SEQUENTIAL BASICS

4

Sequential circuits are the mainstay of digital systems. In this chapter, we start by examining several sequential circuit elements that are widely used in digital systems for storing information and for counting events. We then see how a system can be built from two main sections: a datapath and a control section. We complete the chapter with a discussion of a clocked synchronous timing methodology based on the abstraction of discrete time. This methodology is central to design of complex digital systems.

4.1 STORAGE ELEMENTS

In Chapter 1, we briefly introduced the idea of sequential circuits. We described a sequential circuit as one whose outputs depend not only on the current values of inputs, but also on the previous values of inputs. Such circuits have some form of memory, or storage, of the history of input values. We mentioned that sequential circuits are commonly regulated by a periodic clock signal that divides the passage of time into discrete clock cycles. We also showed one of the simplest elements for storing values, a D flip-flop, that can store one bit of information. In this section, we will look at further uses of the D flip-flop and other storage elements.

4.1.1 FLIP-FLOPS AND REGISTERS

As a reminder, the symbol for a D flip-flop is shown in Figure 4.1, and a timing diagram is shown in Figure 4.2. The flip-flop is edge-triggered, meaning that on each rising edge of the clk input, the current value of the D input is stored within the flip-flop and reflected on the Q output. We illustrated use of D flip-flops in sequential circuits in Example 1.2, where we stored the previous two values of an input signal on successive clock edges so that we could detect a given sequence of input values.

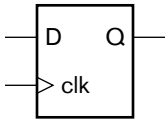

FIGURE 4.1 A D flip-flop.

While it is possible to implement a flip-flop as a combination of gates, it is not very instructive to do so. Moreover, flip-flops are provided as

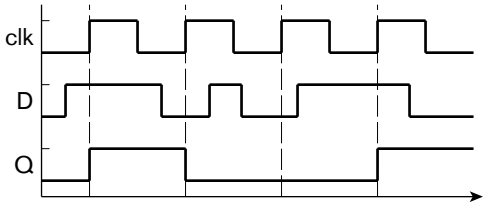

FIGURE 4.2 Timing diagram for a D flip-flop.

primitive elements in most implementation fabrics, so we would only need to implement one using gates in very exceptional circumstances. Advanced books on IC design typically include more detailed treatment of flip-flop implementation (see Section 4.6, Further Reading).

In most digital circuits, flip-flops are not used individually, but in groups to store binary-coded values. A group of flip-flops used in this way is called a *register*. Each flip-flop in the register stores one bit of the code word of the stored value, as shown in Figure 4.3. The circuit at the top of the figure shows that each bit of an input and an output signal is connected to the input and output, respectively, of one of the flip-flops, and that the clock signal is connected in common to the clock input of all of the flip-flops. When there is a rising edge on the clock input, each flip-flop in the register updates its stored bit from the signal connected to its data input and drives the new value on its data output. The symbol for the register is shown at the bottom of Figure 4.3. The difference, compared to the symbol for a single flip-flop, is in the thick lines used for the data input and output, denoting multiple bits. We can think of this as a more abstract component that has similar behavior to a D flip-flop, except that it stores a complete code word rather than a single bit.

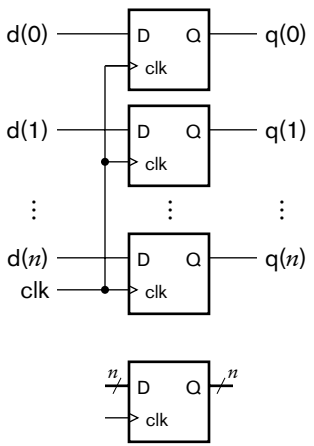

FIGURE 4.3 A register composed of D flip-flops (top), and the symbol for the register (bottom).

We can model simple D flip-flops and registers in Verilog using an always block of the form

```
always @(posedge clk)
  q <= d;
```

This is the first of a small number of always-block templates that we will introduce for modeling sequential circuits. It is important that we adhere to the template structures, since synthesis tools can generally only synthesize sequential circuits that use the templates. A complete description of the templates and the way synthesis tools process them is included in Appendix C.

We would place a block representing a flip-flop or register in the statement part of a module. The notation @(...) after the always keyword is called the block's *event list*, and specifies an event to which the block responds. In this case, the keyword posedge specifies that the event is a positive (rising) edge, a change from 0 to 1, on the clock input clk. When

the event occurs, the block performs the statement that follows. (If there is more than one statement to perform, we can group them using begin … end keywords.) The statement in this case assigns the current value of the data input d to the data output q. Since this assignment only happens on rising edges of clk, and the value of q remains unchanged between rising edges, the block models the behavior we described for an edge-triggered D flip-flop or a register. The distinction between the two arises from the sizes of d and q. If they are single bits, the block models a D flip-flop, storing just a single bit of data. If d and q are vectors, the block models a register.

There are two further points to note about this model for a flip-flop or register. First, the output q must be declared as a variable, for example, using a reg or integer keyword. As we have previously mentioned, assignments within procedural blocks must be made to variables, not nets. Second, we have used a different form of assignment symbol, <= instead of =, in this block. The form using = is called a *blocking assignment*, and can be used in blocks that model combinational logic, as we saw in Chapter 2. The form using <= is called a *nonblocking assignment*, and should be used in assignments to variables representing the outputs of flip-flops or registers. The reason for the distinctions arise from subtleties in the way variables are updated during simulation of Verilog models. We will not go into details in this book. (The details are covered in reference books on Verilog.) Instead, we will simply follow the convention of using nonblocking assignments in blocks modeling outputs of sequential logic.

One use for a register constructed from simple D flip-flops is as a *pipeline register* in a sequential design. We will discuss this in further detail in Chapter 9, focusing on the use of pipelining as a technique for improving performance of a digital system. For now, consider the circuit outlined at the top of Figure 4.4. Successive values of data arriving at the input are processed by a number of combinational subcircuits, for example, by arithmetic subcircuits built from components described in Chapter 3. The total propagation delay of the circuit is the sum of the propagation delays of the individual subcircuits. This total delay must be less than the interval between arriving data values, otherwise data values may be lost. If the total delay is too long, we can divide the circuit into segments by inserting a register after each subcircuit, as shown at the

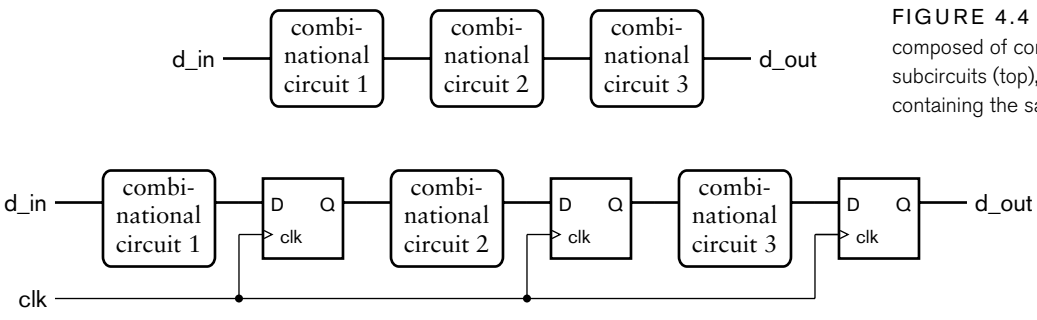

FIGURE 4.4 A circuit composed of combinational subcircuits (top), and a pipeline containing the same subcircuits.

bottom of Figure 4.4. This arrangement is called a *pipeline*, as it allows data and intermediate results to flow through over several clock cycles. A new input value arrives at the beginning of each clock cycle. During a clock cycle, each subcircuit uses the value from the preceding register (or from the input, in the case of the first subcircuit) to perform its combinational function and to yield an intermediate result. On the next rising clock edge, the intermediate results are stored in the registers at the outputs of the subcircuits. Each intermediate result is then used by the next subcircuit during the next clock cycle. Computation is thus performed in assembly-line fashion. A new final result reaches the output on each clock edge, having taken several clock cycles to be computed.

EXAMPLE 4.1 Develop a Verilog model for a pipelined circuit that computes the average of corresponding values in three streams of input values, a, b and c. The pipeline consists of three stages: the first stage sums values of a and b and saves the value of c; the second stage adds on the saved value of c; and the third stage divides by three. The inputs and output are all signed fixed-point numbers indexed from 5 down to −8.

SOLUTION The module definition is

```
module average_pipeline ( output reg signed [5:-8] avg,
                          input      signed [5:-8] a, b, c,
                          input                    clk );

  wire signed [5:-8] a_plus_b, sum, sum_div_3;
  reg  signed [5:-8] saved_a_plus_b, saved_c, saved_sum;

  assign a_plus_b = a + b;

  always @(posedge clk) begin  // Pipeline register 1
    saved_a_plus_b <= a_plus_b;
    saved_c        <= c;
  end

  assign sum = saved_a_plus_b + saved_c;

  always @(posedge clk)  // Pipeline register 2
    saved_sum <= sum;

  assign sum_div_3 = saved_sum * 14'b00000001010101;

  always @(posedge clk)  // Pipeline register 3
    avg <= sum_div_3;

endmodule
```

The nets and variables declared within the module are used for the intermediate results of the arithmetic operations and for the values saved in registers. The simple assignment statements model the arithmetic operations (two additions and a multiplication). We express the division by three as a multiplication by one-third (expressed as the binary fixed-point number 14'b00000001010101), as multipliers are generally simpler circuits than dividers. Moreover, some implementation fabrics have built-in multipliers that can be used. The three always blocks model the pipeline registers storing the intermediate results. Note that the first register actually stores two values together: the sum of a and b, and the input value c. If c were not saved in this way, the wrong value from the input stream c would be added by the second adder, rather than the value corresponding to the saved sum of a and b. Also note that the third register assigns directly to the output avg, as the value saved by the third register is the value required at the output.

The D flip-flop that we have considered so far is somewhat limited in its use, since it stores a new value on every rising edge of the clock input. Many systems only require a flip-flop to store a value when some controlling condition arises. For that, we can use an enhanced form of D flip-flop with a *clock-enable* input (sometimes call a *load-enable* input), illustrated in Figure 4.5. This flip-flop only updates the stored value when the CE input is 1 at the time of a rising clock edge. If the CE input is 0 on a rising clock edge, the flip-flop maintains the stored value unchanged. This behavior is shown in the timing diagram in Figure 4.6. As we mentioned in Section 1.3.6, the value on the data input must be stable for the setup time before and the hold time after the clock edge. A similar constraint applies to the clock-enable input. We say that the clock-enable input is a *synchronous control input,* meaning that it must be stable around a clock edge, and its effect is only acted upon when a clock edge occurs.

As with the simple D flip-flop, we can use multiple flip-flops with clock enable in parallel to form a register with clock enable. This form of register is probably the most common used in sequential digital systems, as it allows for storage of an intermediate result computed during one clock cycle to be used as an input to a subsequent computation any number of

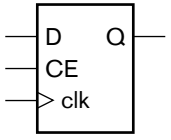

FIGURE 4.5 A D flip-flop with clock-enable input.

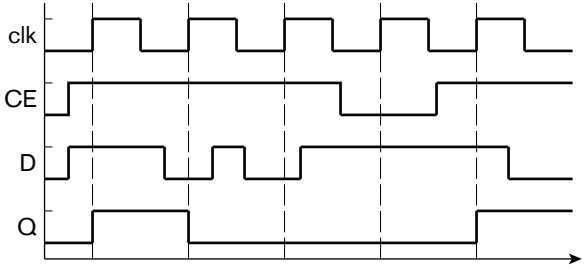

FIGURE 4.6 Timing diagram for a D flip-flop with clock enable.

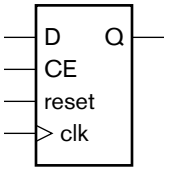

FIGURE 4.7 A D flip-flop with clock-enable and reset inputs.

clock cycles later. We will see in Section 4.3 how we can develop control conditions that govern when data is stored in registers.

We can model flip-flops and registers with clock enable inputs by extending the always-block template used to model simple D flip-flops and registers. The revised template is

```
always @(posedge clk)
  if (ce) q <= d;
```

The difference between this and the previous template is the addition of the if statement. When a rising edge occurs on the clk input, the output signal is only updated if the ce input is 1; otherwise, the stored value is unchanged. As before, the sizes of d and q determine whether the block models a single-bit flip-flop or a multibit register.

A further extension to the simple flip-flop involves adding an input to reset the stored value to 0. This is useful for ensuring that the flip-flop is initialized to a known state when power is first applied to a sequential circuit or when the circuit must be restarted from an initial state. Some circuits include a push button to allow the user to reset the circuit, for example, when it has encountered an error condition from which it cannot recover. Figure 4.7 shows a symbol for a flip-flop with both a clock-enable input and a reset input. The reset input overrides the clock-enable and data inputs. That is, when reset is 1, the stored value and the output Q are both changed to 0, regardless of the values on the CE and D inputs.

An important question to consider is the timing of changes on the reset input and when the reset operation occurs. There are two alternative behaviors, and a flip-flop with reset exhibits one or the other. The first reset behavior is called *synchronous reset*, and treats the reset input as a synchronous control input. This behavior is illustrated in Figure 4.8, in which the reset input causes the flip-flop to be reset on the first, fourth and fifth rising clock edges. Notice that, during the seventh clock cycle, reset changes to 1, but then changes back to 0 before a clock edge occurs. Since reset is 0 at the time of the next clock edge, the flip-flop is not reset.

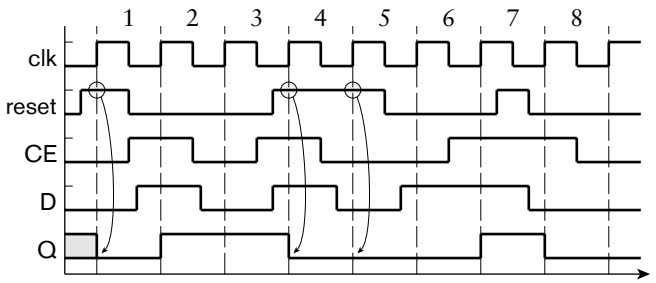

FIGURE 4.8 Timing diagram for a flip-flop with clock-enable and synchronous reset inputs.

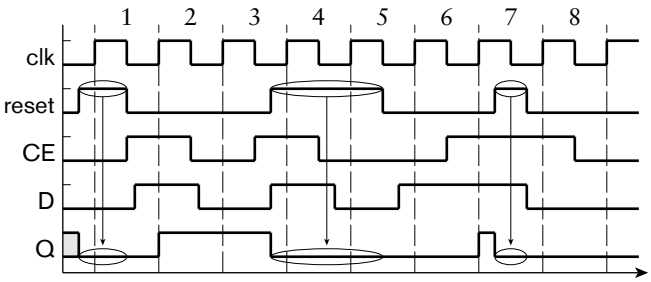

FIGURE 4.9 Timing diagram for a flip-flop with clock-enable and asynchronous reset inputs.

