CHAPTER 8

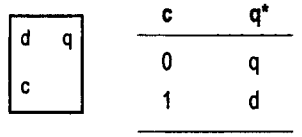
SEQUENTIAL CIRCUIT DESIGN: PRINCIPLE

A sequential circuit is a circuit that has an internal state, or memory. A *synchronous* sequential circuit, in which all memory elements are controlled by a global synchronizing signal, greatly simplifies the design process and is the most important design methodology. Our focus is on this type of circuit. In this chapter and the next chapter, we examine the VHDL description of basic memory elements and study the design of sequential circuits with a "regular structure." Chapters 10, 11 and 12 discuss the design of sequential circuits with a "random structure" (finite state machine) and circuits based on register transfer methodology.

8.1 OVERVIEW OF SEQUENTIAL CIRCUITS

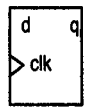
8.1.1 Sequential versus combinational circuits

A combinational circuit, by definition, is a circuit whose output, after the initial transient period, is a function of current input. It has no internal state and therefore is "memoryless" about the past events (or past inputs). A sequential circuit, on the other hand, has an *internal state*, or *memory*. Its output is a function of current input as well as the internal state. The internal state essentially "memorizes" the effect of the past input values. The output thus is affected by current input value as well as past input values (or the entire sequence of input values). That is why we call a circuit with internal state a *sequential circuit*.



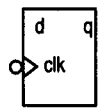
c	q*
0	q
1	d

(a) D latch



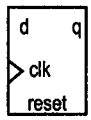
clk	q*
0	q
1	q
↓	d

(b) positive-edge-triggered D FF



clk	q*
0	q
1	q
↓	d

(c) negative-edge-triggered D FF



reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↓	d

(d) D FF with asynchronous reset

Figure 8.1 D latch and D FF.

8.1.2 Basic memory elements

We can add memory to a circuit in two ways. One way is to add closed feedback loops in a combinational circuit, in which the memory is implicitly manifested as system states. Because of potential timing hazards and racing, this approach is very involved and not suitable for synthesis.

The other way is to use predesigned memory components. All device libraries have certain memory cells, which are carefully designed and thoroughly analyzed. These elements can be divided into two broad categories: *latch* and *flip-flop* (*FF*). We review the basic characteristics of a D-type latch (or just D latch) and D-type FF (or just D FF).

D latch The symbol and function table of a D latch are shown in Figure 8.1(a). Note that we use * to represent the next value, and thus q* means the next value of q. The c and d inputs can be considered as a control signal and data input respectively. When c is asserted, input data, d, is passed directly to output, q. When c is deasserted, the output remains the same as the previous value. Since the operation of the D latch depends on the level of the control signal, we say that it is *level sensitive*. A representative timing diagram is shown in the `q_latch` output of Figure 8.2. Note that input data is actually stored into the latch at the falling edge of the control signal.

Since the latch is "transparent" when c is asserted, it may cause racing if a loop exists in the circuit. For example, the circuit in Figure 8.3 attempts to swap the contents of two latches. Unfortunately, racing occurs when c is asserted. Because of the potential complication of timing, we normally do not use latches in synthesis.

D FF The symbol and function table of a *positive-edge-triggered* D FF are shown in Figure 8.1(b). D FF has a special control signal known as a *clock signal*, which is labeled `clk` in the diagram. The D FF is activated only when the clock signal changes from '0' to '1', which is known as the *rising edge* of the clock. At other times, its output remains the same

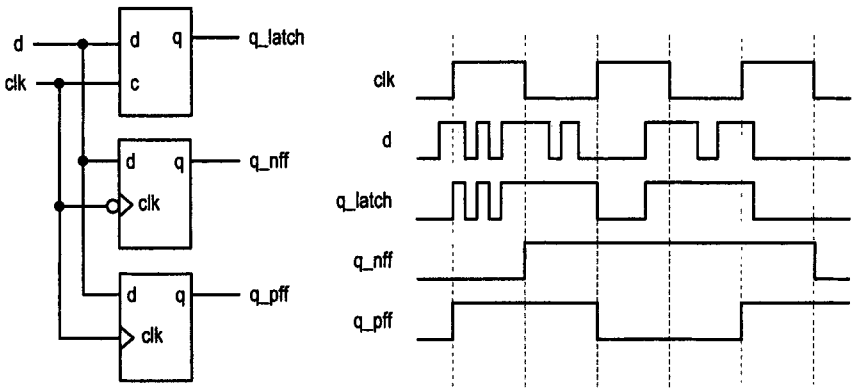


Figure 8.2 Simplified timing diagram of D latch and D FFs.

