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.

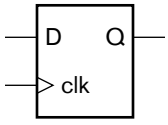


FIGURE 4.1 A D flip-flop.

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

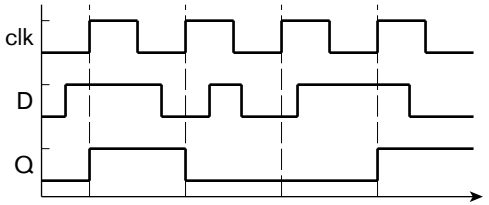


FIGURE 4.2 Timing diagram for a D flip-flop.

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.

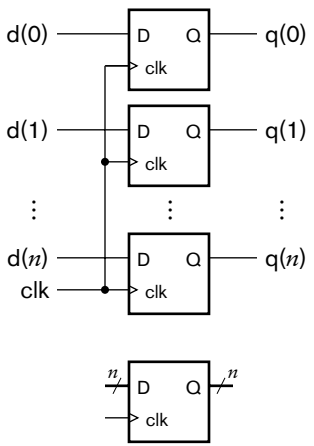


FIGURE 4.3 A register composed of D flip-flops (top), and the symbol for the register (bottom).

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

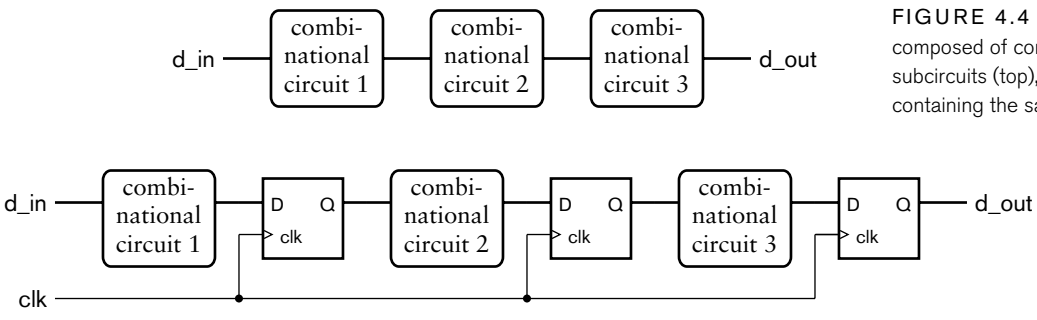


FIGURE 4.4 A circuit composed of combinational subcircuits (top), and a pipeline containing the same subcircuits.

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

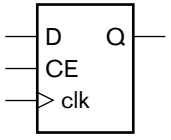


FIGURE 4.5 A D flip-flop with clock-enable input.

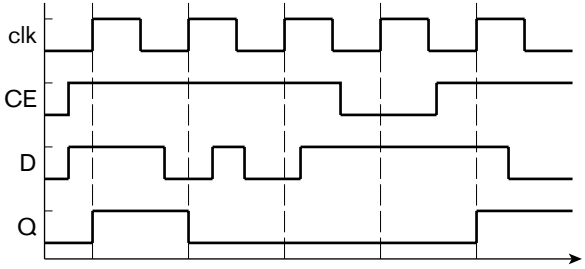


FIGURE 4.6 Timing diagram for a D flip-flop with clock enable.

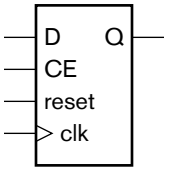


FIGURE 4.7 A D flip-flop with clock-enable and reset inputs.

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.

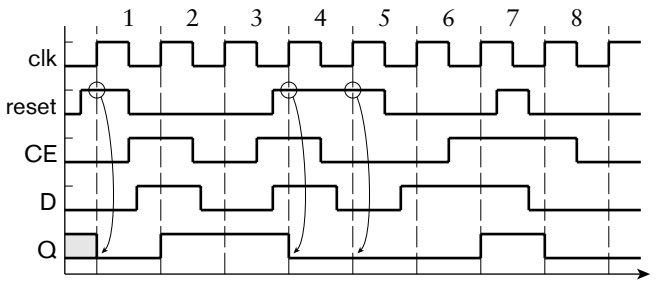


FIGURE 4.8 Timing diagram for a flip-flop with clock-enable and synchronous reset inputs.

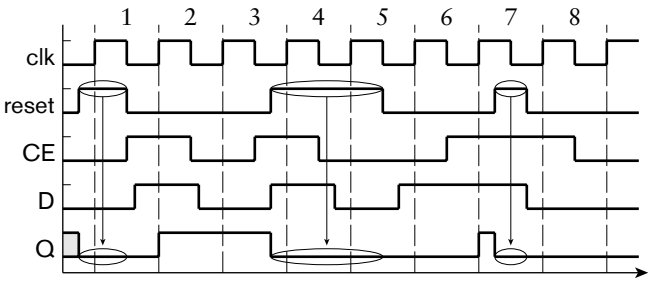


FIGURE 4.9 Timing diagram for a flip-flop with clock-enable and asynchronous reset inputs.

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.

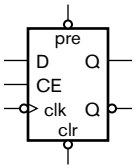


FIGURE 4.10 A negative-edge-triggered flip-flop.

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.

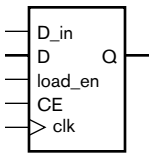


FIGURE 4.11 A symbol for a shift register.

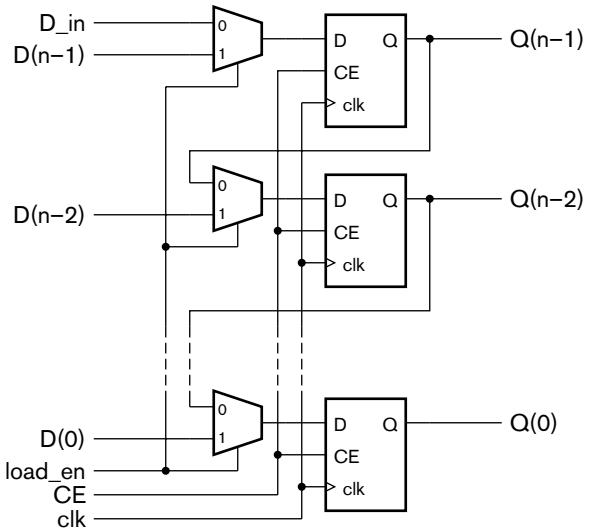


FIGURE 4.12 A shift register implemented with D flip-flops and multiplexers.

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.

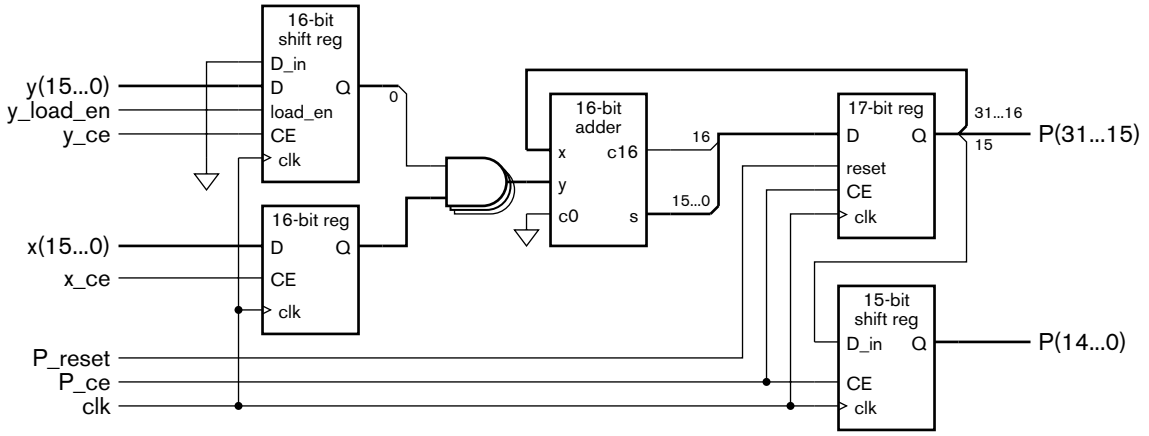


FIGURE 4.13 Registers, shift registers and other components used to form a sequential multiplier.

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.

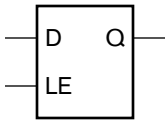


FIGURE 4.14 Symbol for a latch.

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.

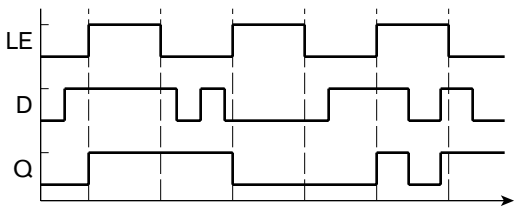


FIGURE 4.15 Timing diagram for a latch.

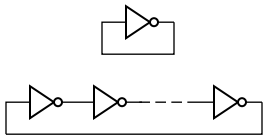


FIGURE 4.16 Inverters connected in feedback loops.

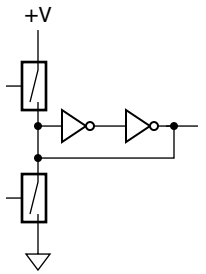


FIGURE 4.17 Using switches to force a node of an inverter ring to 0 or 1.

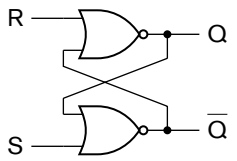


FIGURE 4.18 Cross-coupled RS-latch.

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

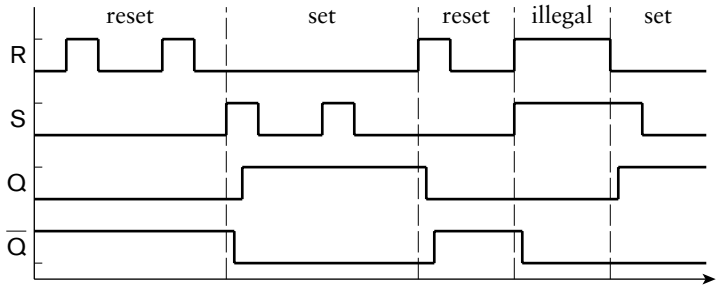


FIGURE 4.19 Timing for an RS-latch, showing the reset and set states, as well as an illegal operating condition.

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

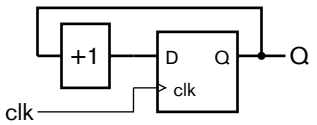


FIGURE 4.20 A simple counter composed of a register and an incrementer.

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.

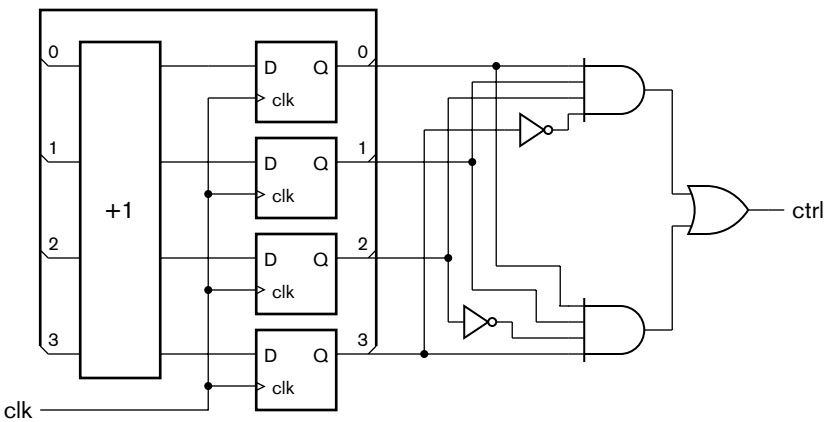


FIGURE 4.21 A counter with decoded outputs.

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.

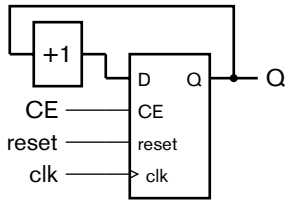


FIGURE 4.22 A counter with clock-enable and reset inputs.

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.

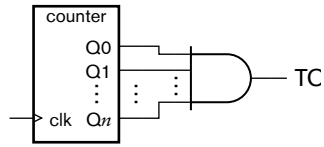


FIGURE 4.23 A counter with terminal-count output.

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.

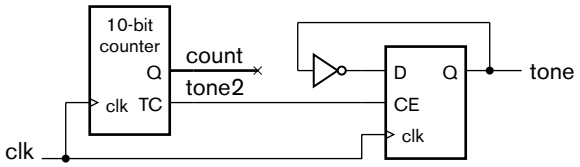


FIGURE 4.24 An alarm clock frequency divider.

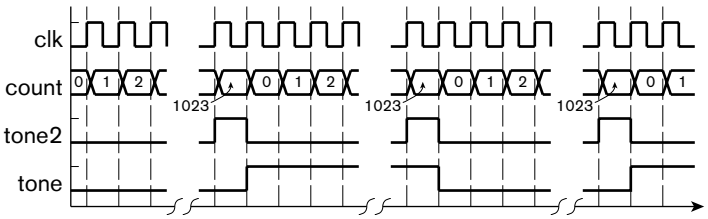


FIGURE 4.25 Timing diagram for an alarm clock frequency divider.

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.

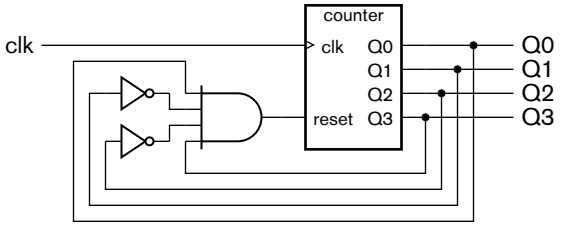


FIGURE 4.26 A decade counter.

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.

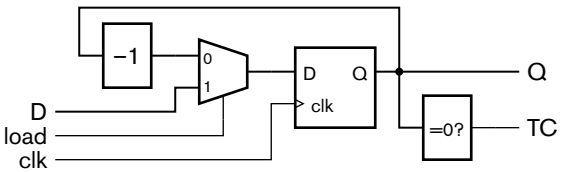


FIGURE 4.27 A down counter with synchronous load.

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

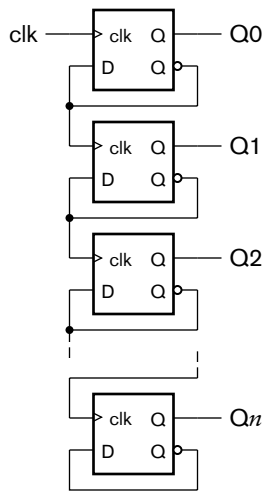


FIGURE 4.28 Structure of a ripple counter.