Notice also that we have shown the initial value of the Q output as neither 0 nor 1, but some unknown value, denoted by the grey shading. The fact that reset is 1 at the first clock edge forces the output to the known 0 value. Finally, we have ensured that the value of reset, like other data and control inputs is stable around each clock edge.

The second reset behavior for flip-flops is called *asynchronous reset*. In this case, the reset input is treated as an *asynchronous control input*, that is, when it changes to 1, it has an immediate effect regardless of the value of the clock or occurrence of clock edges. Moreover, the effect continues for as long as the reset input is 1. This behavior is illustrated in Figure 4.9. The timing of the inputs is the same as in Figure 4.8, but the output timing is different. At the start and in the third cycle, Q changes to 0 as soon as reset changes to 1, rather than waiting until the next clock edge. Furthermore, in the seventh cycle, the reset pulse that was ignored in the previous diagram takes effect in this case.

There is a potential problem that we should be aware of when designing circuits with asynchronous reset. The effect of changing the reset input from 1 back to 0 is to allow flip-flops to resume normal operation. However, if the change occurs close to a clock rising edge, the effect may occur at that edge or be delayed until the subsequent edge. This can cause problems in a system with numerous flip-flops, all of which are connected to the same clock and reset signals. Differences in the wiring delays can cause the change of reset from 1 to 0 to occur at slightly different times relative to clock edges for different flip-flops. Consequently, some flip-flops may be released from reset and resume storing values at one clock edge, whereas others might not resume until the subsequent clock edge, resulting in incorrect circuit operation. The solution to this problem is to ensure that the release of the reset signal from 1 to 0 always occurs synchronously with the clock; that is, to ensure that the change occurs sufficiently before a clock edge that the reset signal is stable around the edge for all flip-flops in the system.

The choice between synchronous and asynchronous reset may be influenced by the implementation fabric used for a design. Some fabrics only provide flip-flops with one or the other form of reset. Others, such as many FPGAs, allow us to program each flip-flop to use one or the other form of reset. Alternatively, the choice between the two forms of reset may be made by a system architect based on requirements for the design or the timing practices adopted for the design project. In that case, the chosen form of reset would be incorporated as a design specification for the subcircuits of the larger system. Generally, we should simplify the timing of a design by adopting one form of reset, either synchronous or asynchronous, uniformly throughout the design.

Just as we can use simpler flip-flops in parallel to form registers, so we can use flip-flops with reset in parallel. The result is a register that can be reset to a code word of all 0s. We can model flip-flops and registers with reset in Verilog by extending our previous always-block templates. The template for a flip-flop with synchronous reset and clock enable is

```
always @(posedge clk)
  if      (reset) q <= 1'b0;
  else if (ce)    q <= d;
```

On a rising clock edge, the block first checks whether the reset input is active, since this input has priority over all of the other logic in the flip-flop. If the reset input is active, the output is reset to 0. If we are modeling a multibit register, we would change the assignment to something like

```
q <= 6'b0;
```

to clear all output bits. The length of the vector will, of course, depend on the number of elements in the vector output signal. The remainder of the always-block template, after the test for reset, is the same as before. Only if `reset` is inactive does the block check the clock-enable input.

If we need to model a flip-flop or register with asynchronous reset, we need to take account of the fact that the reset input has an effect regardless of the value of the clock input. The always-block template for this kind of flip-flop is

```
always @(posedge clk or posedge reset)
  if      (reset) q <= 1'b0;
  else if (ce)    q <= d;
```

We have included the reset input in the event list of the block, since the block may need to update the outputs on a change of value of the reset input, not just on a change of value of the clock input. The revised block checks the value of the reset input first, before it looks at the clock input. If the reset input is 1, the block clears the output immediately. Only if the reset input is 0 does the block proceed to check for activity of the synchronous control input on a rising clock edge. As before, we can change the assignment to the output to reflect the difference between a single-bit flip-flop and a multibit register.

EXAMPLE 4.2 Develop a Verilog model for an accumulator that calculates the sum of a sequence of fixed-point numbers. Each input number is signed with 4 pre-binary-point and 12 post-binary-point bits. The accumulated sum has 8 pre-binary-point and 12 post-binary-point bits. A new number arrives at the input during a clock cycle when the data_en control input is 1. The accumulated sum is cleared to 0 when the reset control input is 1. Both control inputs are synchronous.

SOLUTION The module requires a clock input, two control inputs, a data input and a data output, as follows:

```
module accumulator
  ( output reg signed [7:-12] data_out,
    input      signed [3:-12] data_in,
    input                     data_en, clk, reset );

  wire signed [7:-12] new_sum;

  assign new_sum = data_out + data_in;

  always @(posedge clk)
    if      (reset)   data_out <= 20'b0;
    else if (data_en) data_out <= new_sum;

endmodule
```

The first assignment in the module models the addition of the accumulated sum (data_out) and the data input. The data input is implicitly sign-extended to match the size of the sum. The always block models the register used to accumulate the sum. It is based on the template for a register with synchronous reset and clock enable. When reset is 1, the block clears the register output, represented by the output variable data_out. If reset is 0, the block checks whether a new data value has arrived and been added to the sum. In that case, the register output is updated with the new sum; otherwise, it is unchanged.

We have now covered the main aspects of flip-flops and registers. There are other extensions, but they are just variations on the themes we have seen. One such variation is the addition of a control input to preset a flip-flop to 1. This is much like a reset control input, and may be either synchronous or asynchronous. Another variation is for the reset control input to use active-low logic, that is, for a 0 on the reset input to clear the stored data and output. Likewise, a preset control input might use active-low logic. A further variation is to use active-low logic for the clock input. This involves triggering a change of stored value on a falling edge of the clock signal rather than on a rising edge.

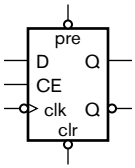

FIGURE 4.10 A negative-edge-triggered flip-flop.

EXAMPLE 4.3 The symbol in Figure 4.10 shows a negative-edge-triggered flip-flop with clock enable, negative-logic asynchronous preset and clear, and both active-high and active-low outputs. It is illegal for both preset and clear to be active together. Develop a Verilog model for this flip-flop.

SOLUTION The module definition is

```
module flip_flop_n ( output reg Q,
                     output     Q_n,
                     input      pre_n, clr_n, D,
                     input      clk_n, CE );

  always @( negedge clk_n or
            negedge pre_n or negedge clr_n ) begin
    if (!pre_n && !clr_n)
      $display("Illegal inputs: pre_n and clr_n both 0");
    if      (!pre_n) Q <= 1'b1;
    else if (!clr_n) Q <= 1'b0;
    else if (CE)     Q <= D;
  end

  assign Q_n = ~Q;

endmodule
```

We adopt the convention of appending "_n" to a name to indicate active-low logic. The always block models the flip-flop behavior. Since the pre_n and clr_n inputs are asynchronous control inputs, we include them, along with the clock input, in the event list of the block. Since they are all active-low inputs, we use negedge to specify that the block should respond to negative (falling) edges, that is, to changes from 1 to 0. Within the block, we check that the illegal condition described in the specification does not arise during use of the flip-flop in a circuit. The remainder of the block is based on the template for a flip-flop with asynchronous control. In this case, we have two asynchronous control inputs, so

we test them, one after the other, before checking for the synchronous clock-enable control input.

4.1.2 SHIFT REGISTERS

A register, as we have seen, stores data and makes it available at the output unchanged. A *shift register*, on the other hand, can perform a shift operation on the stored data. We described shift operations in Chapter 3, and showed how a shift operation has the effect of scaling a numeric value by a power of 2. As we will see in Chapter 8, shift operations are also used to implement serial transfer of data, that is, transfer one bit at a time over a single wire, instead of using separate wires for each of the bits of data. For now, we will just focus on use of shift registers to combine arithmetic scaling with storage functions.

Figure 4.11 shows a symbol for a shift register, and Figure 4.12 shows how it can be implemented with D flip-flops and multiplexers. The shift register is updated on a rising clock edge when CE is 1. In that case, when the load_en signal is 1, the multiplexers select new data on the D(n–1) through D(0) inputs for updating the register. Alternatively, when CE is 1 and load_en is 0, the multiplexers select the existing data, shifted right by one place. The least significant bit is discarded, and the most significant bit is updated with the value of the D_in signal. If we tie D_in to 0, the shift register performs a logical shift right operation on the stored data. Alternatively, if we connect the most significant output bit back to D_in, the shift register performs an arithmetic shift right operation. We will see in Chapter 8 how we connect the D_in input and the Q(0) output for serial transfer of data.

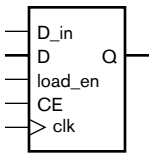

FIGURE 4.11 A symbol for a shift register.

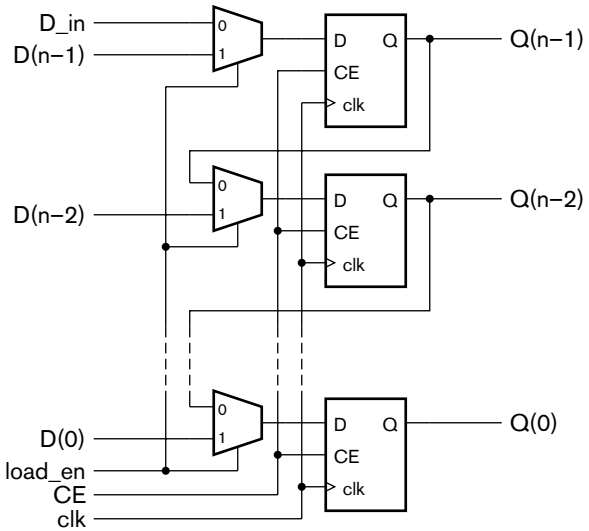

FIGURE 4.12 A shift register implemented with D flip-flops and multiplexers.

EXAMPLE 4.4 In Chapter 3, we showed how to perform multiplication of unsigned integers by addition of partial products. Construct a multiplier for two 16-bit operands containing just one adder that adds successive partial products over successive clock cycles. The final product is 32 bits.

SOLUTION In order to perform the operation over multiple cycles, we need a number of registers to hold intermediate results, as shown in Figure 4.13. The x operand is stored in an ordinary register whose output connects to an array of 16 AND gates that form a partial product. The y operand is stored in a shift register whose least significant bit, Q(0), controls the AND gates. The y operand is shifted on successive cycles, thus giving the 16 successive partial products. The sum of the partial products are accumulated in a 17-bit ordinary register and a 15-bit shift register. Since the shift register is never required to load data other than through the D_in connection, the data and load_en inputs are absent. On each clock cycle, the least significant bit of the ordinary register is shifted into the shift register, and the remaining bits of the ordinary register are added with the next partial product. By shifting the accumulated sum in this way, partial products are added at successively more significant positions of the result.

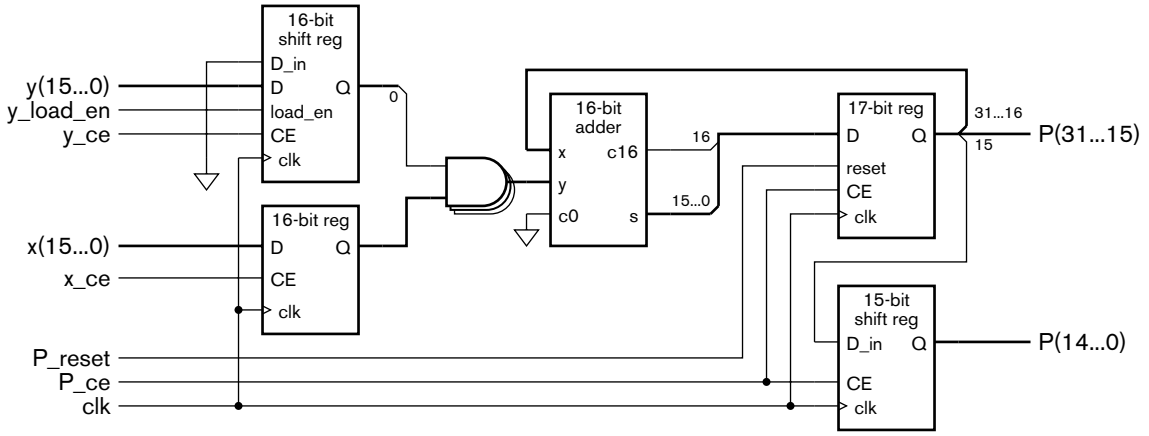

FIGURE 4.13 Registers, shift registers and other components used to form a sequential multiplier.

Making the sequential multiplier perform the required operations over successive clock cycles requires a separate control circuit. We will discuss control sequencing in detail in Section 4.3, and leave detailed design of the multiplier control to Exercise 4.20.

4.1.3 LATCHES

As we have seen, a flip-flop is a basic sequential circuit element that stores one bit. Most digital circuits use edge-triggered flip-flops that store a new data value when the clock signal changes from 0 to 1. No further values are stored while the clock remains at 1, nor when the clock returns to 0.

Some systems, however, use sequential elements called *latches*, with slightly different timing for storage of values. Figure 4.14 shows a symbol for a latch, and Figure 4.15 shows the timing behavior.

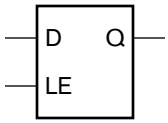

FIGURE 4.14 Symbol for a latch.

The latch has two inputs, a data input, D, and a latch-enable input, LE. It also has a data output, Q. When the latch-enable input is 1, the value at the data input is stored in the latch and transmitted through to the output. As the timing diagram shows, provided the data input remains unchanged for the entire time that the latch-enable input is 1, the behavior is the same as that of a flip-flop. However, if the data input changes while the latch-enable input is 1, the changed value is transmitted to the output. When the latch-enable input eventually changes to 0, the value stored in the latch just before the change is maintained in the latch and at the output. The fact that data is transmitted through to the output while the latch-enable input is 1 leads us also to use the name *transparent latch* for this component. While the latch-enable input is 1, what we see on the output is the value present on the input, so the latch appears to be transparent.

We can model a latch in Verilog using an always block of the form

```
always @(LE or D)
  if(LE) Q <= D;
```

This block includes both the latch-enable input and the data input in the event list. The notation or in the event list specifies that the block responds to changes on either input. However, it only updates the output Q when LE is 1. If the D input changes while the LE input is 1, the change on D is reflected on the output, modeling the transparent state of the latch. On the other hand, if D changes while LE is 0, the output is not assigned and maintains its previous value.