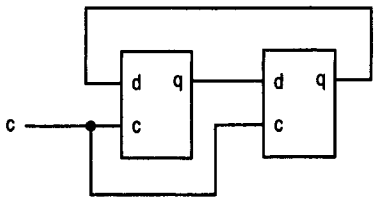


Figure 8.3 Data swapping using D latches.

as its previous value. In other words, at the rising edge of the clock, a D FF takes a sample of input data, stores the value into memory, and passes the value to output. The output, which reflects the stored value, does not change until the next rising edge. Since operation of the D FF depends on the edge of the clock signal, we say that it is *edge sensitive*. A representative timing diagram is shown in the `q_pff` output of Figure 8.2. Note that the clock signal, `clk`, is functioning as a sampling signal, which takes a sample of the input data, d, at the rising edge. The clock signal plays a key role in a sequential circuit and we add a small triangle, as in the `clk` port in Figure 8.1(b), to emphasize use of an edge-triggered FF.

The operation of a *negative-edge-triggered* D FF is similar except that sampling is performed at the falling edge of the clock. Its symbol and function table are shown in Figure 8.1(c). A representative timing diagram is shown in the `q_nff` output of Figure 8.2.

The sampling property of FFs has several advantages. First, variations and glitches between two rising edges have no effect on the content of the memory. Second, there will be no race condition in a closed feedback loop. If we reconstruct the swapping circuit of Figure 8.3 by replacing the D latches with the D FFs, the D FFs swap their contents at each rising edge of the clock and the circuit functions as expected. The disadvantage of the D FF is its circuit size, which is about twice as large as that of a D latch. Since its benefits far outweigh the size disadvantage, today's sequential circuits normally utilize D FFs as the storage elements.

The timing of a D FF is more involved than that of a combinational component. The timing diagram is shown in Figure 8.4. There are three main timing parameters:

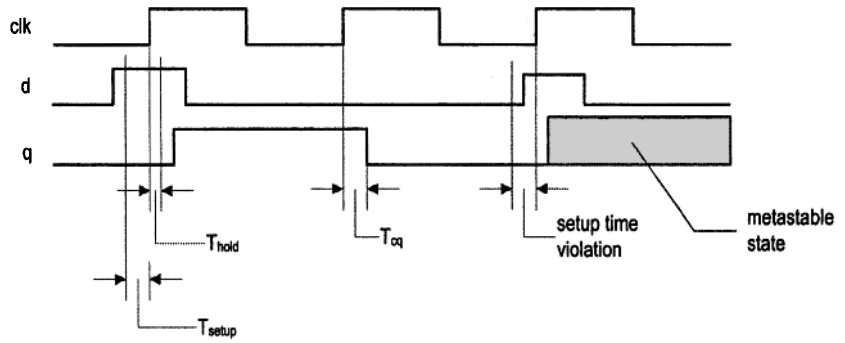


Figure 8.4 Detailed timing diagram of a D FF.

- T_{cq}: *clock-to-q delay*, the propagation delay required for the d input to show up at the q output after the sampling edge of the clock signal.
- T_{setup}: *setup time*, the time interval in which the d signal must be stable *before* the clock edge.
- T_{hold}: *hold time*, the time interval in which the d signal must be stable *after* the clock edge.

T_{cq} corresponds roughly to the propagation delay of a combinational component. T_{setup} and T_{hold}, on the other hand, are *timing constraints*. They specify that the d signal must be stable in a small window around the sampling edge of the clock. If the d signal changes within the setup or hold time window, which is known as *setup time violation* or *hold time violation*, the D FF may enter a *metastable state*, in which the q becomes neither '0' nor '1'. The issue of metastability is discussed in Chapter 16.