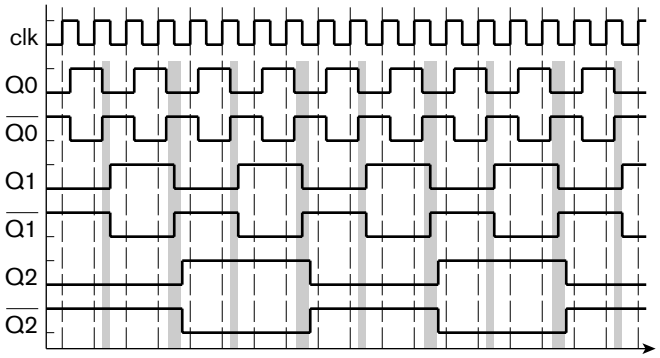


FIGURE 4.29 Timing diagram for a ripple counter.

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

the outputs are the same, place the implied pairs in square *i–j*. (If the next states of *i* and *j* are *m* and *n* for some input *x*, then *m–n* is an implied pair.) If the outputs and next states are the same (or if *i–j* implies only itself), place a check (√) in square *i–j* to indicate that $i \equiv j$.

3. Go through the table square by square. If square *i–j* contains the implied pair *m–n*, and square *m–n* contains an ×, then $i \not\equiv j$, and an × should be placed in square *i–j*.
4. If any ×s were added in step 3, repeat step 3 until no more ×s are added.
5. For each square *i–j* that does not contain an ×, $i \equiv j$.

If desired, row matching can be used to partially reduce the state table before constructing the implication table. Although we have illustrated this procedure for a Mealy table, the same procedure applies to a Moore table.

Two sequential circuits are said to be equivalent if every state in the first circuit has an equivalent state in the second circuit, and vice versa.

Optimization techniques such as this are incorporated in CAD tools. The importance of state minimization has slightly diminished in recent years due to the abundance of transistors on chips; however, it is still important to do obvious state minimizations to reduce the circuit's area and power.

1.10 Sequential Circuit Timing

The correct functioning of sequential circuits involves several timing issues. Propagation delays of flip-flops, gates and wires; setup times and hold times of flip-flops; clock synchronization; clock skew; and the like become important issues in designing sequential circuits. In this section, we look at various topics related to sequential circuit timing.

1.10.1 Propagation Delays, Setup, and Hold Times

There is a certain amount of time, albeit small, that elapses from the time the clock changes to the time the *Q* output changes. This time, called *propagation delay* or ***clock-to-Q delay*** of the flip-flop is indicated in Figure 1-34. The propagation delay can depend on whether the output is changing from high to low or vice versa. In the figure, the propagation delay for a low-to-high change in *Q* is denoted by t_{plh}, and for a high-to-low change it is denoted by t_{phl}.

For an ideal *D* flip-flop, if the *D* input changed at exactly the same time as the active edge of the clock, the flip-flop would operate correctly. However, for a real flip-flop, the *D* input must be stable for a certain amount of time before the active edge of the clock. This interval is called the *setup time* (t_{su}). Furthermore, *D* must be stable for a certain amount of time after the active edge of the clock. This interval is called the *hold time* (t_h). Figure 1-34 illustrates setup and hold times for a *D* flip-flop that changes state on the rising edge of the clock. *D* can change at any time during the shaded region on the diagram, but it must be stable during the time interval t_{su} before the active edge and for t_h after the active edge. If *D* changes at any time during the forbidden interval, it cannot be determined whether the flip-flop will change state. Even worse, the flip-flop may malfunction and output a short pulse or even go into oscillation.

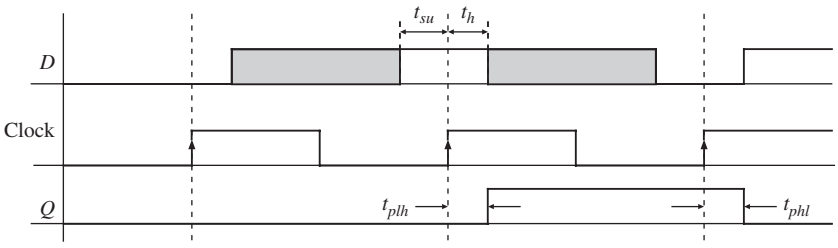


FIGURE 1-34: Setup and Hold Times for D Flip-Flop

Flip-flops typically have a setup time about 3–10x of the propagation delay of an inverter (NOT) gate. The hold times are typically 1–2x of the delay of an inverter. Minimum values for t_{su} and t_h and maximum values for t_{plh} and t_{phl} can be obtained from manufacturers' data sheets or ASIC (Application Specific Integrated Circuit) libraries accompanying design tools.

1.10.2 Timing Conditions for Proper Operation

In a synchronous sequential circuit, state changes occur immediately following the active edge of the clock. The maximum clock frequency for a sequential circuit depends on several factors. The clock period must be long enough so that all flip-flop and register inputs will have time to stabilize before the next active edge of the clock. Propagation delays and setup and hold times create complications in sequential circuit timing.

Static timing analysis (STA) is a method of validating the timing performance of a design by checking all possible paths for timing violations under worst-case conditions. A static analysis path starts at a source flip-flop (or at a primary input) and terminates at a destination flip-flop (or primary output). A static timing path between two flip-flops starts at the input to the source flip-flop and terminates at the input of the destination flip-flop. It does not go through the destination flip-flop. The path terminates when it encounters a clocked device. If a signal goes from register (flip-flop) A to register B and then to register C, the signal contains two paths. The timing paths in a synchronous digital system can be classified into 4 types:

I. Register to register paths (i.e., flip-flop to flip-flop)
II. Primary input to register paths (i.e., input to flip-flop)
III. Register to primary output paths (i.e., flip-flop to output)
IV. Input to output paths (i.e., no flip-flop)

Question: Identify the static timing paths in the following circuit:

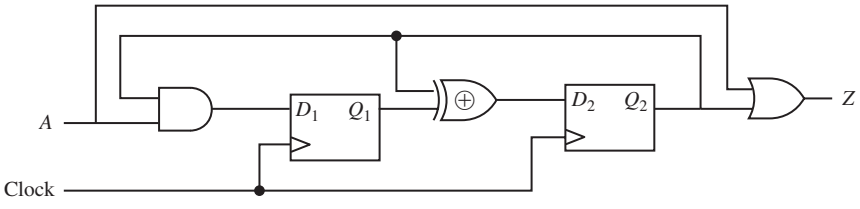


FIGURE 1-35: A Circuit to Illustrate Timing Paths

There are six static timing paths in this circuit:

I. From A to D_1 (primary input to flip-flop)
II. From D_1 to D_2 including the XOR (flip-flop to flip-flop)
III. From D_2 via XOR to D_2 (flip-flop to flip-flop)
IV. From D_2 to D_1 via AND (flip-flop to flip-flop)
V. From D_2 to Z via the OR gate (flip-flop to output)
VI. From A to Z via the OR gate (input to output)

The most complicated paths are the flip-flop to flip-flop paths; the other paths can be treated as special cases of this type of path.

Static timing analysis checks how the data arrives with respect to clock. It detects setup and hold-time violations in the design so that they can be corrected. A **setup time violation** occurs if the data changes just before the clock without providing enough setup time for the flip-flop. A **hold-time violation** occurs if the data changes just after the clock without providing enough hold time for the flip-flop.

Slack is the amount of time still left before a signal will violate a setup or hold-time constraint. Paths must have a positive or zero slack in order to have no violations. Paths that have a zero or very small slack are the speed-limiting paths in the design, because any small changes in clock or gate delays will lead to violations in such circuits. Paths that have a negative slack time have already violated a setup or hold constraint.

Static timing analysis considers the worst possible timing scenarios, but not the logical operation of the circuit. In comparison with circuit simulation, static timing analysis is faster because it doesn't need to simulate multiple test vectors.

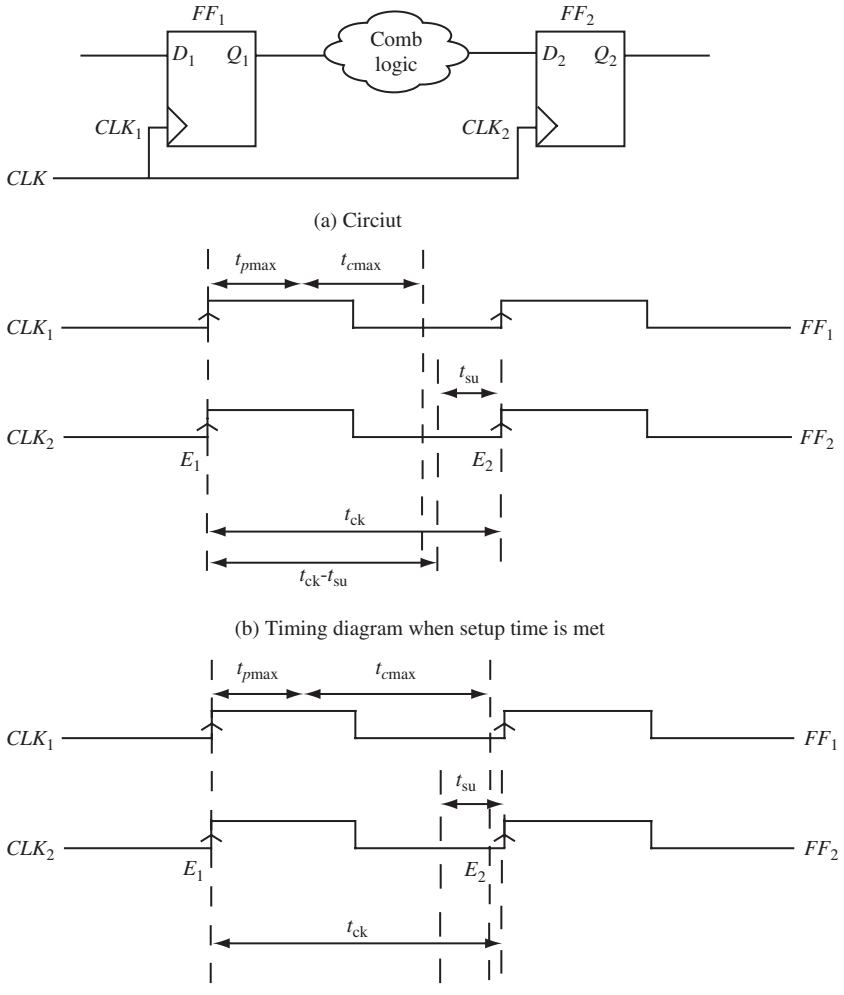
Timing Rules for Flip-Flop to Flip-Flop Paths

For a circuit of the general form of Figure 1-36, assume that the maximum propagation delay through the combinational circuit is $t_{c\max}$ and the maximum ***clock- to-Q delay*** or propagation delay from the time the clock changes to the time the flip-flop output changes is $t_{p\max}$, where $t_{p\max}$ is the maximum of t_{plh} and t_{phl}. Also assume that the minimum propagation delay through the combinational circuit is $t_{c\min}$ and the minimum clock-to-Q delay or propagation delay from the time the clock changes to the time the flip-flop output changes is $t_{p\min}$, where $t_{p\max}$ data is launched from flip-flop 1's D (i.e., D_1) to FF_1's Q (i.e., Q_1) at the positive edge of clock at FF_1 (i.e., CK_1). Data is captured at FF_2's D (i.e., D_2) at the positive clock edge at FF_2 (i.e., CLK_2). FF_1 is called the launching flip-flop, and FF_2 is called the capturing flip-flop. There are two rules this circuit has to meet in order to ensure proper operation.

Rule No. 1: Setup time rule for flip-flop to flip-flop path: Clock period should be long enough to satisfy flip-flop setup time.

For proper synchronous operation, the data launched by FF_1 at edge E_1 of clock CK_1 should be captured by FF_2 at edge E_2 of clock CK_2. The clock period should be long enough to allow the first flip-flop's outputs to change and the combinational circuitry to change while still leaving enough time to satisfy the setup time. Once the clock CK_1 arrives, it could take a delay of up to $t_{p\max}$ before FF_1's output changes. Then it could take a delay of up to $t_{c\max}$ before the output of the combinational circuitry changes. Thus the maximum time from the active edge E_1 of the clock CK_1 to the time the change in Q_1 propagates to the second flip-flop's input (i.e., D_2) is $t_{p\max} + t_{c\max}$. In

FIGURE 1-36: Flip-Flop to Flip-Flop Path via Combinational Logic



order to ensure proper flip-flop operation, the combinational circuit output must be stable at least t_{su} before the end of the clock E_2 reaches FF_2. If the clock period is t_{ck},

$$t_{ck} \geq t_{p\max} + t_{c\max} + t_{su} \tag{1-34-a}$$

Equation (1-34-a) relates the clock frequency of operation of the circuit with setup time of the flip-flops. Therefore, setup time violations can be solved by changing the clock frequency. The difference between t_{ck} and $(t_{p\max} + t_{c\max} + t_{su})$ is referred to as the **setup time margin**. The setup margin has to be zero or positive in order to have a circuit pass timing checks. Figure 1-36 (b) illustrates a situation in which setup time constraint is met, and Figure 1-36 (c) illustrates a situation when setup time constraint is violated. One can check for setup time violations by checking whether

$$t_{ck} - t_{p\max} - t_{c\max} - t_{su} \geq 0 \tag{1-34-b}$$

When a designer creates a design, typically the flip-flops and gates are selected from a vendor's design library. Hence parameters such as $t_{p\max}$ and t_{su} are generally fixed for a designer. Of course the designer can check whether a different design library with more desirable $t_{p\max}$ and t_{su} is available for use, but in general, the strategy during timing analysis is to adjust the clock frequency of the circuit or the overall combinational delay of the logic. Often, the clock frequency specification comes from the customer's requirements or the architecture teams; therefore, the designers often have to "meet" timing by ensuring correct combinational delays.

Rule No. 2 Hold-time rule for flip-flop to flip-flop path: Minimum circuit delays should be long enough to satisfy flip-flop hold time.
For proper synchronous operation, the data launched by flip-flop 1 on edge E_1 of clock CK_1 should not be captured by flip-flop 2 on edge E_1 of clock CK_1. This can be understood by thinking about Rule No. 1. According to Rule No. 1, in Figure 1-37 at edge E_2, FF_2 should capture the data launched by FF_1 on the previous edge (i.e., edge E_1). For this to happen successfully, the old data should remain stable at edge E_2 until FF_2's hold time elapses. When FF_2 is capturing this old data at edge E_2, FF_1 has started to launch new data on edge E_2, which should be captured by FF_2 only at edge E_3. A hold-time violation could occur if the data launched by FF_1 at E_2 is fed through the combinational circuit and causes D_2 to change too soon after the clock edge E_2. The new data being launched by FF_1 takes at least $t_{p\min}$ time to pass through FF_1 and at least $t_{c\min}$ to pass through the combinational circuitry. Hence, the hold time is satisfied if

$$t_{p\min} + t_{c\min} \geq t_h \tag{1-35}$$

Figure 1-37 illustrates a situation where hold-time is satisfied. When checking for hold-time violations, the worst case occurs when the timing parameters have their minimum values. Since $t_{p\min} > t_h$ for normal flip-flops, a hold-time violation due to Q changing does not usually occur.