Just as we can implement multibit registers with flip-flops connected in parallel, so we can implement multibit latches with single-bit latches connected in parallel. The result is a latch in which multiple data bits flow through when the latch-enable input is 1 and are stored when the latch-enable input is 0.

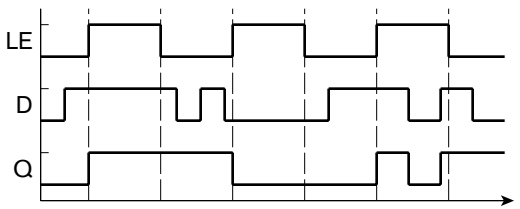

FIGURE 4.15 Timing diagram for a latch.

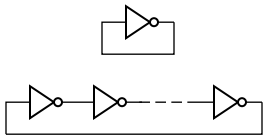

FIGURE 4.16 Inverters connected in feedback loops.

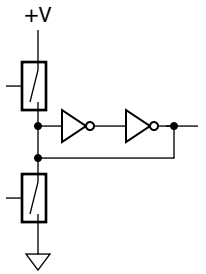

FIGURE 4.17 Using switches to force a node of an inverter ring to 0 or 1.

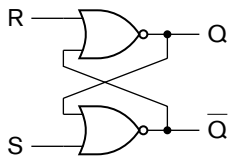

FIGURE 4.18 Cross-coupled RS-latch.

While latch circuits are relatively simple to implement in many fabrics, the fact that data can flow through them transparently can make it harder to design complex systems with correct timing behavior. The usual solution is to use two-phase nonoverlapping clock signals. Since this approach is not widely used now, the details are beyond the scope of this book. (See the books in Section 4.6, Further Reading.) However, we do need to consider how latching behavior can arise inadvertently from Verilog models, since it is a common design error.

First, let's return to our definition of a combinational logic circuit. We said that such a circuit is one whose outputs are defined purely as a function of the current input values, and that have no dependence on previous input values. The way in which a circuit's output can depend on previous input values is for the circuit to have a feedback path, that is, a cycle of connections from the output of a gate through other gates and back to the input of the gate. Perhaps the simplest such circuit is an inverter whose output is connected to its input, as shown at the top of Figure 4.16. Since the output of the inverter is the logical negation of its input, the output will oscillate between 0 and 1 with a frequency that is dependent on the propagation delay through the inverter. (Alternatively, the inverter may exhibit analog circuit behavior and reach an intermediate voltage level that is neither a valid logic low nor a valid logic high.) If we extend the feedback loop with more inverters to give an odd number of inverters in total (as shown at the bottom of Figure 4.16), we reduce the overall frequency of oscillation. This form of oscillator is called a *ring oscillator*. If we extend the ring to have an even number of inverters, the circuit will reach a stable state in which alternate inverters have a 0 at their output and the others have a 1. There are two possible stable states for such a ring of inverters. We could force the ring into one or other of the states by forcing a given node to 0 or 1, for example, by using switches as shown in Figure 4.17. (This is an idealization. In a real circuit, the switches would have some series resistance, thus avoiding damage to the output of the second inverter.) When both switches are open, the circuit retains the state into which is was forced. Hence, its output depends on the previous input value. This is a basic form of one-bit storage, called a *reset-set latch*, or *RS-latch* for short.

A more common implementation of an RS-latch uses cross-coupled gates, as shown in Figure 4.18. The timing behavior of the RS-latch is shown in Figure 4.19. Normally, the reset input R and the set input S are both 0. Assume initially that Q is 0 and $\overline{Q}$ is 1. This is a stable state, called the *reset state*. If the R input changes to 1 in this state, neither output changes and the latch stays in the reset state. However, if the S input changes to 1, $\overline{Q}$ changes to 0. This value is fed back to the other gate, which causes Q to change to 1. This is also a stable state, called the *set state*. When S returns to 0, the latch stays in the set state. Further changes

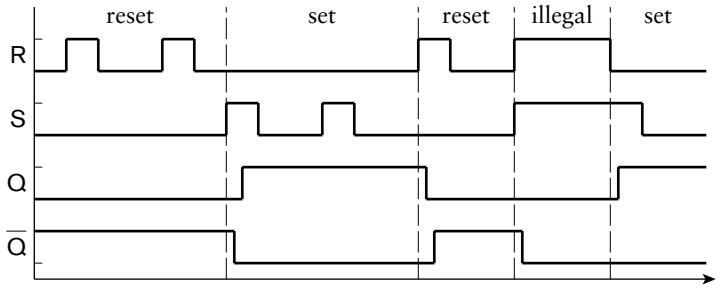

FIGURE 4.19 Timing for an RS-latch, showing the reset and set states, as well as an illegal operating condition.

of S to 1 while the latch is in the set state make no difference. However, if R goes to 1, the feedback causes the latch to change back to the reset state. Thus, which state the latch is in at any time depends on which of the S or R inputs was 1 most recently. Note that if both R and S are 1 at the same time, both Q and $\overline{Q}$ are 0. This is usually considered an illegal operating condition for an RS-latch.

Now that we have seen ways in which feedback can cause latching behavior, let's see how feedback can arise in Verilog models. In Chapter 2, we showed how a combinational circuit is modeled using an assignment statement in an architecture. Normally, we include the inputs to the circuit in the expression on the right-hand side of the assignment symbol and the output of the circuit on the left-hand side. However, if we have an assignment with a given net appearing both on the left-hand side and on the right-hand side, we imply a feedback loop from the output to the input. Most synthesis CAD tools will not synthesize such circuits without complaint, since the timing is not readily predictable and correct operation is not guaranteed. For example, if we write the following in a model:

```
assign a = a + b;
```

we imply an adder with the output feeding back directly into an input. In this sense, assignments modeling combination hardware in Verilog are different from assignments to variables in programming languages. Depending on the propagation delay through the synthesized and implemented circuit, we may add the value of b to itself once, twice, or more times within a given time interval. Moreover, if the delays are different for different bits, the result may not correspond to addition of the value of b at all. Most synthesis tools would either issue a warning or reject an assignment in the above form as erroneous.

A feedback loop can also be implied by a number of assignments in combination, where there is a cycle of dependencies between them. For example, consider the following assignments:

```
assign x = y + 1;
assign y = x + z;
```

Due to the first assignment, the value of x depends on the value of y. Due to the second assignment, the value of x depends on y, and thus indirectly on x itself. A synthesis tool should also issue a warning or flag this as erroneous.

The fact that synthesis tools object to feedback loops in combinational circuits can make it hard to model circuits in which we deliberately include such loops. For example, a Verilog model of the cross-coupled RS-latch of Figure 4.18 might be written as

```
assign Q   = ~(R | Q_n);
assign Q_n = ~(S | Q);
```

These assignments imply a cyclic dependency between Q and Q_n, which is exactly what we want in the synthesized circuit. An alternative way of modeling this behavior is to use an always block and an assignment, as follows:

```
always @(R or S)
  if      (R) Q <= 1'b0;
  else if (S) Q <= 1'b1;

assign Q_n = Q;
```

The assignment simply negates the value of Q, which is generated by the always block. In the block, we have included the R and S inputs in the event list. Thus, the block will be reactivated whenever either input changes. If R is 1, the block updates the Q output to represent the reset state, and if S is 1, the block updates the output to represent the set state. Note that, if neither input is 1, the block makes no assignment to Q. In that case, the outputs remain unchanged; that is, it stores the previously updated state. In general, if there is any execution path through an always block where we do not update an output, then the block represents latching behavior for that output, since the output maintains its previous value. If this is intended, as in the block modeling the RS-latch, we don't have a problem. However, it is a common Verilog modeling error to inadvertently omit an assignment

to an output in an execution path, for example, in one alternative of a complex if statement. The unintended latching behavior for that output can be most perplexing until the error is located and corrected.

EXAMPLE 4.5 The following always block is intended to model multiplexer circuitry that selects between a number of inputs to assign to outputs z1 and z2. Identify the error in the block and describe the behavior that results.

```
always @*
  if (~sel) begin
    z1 <= a1; z2 <= b1;
  end else begin
    z1 <= a2; z3 <= b2;
  end
```

SOLUTION The assignment to z3 in the "else" part of the if statement should assign to z2. As a consequence, z2 is not updated on that execution path and z3 is not updated on the execution path in which sel is 0. Thus, the block implies transparent latches for z2 and z3. The latch for z2 is transparent when sel is 0 and stores a value when sel is 1. The latch for z3 is transparent when sel is 1 and stores a value when sel is 0. This unintended behavior can be corrected simply by changing the target of the assignment from z3 to z2, as it should be.

KNOWLEDGE TEST QUIZ

1. Write a Verilog always block for a simple rising-edge-triggered register.
2. What do we call an arrangement of combinational subcircuits and registers that operate in assembly-line-like fashion?
3. What effect does a clock-enable input have on a register?
4. What is the distinction between an asynchronous reset and a synchronous reset?
5. What additional function does a shift register provide compared to an ordinary register?
6. What is meant by the term "transparent" with respect to a latch?
7. What problem is caused by omitting an assignment to an output in a Verilog always block that models combinational logic?

4.2 COUNTERS

A counter is a sequential component that increments or decrements a stored value. Counters occur in many digital circuit applications. For example, if an application requires a given operation to be performed on

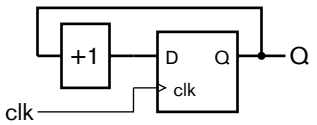

FIGURE 4.20 A simple counter composed of a register and an incrementer.

a number of items of data or to be repeated a number of times, a counter can be used to keep track of how many items have been processed or how many times the operation has been performed. Counters are also used as timers, by counting the number of intervals of a fixed duration that have passed.

A simple form of counter is composed of an edge-triggered register and an incrementer, as shown in Figure 4.20. The value stored in the register is interpreted as an unsigned binary integer. The incrementer can be implemented using the circuit we described for an unsigned incrementer in Section 3.1.2 on page 108. The counter increments the stored value on every clock edge. When the stored count value reaches its maximum value ($2^n - 1$, for an n-bit counter), the incrementer yields a result of all zeros, with the carry out being ignored. This result value is stored on the next clock edge. Thus, the counter acts like the odometer in a car, rolling over to zeros after reaching its maximum value. Mathematically speaking, the counter increments modulo 2^n. The counter goes through all 2^n unsigned binary integer values in order every 2^n clock cycles. One use for such a counter is in conjunction with a decoder to produce periodic control signals.

EXAMPLE 4.6 Design a circuit that counts 16 clock cycles and produces a control signal, ctrl, that is 1 during every eighth and twelfth cycle.

SOLUTION We need a 4-bit counter, since $16 = 2^4$. The counter counts from 0 to 15 and then wraps back to 0. During the eighth cycle, the counter value is 7 (0111_2), and during the twelfth cycle, the counter value is 11 (1011_2). We can generate the control signal by decoding the two required counter values and forming the logical OR of the decoded signals. The required circuit is shown in Figure 4.21.

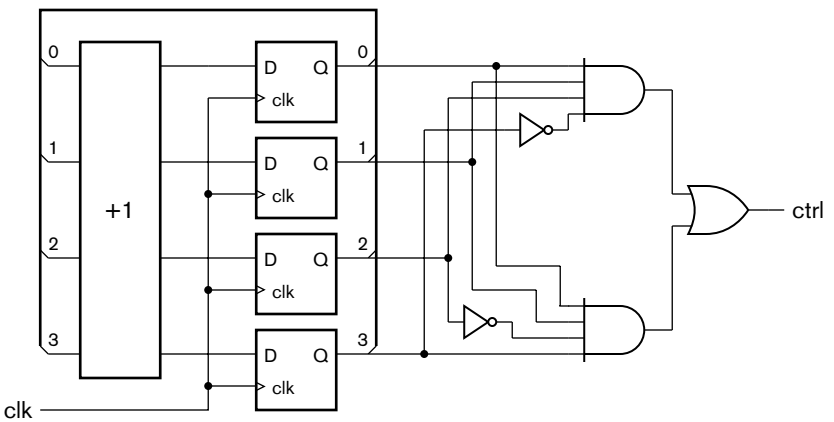

FIGURE 4.21 A counter with decoded outputs.

EXAMPLE 4.7 Develop a Verilog model of the circuit from Example 4.6.

SOLUTION The module definition is

```
module decoded_counter ( output ctrl,
                         input  clk );

  reg [3:0] count_value;

  always @(posedge clk)
    count_value <= count_value + 1;

  assign ctrl = count_value == 4'b0111 ||
                count_value == 4'b1011;

endmodule
```

The module contains an always block that represents the counter. It is similar in form to a block for an edge-triggered register. The difference is that the value assigned to the count_value output on a rising clock edge is the incremented count value. The assignment to count_value represents the update of the value stored in the register, and the addition of 1 represents the incrementer. The final assignment statement in the module represents the decoder.

The counter that we have described so far is free running, incrementing the count value on every clock cycle. We can modify the counter to make it useful in applications that require more control over the count value. Two simple modifications involve adding a clock enable and a reset input to the storage register within a counter. The clock-enable input allows us to control when the counter increments its value, so this input is often called a *count-enable* input. The reset input allows us to clear the count value back to zero. A counter modified in this way is shown in Figure 4.22. This form of counter is very useful for counting occurrences of events. We would connect a signal indicating event occurrence to the count-enable input of the counter. If we need to count events over several intervals, we can reset the counter at the start of each interval.

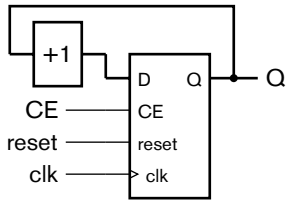

FIGURE 4.22 A counter with clock-enable and reset inputs.

Another modification is a *terminal-count* output. This is simply a decoded output that is 1 when the counter reaches is maximum, or terminal, value. For the counters we have described above, the maximum value of $2^n - 1$ is represented by a count value with all 1 bits. We can use an n-input AND gate to generate the terminal count output, as shown in Figure 4.23. For a free-running counter, the terminal-count output is 1 for a single clock cycle every 2^n clock cycles; that is, it is a periodic signal whose frequency is the input clock frequency divided by 2^n.

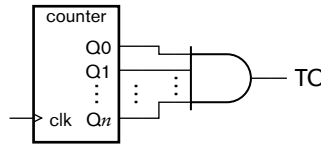

FIGURE 4.23 A counter with terminal-count output.

EXAMPLE 4.8 A digital alarm clock needs to generate a periodic signal at a frequency of approximately 500Hz to drive the speaker for the alarm tone. Use a counter to divide the system's master clock signal, with a frequency of 1 MHz, to derive the alarm tone.