8.1.3 Synchronous versus asynchronous circuits

The clock signal of FFs plays a key role in sequential circuit design. According to the arrangement of the clock, we can divide the sequential circuits into the following classes:

- *Globally synchronous circuit* (or simply *synchronous circuit*). A globally synchronous circuit uses FFs as memory elements, and all FFs are controlled (i.e., *synchronized*) by a single global clock signal. Synchronous design is the most important methodology used to design and develop large, complex digital systems. It not only facilitates the synthesis but also simplifies the verification, testing, and prototyping process. Our discussion is focused mainly on this type of circuit.
- *Globally asynchronous locally synchronous circuit*. Sometimes physical constraints, such as the distance between components, prevent the distribution of a single clock signal. In this case, a system may be divided into several smaller subsystems. Since a subsystem is smaller, it can follow the synchronous design principle. Thus, subsystems are synchronous internally. Since each subsystem utilizes its own clock, operation between the subsystems is asynchronous. We need special interface circuits between the subsystems to ensure correct operation. Chapter 16 discusses the design of the interface circuits.
- *Globally asynchronous circuit*. A globally asynchronous circuit does not use a clock signal to coordinate the memory operation. The state of a memory element changes independently. Globally asynchronous circuits can be divided into two categories.

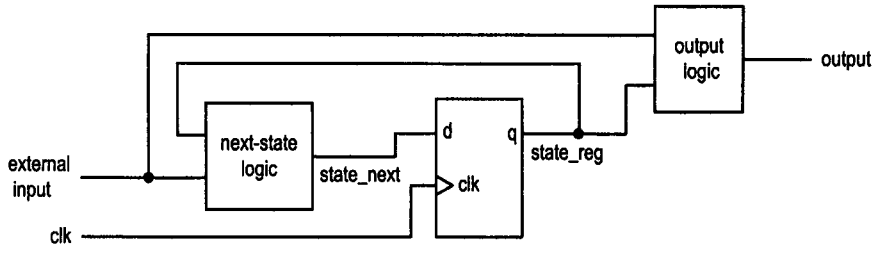


Figure 8.5 Conceptual diagram of a synchronous sequential circuit.

The first category comprises circuits that consist of FFs but do not use the clock in a disciplined way. One example is the *ripple counter*, in which the clock port of an FF is connected to the output of the previous FF. Utilizing FFs in this way is a poor design practice. The second category includes the circuits that contain "clockless" memory components, such as a latch or a combinational circuit with closed feedback loops. This kind of circuit is sometimes simply referred to as an *asynchronous circuit*. The design of asynchronous circuits is very different from that of synchronous circuits and is not recommended for HDL synthesis. The danger is demonstrated by an example in Section 8.3.

8.2 SYNCHRONOUS CIRCUITS

8.2.1 Basic model of a synchronous circuit

The basic diagram of a synchronous circuit is shown in Figure 8.5. The memory element, frequently know as a *state register*, is a collection of D FFs, synchronized by a common global clock signal. The output of the register (i.e., the content stored in the register), the `state_reg` signal, represents the internal state of the system. The *next-state logic* is a combinational circuit that determines the next state of the system. The *output logic* is another combinational circuit that generates the external output signal. Note that the output depends on the external input signal and the current state of the register. The circuit operates as follows:

- At the rising edge of the clock, the value of the `state_next` signal (appearing at the d port) is sampled and propagated to the q port, which becomes the new value of the `state_reg` signal. The value is also stored in FFs and remains unchanged for the rest of the clock period. It represent the *current state* of the system.
- Based on the value of the `state_reg` signal and external input, the next-state logic computes the value of the `state_next` signal and the output logic computes the value of external output.
- At the next rising edge of the clock, the new value of the `state_next` signal is sampled and the `state_reg` signal is updated. The process then repeats.

To satisfy the timing constraints of the FFs, the clock period must be large enough to accommodate the propagation delay of the next-state logic, the clock-to-q delay of the FFs and the setup time of the FFs. This aspect is discussed in Section 8.6.