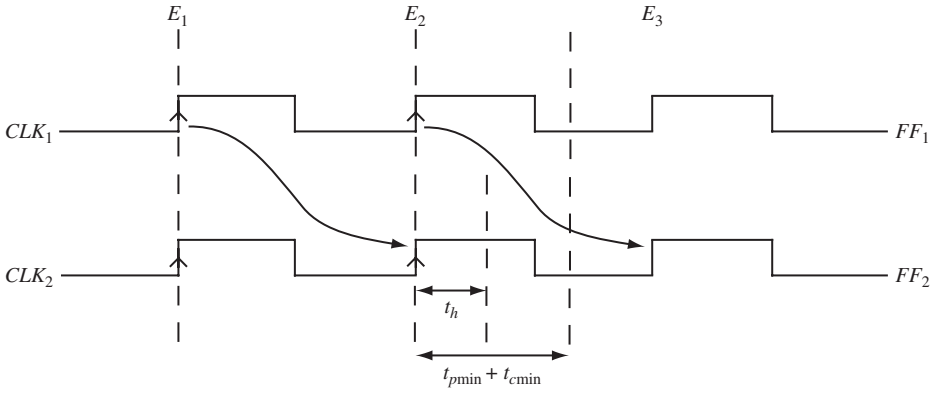


FIGURE 1-37: Timing Diagrams Illustrating Hold Time in Flip-Flop Path

One should note that equation (1-35) does not have the clock frequency in it. Therefore, **if a circuit has a hold-time violation, it cannot be corrected by changing the clock frequency of the circuit.** To correct a hold-time violation, the circuit must be redesigned. In general, to avoid hold-time violations, one needs more combinational delays. Note that this is the opposite of what is desired to meet setup time constraints.

Designing shift registers and counters by chaining together flip-flops is very easy from a functional perspective, however, it is very difficult to meet hold-time constraints, because combinational circuit delay is zero. One way to correct such designs is by inserting buffers between flip-flops as in Figure 1-38.

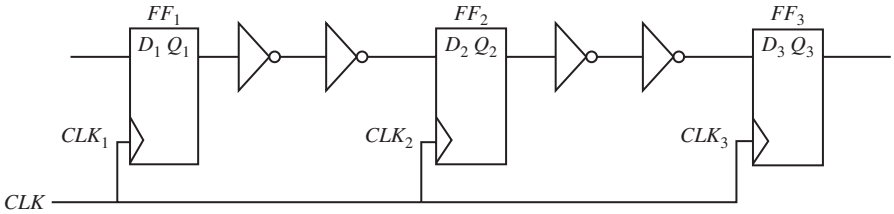


FIGURE 1-38: Shift Register with Buffers for Meeting Hold-Time Constraints

Timing Rules for Input to Flip-Flop Paths

Now let us consider a timing path from primary input to flip-flop as in Figure 1-39. The changes in primary input X should happen such that the value propagates to the flip-flop input satisfying both setup and hold-time constraints. In other words, flip-flop setup time and hold time dictate when primary inputs are allowed to change.

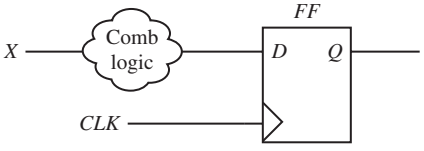


FIGURE 1-39: Input to Flip-Flop Path Timing

Rule No. 3 Setup time rule for input to flip-flop path: External input changes to the circuit should satisfy flip-flop setup time.

A setup time violation could occur if the X input to the circuit changes too close to the active edge of the clock. When the X input to a sequential circuit changes, we must make sure that the input change propagates to the flip-flop inputs such that the setup time is satisfied before the active edge of the clock. If X changes at time t_x before the active edge of the clock (see Figure 1-40), then it could take up to the maximum propagation delay of the combinational circuit before the change in X propagates to the flip-flop input. There should still be a margin of t_{su} left before the edge of the clock. Hence, the setup time is satisfied if

$$t_x \geq t_{cx\max} + t_{su} \tag{1-36}$$

where $t_{cx\max}$ is the maximum propagation delay from X to the flip-flop input.

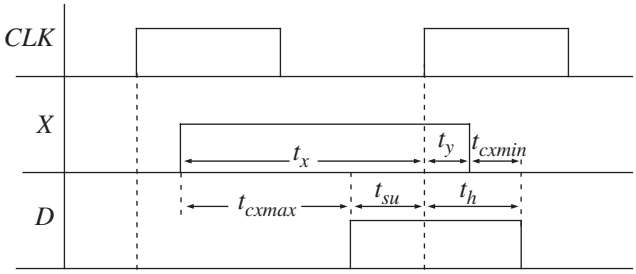
Rule No. 4 Hold-time rule for input to flip-flop path: External input changes to the circuit should satisfy flip-flop hold times.

In order to satisfy the hold time, we must make sure that X does not change too soon after the clock. If a change in X propagates to the flip-flop input in zero time, X should not change for a duration of t_h after the clock edge. Fortunately, it takes some positive propagation delay for the change in X to reach the flip-flop. If $t_{cx\min}$ is the minimum propagation delay from X to the flip-flop input, changes in X will not reach the flip-flop input until at least a time of $t_{cx\min}$ has elapsed after the clock edge. So, if X changes at time t_y after the active edge of the clock, then the hold time is satisfied if

$$t_y \geq t_h - t_{cx\min} \tag{1-37}$$

If t_y is negative, X can change before the active clock edge and still satisfy the hold time.

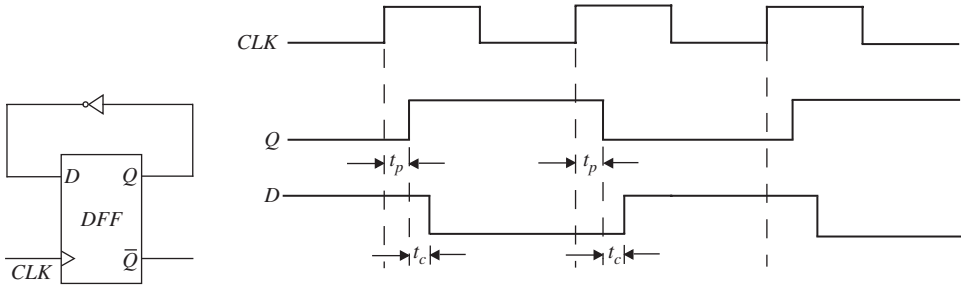
FIGURE 1-40: Setup and Hold Timing for Changes in X



Given a circuit, one can determine the safe frequency of operation and safe regions for input changes using the foregoing principles.

Consider a simple circuit of the form of Figure 1-41(a). The output of a D flip-flop is fed back to its input through an inverter. From a timing perspective, this circuit is equivalent to the circuit in Figure 1-36(a). Assume a clock as indicated by the waveform CLK in Figure 1-41(b). If the current output of the flip-flop is 1, a value of 0 will appear at the flip-flop's D input after the propagation delay of the inverter. Assuming that the next active edge of the clock arrives after the setup time has elapsed, the output of the flip-flop will change to 0. This process will continue yielding the output Q of the flip-flop to be a waveform with twice the period of the clock. Essentially the circuit behaves as a frequency divider.

FIGURE 1-41: Simple Frequency Divider



(a) A frequency divider

(b) Frequency divider timing diagram

If we increase the frequency of the clock slightly, the circuit will still work, yielding half of the increased frequency at the output. However, if we increase the frequency to be very high, the output of the inverter may not have enough time to stabilize and meet the setup time requirements. Similarly, if the inverter was very fast and fed the inverted output to the D input extremely quickly, timing problems will occur, because the hold time of the flip-flop may not be met. So one can easily see a variety of ways in which timing problems could arise from propagation delays and setup- and hold-time requirements.

Timing Rules Nos. 1 and 2 can be applied to this circuit, and it can be seen that the maximum clock frequency of this circuit for proper operation can be derived from equation (1-34). If the minimum clock period is denoted by t_{ckmin},

$$t_{ckmin} = t_{p\max} + t_{c\max} + t_{su}$$

Hence maximum clock frequency $f_{\max}$ is given by:

$$f_{max} + 1/(t_{p\max} + t_{c\max} + t_{su}) \tag{1-38}$$

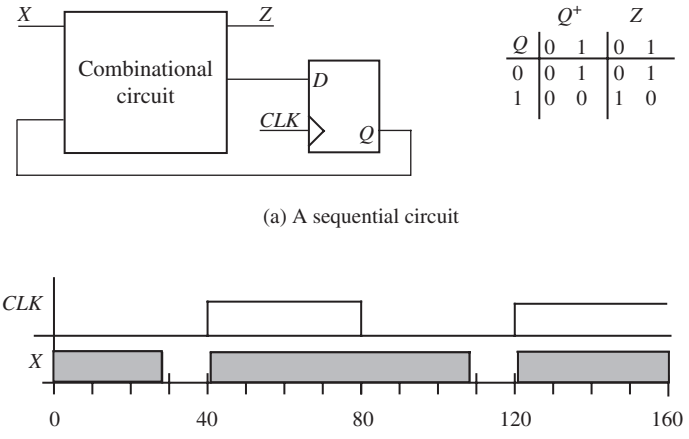
If the minimum and maximum delays of the inverter are 1 ns and 3 ns, and if $t_{p\min}$ and $t_{p\max}$ are 5 ns and 8 ns, the maximum frequency at which it can be clocked can be derived using equation 1-38. Assume that the setup and hold times of the flip-flop are 4 ns and 2 ns. For proper operation, $t_{ck} \geq t_{p\max} + t_{c\max} + t_{su}$. In this example, $t_{p\max}$ for the flip-flops is 8 ns, $t_{c\max}$ is 3 ns, and t_{su} is 4 ns. Hence,

$$t_{ck} \geq 8 + 3 + 4 = 15\,\text{ns}$$

The maximum clock frequency is then $1/t_{ck} = 66.67$ MHz. One should also make sure that the hold-time requirement is satisfied. Hold-time requirement means that the D input should not change before 2 ns after the clock edge. This will be satisfied if $t_{p\min} + t_{c\min} \geq 2$ ns. In this circuit, $t_{p\min}$ is 5 ns and $t_{c\min}$ is 1 ns. Thus the Q output is guaranteed not to change until 5 ns after the clock edge and at least 1 ns more should elapse before the change can propagate through the inverter. Hence the D input will not change until 6 ns after the clock edge, which automatically satisfies the hold-time requirements. Since there are no external inputs, these are the only timing constraints that we need to satisfy.

Now consider a circuit as shown in Figure 1-42(a). Assume that the delay of the combinational circuit is in the range 2 to 4 ns, the flip-flop propagation delays are in the range 5 to 10 ns, the setup time is 8 ns, and hold time is 3 ns. In order to satisfy the setup time, the clock period has to be greater than $t_{p\max} + t_{c\max} + t_{su}$. So

$$t_{ck} \geq 10 + 4 + 8 = 22\,\text{ns}$$



(a) A sequential circuit

(b) Safe regions for changes in X

FIGURE 1-42: Safe Regions for Input Changes

The hold-time requirement is satisfied if the output does not change until 3 ns after the clock. Here, the output is not expected to change until $t_{p\min} + t_{c\min}$. Since $t_{p\min}$ is 5 ns and $t_{c\min}$ is 2 ns, the output is not expected to change until 7 ns, which automatically satisfies the hold-time requirement. This circuit has external inputs that allow us to identify safe regions where the input X can change using

requirements (iii) and (iv) in the foregoing list. The *X* input should be stable for a duration of $t_{cx\max} + t_{su}$ (i.e., 4 ns + 8 ns) before the clock edge. Similarly, it should be stable for a duration of $t_h - t_{cx\min}$ (i.e., 3 ns − 0.2 ns) after the clock edge. Thus, the *X* input should not change 12 ns before the clock edge and 1 ns after the clock edge. Although the hold time is 3 ns, we see that the input *X* can change 1 ns after the clock edge, because it takes at least another 2 ns (minimum delay of combinational circuit) before the input change can propagate to the *D* input of the flip-flop. The shaded regions in the waveform for *X* indicate safe regions where the input signal *X* may change without causing erroneous operation in the circuit.

In a typical sequential circuit, there are often millions of timing paths that need to be considered in deriving the maximum clock frequency. The maximum frequency must be determined by locating the longest path among all the timing paths in the circuit.

Example

Consider the circuit in Figure 1-43 with the following minimum/maximum delays:

CLK-to-Q for flip-flop A: 7 ns/9 ns

CLK-to-Q for flip-flop B: 8 ns/10 ns

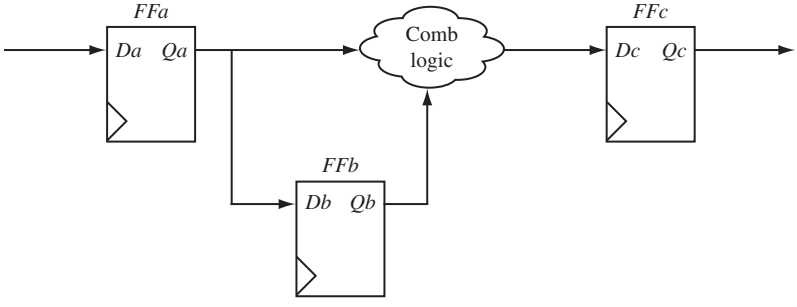
CLK-to-Q for flip-flop C: 9 ns/11 ns

Combinational logic: 3 ns/4 ns

Setup time for flip-flops: 2 ns

Hold time for flip-flops: 1 ns

FIGURE 1-43: Circuit with Three Flip-Flops



Compute the delays for all timing paths in this circuit and determine the maximum clock frequency allowed in this circuit.

Answer: Remember that a timing path starts at either a primary input or at the input of a flip-flop. A path terminates at the input of a flip-flop or at a primary output.

Delay for path from flip-flop A to B = $t_{clk\text{-}to\text{-}Q(A)} + t_{su}(B) = 9\text{ ns} + 2\text{ ns} = 11\text{ ns}$

Delay for path from flip-flop A to C = $t_{clk\text{-}to\text{-}Q(A)} + t_{combo} + t_{su}(C) = 9\text{ ns} + 4\text{ ns} + 2\text{ ns} = 15\text{ ns}$

Delay for path from flip-flop B to C = $t_{clk\text{-}to\text{-}Q(B)} + t_{combo} + t_{su}(C) = 10\text{ ns} + 4\text{ ns} + 2\text{ ns} = 16\text{ ns}$

Delay for path from input to flip-flop A = $t_{su}(A) = 2\text{ ns} = 2\text{ ns}$

Delay for path from flip-flop C to output = $t_{clk\text{-}to\text{-}Q(C)} = 11\text{ ns}$

Since the delay for path from B to C is the largest of the path delays, the maximum clock frequency is determined by this delay of 16 ns. The frequency is $1/t_{\min} = 1/16\,\text{ns} = 62.5$ MHz.

