behavior.

When we introduce pipelining into a Y86-64 processor, we must deal with feedback effects properly. Clearly, it would be unacceptable to alter the system behavior as occurred in the example of Figure 4.38. Somehow we must deal with the data and control dependencies between instructions so that the resulting behavior matches the model defined by the ISA.

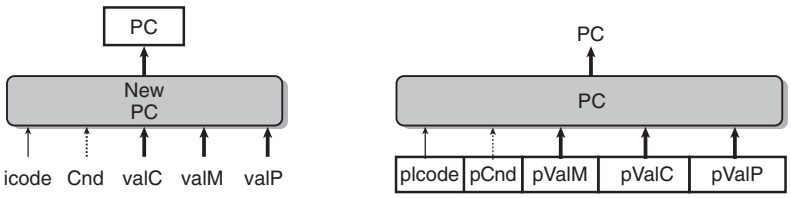
4.5 Pipelined Y86-64 Implementations

We are finally ready for the major task of this chapter—designing a pipelined Y86-64 processor. We start by making a small adaptation of the sequential processor SEQ to shift the computation of the PC into the fetch stage. We then add pipeline registers between the stages. Our first attempt at this does not handle the different data and control dependencies properly. By making some modifications, however, we achieve our goal of an efficient pipelined processor that implements the Y86-64 ISA.

4.5.1 SEQ+: Rearranging the Computation Stages

As a transitional step toward a pipelined design, we must slightly rearrange the order of the five stages in SEQ so that the PC update stage comes at the beginning of the clock cycle, rather than at the end. This transformation requires only minimal change to the overall hardware structure, and it will work better with the sequencing of activities within the pipeline stages. We refer to this modified design as SEQ+.

We can move the PC update stage so that its logic is active at the beginning of the clock cycle by making it compute the PC value for the *current* instruction. Figure 4.39 shows how SEQ and SEQ+ differ in their PC computation. With SEQ (Figure 4.39(a)), the PC computation takes place at the end of the clock cycle, computing the new value for the PC register based on the values of signals computed during the current clock cycle. With SEQ+ (Figure 4.39(b)), we create state registers to hold the signals computed during an instruction. Then, as a new clock cycle begins, the values propagate through the exact same logic to compute the PC for the now-current instruction. We label the registers "pIcode,"



(a) SEQ new PC computation (b) SEQ+ PC selection

Figure 4.39 Shifting the timing of the PC computation. With SEQ+, we compute the value of the program counter for the current state as the first step in instruction execution.