There are several advantages of synchronous design. First, it simplifies circuit timing. Satisfying the timing constraints (i.e., avoiding setup time and hold time violation) is one

of the most difficult design tasks. When a circuit has hundreds or even thousands of FFs and each FF is driven by an individual clock, the design and analysis will be overwhelming. Since in a synchronous circuit all FFs are driven by the identical clock signal, the sampling of the clock edge occurs simultaneously. We only need to consider the timing constraints of a single memory component. Second, the synchronous model clearly separates the combinational circuits and the memory element. We can easily isolate the combinational part of the system, and design and analyze it as a regular combinational circuit. Third, the synchronous design can easily accommodate the timing hazards. As we discussed in Section 6.5.3, the timing hazards are unavoidable in a large synthesized combinational circuit. In a synchronous circuit, inputs are sampled and stored at the rising edge of the clock. The glitches do not matter as long as they are settled at the time of sampling. Instead of considering all the possible timing scenarios, we only need to focus on worst-case propagation delays of the combinational circuit.

8.2.2 Synchronous circuits and design automation

The synchronous model essentially reduces a complex sequential circuit to a single closed feedback loop and greatly simplifies the design process. We only need to analyze the timing of a simple loop. Once it is done, the memory elements can be isolated and separated from the circuit. The sequential design now becomes a combinational design and we can apply the previous optimization and synthesizing schemes of combinational circuits to construct sequential circuits. Because of this, the synchronous model is the most dominant methodology in today's design environment. Most EDA tools are based on this model.

The benefit of synchronous methodology is not just limited to synthesis. It can facilitate the other tasks of the development process. The impact of synchronous methodology is summarized below.

- *Synthesis.* Since we can separate the memory elements, the system is reduced to a combinational circuit. All optimization algorithms and techniques used in combinational circuit synthesis can be applied accordingly.
- *Timing analysis.* The analysis involves only a single closed feedback loop. It is straightforward once the propagation delay of the combination circuit is known. Thus, the timing analysis of the sequential circuit is essentially reduced to the timing analysis of its combinational part.
- *Cycle-based simulation.* Cycle-based simulation ignores the exact propagation delay but simulates the circuit operation from one clock cycle to another clock cycle. Since we can easily identify the memory elements and their clock, cycle-based simulation can be used for synchronous design.
- *Testing.* One key testing technique is to use scan registers to shift in test patterns and shift out the results. Because the memory elements are isolated, we can easily replace them with scan registers when needed.
- *Design reuse.* The main timing constraint of the synchronous design is embedded in the period of the clock signal (to be discussed in Section 8.6), which depends mainly on the propagation delay of the combination part. As long as the clock period is large enough, the same design can be implemented by different device technologies.
- *Hardware emulation.* Because the same synchronous design can be targeted to different device technologies, it is possible to first construct the design in FPGA technology, run and verify the circuit at a slower clock rate, and then fabricate it in ASIC technology.

8.2.3 Types of synchronous circuits

Based on the "representation and transition patterns" of state, we divide synchronous circuits into three types. These divisions are informal, just for clarity of coding. The three types of sequential circuits are:

- *Regular sequential circuit.* The state representation and state transitions have a simple, *regular* pattern, as in a counter and a shift register. Similarly, the next-state logic can be implemented by regular, structural components, such as an incrementor and shifter.
- *Random sequential circuit.* The state transitions are more complicated and there is no special relation between the states and their binary representations. The next-state logic must be constructed from scratch (i.e., by *random* logic). This kind of circuit is known as a *finite state machine* (*FSM*).
- *Combined sequential circuit.* A combined sequential circuit consists of both a regular sequential circuit and an FSM. The FSM is used to control operation of the regular sequential circuit. This kind of circuit is based on the *register transfer methodology* and is sometimes known as *finite state machine with data path* (*FSMD*).

We discuss the design and description of regular sequential circuits in this chapter and the next chapter, and we cover the FSM and FSMD in Chapters 10, 11 and 12.