Example

Consider the circuit in Figure 1-35 with the following minimum/maximum delays:

CLK-to-Q for flip-flop 1: 5 ns/8 ns

CLK-to-Q for flip-flop 2: 7 ns/9 ns

XOR Gate: 4 ns/6 ns

AND Gate: 1 ns/3 ns

Setup time for flip-flops: 5 ns

Hold time for flip-flops: 2 ns

(a) What is the minimum clock period that this circuit can be safely clocked at?

Answer: Since XOR gate delay is higher than the AND gate delay, and the second flip-flop's delay is greater than that of the first flip-flop, the path from the second flip-flop to input of the second flip-flop via the XOR is the longest path. This path determines the maximum clock frequency. The maximum frequency is dictated by

$$f_{max} = 1/(t_{flip\text{-}flop\text{-}max} + t_{XORmax} + t_{su})$$
$$= 1/(9 + 6 + 5) = 1/20\,\text{ns} = 50\ \text{MHz}$$

(b) What is the earliest time after the rising clock edge that input A can safely change?

Answer: The earliest time after the rising clock edge that A can safely change can be obtained from equation (1-37)

$$t_y = t_h - t_{ANDmin} = 2\,\text{ns} - 1\,\text{ns} = 1\,\text{ns}$$

(c) What is the latest time before the rising clock edge that input A can safely change?

Answer: The latest time before the rising clock edge that A can safely change can be obtained from equation (1-36)

$$t_x = t_{AND\max} + t_{su} = 3\,\text{ns} + 5\,\text{ns} = 8\,\text{ns}$$

1.10.3 Glitches In Sequential Circuits

Sequential circuits often have external inputs that are asynchronous. Temporary false values called glitches can appear at the outputs and next states. For example, if the state table of Figure 1-23(b) is implemented in the form of Figure 1-17, the timing waveforms are as shown in Figure 1-44. Propagation delays in the flip-flop have been neglected; hence state changes are shown to coincide with clock edges. In this example, the input sequence is 00101001, and X is assumed to change in the middle of the clock pulse. At any given time, the next state and Z output can be read from the next state table. For example, at time t_a, State = S_5 and $X = 0$, so Next State = S_0 and $Z = 0$. At time t_b following the rising edge of the clock,

State = S_0 and X is still 0, so Next State = S_1 and $Z = 1$. Then X changes to 1, and at time t_c Next State = S_2 and $Z = 0$. Note that there is a *glitch* (sometimes called a false output) at t_b. The Z output momentarily has an incorrect value at t_b, because the change in X is not exactly synchronized with the active edge of the clock. The correct output sequence, as indicated on the waveform, is 1 1 1 0 0 0 1 1. Several glitches appear between the correct outputs; however, these are of no consequence if Z is read at the right time. The glitch in the next state at t_b (S_1) also does not cause a problem, because the next state has the correct value at the active edge of the clock.

The timing waveforms derived from the circuit of Figure 1-26 are shown in Figure 1-45. They are similar to the general timing waveforms given in Figure 1-44 except that State has been replaced with the states of the three flip-flops, and a propagation delay of 10 ns has been assumed for each gate and flip-flop.

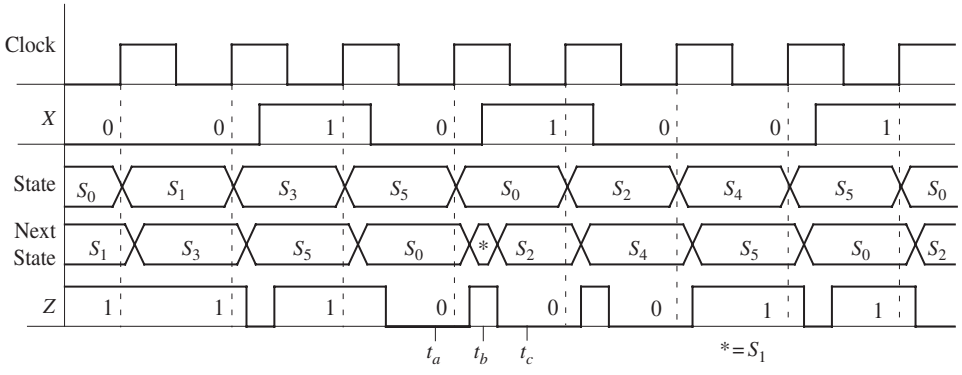


FIGURE 1-44: Timing Diagram for Code Converter

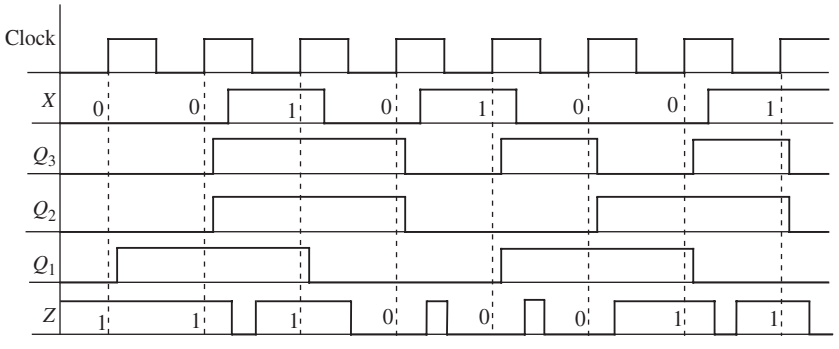


FIGURE 1-45: Timing Diagram for Figure 1-26

1.10.4 Synchronous Design

One of the most commonly used digital design techniques is *synchronous design*. In this type of design, a clock is used to synchronize the operation of all flip-flops, registers, and counters in the system. Synchronous circuits are more reliable than asynchronous circuits. In synchronous circuits, events are expected to occur immediately following the active edge of the clock. Outputs from one part have a full clock cycle to propagate to the next part of the circuit. Synchronous design philosophy makes design and debugging easier as compared with asynchronous techniques. But synchronous designs consume more power than asynchronous designs because of the power consumed in the clock distribution network. Although asynchronous designs can reduce power consumption, it is very difficult

to get timing issues under control; hence, despite their high power consumption, designers favor synchronous designs.

Figure 1-46 illustrates a synchronous digital system. Assume that the system is built from several modules or devices. The devices could be flip-flops, registers, counters, adders, multipliers, and so forth. All of the sequential devices are synchronized with respect to the same clock in a synchronous system. A traditional way to view a digital system is to consider it as a control section plus a data section. The various devices shown in Figure 1-46 are part of the data section. The control section is a sequential machine that generates control signals to control the operation of the data section. For example, if the data section contains a shift register, the control section may generate signals that determine when the register is to be loaded (*Ld)* and when it is to be shifted (*Sh)*. A common clock synchronizes the operation of the control and data sections. The data section may generate status signals (not shown in this figure) that affect the control sequence. For example, if a data operation produces an arithmetic overflow, then the data section might generate a condition signal *V* to indicate an overflow. The control section is also called *controller* and the data section is often called *architecture* or *data path*.

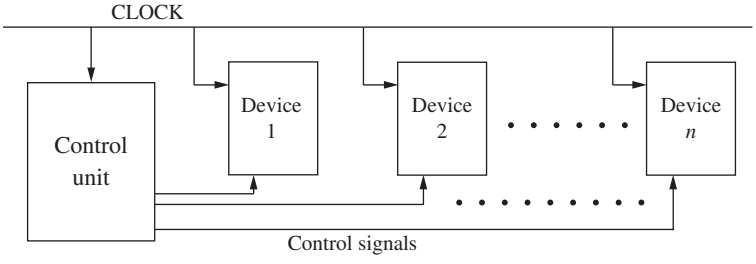


FIGURE 1-46: A Synchronous Digital System

In a synchronous digital system, one desires to see all changes happen immediately at the active edge of the clock, but that might not happen in a practical circuit. Modern integrated circuits (ICs) are fabricated at feature sizes such as or smaller than 0.1 microns. Modern microprocessors are clocked at several gigahertz. In these chips, wire delays are significant as compared with the clock period. Even if two flip-flops are connected to the same clock, the clock edge might arrive at the two flip-flops at different times due to unequal wire delays. If unequal amounts of combinational circuitry (e.g., buffers or inverters) are used in the clock path to different devices, that also could result in unequal delays, making the clock reach different devices at slightly different times. This problem is called **clock skew**. **Clock skew** refers to the absolute time difference in clock signal arrival between two points in the clock network. Clock skew is often caused by delays in the interconnect within the clock distribution network. It can also be caused by the combinational logic used to selectively gate the clock of certain devices.

Timing Rules for Circuits with Skew

When clock skew is present in a circuit, the timing rules Nos. 1 and 2 get appropriately modified. A positive skew means the capturing flip-flop gets the clock delayed with reference to the launching flip-flop. For a circuit with a positive skew as shown in Figure 1-47(a), the timing rules are as follows:

Rule No. 5: $$t_{ck} \geq t_{p\max} + t_{c\max} - t_{skew} + t_{su} \tag{1-39}$$

Rule No. 6: $$t_{p\min} + t_{c\min} \geq t_h + t_{skew} \tag{1-40}$$

Positive skew is good for setup time, but it is bad for hold time.

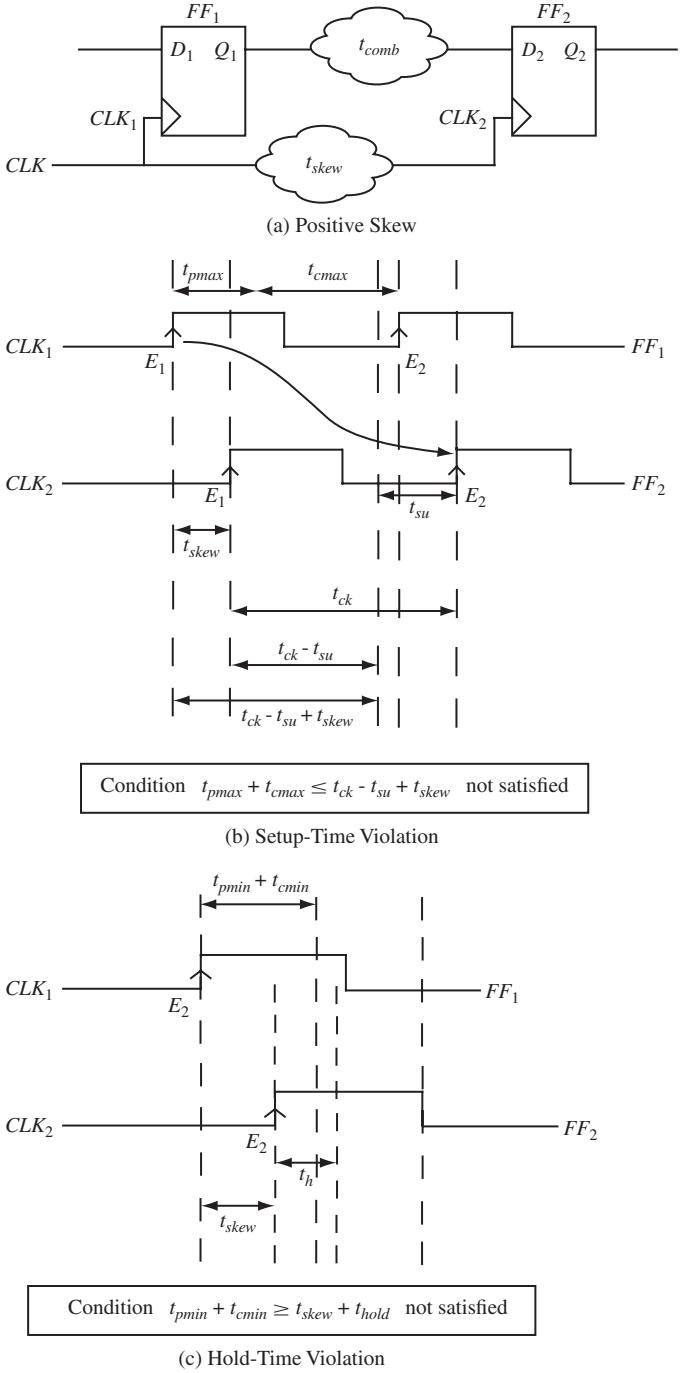


FIGURE 1-47: Illustration of Skew and Timing Violations

Negative skew means that the launching flip-flop gets the clock delayed with reference to the capturing flip-flop. Negative skew is illustrated in Figure 1-48.

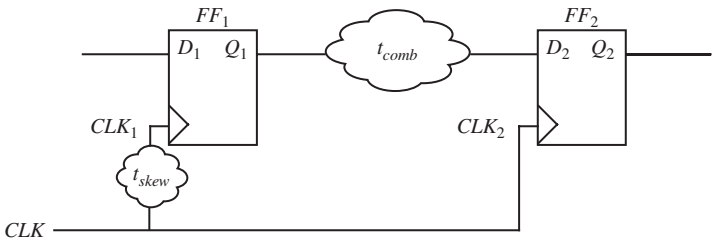


FIGURE 1-48: Negative Skew - CLK1 Is Delayed with Respect to CLK2

For a circuit that has a negative skew, the timing rules are given by the following equations:

$$t_{ck} \geq t_{p\max} + t_{c\max} + t_{skew} + t_{su} \tag{1-41}$$

$$t_{p\min} + t_{c\min} \geq t_h - t_{skew} \tag{1-42}$$

Negative skew is good for hold time, but it is bad for setup time.

Example

Consider the circuit shown in Figure 1-47(a) with the following delays:

CLK-to-Q for Flip-flops: 7 ns/9 ns

Combinational Delay: 4 ns/6 ns

Setup Time for Flip-Flops: 5 ns

Hold Time for Flip-Flops: 2 ns

(a) If skew for the second flip-flop is 3 ns, what is the maximum clock frequency? Compare it with the clock frequency if no skew is present.

Answer: This is a case of positive skew.

$$\begin{aligned} t_{ck} &= t_{p\max} + t_{c\max} - t_{skew} + t_{su} \\ &= 9\,\text{ns} + 6\,\text{ns} - 3\,\text{ns} + 5\,\text{ns} \\ &= 17\,\text{ns} \end{aligned}$$

The maximum clock frequency when skew is present is 1/17 ns (i.e., 58.82 MHz), whereas without skew the circuit could handle only a maximum frequency of 1/20 ns (i.e., 50 MHz).