SOLUTION We need to divide the master clock signal by approximately 2000. We can use a divisor of $2^{11} = 2048$, which gives us an alarm tone frequency of 488Hz, which is close enough to 500 Hz. Thus, we could use the terminal-count output of an 11-bit counter for the tone signal. However, the duty cycle (the ratio of time for which the signal is 1 to the time for which it is 0) would only be 1/2048, which would have very low AC energy. We can rectify this by dividing the master clock by 2^{10} with a 10-bit counter, and using the terminal-count output as the count-enable input to a divide-by-2 counter. A circuit is shown in Figure 4.24, and a timing diagram in Figure 4.25. The output of the divide-by-2 counter alternates between 0 and 1 for every pulse on its clock-enable input. The output thus has a 50% duty cycle, which will drive a speaker much more efficiently.

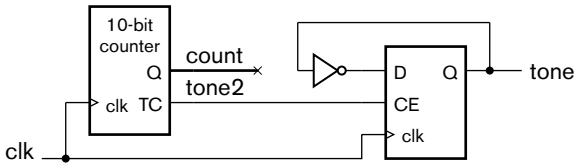

FIGURE 4.24 An alarm clock frequency divider.

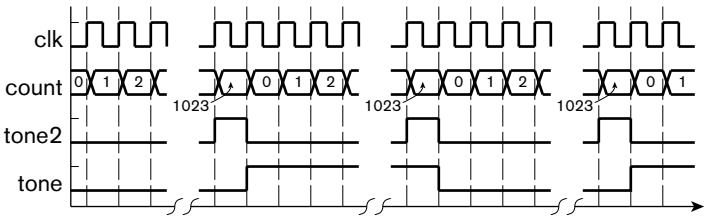

FIGURE 4.25 Timing diagram for an alarm clock frequency divider.

Not all free-running counter applications need to divide by a power of 2. If we need to divide by some other value, k, we need the counter to wrap back to 0 after reaching a terminal count of $k - 1$. Mathematically speaking, the counter increments modulo k. We can construct such a counter by decoding the unsigned binary code word for $k - 1$ and using that as the terminal count output. We can feed the terminal count signal back to a synchronous reset input to the storage register within the counter.

EXAMPLE 4.9 Design a circuit for a modulo 10 counter, otherwise known as a *decade counter*.

SOLUTION The maximum count value is 9, so we need 4 bits for the counter. The unsigned binary code word for 9 is 1001_2. We can decode this value and use it to reset to counter to 0 on the next clock cycle. The circuit is shown in Figure 4.26.

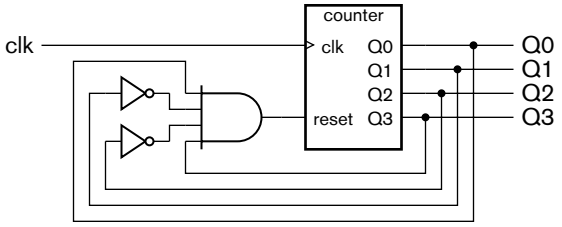

FIGURE 4.26 A decade counter.

EXAMPLE 4.10 Develop a Verilog model for the decade counter of Example 4.9.

SOLUTION The module definition is

```
module decade_counter ( output reg [3:0] q,
                        input            clk );

  always @(posedge clk)
    q <= q == 9 ? 0 : q + 1;

endmodule
```

We model the output port for the count value using an unsigned vector, since it is represents a binary-coded integer value. On a rising clock edge, the always block checks whether the counter has reached the terminal count value. If so, the count value wraps back to 0; otherwise, the block adds 1 to yield the new count value.

Another form of counter that is useful in timing applications is a *down counter with load*. This counter is loaded with an input value, and then decrements the count value. The terminal count output is activated when the count value reaches zero. A circuit for the counter is shown in Figure 4.27. It consists of a register whose input comes either from the input value to be loaded or from the decremented count value. In this case, the loading of input data is synchronous, since it occurs on a rising clock edge.

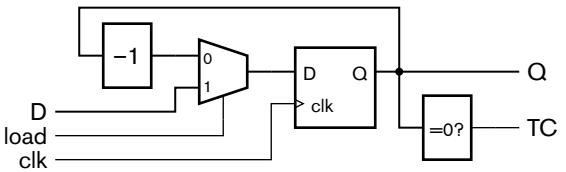

FIGURE 4.27 A down counter with synchronous load.

If the clock input to the counter is a periodic signal with period t and the counter is loaded with a value k, the terminal count is reached after an interval of $k \times t$. Thus, this form of counter can be used as an *interval timer*, where the terminal-count output signal is used to trigger an activity after expiration of a given time interval.

EXAMPLE 4.11 Develop a Verilog model for an interval timer that has clock, load and data input ports and a terminal-count output port. The timer must be able to count intervals of up to 1000 clock cycles.

SOLUTION The data input and counter need to be 10 bits wide, since that is the minimum number of bits needed to represent 1000. The module definition is

```
module interval_timer_rtl ( output      tc,
                            input [9:0] data,
                            input       load, clk );

  reg [9:0] count_value;

  always @(posedge clk)
    if (load) count_value <= data;
    else      count_value <= count_value - 1;

  assign tc = count_value == 0;

endmodule
```

On a rising clock edge, the always block uses the load input to determine whether to update the count value with the data input or the decremented count value. The decrement operation is performed using an unsigned subtraction without borrow out. So after reaching zero, the count value wraps back to the largest 10-bit value, namely, 1023. The final assignment in the architecture drives the terminal count to 1 when the count value reaches zero.

EXAMPLE 4.12 Modify the interval timer so that, when it reaches zero, it reloads the previously loaded value rather than wrapping around to the largest count value.

SOLUTION We need to use a separate register to store the data value to load into the counter. When the load input is activated, a new data value is loaded into the storage register as well as into the counter. When the terminal count is reached, the counter should be loaded from the storage register. The inputs and outputs of the revised interval timer are the same, so we don't need to change the ports of the module definition. The revised module is

```
module interval_timer_repetitive ( output      tc,
                                   input [9:0] data,
                                   input       load, clk );

  reg [9:0] load_value, count_value;

  always @(posedge clk)
    if (load) begin
      load_value <= data;
      count_value <= data;
    end
    else if (count_value == 0)
      count_value <= load_value;
    else
      count_value <= count_value - 1;

  assign tc = count_value == 0;

endmodule
```

In this module, we have added a separate variable, load_value, to represent the storage register. The always block is revised so that, when load is 1 on a rising clock edge, both the load_value variable and the count_value variable are updated from the data input. Also, when the count value is 0 on a rising clock edge (provided load is not 1), the count value is updated from the load_value variable. Otherwise, the count value is decremented as before.

The last kind of counter that we will describe in this section is a *ripple counter* (distinct from ripple carry used in an incrementer of a counter), shown in Figure 4.28. It is somewhat different in structure from the synchronous counters we have previously examined. Like those counters, it has a collection of flip-flops for storing the count value. However, unlike them, the clock signal is not connected in common to all of the flip-flop clock inputs. Rather, the clock input just triggers the flip-flop for the least significant bit, causing it to toggle between 0 and 1 on each rising clock edge. When the Q output changes to 0, the $\overline{Q}$ output changes to 1, triggering the next flip-flop to toggle between 0 and 1. This flip-flop behaves similarly, causing the third flip-flop to toggle when it (the second

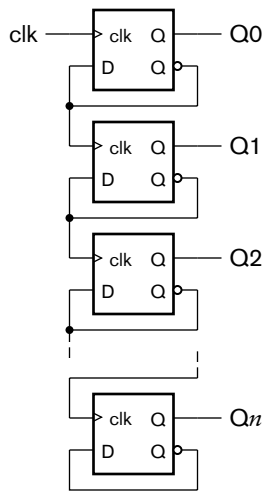

FIGURE 4.28 Structure of a ripple counter.

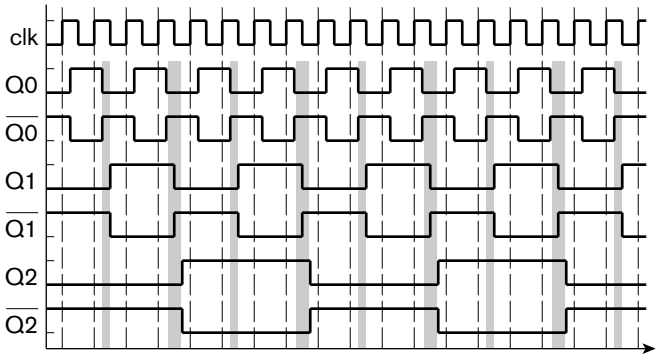

FIGURE 4.29 Timing diagram for a ripple counter.

flip-flop) changes from 1 to 0. In general, we can think of the flip-flops for bits 0 to $i-1$ as forming an i-bit counter. The most significant bit of this counter changes from 1 to 0 when it overflows. When that happens, the next flip-flop, for bit i, toggles between 0 and 1. This behavior is shown in the timing diagram of Figure 4.29.

An important timing issue arises from the fact that the flip-flops in a ripple counter are not all clocked together. Each flip-flop has a propagation delay between a rising edge occurring on its clock input and the outputs changing value. These propagation delays are shown in Figure 4.29. Since each flip-flop is clocked from the output of the previous flip-flop, the propagation delays accumulate. The outputs of the counter don't all change at once on a change of the counter's clock input. Instead, the output changes "ripple" along the counter as they propagate through the flip-flops; hence, the name of this kind of counter. The shaded areas in the timing diagram show intervals where the count value is not correct, due to changes not having propagated completely through the counter. Whether this lack of synchronization among output changes is a problem or not depends on the particular application under consideration. Some factors to consider include:

- The length of the counter. For longer counters, there are more flip-flops through which changes have to propagate, making the maximum accumulated delay larger. For short counters, the delay may be acceptable.
- The period of the input clock relative to the propagation delays of the counter. For a short clock period, the accumulated delay may exceed the clock period. In that case, there will be clock cycles during which the counter outputs don't reach the correct value before the end of the cycle. For systems with long clock periods, the count value will settle early in the clock cycle.

- The tolerance for transient incorrect count values. If the count value may be sampled before it has settled, incorrect operation may result. However, if the count value is not sampled until it is guaranteed settled, operation is correct.

The main advantages of a ripple counter are that it uses much less circuitry in its implementation (since an incrementer is not required) and that it consumes less power. Hence, it is useful in those applications that are sensitive to area, cost and power and that have less stringent timing constraints. As an example, a digital alarm clock might use ripple counters to count the time, since changes occur infrequently relative to the propagation delay (seconds compared to nanoseconds).

KNOWLEDGE TEST QUIZ

1. Show in a diagram how an incrementer and a register can be connected to form a simple counter.
2. What is the maximum count value for an n-bit counter? What value does it then advance to?
3. How is a modulo k counter constructed?
4. What is a decade counter?
5. What is an interval timer?
6. Why might a long ripple counter be unsuitable for an application with a fast clock?

4.3 SEQUENTIAL DATAPATHS AND CONTROL

We have now arrived at a key point in our discussion of digital logic design. We have seen how information can be binary encoded, how encoded information can be operated upon using combinational circuits, and how encoded information can be stored using registers. We have also seen that registers are needed both to avoid feedback loops in combinational circuits and to deal with data that arrives at the inputs sequentially. We have discussed counters as examples of combining registers and combinational circuits to perform sequential operations, that is, operations that proceed over a number of discrete intervals of time. We are now in a position to take a more general view of sequential operations. This general view will form the basis of our subsequent discussions of digital systems and embedded systems.

In many digital systems, the operations to be performed on input data are expressed as a combination of simpler operations, such as arithmetic operations and selection between alternative data values. Our general view of a digital system divides the circuit that implements the operations into a

datapath and a *control section*. The datapath contains the combinational circuits that implement the basic operations and the registers that store intermediate results. The control section generates *control signals* that govern the operation of the datapath elements: selecting operations to be performed and enabling registers. In particular, the control section ensures that control signals are activated in the right order and at the right times to cause the datapath to perform the required operations on the data flowing through it. Hence, we say that the control section performs *control sequencing*. In many cases, the control section makes use of *status signals* generated by the datapath. The status signals indicate whether certain conditions of interest are true, for example, whether data has certain values, or whether input data is available. The values of the status signals can influence the control sequence.

One of the most challenging tasks in digital design is designing a datapath and corresponding control section to meet the given requirements and constraints. There are usually many alternative datapaths that could meet the functional requirements. Choosing among them usually involves trading off between area and performance.

EXAMPLE 4.13 Develop a datapath to perform a complex multiplication of two complex numbers. The operands and product are all in Cartesian form. The real and imaginary parts of the operands are represented as signed fixed-point numbers with 4 pre-binary-point and 12 post-binary-point bits. The real and imaginary parts of the product are similarly represented, but with 8 pre-binary-point and 24 post-binary-point bits. The complex multiplier is subject to constraints that strongly limit the circuit area.

SOLUTION Given two complex numbers $a = a_r + ja_i$ and $b = b_r + jb_i$, the complex product is

$$p = ab = p_r + jp_i = (a_r b_r - a_i b_i) + j(a_r b_i + a_i b_r) \tag{4.1}$$

This computation requires four fixed-point multiplications, one subtraction and one addition. If we were to implement the complex multiplier as a combinational circuit, separate components would be needed for each of these operations, consuming a large amount of circuit area. Since area is a strong constraint, we can reduce the area by using one multiplier to perform the four multiplications in sequence, and one adder/subtracter to form the real and imaginary parts of the product. We will need registers to store the intermediate results. The full computation will take place over several clock cycles.

A datapath to perform the sequential complex multiplication is shown in Figure 4.30. Since the multiplier is shared, multiplexers at the multiplier inputs are needed to select the operands. The result of a given multiplication is stored in one or other of the partial-product registers. To form the real part of the complex product, two partial products are subtracted by the adder/subtracter.

To form the imaginary part, two partial products are added. In each case, the part of the complex product is stored in an output register.

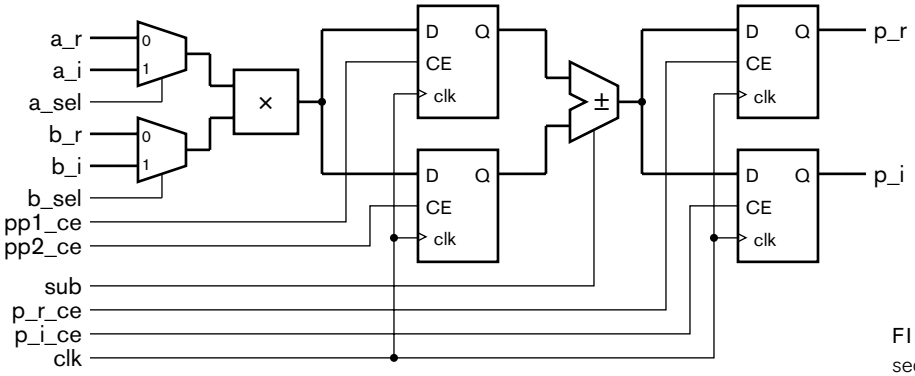

FIGURE 4.30 Datapath for a sequential complex multiplier.