8.3 DANGER OF SYNTHESIS THAT USES PRIMITIVE GATES

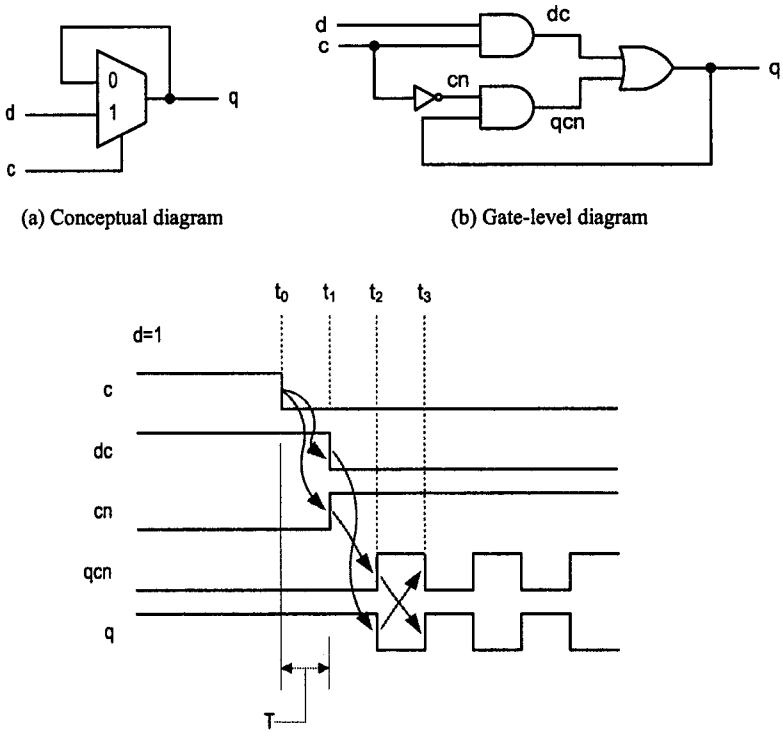
As we discussed earlier, an asynchronous sequential circuit can be constructed from scratch by adding a feedback loop to the combinational components. Although asynchronous circuits potentially can run faster and consume less power, designing an asynchronous circuit is difficult because of the potential races and oscillations. The design procedure is totally different from the synchronous methodology, and we should avoid using normal EDA software to synthesize asynchronous circuits. Since this book focuses on RT-level synthesis, we do not discuss this topic in detail. The following example illustrates the potential danger of using the normal synthesis procedure to construct an asynchronous circuit.

Consider the D latch discussed in Section 8.1.2. We can easily translate the truth table into VHDL code, as shown in Listing 8.1.

Listing 8.1 D latch from scratch

```
library ieee;
use ieee.std_logic_1164.all;
entity dlatch is
   port(
      c: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end dlatch;

architecture demo_arch of dlatch is
   signal q_latch: std_logic;
begin
   process(c,d,q_latch)
   begin
```



(a) Conceptual diagram

(b) Gate-level diagram

(c) Timing diagram

Figure 8.6 Synthesizing a D latch from scratch.

```
        if (c='1') then
           q_latch <= d;
        else
           q_latch <= q_latch;
20      end if;
    end process;
    q <= q_latch;
end demo_arch;
```

Synthesis software can normally recognize that this code is for a D latch and should infer a predesigned D-latch cell from the cell library accordingly. For demonstration purposes, let us try to use simple gates to synthesize it from scratch. We can derive the conceptual diagram and expand it to a gate-level diagram following the procedure to synthesize a combinational circuit, as shown in Figure 8.6(a) and (b).

At first glance, the circuit is just like a combinational circuit except that the output is looped back as an input. However, there is a serious timing problem for this circuit. Let us assume that all gates have a propagation delay of T and the wire delays are negligible, and that c, d and q are '1' initially. Now consider what happens when c changes from '1' to '0' at time t_0. According to the function table, we expect that q should be latched to the value of d and thus should remain '1'. Following the circuit diagram, we can derive a detailed timing diagram, as shown in Figure 8.6(c). The events are summarized below.

- At t_0, c changes to '0'.
- At t_1 (after a delay of T), dc and cn change.
- At t_2 (after a delay of $2T$), qcn changes (due to cn) and q changes (due to dc).
- At t_3 (after a delay of $3T$), q changes (due to qcn) and qcn changes (due to q).

Clearly, the output q continues to oscillate at a period of $2T$ and the circuit is unstable.

Recall that in Section 6.5.4, we discussed delay-sensitive circuit, in which the correctness of circuit function depends on the delays of various components. Asynchronous circuits belong to this category and thus are not suitable for synthesis. If we really wish to implement an asynchronous circuit from scratch, it is better to do it manually using a schematic rather than relying on synthesis.