(b) What is the biggest skew that the circuit in Figure 1-47(a) can take while meeting the hold-time constraint for this circuit?

Answer:

$$\begin{aligned} & t_{p\min} + t_{c\min} \geq t_h + t_{skew} \\ & 7\,\text{ns} + 4\,\text{ns} \geq 2\,\text{ns} + t_{skew} \\ & 9\,\text{ns} \geq t_{skew} \end{aligned}$$

Skew must be less than 9 ns.

(c) If skew for the first flip-flop in Figure 1-48 is 3 ns, what is the maximum clock frequency? Compare it with the clock frequency if no skew is present.

Answer:

$$t_{ck} = t_{p\max} + t_{c\max} + t_{skew} + t_{su}$$

$$= 9\,\text{ns} + 6\,\text{ns} + 3\,\text{ns} + 5\,\text{ns}$$

$$= 23\,\text{ns}$$

The maximum clock frequency when skew is present is 1/23 ns (i.e., 43.47 MHz), whereas without skew the circuit could handle a maximum frequency of 1/20 ns (i.e., 50 MHz).

(d) What is the biggest skew that the circuit in Figure 1-48 can take while meeting the hold-time constraint for this circuit?

Answer:

$$t_{p\min} + t_{c\min} \geq t_h - t_{skew}$$

$$7\,\text{ns} + 4\,\text{ns} + t_{skew} \geq 2\,\text{ns}$$

Since the first flip-flop's clock is delayed by t_{skew} and it takes an additional 11 ns to reach the second flip-flop, there is no possibility this signal change can cause a hold-time violation.

$$t_{skew} \geq -9\,\text{ns}$$

If the skew at flip-flop 1 increases, there will be no hold-time violation, but of course the maximum allowable clock frequency will reduce.

There are also problems that occur due to glitches in control signals. Consider Figure 1-49, which illustrates the operation of a digital system that uses devices that change state on the falling edge of the clock. Several flip-flops may change state in response to this falling edge. The time at which each flip-flop changes state is determined by the propagation delay for that flip-flop. The changes in flip-flop states in the control section will propagate through the combinational circuit that generates the control signals, and some of the control signals may change as a result. The exact times at which the control signals change depend on the propagation delays in the gate circuits that generate the signals as well as the flip-flop delays. Thus, after the falling edge of the clock, there is a period of uncertainty during which control signals may change. Glitches and spikes may occur in the control signals due to hazards. Furthermore, when signals are changing in one part of the circuit, noise may be induced in another part of the circuit. As indicated by the cross-hatching in Figure 1-49, there is a time interval after each falling edge of the clock in which there may be noise in a control signal (*CS*), and the exact time at which the control signal changes is not known.

If we want a device in the data section to change state on the falling edge of the clock only if the control signal $CS = 1$, we can AND the clock with *CS*, as shown in Figure 1-50(a). This technique is called clock gating. The transitions will occur in synchronization with the clock CLK except for a small delay in the AND gate. The gated CLK signal is clean because the clock is 0 during the time interval in which the switching transients occur in *CS*.

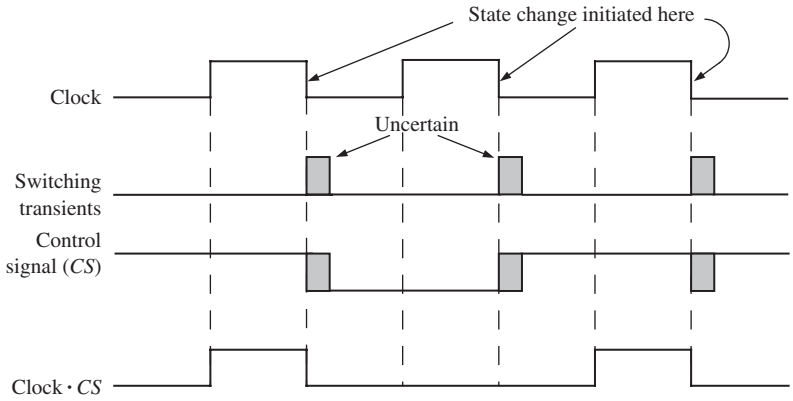
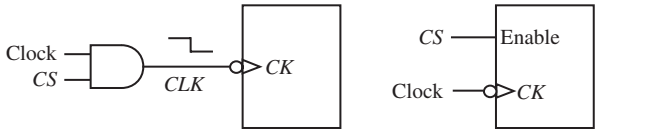


FIGURE 1-49: Timing Chart for System with Falling-Edge Devices

Gating the clock with the control signal, as illustrated in Figure 1-50(a) can solve some synchronization problems. However clock gating can also lead to clock skew and additional timing problems in high-speed circuits. Instead of gating the clock with the control signal, it is more desirable to use devices with clock enable (CE) pins and feed the control signal to the enable pin, as illustrated in Figure 1-50(b). Many registers, counters, and other devices used in synchronous systems have an enable input. When enable = 1, the device changes state in response to the clock, and when enable = 0, no state change occurs. Use of the enable input eliminates the need for a gate on the clock input, and associated timing problems are avoided.



(a) Control signal gating

(b) Use of a clock enable (*CE*) input to synchronize

FIGURE 1-50: Techniques Used to Synchronize Control Signals

We discourage designers from gating clocks or feeding the output of combinational circuits to clock inputs. While clock skew from wire delays is unavoidable to some extent, clock skew due to combinational circuitry in the clock path can easily be avoided. Circuits as in Figure 1-51 should be avoided as much as possible to minimize timing problems.

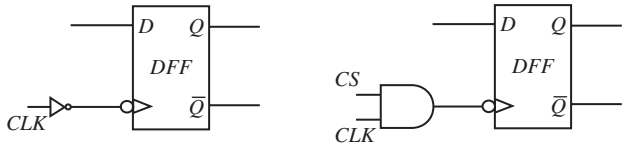


FIGURE 1-51: Examples of Circuits to Avoid

If devices do not have enables and synchronous operation cannot be obtained without clock gating, one should pay attention to gating the clocks correctly. A device with negative edge triggering can be made to function correctly by ANDing

the clock signal with the control signal as shown in Figure 1-50(a). In the following paragraphs, we describe issues associated with control signal gating for positive-edge–triggered devices.

Figure 1-52 illustrates the operation of a digital system that uses devices that change state on the rising edge of the clock. In this case, the switching transients that result in noise and uncertainty will occur following the rising edge of the clock. The cross-hatching indicates the time interval in which the control signal *CS* may be noisy. If we want a device to change state on the rising edge of the clock when $CS = 1$, transition is expected at (a) and (c), but no change is expected at (b) since $CS = 0$ when the clock edge arrives. In order to create a gated control signal, it is tempting to AND the clock with *CS*, as shown in Figure 1-53(a). The resulting signal, which goes to the CK input of the device, may be noisy and timed incorrectly. In particular, the CLK_1 pulse at (a) will be short and noisy. It may be too short to trigger the device, or it may be noisy and trigger the device more than once. In general, it will be out of synchronization with the clock, because the control signal does not change until after some of the flip-flops in the control circuit have changed state. The rising edge of the pulse at (b) again will be out of synch with the clock, and it may be noisy. But even worse, the device will trigger near point (b) when it should not trigger there at all. Since $CS = 0$ at the time of the rising edge of the clock, triggering should not occur until the next rising edge, when $CS = 1$.

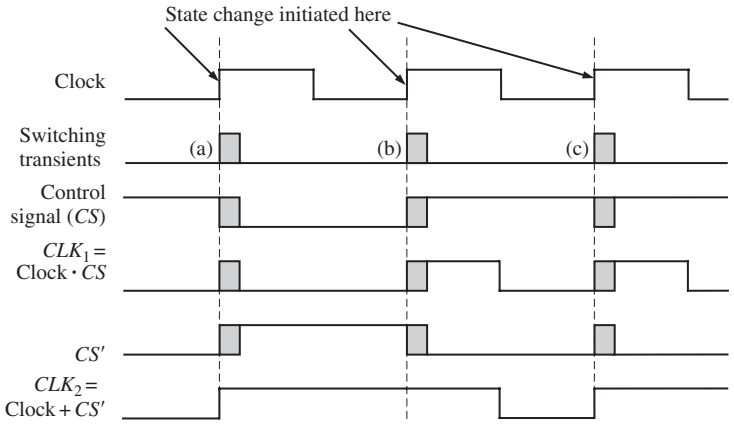
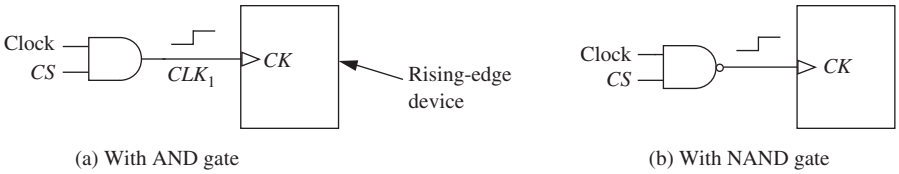


FIGURE 1-52: Timing Chart for System with Rising-Edge Devices

For a rising-edge device, if one changed the AND gate in Figure 1-50 to NAND gate as in Figure 1-53(b), it would be incorrect because the synchronization will happen at the wrong edge. The correct way to gate the control signal will be as in Figure 1-54, which will result in the *CK* input to the device having a positive edge only when the control signal is positive and the clock is going to have a positive edge. The *CK* input is then

$$CLK_2 = (CS \cdot clock')' = CS' + clock$$

The last waveform in Figure 1-45 illustrates this gated control signal. Even though this circuit can solve the synchronization problem, we encourage designers to refrain from gating clocks at all, if possible.



(a) With AND gate (b) With NAND gate

FIGURE 1-53: Incorrect Clock Gating for Rising-Edge Devices

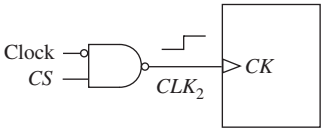


FIGURE 1-54: Correct Control Signal Gating for Rising-Edge Device

In summary, synchronous design is based on the following principles:

- Method: All clock inputs to flip-flops, registers, counters, and the like are driven directly from the system clock.
- Result: All state changes occur immediately following the active edge of the clock signal.
- Advantage: All switching transients, switching noise, and the like occur between clock pulses and have no effect on system performance.

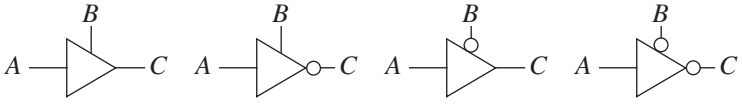
Asynchronous design is generally more problematical than synchronous design. Since there is no clock to synchronize the state changes, problems may arise when several state variables must change at the same time. A race occurs if the final state depends on the order in which the variables change. Asynchronous design requires special techniques to eliminate problems with races and hazards. On the other hand, synchronous design has several disadvantages: In high-speed circuits where the propagation delay in the wiring is significant, the clock signal must be carefully routed so that it reaches all the clock inputs at essentially the same time (i.e., to minimize clock skew). The maximum clock rate is determined by the worst-case delay of the longest path. Because the system inputs may not be synchronized with the clock, use of synchronizers may be required. Synchronous systems also consume more power than asynchronous systems. The clock distribution circuitry in synchronous chips often consumes a significant fraction of the chip's power.

1.11 Tristate Logic and Busses

Normally, if we connect the outputs of two gates or flip-flops together, the circuit will not operate properly. It can also cause damage to the circuit. Hence, when one needs to connect multiple gate outputs to the same wire or channel, one way to do that is by using tristate buffers. Tristate buffers are gates with a high-impedance state (hi-Z) in addition to high and low logic states. The high-impedance state is equivalent to an open circuit. In digital systems, transferring data back and forth between several system components is often necessary. Tristate busses can be used to facilitate data transfers between registers. When several gates are connected on to a wire, what one expects is that at any one point, one of the gates is going to actually drive the wire and the other gates should behave as if they were not connected to the wire. The high-impedance state achieves this.

Tristate buffers can be inverting or non-inverting. The control input can be active high or active low. Figure 1-55 shows four kinds of tristate buffers. *B* is the control input used to enable or disable the buffer output. When a buffer is enabled, the output (*C*) is equal to the input (*A*) or its complement. However, we can connect two tristate buffer outputs, provided that only one output is enabled at a time.

FIGURE 1-55: Four Kinds of Tristate Buffers



B	A	C
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

(a)

B	A	C
0	0	Hi-Z
0	1	Hi-Z
1	0	1
1	1	0

(b)

B	A	C
0	0	0
0	1	1
1	0	Hi-Z
1	1	Hi-Z

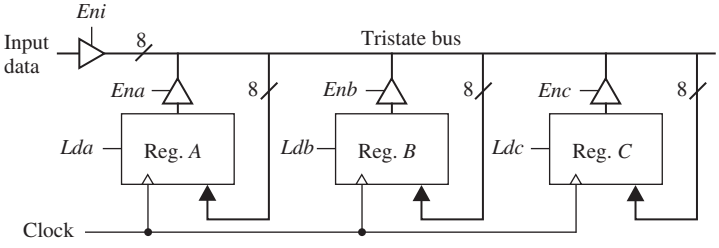
(c)

B	A	C
0	0	1
0	1	0
1	0	Hi-Z
1	1	Hi-Z

(d)

Figure 1-56 shows a system with three registers connected to a tristate bus. Each register is 8 bits wide, and the bus consists of 8 wires connected in parallel. Each tristate buffer symbol in the figure represents 8 buffers operating in parallel with a common enable input. Only one group of buffers is enabled at a time. For example, if *Enb* = 1, the register *B* output is driven onto the bus. The data on the bus is routed to the inputs of register *A*, register *B*, and register *C*. However, data is loaded into a register only when its load input is 1 and the register is clocked. Thus, if *Enb* = *Ldc* = 1, the data in register *B* will be copied into register *C* when the active edge of the clock occurs. If *Eni* = *Lda* = *Ldb* = 1, the input data will be loaded in registers *A* and *B* when the registers are clocked.

FIGURE 1-56: Data Transfer Using Tristate Bus



Problems

1.1 Write out the truth table for the following equation:

$$F = (A \oplus B) \cdot C + A' \cdot (B' \oplus C)$$

1.2 A full subtracter computes the difference of three inputs *X*, *Y*, and B_{in}, where $Diff = X - Y - B_{in}$. When $X < (Y + B_{in})$, the borrow output B_{out} is set. Fill in the

CHAPTER 12
SYNTHESIS

THE SUCCESS of logic synthesis has dramatically cut the design time and pushed HDLs into the forefront of digital designs. Accompanying this success, massive consumer's products are produced such as MP3 players and DVD players, and have changed people's daily lives.