In the diagram, the signals upon which data flows are drawn with thicker lines, since they carry multibit binary-coded values. The remaining signals, drawn with lighter weight lines, are the clock and the control signals. They include select signals for the multiplexers, clock-enable signals for the registers, and a signal to choose the operation to be performed by the adder/subtracter. The values of the control signals are driven by a separate control section, not shown on the diagram.

EXAMPLE 4.14 Develop a Verilog model of the complex multiplier datapath.

SOLUTION The module includes ports for the data inputs and outputs, as well as clock and reset inputs and an input to indicate the arrival of new data. We will return to the last of these inputs later. The module definition is

```
module multiplier
  ( output reg signed [7:-24] p_r, p_i,
    input      signed [3:-12] a_r, a_i, b_r, b_i,
    input                     clk, reset, input_rdy );

  reg a_sel, b_sel, pp1_ce, pp2_ce, sub, p_r_ce, p_i_ce;

  wire signed [3:-12] a_operand, b_operand;
  wire signed [7:-24] pp, sum;
  reg  signed [7:-24] pp1, pp2;

  ...
```

(continued)

```
  assign a_operand = ~a_sel ? a_r : a_i;
  assign b_operand = ~b_sel ? b_r : b_i;

  assign pp = {{4{a_operand[3]}}, a_operand, 12'b0} *
              {{4{b_operand[3]}}, b_operand, 12'b0};

  always @(posedge clk)  // Partial product 1 register
    if (pp1_ce) pp1 <= pp;

  always @(posedge clk)  // Partial product 2 register
    if (pp2_ce) pp2 <= pp;

  assign sum = ~sub ? pp1 + pp2 : pp1 - pp2;

  always @(posedge clk)  // Product real-part register
    if (p_r_ce) p_r <= sum;

  always @(posedge clk)  // Product imaginary-part register
    if (p_i_ce) p_i <= sum;

  ...

endmodule
```

The nets and variables declared within the module represent the control signals and the internal data connections. There are further declarations for the control section that we will return to later. In the statement part of the architecture, the assignments to a_operand and b_operand represent the multiplexers, and the assignment to pp represents the multiplier. (The multiplier operands are extended so that the result size matches the sizes of the real and imaginary parts of the product.) The first two always blocks represent the partial-product registers. The assignment to sum represents the adder/subtracter, and the second two always blocks represent the output registers. We will return to further statements that represent the control section later.

EXAMPLE 4.15 Design a control sequence for the control signals of the sequential complex multiplier.

SOLUTION We first need to determine a sequence of operations to be performed by the datapath to implement the required function expressed in Equation 4.1. There are many possible sequences, but we must ensure that there is no conflict for resources; that is, we must ensure that we don't try to use an element of the datapath for more than one operation at a time. One possible sequence, initiated by input_rdy being 1, is:

1. Multiply a_r and b_r, and store the result in partial product register 1.
2. Multiply a_i and b_i, and store the result in partial product register 2.

3. Subtract the partial product register values and store the result in the product real part register.
4. Multiply a_r and b_i, and store the result in partial product register 1.
5. Multiply a_i and b_r, and store the result in partial product register 2.
6. Add the partial product register values and store the result in the product imaginary part register.

This sequence would take six clock cycles to complete. In each cycle, only one of the arithmetic components is used, so there is no conflict for resources. However, we can reduce the number of cycles required, without creating conflict, by using the multiplier and the adder/subtracter concurrently. Specifically, we can merge steps 3 and 4 into one step, in which we subtract partial products to form the real part of the product and we multiply a_r and b_i to form a further partial product.

Given this 5-step sequence, the control signals that need to be activated in each step are shown in Table 4.1. The combination of control signal values in each step cause the datapath components to perform the required operations for that step. Note that in some steps, the multiplexers and adder/subtracter are not used. We don't care what values are driven for the control signals governing those components in those steps.

step	a_sel	b_sel	pp1_ce	pp2_ce	sub	p_r_ce	p_i_ce
1	0	0	1	0	–	0	0
2	1	1	0	1	–	0	0
3	0	1	1	0	1	1	0
4	1	0	0	1	–	0	0
5	–	–	0	0	0	0	1

TABLE 4.1 Control sequence for the complex multiplier.

4.3.1 FINITE-STATE MACHINES

Example 4.15 describes a control sequence for a sequential datapath, but we have yet to show how to design a circuit for the control section that generates the control sequence. We will introduce an abstraction called a *finite-state machine* for this purpose. There is a substantial body of mathematical theory underlying finite-state machines. Some of the useful results from this theory are implemented in CAD tools that transform finite-state machines to optimize sequential circuits. However, we will take a pragmatic approach, focusing on the design of control sections to sequence the operation of datapaths.

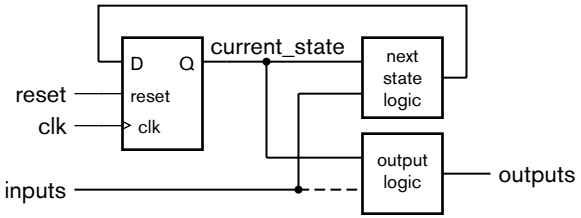

FIGURE 4.31 Circuit structure for a finite-state machine.

In general terms, a finite-state machine is defined by a set of *inputs*, a set of *outputs*, a set of *states*, a *transition function* that governs transitions between states, and an *output function*. The states are just abstract values that mark steps in a sequence of operations. The machine is called "finite-state" because the set of states is finite in size. The finite-state machine has a *current state* in a given clock cycle. The transition function determines the *next state* for the next clock cycle based on the current state and, possibly, the values of inputs in the given clock cycle. The output function determines the values of the outputs in a given clock cycle based on the current state and, possibly, the values of inputs in the given clock cycle.

Figure 4.31 shows a schematic representation of a finite-state machine. The register stores the current state in binary coded form. One of the states in the state set is designated the *initial state*. When the system is reset, the register is reset to the binary code for the initial state; thus, the finite-state machine assumes the initial state as its current state. During each clock cycle, the value of the next state is computed by the next state logic, which is a combinational circuit that implements the transition function. Also, the outputs are driven with the value computed by the output logic, which is a combinational circuit that implements the output function. The outputs are the control signals that govern operation of a datapath. On the rising clock edge marking the beginning of the next clock cycle, the current state is updated with the computed next-state value. The next state may be the same as the previous state, or it may be a different state.

Finite-state machines are often divided into two classes. In a *Mealy* finite-state machine, the output function depends on both the current state and the values of the inputs. In such a machine, the connection drawn with a dashed line in Figure 4.31 is present. If the input values change during a clock cycle, the output values may change as a consequence. In a *Moore* finite-state machine, on the other hand, the output function depends only on the current state, and not on the input values. The dashed connection in Figure 4.31 is absent in a Moore machine. If the input values change during a clock cycle, the outputs remain unchanged.

In theory, for any Mealy machine, there is an equivalent Moore machine, and *vice versa*. However, in practice, one or the other kind of machine will be most appropriate. A Mealy machine may be able to implement a given control sequence with fewer states, but it may be harder to meet timing constraints, due to delays in arrival of inputs used to compute the next state. As we present examples of finite-state machines, we will identify whether they are Mealy or Moore machines.

In many finite-state machines, there is an idle state that indicates that the system is waiting to start a sequence of operations. When an input indicates that the sequence should start, the finite-state machine follows a sequence of states on successive clock cycles, with the output values controlling the operations in a datapath. Eventually, when the sequence of operations is complete, the finite-state machine returns to the idle state.

EXAMPLE 4.16 Design a finite-state machine to implement the control sequence for the complex multiplier described in Example 4.15. The control sequence is initiated by input_rdy being 1 during the clock cycle in which new data arrives at the datapath inputs.

SOLUTION Our finite-state machine needs five states, one for each of the steps of the control sequence. Let's call them step1 through step5. We also need to deal with the case of waiting for input data to arrive. We could consider a separate idle state for that case. When, in the idle state, input_rdy is 1, we would then transition to state1 to start the multiplication; otherwise, we would stay in the idle state. The problem with this is that it wastes a clock cycle, since we would not perform the first multiplication until after the cycle in which data arrived.

The alternative is to use step1 as the idle state. If it turns out that new data has not arrived in a given clock cycle while in this state, we simply repeat step1 as the next state. On the other hand, if new data has arrived, indicated by input_rdy being 1 in the clock cycle, the real parts are multiplied during that clock cycle and can be stored on the next clock edge. We would then transition to step2, and on subsequent clock cycles to step3, step4 and step5. At the end of the step5 clock cycle, the complete complex product is stored in the output registers of the datapath, so we can transition back to step1 in the next clock cycle.

In summary, our finite-state machine has the signal input_rdy as its single input, and the control signals listed in Example 4.15 as outputs. The state set is {step1, step2, step3, step4, step5}, with step1 being the initial state. The transition function is defined in Table 4.2. The output function is defined in Table 4.1. Since the output function depends only on the current state and not on the input value, this finite-state machine is a Moore machine.

---

current_ state	input_ rdy	next_ state
step1	0	step1
step1	1	step2
step2	–	step3
step3	–	step4
step4	–	step5
step5	–	step1

TABLE 4.2 The transition function for the complex multiplier finite-state machine.

An important issue to consider when designing a finite-state machine is how to encode the state values. We glossed over that in Example 4.16 by treating the states as abstract values. As we discussed in Chapter 2, if we have N states, we need at least $\lceil \log_2 N \rceil$ bits in our code. However, we may choose to have more if that simplifies circuitry that uses encoded states. In particular, while a longer than minimal code length requires more flip-flops in the state register and more wires for the state signals, it may make the next-state and output logic circuits simpler and smaller. In general choosing an optimal state encoding is a complex mathematical problem. However, synthesis CAD tools incorporate methods for choosing a state encoding, so we may be able to let a tool make the choice for us. One aspect of state encoding is the choice of a code word to represent the initial state. In many cases, a good choice is a code word with all 0 bits, since that allows us to use a simple register with reset for the state register. If some other code word is chosen for the initial state, that code word must be loaded into the register on system reset.

Modeling Finite-State Machines in Verilog

Since a finite-state machine is composed of a register, next-state logic and output logic, a straightforward way to model a finite-state machine is to use the Verilog features that we already know for modeling registers and combinational logic. The only aspect we have not addressed is how to represent the state set, particularly when we want to take an abstract view and leave state encoding to the synthesis tool. In Verilog, we can use *parameter definitions* to specify a set of symbolic names associated with the binary code words for the states. For example, we can define parameters for the states in Example 4.16 as follows:

```
parameter [2:0] step1 = 3'b000, step2 = 3'b001,
                step3 = 3'b010, step4 = 3'b011,
                step5 = 3'b100;
```

This defines five parameters, named `step1` through `step5`, corresponding to the binary code words 000 through 100, respectively. In the rest of the state machine model, we just use the symbolic names, not the code word values. A synthesis tool may be able to recode the state parameters, that is, to choose an alternate encoding for the state set, to optimize the generated hardware for the state machine.

We can declare a variable to represent the current state of a state machine as follows:

```
reg [2:0] current_state;
```

This specifies that current_state is a vector that can take on parameter values representing states. So, for example, we could make the following assignment in a procedural block:

```
current_state <= step4;
```

to assign the value step4 to the variable.

EXAMPLE 4.17 Develop a Verilog model of the finite-state machine in Example 4.16.

SOLUTION We will augment the architecture declaration of Example 4.14 with the Verilog representation of the control section. The additional declarations of parameters for the set of states and variables for the current and next state are

```
parameter [2:0] step1 = 3'b000, step2 = 3'b001,
                step3 = 3'b010, step4 = 3'b011,
                step5 = 3'b100;
reg [2:0] current_state, next_state ;
```

The additional statements added to the module are

```
 always @(posedge clk or posedge reset)  // State register
   if (reset) current_state <= step1;
   else       current_state <= next_state;

  always @*  // Next-state logic
   case (current_state)
     step1: if (!input_rdy) next_state = step1;
            else            next_state = step2;
     step2:                 next_state = step3;
     step3:                 next_state = step4;
     step4:                 next_state = step5;
     step5:                 next_state = step1;
   endcase

  always @* begin  // Output_logic
   a_sel = 1'b0; b_sel = 1'b0;  pp1_ce = 1'b0; pp2_ce = 1'b0;
   sub = 1'b0;   p_r_ce = 1'b0; p_i_ce = 1'b0;
   case (current_state)
     step1: begin
              pp1_ce = 1'b1;
            end
```

(continued)

```
      step2: begin
               a_sel = 1'b1; b_sel = 1'b1; pp2_ce = 1'b1;
             end
      step3: begin
               b_sel = 1'b1; pp1_ce = 1'b1;
               sub = 1'b1;   p_r_ce = 1'b1;
             end
      step4: begin
               a_sel = 1'b1; pp2_ce = 1'b1;
             end
      step5: begin
               p_i_ce = 1'b1;
             end
  endcase
end
```

The first always block models the state storage for the finite-state machine. It is based on the template for a register with asynchronous reset. When the reset input is active, the block resets the current state to the initial state, step1. Otherwise, on a rising clock edge, the block updates the current state with the computed next state.

The next state is computed by the second always block, which models the transition function of Table 4.2. The statement inside the block is a *case statement*. It uses the value of the current_state variable to choose among alternatives for updating next_state. The alternative for step1 uses a nested if statement to determine whether to proceed to step2 or stay in step1, depending on the value of input_rdy. All other alternatives simply advance the state unconditionally.

The output values are computed by the third always block, which models the output function of Table 4.1. This block also includes a case statement that chooses alternatives for assigning values to the outputs depending on the value of current_state. Rather than including an assignment for every output in each alternative of the case statement, we precede the case statement with a default assignment of 0 for each output, and only include overriding assignments of 1 in those alternatives where they are required. This style for modeling the output function usually makes the always block more succinct, and helps to avoid inadvertent introduction of latches due to omission of an output assignment in an alternative.