8.4 INFERENCE OF BASIC MEMORY ELEMENTS

All device libraries have predesigned memory cells. Internally, these cells are designed as asynchronous sequential circuits. They are carefully crafted and thoroughly analyzed and verified. These cells are treated as "leaf units," and no further synthesis or optimization will be performed. The previous section has shown the danger of deriving a memory element from scratch. To avoid this, we must express our intent clearly and precisely in VHDL code so that these predesigned latches or FFs can be inferred. While we should be innovative about the design, it is a good idea to follow the standard VHDL description of latch and FF to avoid any unwanted surprise.

8.4.1 D latch

The function table of a D latch was shown in Figure 8.1(a). The corresponding VHDL code is shown in Listing 8.2. It is the standard description. Synthesis software should infer a predesigned D latch from the device library.

Listing 8.2 D latch

```
library ieee;
use ieee.std_logic_1164.all;
entity dlatch is
   port(
      c: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end dlatch;

architecture arch of dlatch is
begin
   process(c,d)
   begin
      if (c='1') then
         q <= d;
      end if;
   end process;
end arch;
```

In this code, the value of d is passed to q when c is '1'. Note that there is no else branch in the if statement. According to the VHDL definition, q will keep its previous value when c is not '1' (i.e., c is '0'). This is just what we want for the D latch. Alternatively, we can explicitly include the else branch to express that q has its previous value when c is '0', as in the VHDL code in Listing 8.1. The code is not as compact or clear and is not recommended.

8.4.2 D FF

Positive-edge-triggered D FF The function table of a positive-edge-triggered D FF was shown in Figure 8.1(b). The corresponding VHDL code is shown in Listing 8.3. This is a standard description and should be recognized by all synthesis software. A predesigned D FF should be inferred accordingly.

Listing 8.3 D FF

```
library ieee;
use ieee.std_logic_1164.all;
entity dff is
   port(
      clk: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end dff;

architecture arch of dff is
begin
   process(clk)
   begin
      if (clk'event and clk='1') then
         q <= d;
      end if;
   end process;
end arch;
```

The key expression to infer the D FF is the Boolean expression

```
clk'event and clk='1'
```

The `'event` term is a VHDL attribute returning `true` when there is a change in signal value (i.e., an *event*). Thus, when `clk'event` is `true`, it means that the value of `clk` has changed. When the `clk='1'` expression is `true`, it means that the new value of `clk` is `'1'`. When both expressions are `true`, it indicates that the `clk` signal changes to `'1'`, which is the rising edge of the `clk` signal.

The if statement states that at the rising of the `clk` signal, q gets the value of d. Since there is no else branch, it means that q keeps its previous value otherwise. Thus, the VHDL code accurately describes the function of a D FF. Note that the d signal is not in the sensitivity list. It is reasonable since the output only responds to `clk` and does nothing when d changes its value.

We can also add an extra condition `clk'last_value='0'` to the Boolean expression:

```
clk'event and clk='1' and clk'last_value='0'
```

to ensure that the transition is from `'0'` to `'1'` rather than from a metavalue to `'1'`. This may affect simulation but has no impact on synthesis. The above Boolean expression is

defined as a function, `rising_edge()`, in the IEEE `std_logic_1164` package. We can rewrite the previous VHDL code as

```
architecture arch of dff is
begin
   process(clk)
   begin
      if rising_edge(clk) then
         q <= d;
      end if;
   end process;
end arch;
```

We can also use wait statement inside the process to infer the D FF:

```
architecture wait_arch of dff is
begin
   process
   begin
      wait until clk'event and clk='1';
      q <= d ;
   end process ;
end wait_arch;
```

However, since the sensitivity list makes the code easier to understand, we do not use this format in this book.

Theoretically, a then branch can be added to the code:

```
if (clk'event and clk='1') then
   q <= d;
else
   q <= '1';
end if;
```

Although it is syntactically correct, it is meaningless for synthesis purpose.

Negative-edge-triggered D FF A negative-edge-triggered D FF is similar to a positive-edge-triggered D FF except that the input data is sampled at the falling edge of the clock. To specify the falling edge, we must revise the Boolean expression of the if statement:

```
if (clk'event and clk='0') then
```

We can also use the Boolean expression

```
clk'event and clk='0' and clk'last_value='1'
```

to ensure the `'1'` to `'0'` transition or use the shorthand function, `falling_edge()`, defined in the IEEE `std_logic_1164` package.