In this chapter, we deal with the principles of logic synthesis and the general architecture of synthesis tools. In general, synthesis can be divided into logic synthesis and high-level synthesis. The former transforms an RTL representation into gate-level netlists while the latter transforms a high-level representation into an RTL representation. The general architecture of synthesis tools is divided into two parts: the front end and the back end. The front end consists of two phases: parsing and elaboration. The back end contains three phases: analysis, optimization and netlist generation.

In order to make good use of synthesis tools, we need to provide the design environment and design constraints along with the RTL code and technology library. The design environment provides the process parameters, I/O port attributes and statistical wireload models for the synthesis tools to synthesize a design. The design constraints provide the clock related parameters, input and output delays and timing exceptions.

Finally, we give some guidelines about how to write a good Verilog HDL code such that it can be acceptable by most logic synthesis tools and can achieve the best compilation times and synthesis results. These guidelines also include clock signals, reset signals and how to partition a design.

12.1 DESIGN FLOW OF ASICS AND FPGA-BASED SYSTEMS

Quite often, it is required to follow a design flow when designing an ASIC or an FPGA-based system. A *design flow* is a set of procedures that allows designers to progress from a specification for an ASIC or FPGA-based system to the final chip or

FPGA implementation in an efficient and error-free way. In this section, we describe the general design flow of designing ASICs and FPGA-based systems.

12.1.1 The General Design Flow

Recall that the RTL design is a mixed style combining both dataflow and behavioral styles with the constraint that the resulting description can be acceptable by synthesis tools. The general design flow of an ASIC and FPGA-based design is shown in Figure 12.1.

From this figure, we know that the design flow can be divided into two major parts: *front end* and *back end*. The front end contains three phases, starting from product requirement, behavioral/RTL description and ending with RTL synthesis, which generates a gate-level netlist. The back end also contains three phases, starting from the structural description of gate-level netlist, physical synthesis and ending with physical specification. In other words, the RTL synthesis is at the heart of the front-end part and the physical synthesis is the essential component of the back-end part.

12.1.1.1 RTL Synthesis Flow The general RTL design flow is shown in Figure 12.2. The RTL design flow starts with product requirement, which is converted into a *design specification*. The specification is then described with RTL behavioral style in Verilog HDL or VHDL. The results are then verified by using a set of test benches written by HDL. This verifying process is called *RTL functional verification*.

The design is synthesized by a logic synthesizer after its function has been verified correctly. This process is denoted as *RTL synthesis* or *logic synthesis*. The function of the logic synthesizer is to convert an RTL description to generic gates and registers and then optimize the logic to improve speed and area. In addition, finite state machine decomposition, data path optimization and power optimization may also be performed

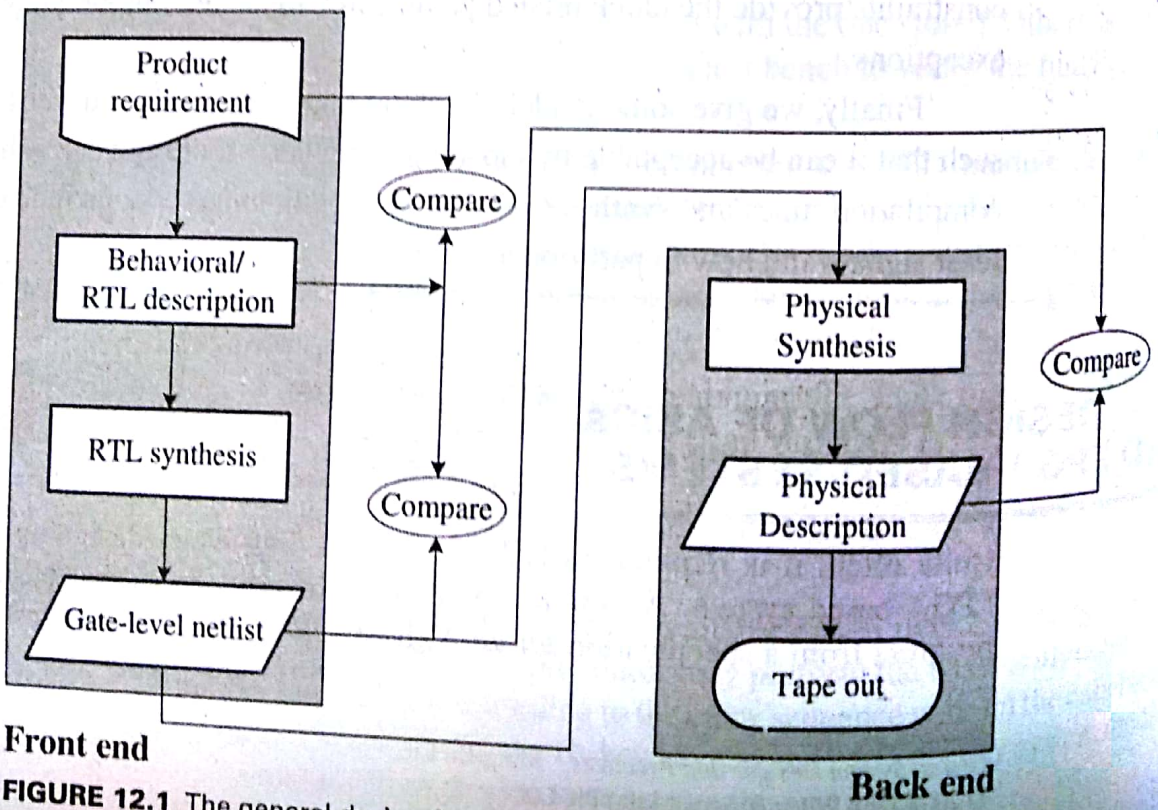


FIGURE 12.1 The general design flow of an ASIC and FPGA-based design

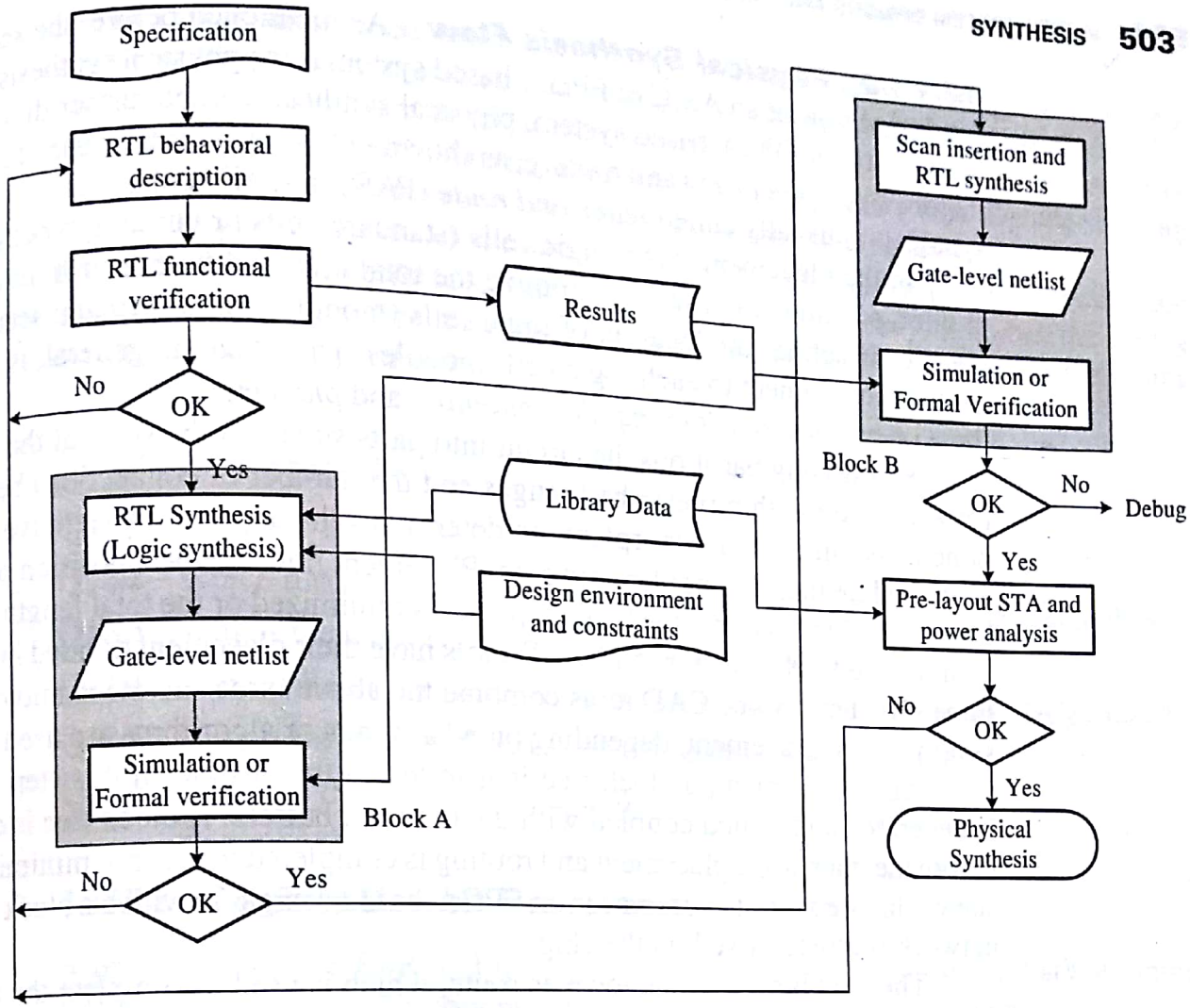


FIGURE 12.2 The general flow of RTL synthesis

at this stage. In general, a logic synthesizer accepts three inputs, *RTL code*, *technology library* and *design environment and constraints*, and generates a gate-level netlist.

After a gate-level netlist is generated, it is necessary to re-run the test benches used in the stage of RTL functional verification to check if they produce exactly the same output for the behavioral and structural descriptions or to perform an RTL versus gate equivalence checking to compare the logical equivalence of the two descriptions.

The next three steps often used in ASIC (namely, cell-based design), but not in an FPGA-based design are shown in the shaded block B, which incorporates the *scan-chain logic insertion*, re-synthesis and verification. In fact, this block may be combined together with block A. The scan-chain (or test logic) insertion step is to insert or modify logic and registers to aid in manufacturing tests. Automatic test pattern generation (ATPG) and built-in self-test (BIST) are usually used in most modern ASIC designs. The details of these topics are addressed in Chapter 16.

The final stage of RTL design flow is the pre-layout static timing analysis (STA) and power consumption analysis. Static timing analysis checks the temporal requirements of the design. We describe the STA in greater detail in Section 13.5. Power analysis estimates the power consumption of the circuit. The power consumption depends on the activity factors of the gates. Power analysis can be performed for a particular set of test vectors by running the simulator and evaluating the total capacitances switched at each clock transition of each node.

12.1.1.2 Physical Synthesis Flow As mentioned before, the second part of the design flow of an ASIC or FPGA-based system is the physical synthesis. Regardless of ASIC or an FPGA-based system, physical synthesis can be further divided into two major stages: *placement* and *routing*, as shown in Figure 12.3. Because of this, physical synthesis is usually called *place and route* (PAR) in CAD tools.

In the placement stage, logic cells (standard cells or building blocks) are placed at fixed positions in order to minimize the total area and wire length. In other words, placement defines the location of logic cells (modules) on a chip and sets aside space for the interconnect to each logic cell (module). This stage, in general, is a mixture of three operations: *partitioning*, *floorplanning* and *placement*.

Partitioning partitions the circuit into parts such that the sizes of the components (modules) are within prescribed ranges and the number of connections between components is minimized. Floorplanning determines the appropriate (relative) location of each module in a rectangular chip area. Placement finds the best position of each module on the chip such that the total chip area is minimized or the total length of the wires is minimized. Of course, not all CAD tools have their placement divided into the above three sub-steps. Some CAD tools combine the above three sub-steps into one big step, simply called placement, depending on what kinds of algorithms are used.

After placement, a clock tree is inserted in the design. In this step a clock tree is generated and routed coupled with the required buffers. A clock tree is often placed before the main logic placement and routing is completed in order to minimize the clock skew. This step is not necessary in an FPGA-based design, in which a clock distribution network is already fixed on the chip.

The next big stage is known as route, which is used to complete the connections of signal nets among the cell modules placed by placement. This stage is further divided

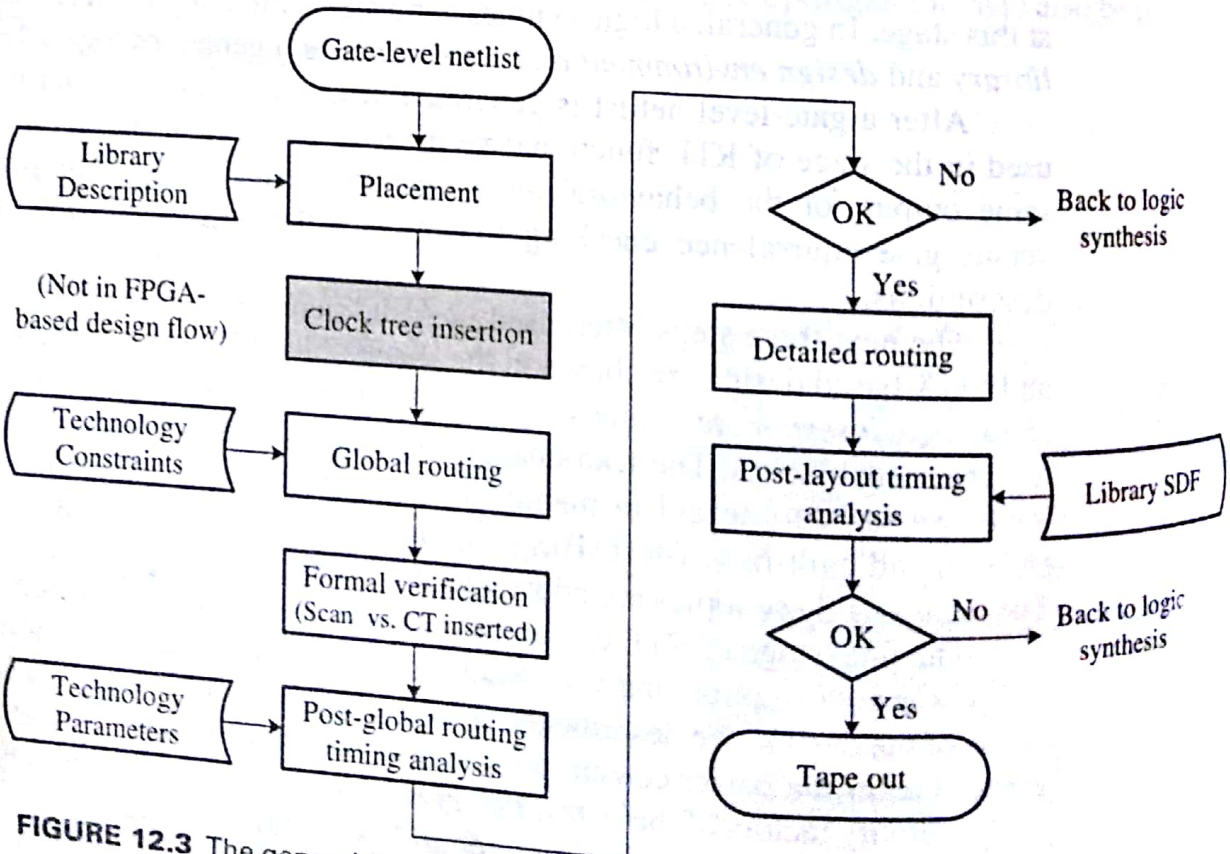


FIGURE 12.3 The general flow of physical synthesis

into two sub-stages: *global routing* and *detailed routing*. Global routing decomposes a large routing problem into small and manageable sub-problems (detailed routing) by finding a rough path for each net in order to reduce chip size, shorten the total length of the wires, and evenly distribute the congestion over the routing area. Detailed routing carries out the actual connections of signal nets among the modules.