State Transition Diagrams

A *state transition diagram* is an abstract diagrammatic representation of a finite-state machine. It uses a circle, or "bubble," to represent each state. Directed arcs between state bubbles represent transitions from one state to another. An arc may be labeled with a combination of input values

that allow the transition to occur. To illustrate, Figure 4.32 shows a state transition diagram for a finite-state machine with states s1, s2 and s3. Each arc is labeled with the values of two inputs, a_1 and a_2, that are required for the transition. Thus, when the finite-state machine is in state s1 and the inputs are both 1, the state of the machine in the next clock cycles is s3. If the machine is in state s1 and both inputs are 0, the machine stays in state s1. From state s1, if the inputs are 0 and 1, or 1 and 0, the machine transitions to state s2. Note that we have omitted a label on the arc from s2 to s3. This is a common convention to indicate an unconditional transition; that is, when the machine is in state s2, the next state is s3 regardless of the input values. Another important point is that all possible combinations of input values are accounted for in each state, and that no combination is repeated on more than one arc from a given state.

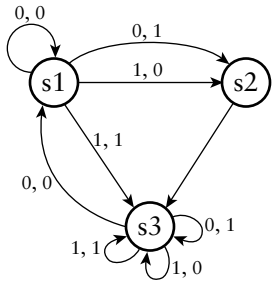

FIGURE 4.32 A state transition diagram.

A bubble diagram may also be labeled with the values of outputs. Since Moore-machine outputs depend only on the current state, we attach the labels for such outputs to the state bubbles. This is shown on the augmented bubble diagram in Figure 4.33. For each state, we list the values of two Moore-style outputs, x_1 and x_2, in that order.

Mealy-machine outputs, on the other hand, depend on both the current state and the current input values. Usually, the input conditions are the same as those that determine the next state, so we usually attach Mealy-output labels to the arcs. This does not imply that the outputs change at the time of the transition, only that the output values are driven when the current state is the source state of the arc and the input values are those of the arc label. If the inputs change while in the source state, the outputs change to those listed on some other arc labeled with

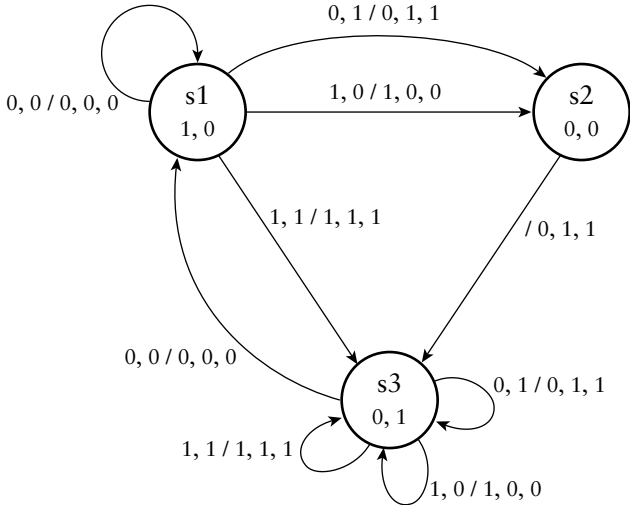

FIGURE 4.33 A state transition diagram augmented with Moore- and Mealy-style output values.

the new input values. Mealy-style outputs are also shown on the arcs in Figure 4.33. In each case, the output values are listed after the "/" in the order y_1, y_2 and y_3.

EXAMPLE 4.18 Draw a state transition diagram for the finite-state machine of Example 4.16. Include the output values in the order of their occurrence in Table 4.1.

SOLUTION The diagram is shown in Figure 4.34. There is a transition from step1 to step2 that occurs when input_rdy is 1, and a transition from step1 back to itself when input_rdy is 0. All other transitions are unconditional. Since it is a Moore machine, the output values are all drawn in the state bubbles.

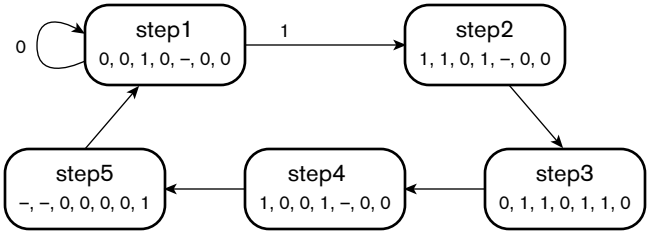

FIGURE 4.34 State transition diagram for the complex multiplier.

In many applications, a state transition diagram is a useful notation, since it graphically conveys the control organization of a sequential design. Many CAD tools provide graphical editors for entering state transition diagrams, and can automatically generate Verilog code for simulation and synthesis. The disadvantage of the notation is that the annotations of input conditions and output values can clutter the diagram, obscuring the control organization. Also, for large and complex state machines, the diagram can become unwieldy. In those cases, a Verilog model in textual form may be more intelligible. Ultimately, since state transition diagrams and Verilog models of state machines encapsulate the same information, it is a question of personal preference or project guidelines that determine the method to use.

KNOWLEDGE TEST QUIZ

1. What is the purpose of the datapath in a digital system?
2. What is the purpose of the control section in a digital system?
3. What are control signals and status signals?
4. What is the distinction between a Moore and a Mealy finite-state machine?

5. Write a Verilog parameter definition for the set of states s0, s1, s2 and s3.
6. In a state transition diagram, where are labels written for Mealy-style outputs and for Moore-style outputs?

4.4 CLOCKED SYNCHRONOUS TIMING METHODOLOGY

We now have a general view of a digital system, shown in Figure 4.35. It comprises a datapath that stores and transforms binary-coded information and a control section that sequences operations within the datapath. The datapath, in turn, includes combinational subcircuits that perform operations on the data and registers that store the data. Stored data can be fed back to earlier stages of the datapath or fed forward to subsequent stages. The control section drives the control signals that govern operation of the combinational subcircuits and storage of data in the registers. The control section can also use status information about the data values to determine what operations to perform and in what sequence. Given that data is transferred between registers through combinational subcircuits, this view of a system is often called a *register-transfer level* (RTL) view. The word "level" refers to the level of abstraction. Register-transfer level is more abstract than a gate-level view, but less abstract than an algorithmic view.

In Chapter 1, we identified division of time into discrete intervals as a key abstraction for managing the complexity of timing in digital systems. We also described some of the specific timing characteristics of flip-flops (and hence registers) over which the discrete-timing approach abstracts. Now that we have seen some more complex digital systems, we can begin to see the value of the discrete-timing abstraction. It is based on driving all of the registers shown in Figure 4.35 with a common periodic clock signal. We say that the registers are all clocked *synchronously* on each rising clock edge. The combinational subcircuits perform their operations in the interval between one clock edge and the next, called a *clock*

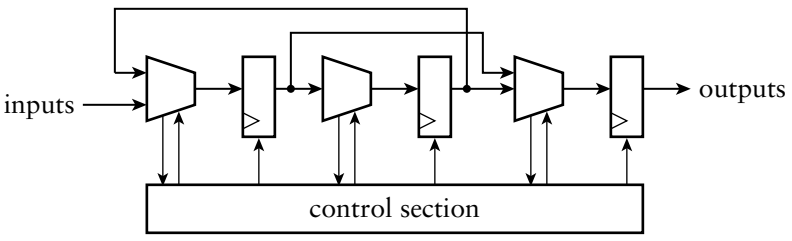

FIGURE 4.35 A general view of a digital system.

cycle. This *clocked synchronous timing methodology* helps us ensure that operations are completed by combinational subcircuits by the time their results are needed, and simplifies composition of large systems from smaller subsystems.

Since registers are composed of flip-flops connected in parallel, we can derive the timing characteristics of registers from those of flip-flops. We will make the simplifying assumption that all of the flip-flops in a given register have the same timing characteristics, or that any differences are negligible. We can thus identify the setup time (t_{su}), hold time (t_h) and clock-to-output delay (t_{co}) of a register as being the same as those characteristics of the constituent flip-flops. All of the bits of data to be stored in a register must be stable at the input for at least the setup time before a clock edge and for at least the hold time after the clock edge. We can only guarantee that all bits of the stored data will be available at the output after the clock-to-output delay following the clock edge.

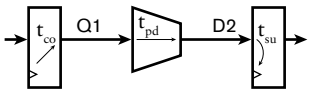

FIGURE 4.36 A register-to-register path.

These considerations lead us to the register-to-register timing for a path in the system shown in Figure 4.36. Q1 is the output of one register that feeds into a combinational subcircuit. D2 is the output of the subcircuit, feeding into the next register. The timing parameters are illustrated in Figure 4.37. After a clock rising edge, Q1 changes to the new stored value and stabilizes by the end of the interval t_{co}. The new value then propagates through the combinational subcircuit, stabilizing at the output D2 by the end of the interval t_{pd}, the propagation delay of the subcircuit. The value on D2 must be stable at least t_{su} before the next clock edge, so there is a slack period, t_{slack}, where nothing changes. The diagram shows that the sum of these intervals must be equal to the clock cycle time, t_c. Alternatively, we can express this as an inequality:

$$t_{co} + t_{pd} + t_{su} < t_c \tag{4.2}$$

Another important path in the digital system is the control path shown in Figure 4.38. At the top of the figure is a register-to-register section of the datapath, and at the bottom is the finite-state machine in

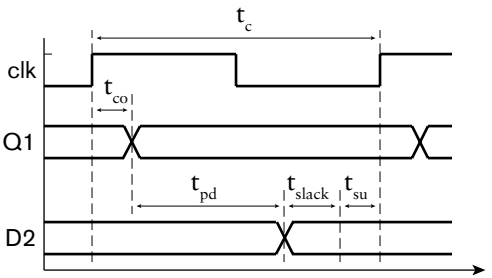

FIGURE 4.37 Register-to-register timing.

the control section. The status signals driven by the combinational subcircuit are inputs to the output logic and next-state logic in the control section. The control signals driven by the output logic govern the operation of the combinational subcircuit and the target register. (In general, status signals from one combinational subcircuit would influence operation of some other combinational subcircuit, but the same timing considerations apply.) Our timing analysis for these control paths is similar to that for the register-to-register datapath. We simply aggregate the combinational propagation delays through the combinational subcircuit and output logic to derive the inequality:

$$t_{co} + t_{pd\text{-}s} + t_{pd\text{-}o} + t_{pd\text{-}c} + t_{su} < t_c \tag{4.3}$$

Here, t_{pd-s} is the propagation delay through the combinational subcircuit to drive the status signals, t_{pd-o} is the propagation delay through the output logic to drive the control signals, and t_{pd-c} is the propagation delay through the combinational subcircuit for a change in the control signal to affect the output data. For a Moore-style control signal that does not depend on a status input, we can ignore the parameter t_{pd-s} in this inequality. In a similar way, we can derive the following inequality for the path that generates the next-state value:

$$t_{co} + t_{pd\text{-}s} + t_{pd\text{-}ns} + t_{su} < t_c \tag{4.4}$$

where t_{pd-ns} is the propagation delay through the next-state logic.

The inequalities in Equations 4.2 through 4.4 must hold for all of the register-to-register and control paths in the system. Since the clock is common to all registers, t_c is the same for all paths. Similarly, if we assume that the same kinds of registers are used throughout the system (which is the case in fabrics such as FPGAs), t_{co} and t_{su} are the same for all paths. That only leaves the propagation delay parameters as the difference among paths.

The path with the longest propagation delay is called the *critical path*. It determines the shortest possible clock cycle time for the system. Since all operations are performed in times determined by the clock, the critical path determines the overall system performance. Hence, if we need to address performance issues, we need to identify which combinational subcircuit is on the critical path and attempt to reduce its delay. In most systems, the critical path will be a register-to-register path in the datapath of the system. For example, if there is such a path that performs an arithmetic operation or that includes a counter, the carry chain may be the critical path. Alternatively, if a system uses a Mealy finite-state machine and a control path corresponding to Equation 4.3 is on the critical path, it may be possible to use an equivalent Moore machine to avoid the status-signal delay in the control path. Of course, once the delay on the critical path is reduced below that of another path, that other path becomes the critical

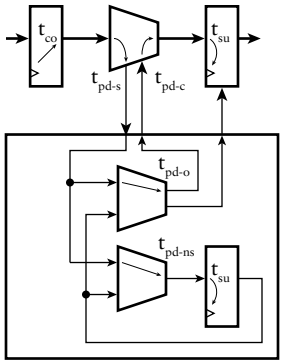

FIGURE 4.38 Control path in a digital system.

path. Hence, attention may need to be paid to several paths in a system to address performance issues.

Depending on the requirements and constraints for the system, we can interpret Equations 4.2 through 4.4 in two ways. One interpretation involves treating the propagation delays as independent parameters and determining the resulting minimum clock period. The system can then be operated with any clock period greater than the minimum. This interpretation is appropriate for systems where high performance is not a requirement.

The other interpretation involves treating the clock cycle time as the independent parameter and determining the propagation delays from it. We might be given a target clock cycle time by a system architect or our marketing department and be asked to design the system to meet that target. In that case, the inequalities place constraints on the propagation delays through the combinational data and control paths. If we meet the constraints with plenty of slack, we might try to optimize the design to reduce cost, for example, by using subcircuits with less area. If we don't meet the constraints, we need to focus attention on the critical path or paths to reduce their delay. It may be that we have designed the system with too much computation to be performed in one or more combinational subcircuits to allow sufficient reduction of the critical path propagation delay. In that case, we could divide the computation into a number of smaller steps that can be done sequentially or in parallel. The combinational subcircuits for the simpler steps should have smaller propagation delay than the original. Thus, even if more steps are required overall to perform the system's operation, the fact that the clock cycle time is reduced may allow us to meet our performance target.

EXAMPLE 4.19 Suppose we have designed a system that includes a multiplication operation on 16-bit unsigned binary-coded integers. The system is required to operate at 50 MHz (a clock cycle time of 20 ns). We have included a combinational multiplier to perform the multiplication, but its propagation delay is 35 ns. All other data and control paths have plenty of slack with the 20 ns clock cycle time. The result of the multiplication is not needed until 20 cycles after the operands are available. Describe how use of the sequential multiplier of Example 4.4 could help us meet our timing requirement.

SOLUTION The sequential multiplier performs the multiplication operation in 17 steps with one adder. In the first step, we store the operands and reset the output register to zero. Then on each of the 16 subsequent steps, we add the partial products. Each step involves only an AND operation and an addition. Thus, the combinational subcircuit between the operand registers and the product output registers will have significantly smaller propagation delay than the 35 ns delay of the full combinational multiplier. This reduction should allow the clock period to be reduced to meet the timing constraint.

Further timing considerations arise from the way the clock signal is connected to all of the registers in a circuit. Suppose, in a register-to-register path, the clock signal to the target register is connected via a long wire with significant delay, as shown in Figure 4.39. A rising clock edge arrives at the source register earlier than at the target register. This phenomenon is called *clock skew*. If the propagation delay through the combinational subcircuit is small (for example, if the subcircuit is just a direct connection to the target register with negligible delay), the value from the previous cycle may not remain stable for the hold time after the clock edge, as shown in Figure 4.40. In most implementation fabrics, the hold time is very small, or may even be negative, thus reducing the likelihood of this problem. (A negative hold time simply means that the data may start changing before the clock edge.) However, if we don't take care to minimize clock skew in a design, the circuit may operate unreliably. Given the importance of minimizing skew across the clock connection network, together with the need for buffers to drive the large number of flip-flop clock inputs as described in Section 2.1.1, we usually leave implementation of the clock signal to CAD tools. As part of the physical design, a tool will insert clock buffers into the circuit and route the connections so as to minimize skew. In FPGA fabrics, dedicated buffer and wiring resources for clock distribution are built into the chip.