After global and detailed routing, the post-global routing and post-layout static timing analysis are performed, separately. These timing analyses re-run the timing analysis with the actual routing loads placed on the gates to check whether the timing constraints are still valid or not.

Review Questions

Q12.1 What is a design flow?

Q12.2 Which phases are included in the front-end part of the general design flow?

Q12.3 Why is the front-end part often called logic synthesis?

Q12.4 Which phases are included in the back-end part of the general design flow?

Q12.5 Why is the back-end part often called physical synthesis?

Q12.6 What are the two major stages of physical synthesis?

12.1.2 Timing-Driven Placement

A problem with the PAR and timing analysis described previously is that the timing information can only be obtained after the layout has been completed. As a result, in order to satisfy the post-layout timing requirement, it may go back to the logic synthesis and starts there again, as illustrated in Figure 12.3. Even more, this process may be repeated several times. One technique to solve this is by incorporating the timing analysis into the placement stage so that the critical path can be placed into the layout with priority. This technique is called *timing-driven placement*. In this section, we will introduce a simple timing-driven placement based on the concept of *slack time*.

Before describing the meaning of slack time, we need to define the *arrival time* and *required time*, as shown in Figure 12.4. Assume that d_i, for $0 \leq i \leq m$, are net

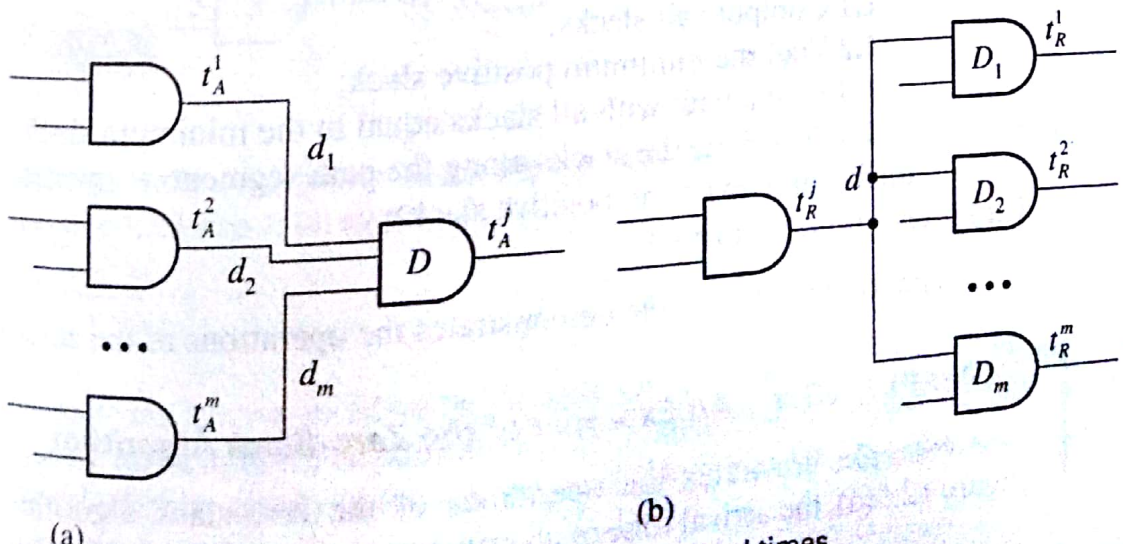


FIGURE 12.4 Definitions of (a) arrival and (b) required times

reason why the synthesis tool produces such a result? Moreover, what would happen if we replace all nonblocking assignments with blocking assignments within the `always` block. Try it on your own system to confirm whatever your answer is.

12.4.6 Memory and Register Files

As we have discussed in Chapter 11, memory and registers or register files often play important roles in designing a digital system. The most commonly used memory modules in digital system design based on synthesis flow are static random-access memories (SRAMs), called RAMs for short. The memory modules have a variety of types, such as RAM, register file, first-in first-out (FIFO) buffer and dual-port RAM.

Memory modules can be constructed in many ways. One way is to use flip-flops or latches. This approach is independent of any synthesis software and type of target system, cell-based or FPGA-based. It is easy to use but inefficient in terms of area because, in general, a flip-flop may take up 10 to 20 times the area of a 6-transistor static RAM cell. Another way is by using standard components supplied by cell library or FPGA vendors. Of course, they depend on the process technology. The third way is to use the RAM compiler supplied by most cell-library vendors. This may be the most area-efficient approach because the size of the memory module can be generally customized to fit into the actual requirement.

Another kind of memory module often used in digital system design is the register file. Register files are usually generated by using a synthesis directive or hand instantiation RAM.

In summary, a flip-flop- or latch-based RAM or register is only applied to the cases in which the size required is small. For the cases where a large size of RAM module or register is required, it is better to use RAM compilers or standard components supplied by the vendors.

Review Questions

Q12.39 Explain why the following statement will infer a latch.

```
always @(enable or data)
   if (enable) y = data;  //infer a latch
```

Q12.40 Explain why the `else` statement in the following is a redundant expression.

```
always @(posedge clk)
   if (enable) y <= data;
   else y <= y;   // a redundant expression
```

12.5 CODING GUIDELINES

Successful synthesis strongly depends on proper partitioning of the design together with a good HDL coding style. A good coding style not only results in reduction of the

chip area (namely, hardware cost) and aids in top-level timing but also produces faster logic. A frequent obstacle to writing HDL code is the software mind-set. *You should think in hardware* when designing a hardware module. In this section, we introduce some coding guidelines to help the reader write a code that achieves the best synthesis results and reduction in compile time [1, 7].

12.5.1 Guidelines for Clocks

Clock signals are the heart of any digital system. For a digital system to be worked properly, the clock signals must be applied very carefully. In this sub-section, we discuss several issues related to clock signals. These are using a single global clock, avoiding using gated clocks, avoiding mixed use of both positive and negative edge-triggered flip-flops in the same design and avoiding using internally generated clock signals.

Using a Single Global Clock In designing a digital system, it should use a single global clock as the preferred clock structure and use positive edge-triggered flip flops as the only memory components, as shown in Figure 12.23(a). For ASIC design, it is necessary to avoid instantiating clock buffers in RTL code manually because clock buffers are normally inserted after synthesis as part of the physical synthesis, as we have mentioned earlier (see Figure 12.3).

Avoiding Using Gated Clocks We should avoid coding gated clocks in an RTL code because of the following reasons. First, clock gating circuits tend to be timing dependent and technology specific. Second, gated clocks may cause clock skews of local clock signals which then may cause hold time violations. Third, gated clocks limit the testability because the logic clocked by a gated clock cannot be made part of a scan chain. A multiplexer may be used to bypass the gated clock in the test mode if gated clocks are required for some reasons.

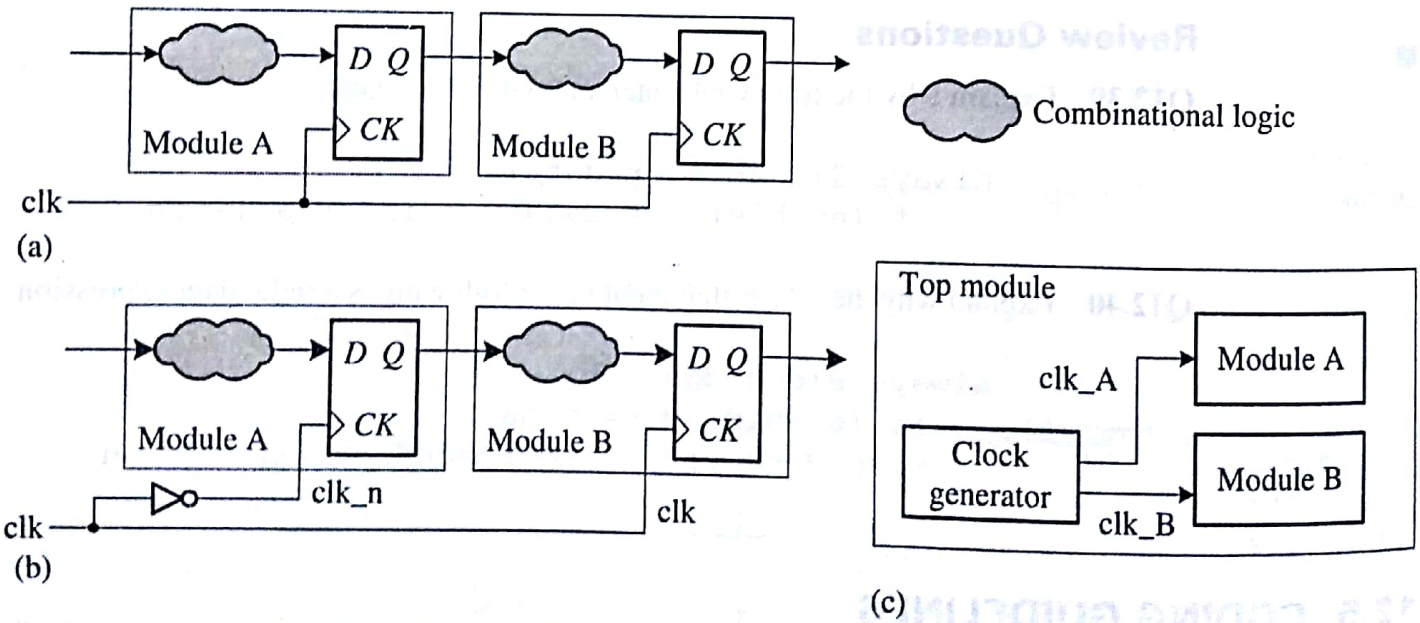


FIGURE 12.23 The clocking schemes for general digital systems: (a) an ideal clock scheme; (b) an example of using both positive and negative edge-triggered flip-flops; (c) using a separate clock module at the top level

Avoiding Mixed use of Both Positive and Negative Edge-Triggered Flip-Flops Although the mixed use of both positive and negative edge-triggered flip-flops in a design is so convenient, it is considered as a bad coding style nowadays because it needs to tackle the stringent timing requirements. However, if this is indeed needed due to performance reasons, the following two problems must be dealt with caution. First, the duty cycle of the clock now becomes a critical issue in the timing analysis. Second, most scan-based testing methodologies require separate handling of positive and negative edge-triggered flip-flops. A good example of using both positive and negative edge-triggered flip-flops is depicted in Figure 12.23(b).

Avoiding Using Internally Generated Clock Signals The final concern about the issues of clock signals is to avoid using internally generated clocks in the design as much as possible. Internally generated clocks, at least, have the following two drawbacks. First, they make it more difficult to constrain the design for synthesis. Second, they limit the testability because the logic driven by the internal clock cannot be made part of a scan chain much like the case of gated clocks. When an internally generated clock, reset or gated clock is required, it is good practice to keep the clock and/or reset generation circuit as a separate module at the top level of the design, as shown in Figure 12.23(c).

12.5.2 Guidelines for Resets

The reset signal plays an important role in any digital system because it initializes the system to its known status by clearing all flip-flops. From the viewpoint of designers, the basic design issues related to reset signals are asynchronous versus synchronous, an internal or external power-on reset, as well as hard versus soft reset when more than one reset signals are used.

Asynchronous Versus Synchronous Reset Both asynchronous and synchronous reset signal generations have their features. Asynchronous reset does not require a free-running clock and does not affect flip-flop data timing. FPGAs provide global reset control signals for all flip-flops associated with them. For a cell-based design, even though all flip-flops have asynchronous reset control signals, it is still harder to implement the system reset circuit since reset is a special signal like clock, which requires a tree of buffers to be inserted at place and route. Moreover, it makes both static timing analysis (or cycle-based simulation) and the automatic scan-chain (test structure) insertion more difficult. In contrast, synchronous reset is easy to synthesize because it is just another synchronous signal to the input. However, it requires a free-running clock, in particular at power-up, for reset to occur. In addition, all cell-based or FPGA-based flip-flops do not support this kind of reset mechanism.

The basic coding styles for both asynchronous and synchronous reset are as follows:

```
// asynchronous reset
always @(posedge clk or posedge reset)
   if  (reset) ...
   else ...
```

```
// synchronous reset
always @(posedge clk)
   if  (reset) ...
   else ...
```

Avoid Internally Generated Conditional Resets The reset signal may be generated internally or externally when power is turned on. However, it is necessary to avoid internally generated conditional resets if possible. When a conditional reset is required, it is necessary to create a separate signal for the reset signal and isolate the conditional reset logic in a separate logic block in order to improve the synthesis result and make the code more readable. For instance, in the following `always` block, the `timer_load_clear` is a conditional reset used to clear the `timer_load` flip-flop to 0 whenever it is activated.

```
always @(posedge gate or negedge reset_n or posedge
 timer_load_clear)
   if  (!reset_n || timer_load_clear)
        timer_load <= 1'b0;
   else timer_load <= 1'b1;
```

The better coding style is to combine both the system reset signals `!reset_n` with the conditional reset signal `timer_load_clear` into one separate block, say `timer_load_reset`, and then apply to the flip-flop, as shown in the following:

```
assign timer_load_reset = !reset_n || timer_load_clear;
always @(posedge gate or posedge timer_load_reset)
   if  (timer_load_reset) timer_load <= 1'b0;
   else timer_load <= 1'b1;
```

More detailed examples concerning this topic can be found in Section 15.4.

12.5.3 Partitioning for Synthesis

Due to the increasing complexity of digital systems, it is necessary to partition a design into many smaller modules so that they can be handled easily. In addition, the purposes of a partition are to obtain a faster compile time and a better synthesis result, as well as the capability to use simpler synthesis strategies to meet timing requirements. In the following, we give some guidelines to carry out a partition.