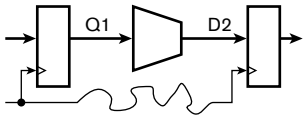

FIGURE 4.39 A register-to-register path with clock skew.

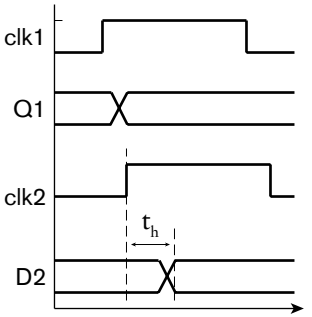

FIGURE 4.40 A timing problem arising from clock skew.

The timing parameters and constraints that we have considered so far apply to the datapath and control section within an integrated circuit chip. When we use that chip as a component of a larger system, we also need to take account of the effect of the input and output pins that connect the chip to other components via wires on a printed circuit board. The inputs have internal buffers that protect the chip from excessive voltage swings and static discharge, and the outputs have buffers to drive the relatively large capacitances and inductances that occur outside the chip. These buffers, together with the associated wiring connecting the integrated circuit chip to the package pins, introduce propagation delays. So when we analyze the timing behavior of the complete system, we need to include the pin and wiring delays. We can apply the same path-based analysis that we used for internal paths. Figure 4.41 shows a register-to-register path between a source register on one chip and a target register on another. The path includes output combinational logic, the output buffer and pin, the printed-circuit-board wiring, the input pin and buffer, and input combinational logic. The sum of the propagation delays plus the register clock-to-output and setup times must be less than the system's clock cycle time. For high-speed systems, this can be a difficult constraint to meet. In such systems, we usually avoid having any combinational input or

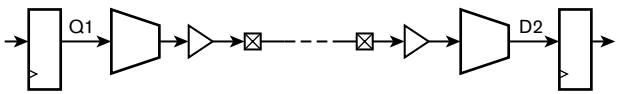

FIGURE 4.41 A register-to-register path between chips.

output logic. An input that connects directly to an input register is often called a *registered input*, and an output that is driven directly from an output register is called a *registered output*. High-speed design methodologies often require registered inputs, registered outputs or a combination of both. Using both allows a whole clock cycle for inter-chip transmission.

4.4.1 ASYNCHRONOUS INPUTS

Our clocked synchronous timing methodology requires us to ensure that inputs to registers are stable during an interval around each clock edge. For those signals that are generated within the circuit, we can ensure that we meet this constraint. However, most circuits must deal with some inputs that are generated externally, either by transducers whose outputs represent real-world quantities or events, or by separate systems that do not share a common clock. We call such signals *asynchronous inputs*. We have no control over the times at which they change value; hence, we cannot guarantee that they meet our timing constraints for register inputs.

Before we describe how to deal with asynchronous inputs, let's examine the behavior of a register, or more specifically, a flip-flop, when its input can change at any time. A flip-flop circuit internally uses a combination of charge storage and positive feedback to store a 0 or a 1 value. Figure 4.17 on page 164 gives a general idea of how this might work in a latch. A D flip-flop circuit elaborates on this structure to make storage edge-triggered. In order to change from storing a 0 to storing a 1, or *vice versa*, some energy input is required. A common analogy is to consider a ball resting in one of two holes, with a hill in between, as shown in Figure 4.42. The ball resting in one hole corresponds to storing a 0, and the ball resting in the other to storing a 1. In order to change the stored value, energy must be supplied to push the ball over the hill. In the case of a D flip-flop, a pulse of energy is sampled from the D input when the clock rises. If the input is 0, the ball is pushed toward the 0 hole, and if the input is 1, the ball is pushed toward the 1 hole.

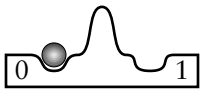

FIGURE 4.42 An analogy for the behavior of a flip-flop.

Now if the input changes close to the time the clock rises, insufficient energy may be sampled. For example, if the ball is in the 0 hole and the input changes to 1, there may be insufficient energy to push the ball to the 1 hole. The ball may get close to the top of the hill then fall back again. This corresponds to the flip-flop output starting to change from 0 to 1, but then reverting to 0. A particularly significant case arises if there is just sufficient energy to push the ball to the top of the hill, as shown in Figure 4.43, but not to push it straight over. In that case, the ball teeters on the top for some time before falling one way or the other. The time for which it teeters and the direction in which it falls are unpredictable. This condition is called *metastability*. The behavior of a real flip-flop in a *metastable state* depends on the details of the internal electrical and physical

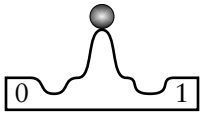

FIGURE 4.43 An analogy for the behavior of a flip-flop.

design of the flip-flop. Some flip-flops may delay a change between 0 and 1, some may oscillate, and others may have an invalid logic level at the output for some time. The problem is not so much the indeterminate behavior of the flip-flop output while the metastable state persists, but the fact that the delay until the output is stable is not bounded. As a consequence, we can't guarantee that the timing constraints for the circuits connected to the flip-flop output will be met.

Mathematical models of flip-flop behavior can be developed to help us understand how asynchronous inputs affect circuit operation. The details of these models are beyond the scope of this book, so we just summarize the conclusions here. Suppose an asynchronous input changes with a frequency of f_1 and the clock frequency of the system is f_2. We sample the output value of the flip-flop to which the asynchronous input is connected after a period t. Occasionally, the sampled value will be incorrect due to metastability in the flip-flop, and that will cause some form of failure. The mathematical model gives us the mean time between failures (MTBF):

$$\text{MTBF} = \frac{e^{k_2 t}}{k_1 f_1 f_2} \tag{4.5}$$

The constants k_1 and k_2 are measured for a particular flip-flop. Since the MTBF is inversely proportional to the frequencies, higher frequencies lead to shorter MTBF, that is, to more frequent failure. More significant, however, is that the MTBF is nonlinearly related to the time before sampling. The value of k_2 is typically large and positive, so a small increase in the time before sampling yields a significant increase in the MTBF.

The usual approach to dealing with asynchronous inputs is to connect them to a *synchronizer*, and to use the output of the synchronizer in the rest of the system. A simple synchronizer is shown in Figure 4.44. The first flip-flop samples the value of the asynchronous input at each clock edge. Usually, the value is passed on to the flip-flop's output within the clock-to-output delay of the flip-flop and sampled on the next clock edge by the second flip-flop. The output of the second flip-flop is used in the rest of the system. On those occasions where the asynchronous input changes close to a clock edge, the first flip-flop may enter the metastable state. However, its output is not sampled for an entire clock cycle, giving the flip-flop time to resolve the metastability. In terms of Equation 4.5, the sampling interval t is one clock cycle period, t_c.

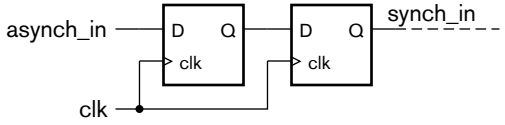

FIGURE 4.44 A synchronizer for an asynchronous input.

It is only in fairly recent times that component manufacturers have developed a complete understanding of metastability and its effects on system reliability. Earlier than 15 years or so ago, published data on the metastability characteristics of flip-flops was hard to find. Since then, manufacturers have improved both their device behavior and their published data. For most applications using modern implementation fabrics, the simple synchronizer shown in Figure 4.44 is sufficient to give a MTBF that is much longer than the lifetime of the system. However, for those applications in which reliability is a key requirement and that have many asynchronous inputs, we should study the published data for implementation fabric we use and follow the manufacturer's advice on synchronizing inputs.

Switch Inputs and Debouncing

We mentioned that externally generated signals are often asynchronous inputs to a system. A common example is connection of switches that form a user interface to the system. This includes push-button, slider, toggle and rotary switches. A user can change a switch position at random times, so we cannot assume synchronization with a clock signal. Similarly, a microswitch used to sense mechanical input may change asynchronously. There is a further problem that we must also deal with. Switches are electromechanical devices containing electrical contacts that open and close a circuit in response to mechanical movement. As the contacts close, they *bounce*, causing the circuit to open and close one or more times before finally setting in the closed position. Similarly, as the contacts open, they may also bounce. If we are to avoid spurious activation of the system's response to switch movements, we must debounce the switch input. This involves waiting for some period of time after an initial change in circuit closure is detected before treating the switch input as valid. For most switches, the time taken to settle is of the order of a few millisecond, so a debounce delay of up to 10ms is common practice. Delaying too long causes the user to notice the lag in response to switch activation. A response time of less than 50ms is generally imperceptible.

There are probably as many solutions to switch debouncing as there are design engineers. One simple approach is shown in Figure 4.45.

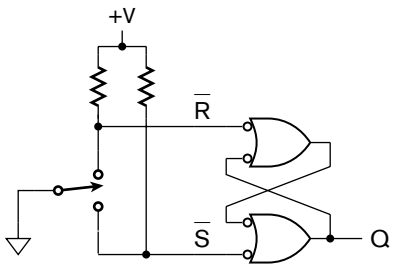

FIGURE 4.45 A switch debouncer using an RS-latch.

It uses an RS-latch with negative-logic inputs and a double-throw switch. When the switch is in the position shown, it holds the reset input of the latch active, producing a 0 at the Q output. When the switch is toggled, we assume that one contact is opened before the other contact is closed. (This is sometimes called "break before make.") Bouncing on the contact to be opened simply leaves the latch in the reset state. When the first bounce occurs on the contact to be closed, the set input is activated, causing the Q output to change to 1. Subsequent bounces leave the latch in the set state. The behavior is similar when the switch is toggled in the other direction.

While this approach is very effective, it has two drawbacks. First, it requires two inputs to the digital system for what is really just one input. Second, it requires a double-throw switch, whereas many low-cost applications require a single-throw switch consisting of two contacts that are shorted together by a push button. Simple circuits for debouncing single-throw switches generally rely on analog circuit design techniques and require components external to the main digital chip. We will not discuss them here, but refer to Section 4.6, Further Reading. Instead, we will outline a fully digital approach to debouncing that can be designed into the main digital circuit of a system.

A simple way of connecting a single-throw or momentary-contact switch to a digital circuit input is shown in Figure 4.46. When the switch is open, the input is pulled to 1, and when the switch is closed, the input is pulled to 0. A change of switch position causes the input to toggle between 0 and 1 until the bouncing stops and the input settles at its final value. Rather than using the input value directly within the system, we sample it at intervals longer than the bounce time. When we get two successive samples that have the same value, we use that value as the stable state of the switch input.

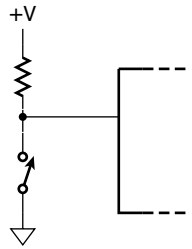

FIGURE 4.46 Simple switch input connection.

EXAMPLE 4.20 Develop a Verilog model of a debouncer for a push-button switch that uses a debounce interval of 10ms. Assume the system clock frequency is 50MHz.

SOLUTION The module definition is

```
module debouncer ( output reg pb_debounced,
                   input      pb,
                   input      clk, reset );

  reg [18:0] count500000; // values are in the range
                          // 0 to 499999
  wire       clk_100Hz;
  reg        pb_sampled;
```

(continued)

```
  always @(posedge clk or posedge reset)
    if      (reset)     count500000 <= 499999;
    else if (clk_100Hz) count500000 <= 499999;
    else                count500000 <= count500000 - 1;

  assign clk_100Hz = count500000 == 499999;

  always @(posedge clk)
    if (clk_100Hz) begin
      if (pb == pb_sampled) pb_debounced <= pb;
      pb_sampled <= pb;
    end

endmodule
```

The first always block represents a down counter that divides the clock by 500,000. The assignment following the block decodes the terminal count to derive a sampling clock that pulses to 1 every 10ms. When the sampling clock is 1, the second always block compares the current push-button input value (pb) with a previously sampled value (pb_sampled). If they are the same, the block updates the debounced output with the current value. If they are not the same, the output is unchanged. Also, when the sampling clock is 1, the block updates the sampled value with the current value.

It is important to note that, even though the debouncer of Example 4.20 uses much more circuitry than the simple debouncer of Figure 4.45, it will probably be cheaper to implement. It uses a simple single-throw switch and only a single resistor external to the integrated circuit, and only requires one input pin. The saving in packaging resources and printed circuit board assembly costs would be more significant in a large-volume application than the expense of additional circuit resources used within the integrated circuit. We might also consider implementing the debounce operation in software run on an embedded processor, if the application requires a processor to be included anyway. If the processor has sufficient time in its task schedule to perform debouncing, that might be a more efficient use of resources. The lesson to learn is that, when we make these trade-off decisions, we must consider all of the costs and resources for the entire system, not just for one aspect in isolation.

4.4.2 VERIFICATION OF SEQUENTIAL CIRCUITS

Now that we have described the design of clocked sequential circuits and the timing constraints that apply, we can return to the verification steps outlined in the design methodology in Section 1.5. We need to consider functional verification (that the sequential circuit performs its

function correctly) and timing verification (that the circuit meets timing constraints). We outlined in Section 1.5 how tools perform static timing analysis to verify timing constraints. Here, we will discuss functional verification using Verilog models, expanding on the ideas introduced in Section 2.4 relating to verification of combinational circuits.

When verifying a combinational circuit, we saw that we need to wait for some time after applying a test case to the circuit's inputs before checking the circuit's outputs, to allow for the propagation delay of the circuit. Similarly, when verifying a sequential circuit, we need to take account of the fact that operations take one or more clock cycles to complete. We need to ensure that the procedural block that checks the output is synchronized with the stimulus block, and knows how many clock cycles after application of a test case to wait before checking the output. If all operations complete in the same number of cycles, and only one operation takes place at a time, this is relatively straightforward. On the other hand, if operations take varying numbers of cycles to complete, the checker needs to check both that the operation completes at the correct time and that the correct result is produced. If multiple operations can take place concurrently, for example, if the datapath is a pipeline, the checker needs to ensure that all operations that start also complete, and that no spurious results are produced.

Developing testbench models for complex sequential circuits can itself become a complex endeavor. We will discuss some of the general techniques that can be used in Chapter 10. For now, we will illustrate a simulation-based approach for verifying circuits that we introduced in previous examples.

EXAMPLE 4.21 Develop a testbench model for the sequential multiplier of Example 4.14. Verify that the result computed by the multiplier is the same (within the limits of the precision of the operands) as that produced using multiplication with the built-in Verilog type real.

SOLUTION The testbench has no external connections, and so the module definition is

```
`timescale 1ns/1ns

module multiplier_testbench;

  parameter t_c = 50;

  reg               clk, reset;
  reg               input_rdy;
```

(continued)

```
wire signed [3:-12] a_r, a_i, b_r, b_i;
wire signed [7:-24] p_r, p_i;

real real_a_r, real_a_i, real_b_r, real_b_i,
     real_p_r, real_p_i, err_p_r, err_p_i;

task apply_test ( input real a_r_test, a_i_test,
                             b_r_test, b_i_test );
  begin
    real_a_r = a_r_test; real_a_i = a_i_test;
    real_b_r = b_r_test; real_b_i = b_i_test;
    input_rdy = 1'b1;
    @(negedge clk) input_rdy = 1'b0;
    repeat (5) @(negedge clk);
  end
endtask

multiplier duv ( .clk(clk), .reset(reset),
                 .input_rdy(input_rdy),
                 .a_r(a_r), .a_i(a_i),
                 .b_r(b_r), .b_i(b_i),
                 .p_r(p_r), .p_i(p_i) );

always begin  // Clock generator
  #(t_c/2)       clk = 1'b1;
  #(t_c - t_c/2) clk = 1'b0;
end

initial begin  // Reset generator
  reset <= 1'b1;
  #(2*t_c) reset = 1'b0;
end

initial begin  // Apply test cases
  @(negedge reset)
  @(negedge clk)
  apply_test(0.0, 0.0, 1.0, 2.0);
  apply_test(1.0, 1.0, 1.0, 1.0);
  // further test cases ...
  $finish;
end

assign a_r = $rtoi(real_a_r * 2**12);
assign a_i = $rtoi(real_a_i * 2**12);
assign b_r = $rtoi(real_b_r * 2**12);
assign b_i = $rtoi(real_b_i * 2**12);

always @(posedge clk)  // Check outputs
  if (input_rdy) begin
    real_p_r = real_a_r * real_b_r - real_a_i * real_b_i;
```

(*continued*)

```
      real_p_i = real_a_r * real_b_i + real_a_i * real_b_r;
      repeat (5) @(negedge clk);
      err_p_r = $itor(p_r)/2**(-24) - real_p_r;
      err_p_i = $itor(p_i)/2**(-24) - real_p_i;
     if (!( -(2.0**(-12)) < err_p_r && err_p_r < 2.0**(-12) &&
            -(2.0**(-12)) < err_p_i && err_p_i < 2.0**(-12) ))
         $display("Result precision requirement not met");
    end

endmodule
```

Within the module, we have instantiated the multiplier module as the device under verification. The instance is connected to testbench nets and variables declared in the module.

Since the multiplier is clocked, we need to generate a clock signal to drive it. This is done by the first always block. It uses a parameter, called t_c, for the clock cycle time. Using a parameter like this allows us to change the clock cycle time without having to chase down every number that varies as a consequence of the change. The block delays for half a clock cycle time, sets the clock to 1, delays a further half a clock cycle time, then sets the clock to 0. (The expression for the duration of the second half clock cycle time is structured so as to compensate for any rounding that may occur in the expression for the first half cycle duration.) After that, the block repeats from the beginning. We also need to generate a reset pulse for the device under verification. This is done by the first initial block. The block sets reset to 1 immediately, then back to 0 after a delay of two clock cycles.

The second initial block stimulates the device under verification with input data. The block uses a task to abstract out the common operations in applying each test-case. Rather than generating fixed-point values directly, the block generates test-case operands of type real on the variables real_a_r, real_a_i, real_b_r and real_b_i. The assignments following the stimulus initial block use the $rtoi conversion function, which converts a real value to an integer value, to assign test-case values to the input inputs of the device under verification. The scaling by 2^{12} is required, since the binary point in each input is 12 places from the right.

Within the stimulus initial block, we must ensure that we generate input stimulus values that meet the timing requirements of the device under verification. The operand values and the input_rdy signal must be set up before a clock edge. The operand values must be held for four cycles while the operation proceeds. To satisfy these requirements, we wait until the first falling clock edge after reset has returned to 0. We do this using the @ notation to delay until the required events occur. The call to the apply_test task then assigns the first test-case operands to

the inputs and sets input_rdy to 1. Next, the task waits for the subsequent falling clock edge before resetting input_rdy back to 0. It then waits a further five cycles, giving the device under verification time to produce its output. After that, subsequent calls to the task repeat these steps with the further test-case operands.

The output-checking always block verifies that the multiplier produces the correct results. It must synchronize with the input stimulus to ensure that it checks the results at the right time. It waits on the same condition as the multiplier's controller finite-state machine, namely, input_rdy being 1 on a rising clock edge. When that occurs, the block reads the stimulus operand values from the variables real_a_r, real_a_i, real_b_r and real_b_i, forms the complex product using the real multiplication operator, and saves the product in the variables real_p_r and real_p_i. The block then waits until the fifth subsequent falling clock edge, by which time the device under verification has stored its result in its output registers. The results are available on the p_r and p_i nets. The block converts them to real form and compares them with the real and imaginary parts saved in real_p_r and real_p_i. It uses the $itor conversion function to convert values from integer to real, and scales by 2^{24} to deal with the assumed position of the binary point 24 places from the right. Since the type real and our fixed-point representation are discrete approximations to mathematical real numbers, an exact equality test is unlikely to succeed. Instead, we check whether the absolute value of the difference is within the required precision, in this case, the precision of the input-operand representation.

4.4.3 ASYNCHRONOUS TIMING METHODOLOGIES

We will close this section on timing methodology with a brief discussion of some alternative approaches. While the clocked synchronous approach yields significant simplifications, there are some applications where it breaks down. Two key assumptions are that the clock signal is distributed globally (that is, across the entire system) with minimal skew, and that the propagation delay between registers is less than a clock cycle. In large high-speed systems, these assumptions are very difficult to maintain. For example, in a large integrated circuit operating with a clock frequency of several GHz, the time taken for a change of signal value to propagate along a wire that stretches across the chip may be a large proportion of a clock cycle, or even more than a clock cycle.

One emerging solution is to reconsider the assumption of a single global clock signal for the entire chip or system. Instead, the system is divided into several regions, each with its own local clock. Where signals connect from one region to another, they are treated as asynchronous inputs. The timing for the system is said to be *globally asynchronous, locally synchronous* (GALS). The benefit of this approach is that it makes

the constraints on clock distribution and timing within each region simpler to manage. The downside is that inter-region connections must be synchronized, thus adding delay to communication. The challenge for the system architect is to find a partitioning for the system that minimizes the amount of communication between regions, or that avoids sensitivity to delay in inter-region communication.

The difficulty in distributing high-speed clock signals and managing timing is even greater in the context of a complete circuit board consisting of several integrated circuits, or a large system consisting of several circuit boards. It is simply not practical to distribute a high-speed clock across a large system. Instead, a slower clock is often used externally to high-speed chips, and operations between chips are synchronized to that external clock. The internal clocks operate at a frequency that is a multiple of the external clock, allowing for synchronization of clock edges. The separate boards in a high-speed system typically are not synchronized, but have independent clocks. Data transmitted from one board to another is treated as an asynchronous input by the receiving board.

Another aspect of timing in clocked synchronous systems is that all register-to-register operations take one clock cycle, whether the combinational subcircuit is on the critical path or not. In principle, the slack time in a clock cycle is wasted; all operations are held back to the time taken by the slowest. It is possible to design *asynchronous* circuits in which completion of one operation triggers dependent operations. Such circuits are also called *delay insensitive*, since they operate as fast as the components and the data allow. However, appropriate design techniques are far less mature than those for clocked circuits, and there is negligible CAD tool support for asynchronous methodologies. Hence, products using asynchronous circuits are very uncommon.

A separate issue with the clocked approach is that clocked circuits consume significant amounts of power. Even if a flip-flop does not change its stored value, changing the clock input between 0 and 1 involves switching transistors on and off, thus consuming extra power. In applications with very low power budgets, such as battery powered mobile devices, this waste of power is unacceptable. One approach to dealing with it is to avoid clocking parts of a system that are inactive. *Clock gating*, as it is called, is becoming a more common design technique as the number of low-power applications increases. Asynchronous circuits are an alternative, since logic levels only change when data values change. If there is no new data to operate upon, the circuit becomes quiescent. A few low-power products using asynchronous circuits have been successfully fielded. Low-power applications may be a more significant motivation for asynchronous design than the potential performance gains.

KNOWLEDGE TEST QUIZ

1. What is meant by the term *register transfer level*?
2. Write the timing condition that must apply on a register-to-register path.
3. What is the critical path in a system?
4. How does the critical path delay affect the clock cycle time of the system?
5. If a given clock cycle time is required, but the critical path delay is too long to achieve it, where should optimization effort be focused?
6. What is meant by the term *clock skew*?
7. Why are registered inputs and outputs used in high-speed systems?
8. What problem can be caused in input registers by asynchronous inputs?
9. Why must inputs from electromechanical switches be debounced?
10. What is the main difference between a testbench for a combinational circuit and a testbench for a sequential circuit?
11. What is meant by the term *globally asynchronous, locally synchronous* (GALS)?

4.5 CHAPTER SUMMARY

- Registers are storage components composed of flip-flops. Simple registers can be augmented with clock-enable, reset and preset control inputs.
- Synchronous control inputs are acted upon on a clock edge. Asynchronous control inputs are acted upon immediately.
- Latching behavior is produced by feedback paths in digital circuits. A transparent latch passes data through while the enable input is 1 and stores data when the enable input is 0.
- A simple free-running counter consists of an incrementer and a register. Substituting a decrementer for the incrementer causes the counter to count down instead of up. Adding a clock-enable input to the register allows control over when the counter increments. Adding a reset input to the register allows the count value to be cleared to 0.
- An n-bit counter counts modulo 2^n; that is, it counts to $2^n - 1$ then wraps to 0. A modulo k up counter decodes the value $k - 1$ and uses it to reset the counter. A modulo k down counter decrements down to 0 and then reloads the value $k - 1$.
- A ripple counter uses the output of one flip-flop to trigger the next flip-flop. It uses less circuitry and consumes less power than a synchronous counter, and can be used in applications where timing constraints allow and power constraints are significant.
- A digital system, in general, consists of a datapath and a control section. The datapath contains combinational subcircuits for operating on data and registers for storing data. The control section sequences operations in the datapath by activating control signals at various times. The control section uses status signals to influence the control sequence.
- A finite-state machine (FSM) has a set of inputs, a set of outputs, a set of states, a transition function and an output function. For a given clock cycle, the FSM has a current state. The transition function determines the next state given the current state and the input values. The output function determines the output values given just the current state (Moore machine), or given the current state and the input values (Mealy machine).
- The state encoding of an FSM can influence the complexity of the next-state and output logic. Synthesis CAD tools are usually able to optimize the state encoding.

- A state transition diagram represents a finite state machine with bubbles for states, arcs for transitions, and labels for input conditions and output values. Labels for Moore-style outputs are written in the bubbles, and labels for Mealy-style outputs are written on arcs.
- At the register-transfer level of abstraction, operation of a system is described in terms of transfer of data between registers through combinational circuits that operate on the data.
- The clocked synchronous timing methodology involves a common clock for all registers, and operation on data by combinational circuits between clock edges.
- For each path from register output to register input, the sum of the clock-to-output delay, combinational propagation delay and setup time must be less than the clock cycle time. The path with the least slack time is the critical path.
- The critical path delay places a lower bound on the clock cycle time. Alternatively, a required clock cycle time places an upper bound on the critical path delay.
- Clock skew is the difference in arrival time of a clock edge at different flip-flops in a system. Clock skew must be minimized to ensure that clocked synchronous circuits operate correctly. CAD tools typically implement clock distribution to minimize skew.
- Registered inputs and outputs reduce combinational delays in interchip register-to-register paths, and thus help in meeting timing constraints.
- Asynchronous inputs are those that are not guaranteed to be stable around clock edges. They can cause metastability in input registers. Synchronizers are required to avoid system failure due to metastability.
- Testbenches for clocked sequential circuits must ensure that stimulus inputs are applied so as to meet timing constraints, and must wait until outputs are valid before checking them.
- A globally asynchronous, locally synchronous (GALS) system has regions with local clocks, and treats inter-region connections as asynchronous inputs.

4.6 FURTHER READING

Digital Design: Principles and Practices, 3rd Edition, John F. Wakerly, Prentice Hall, 2001. Describes flip-flops and latches in detail, presents detailed low-level design procedures for finite-state machines,

provides an analysis procedure for feedback circuits, and discusses metastability and synchronizers in detail.

CMOS VLSI Design: A Circuits and Systems Perspective, 3rd Edition, Neil H. E. Weste and David Harris, Addison-Wesley, 2005. Among many other aspects of CMOS circuit design, this book discusses detailed design of flip-flops and latches and addresses both single-phase and two-phase clocking schemes.

Asynchronous Circuit Design, Chris J. Myers, Wiley-Interscience, 2001. An in-depth treatment of theory and practice.

A Guide to Debouncing, Jack G. Ganssle, The Ganssle Group, 2004, www.ganssle.com/debouncing.pdf. Presents empirical data on switch bounce behavior, and describes hardware and software approaches to debouncing.

Comprehensive Functional Verification: The Complete Industry Cycle, Bruce Wile, John C. Goss and Wolfgang Roesner, Morgan Kaufmann Publishers, 2005. Describes strategies and techniques for stimulus generation and result checking in simulation-based verification.

EXERCISES

EXERCISE 4.1 Draw a schematic for a 6-bit register, constructed from D flip-flops, that updates the stored value on every clock cycle.

EXERCISE 4.2 Write a Verilog model for a 12-bit register that stores an unsigned integer value.

EXERCISE 4.3 Develop a Verilog model of a pipelined circuit that computes the maximum of corresponding values in three streams of input values, a, b and c. The pipeline should have two stages: the first stage determines the larger of a and b and saves the value of c; the second stage finds the larger of c and the maximum of a and b. The inputs and outputs are all 14-bit signed 2s-complement integers.

EXERCISE 4.4 Revise the schematic of Exercise 4.1 to include a clock enable and a reset input to the register, using flip-flops with clock-enable and reset inputs.

EXERCISE 4.5 Write a Verilog model for a register with clock-enable and synchronous reset that stores a 16-bit 2s-complement signed integer value.

EXERCISE 4.6 Draw a datapath for a pipelined complex multiplier. Unlike the sequential multiplier in Example 4.13 that takes five cycles to do each