Register All Outputs In order to make the output drive strengths and input delays predictable, all outputs need to be registered, as shown in Figure 12.24. In this case, there are no combinational logic being placed between the register and the output port (pad).

Keep Related Logic within the Same Module When partitioning a design into smaller modules, it needs to keep in mind that all related combinational logic should

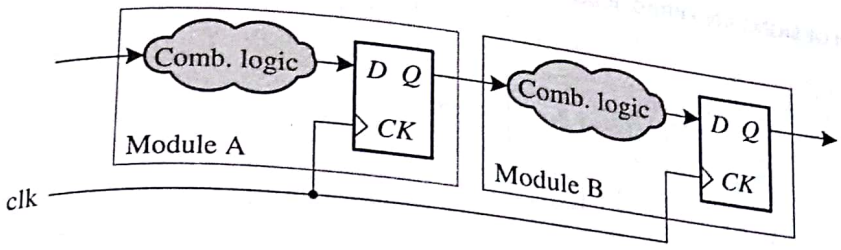


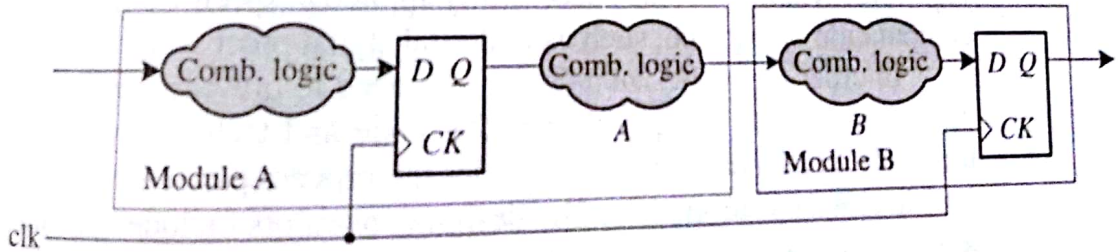
FIGURE 12.24 Register all outputs in a digital system

be put into the same module as possibly as you can. For example, as shown in Figure 12.25(a), the combinational logic *A* and *B* are displaced in two separate modules. It is better to combine them into one module, as shown in Figure 12.25(b). Here, the resulting two modules are also output registered.

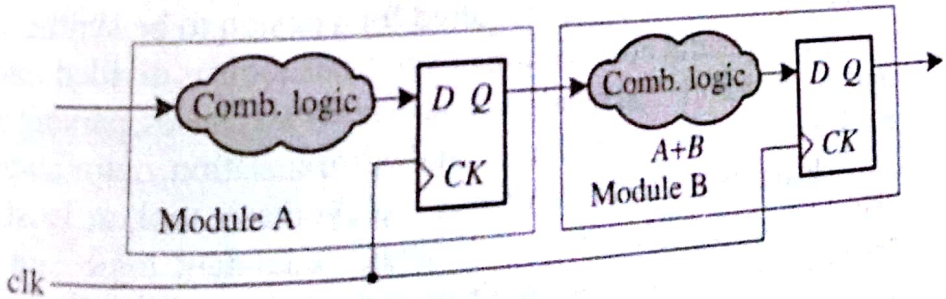
Separating Structural Logic from Random Logic Another guideline for partitioning a design is to separate structural logic from random logic. When partitioning a design, the following features must be kept in mind at all times: to limit a reasonable block size, to partition the top level and to separate I/O pads from boundary scan as well as core logic. In addition, it is necessary to remember that it should not add glue-logic at the top level. Moreover, it should avoid using asynchronous logic in a design.

Synthesis Tools Tend to Maintain the Original Hierarchy As we have mentioned previously, synthesis tools tend to maintain the original hierarchy of a design and only optimize the codes within the same module. This feature has two implications. First, the codes in a design that have different design goals should be separated into different modules so that they can be optimized separately. For example, for a module with the critical path logic, speed optimization is applied to optimize the logic. For a module with the noncritical path logic, the area optimization is applied to optimize the logic.

Second, in order for synthesis tools to consider resource sharing, all relevant resources need to be in the same module. For example, the two adders shown in



(a)



(b)

FIGURE 12.25 Keep all related combinational logic circuits together: (a) bad style; (b) good style

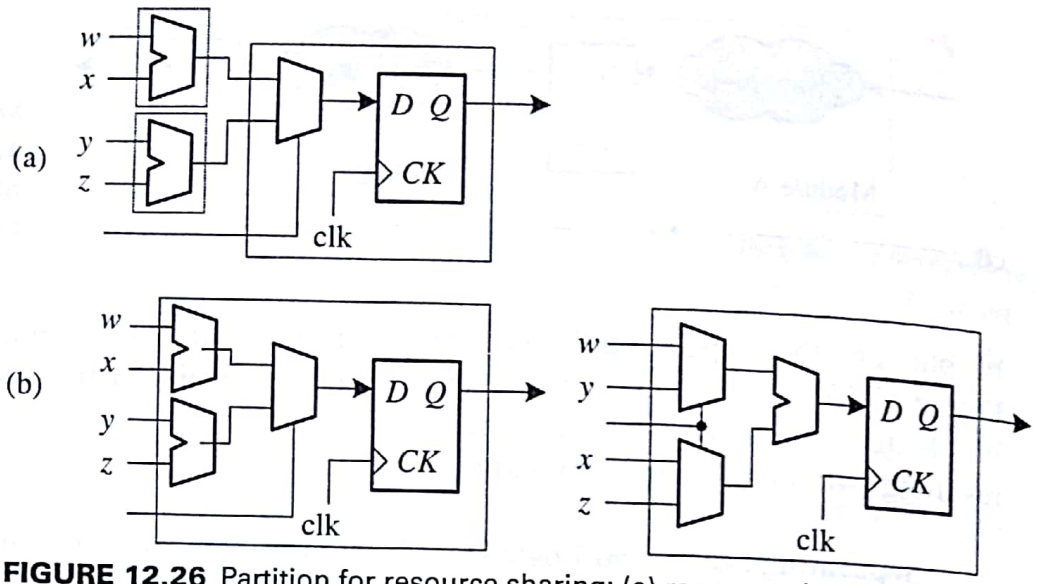


FIGURE 12.26 Partition for resource sharing: (a) resources in different modules cannot be shared; (b) resources in the same module can be shared

Figure 12.26(a) cannot be shared because they are in separate modules. However, the two adders shown in Figure 12.26(b) can be shared because they are in the same module.

SUMMARY

In this chapter, we have dealt with the principles of logic synthesis and introduced how to write a good Verilog HDL code such that it can be acceptable by most logic synthesis tools. In addition, we gave some comments about the constructs that can and cannot be synthesizable.

In general, synthesis can be divided into logic synthesis and high-level synthesis. The former transforms an RTL representation into gate-level netlists while the latter transforms a high-level representation into an RTL representation. At present, logic synthesis is the most commonly used approach when designing general digital systems and high-level synthesis is only successfully applied in specific domains for which intensive computation is required, such as in digital signal processing (DSP) applications.

In order to synthesize a design, we need to provide the design environment and design constraints along with the RTL code and technology library to the synthesis tools. The design environment includes the process parameters, I/O port attributes and statistical wire-load models. The design constraints include the clock related parameters, input and output delays and timing exceptions. Note that both design environment and design constraints are required for a design to be synthesized.

The general architecture of synthesis tools is divided into two parts: the front end and the back end. The front end consists of two phases, parsing and elaboration, while the back end contains three phases: analysis/translation, optimization and netlist generation.

From the viewpoint of users, a synthesis tool, at least, performs the following critical tasks: to detect and eliminate redundant logic and combinational feedback loops as well, to exploit don't-care conditions, to detect unused states and collapse equivalent states, to make state assignments and to synthesize optimal multilevel logic subject to constraints.

It is good practice to use a single global clock as the preferred clock structure and positive edge-triggered flip flops as the only sequential memories. For ASIC design, it is necessary to avoid hand instantiating clock buffers in the RTL code because clock buffers are normally inserted after synthesis as part of the physical synthesis.

The reset signal plays an important role in any digital system because it initializes the system to its known status; namely, it clears all flip-flops. From the viewpoint of designers, the basic design issues of reset signals are: asynchronous versus synchronous, an internal or external power-on reset, as well as hard versus soft reset when more than one reset signals are used.

REFERENCES

1. H. Bhatnagar, *Advanced ASIC Chip Synthesis: Using Synopses Design Compiler and Prime Time,* Kluwer Academic Publishers, Boston, MA, USA, 1999.
2. R.K. Brayton, C. McMullen, G.D. Hachtel and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis,* Kluwer Academic Publishers, Norwell, MA, USA, 1984.
3. M.D. Ciletti, *Modeling, Synthesis and Rapid Prototyping with the Verilog HDL,* Prentice-Hall, Upper Saddle River, NJ, USA, 1999.
4. J. Cong and Y. Ding, "FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* **13**(1), 1–12, 1994.
5. S.H. Gerez, *Algorithms for VLSI Design Automation,* John Wiley & Sons, Inc., New York, NY, USA, 1999.
6. IEEE 1364-2001 Standard, *IEEE Standard Verilog Hardware Description Language,* 2001.
7. M. Keating and P. Bricaud, *Reuse Methodology Manual: For System-on-a-Chip Designs,* Kluwer Academic Publishers, Boston, MA, USA, 2002.
8. M.-Bo. Lin, *Digital System Design: Principles, Practices and ASIC Realization,* 3rd Edn, Chuan Hwa Book Company, Taipei, Taiwan, 2002.
9. S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis,* 2nd Edn, SunSoft Press, 2003.
10. A. Sangiovanni-Vincentelli, A. El Gamal and J. Rose, "Synthesis methods for field programmable gate arrays", *Proceedings of the IEEE,* **81**(7), 1057–1083, 1993.
11. M. Sarrafzadeh and C.K. Wong, *An Introduction to VLSI Physical Design,* McGraw-Hill, New York, NY, USA, 1996.
12. W. Wolf, *FPGA-Based System Design,* Prentice-Hall, Upper Saddle River, NJ, USA, 2004.

PROBLEMS

12.1 Suppose that the switching functions f, g, and h are as follows:

$$f(v, w, x, y, z) = vxy + wxy + z$$

$$g(v, w, x, y, z) = v + wx$$

$$h(v, w, x, y, z) = v + w$$

Find the quotient functions of f/g and f/h.

12.2 Find the kernel and cokernel sets of the following switching expression:

$$f(s, t, u, v, w, x, y, z) = tsu + tsv + wz + xz + yz + xy$$

12.3 Find the kernel and cokernel sets of the following switching expressions:

(a) $f(t, u, v, w, x, y, z) = twy + txy + uwy + uxy + vwy + vxy + z$

(b) $g(t, u, v, w, x, y, z) = tx + ty + uvx + uwx + uvy + uwy + z$

12.4 Use the kernel approach to implement the following multilevel logic circuits and calculate the number of literals of each multiple-output switching function:

(a) $f_1(v, w, x, y, z) = vx + vy + vz$
$f_2(v, w, x, y, z) = wx + wy + wz$

(b) $f_1(u, v, w, x, y, z) = uy + uz + vy + vz$
$f_2(u, v, w, x, y, z) = uy + uz + wy + wz$
$f_3(u, v, w, x, y, z) = vy + vz + xy + xz$

12.5 Use the kernel approach to implement the following multilevel logic circuits and calculate the number of literals of each multiple-output switching function:

(a) $f_1(v, w, x, y, z) = vw' + v'w$
$f_2(v, w, x, y, z) = wz + v'z + v'xy + wxy$

(b) $f_1(u, v, w, x, y, z) = v + w$
$f_2(u, v, w, x, y, z) = vx + vy + wx + wy + z$

12.6 Use the kernel approach to implement the following multilevel logic circuits and calculate the number of literals of each multiple-output switching function:

$$f_1(t, u, v, w, x, y, z) = tuvwz + tuvxz + tuvyz$$

$$f_2(t, u, v, w, x, y, z) = tuvwz + tuwxz + tuwyz$$

12.7 Use the simple two-step technology mapping technique to map Problem 12.4(a) into 4-input LUTs.

(a) How many LUTs are required for the original switching expressions?

(b) How many LUTs are required for the results after being implemented by using the kernel approach?

12.8 Use the simple two-step technology mapping technique to map Problem 12.4(b) into 4-input LUTs.

(a) How many LUTs are required for the original switching expressions?

(b) How many LUTs are required for the results after being implemented by using the kernel approach?

12.9 Use the simple two-step technology mapping technique to map Problem 12.5(a) into 4-input LUTs.

(a) How many LUTs are required for the original switching expressions?

(b) How many LUTs are required for the results after being implemented by using the kernel approach?

12.10 Use the simple two-step technology mapping technique to map Problem 12.5(b) into 4-input LUTs.

(a) How many LUTs are required for the original switching expressions?

(b) How many LUTs are required for the results after being implemented by using the kernel approach?

12.11 Use the simple two-step technology mapping technique to map Problem 12.6 into 4-input LUTs.

(a) How many LUTs are required for the original switching expressions?

(b) How many LUTs are required for the results after being implemented by using the kernel approach?

12.12 If we want to use four inverters cascaded together in some applications, write a Verilog HDL to model it. Synthesize your design with an available FPGA device and see what happens.

12.13 A simple frequency doubler uses XOR gates to extract the edge information from an input clock signal, such as the one shown in Figure 12.27.

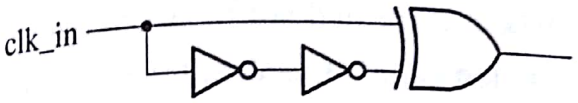


FIGURE 12.27 A simple frequency doubler

(a) Describe the circuit shown in Figure 12.27 in Verilog HDL.

(b) Synthesize your design with an available FPGA device and see what happens.

12.14 Synthesize the following two arithmetic expressions and see what happens.

$$f = w + x + y + z$$

$$g = (w + x) + (y + z)$$

12.15 For every cycle, we want to compute the following for loop and then store the result at the next positive edge of the clock signal clk:

```
for (i = 0; i < m; i = i + 1)
  if (data_a[i] == 1) then total = total + data_b;
```

(a) Explain why the `multiple_iteration_example_a` described in Section 12.4.5 cannot correctly compute the desired results. Of course, you may synthesize it by using synthesis tools and examine the synthesized result carefully. Perhaps this may help you to explore the reason why it cannot work properly.

(b) Of course, if we change the nonblocking operators into blocking ones, the results will be correct. Please explain why?

(c) Assume that we want to insist using the coding style set in this book, namely, still using a nonblocking assignment. Rewrite or modify the code so that it can work properly.