D FF with asynchronous reset A D FF may contain an asynchronous reset signal that clears the D FF to '0'. The symbol and function table are shown in Figure 8.1(d). Note that the reset operation does not depend on the level or edge of the clock signal. Actually, we can consider that it has a higher priority than the clock-controlled operation. The VHDL code is shown in Listing 8.4.

Listing 8.4 D FF with asynchronous reset

```
library ieee;
use ieee.std_logic_1164.all;
entity dffr is
   port(
      clk: in std_logic;
      reset: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end dffr;

architecture arch of dffr is
begin
   process(clk,reset)
   begin
      if (reset='1') then
         q <='0';
      elsif (clk'event and clk='1') then
         q <= d;
      end if;
   end process;
end arch;
```

Both the `reset` and `clk` signals are in the sensitivity list since either can invoke the process. When the process is invoked, it first checks the `reset` signal. If it is `'1'`, the D FF is cleared to `'0'`. Otherwise, the process continues checking the rising-edge condition, as in a regular D FF. Note that there is no else branch.

Since the reset operation is independent of the clock, it cannot be synthesized from a regular D FF. A D FF with asynchronous reset is another leaf unit. The synthesis software recognizes this format and should infer the desired D FF cell from the device library.

Asynchronous reset, as its name implies, is not synchronized by the clock signal and thus should not be used in normal synchronous operation. The major use of a reset signal is to clear the memory elements and set the system to an initial state. Once the system enters the initial state, it starts to operate synchronously and will never use the reset signal again. In many digital systems, a short reset pulse is generated when the power is turned on.

Some D FFs may also have an asynchronous preset signal that sets the D FF to '1'. The VHDL code is shown in Listing 8.5.

Listing 8.5 D FF with asynchronous reset and preset

```
library ieee;
use ieee.std_logic_1164.all;
entity dffrp is
   port(
      clk: in std_logic;
      reset, preset: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end dffrp;

architecture arch of dffrp is
```

```
begin
   process(clk,reset,preset)
   begin
      if (reset='1') then
         q <='0';
      elsif (preset='1') then
         q <= '1';
      elsif (clk'event and clk='1') then
         q <= d;
      end if;
   end process;
end arch;
```

Since the asynchronous signal is normally used for system initialization, a single preset or reset signal should be adequate most of the time.

8.4.3 Register

A register is a collection of a D FFs that is driven by the same clock and reset signals. The VHDL code of an 8-bit register is shown in Listing 8.6.

Listing 8.6 Register

```
library ieee;
use ieee.std_logic_1164.all;
entity reg8 is
   port(
      clk: in std_logic;
      reset: in std_logic;
      d: in std_logic_vector(7 downto 0);
      q: out std_logic_vector(7 downto 0)
   );
end reg8;

architecture arch of reg8 is
begin
   process(clk,reset)
   begin
      if (reset='1') then
         q <=(others=>'0');
      elsif (clk'event and clk='1') then
         q <= d;
      end if;
   end process;
end arch;
```

The code is similar to D FF except that the d input and the q output are now 8 bits wide. We use the symbol of D FF for the register. The size of the register can be derived by checking the bus width marks of the input and output connections.

8.4.4 RAM

Random access memory (*RAM*) can be considered as a collection of latches with special interface circuits. It is used to provide massive storage. While technically it is possible

to synthesize a RAM from scratch by assembling D-latch cells and control circuits, the result is bulky and inefficient. Utilizing the device library's predesigned RAM module, whose memory cells are crafted and optimized at the transistor level, is a much better alternative. Although the basic structure of RAMs is similar, their sizes, speeds, interfaces, and timing characteristics vary widely, and thus it is not possible to derive a portable, device-independent VHDL code to infer the desired RAM module. We normally need to use explicit component instantiation statement for this task.

8.5 SIMPLE DESIGN EXAMPLES

The most effective way to derive a sequential circuit is to follow the block diagram in Figure 8.5. We first identify and separate the memory elements and then derive the next-state logic and output logic. After separating the memory elements, we are essentially designing the combinational circuits, and all the schemes we learned earlier can be applied accordingly. A clear separation between memory elements and combinational circuits is essential for the synthesis of large, complex design and is helpful for the verification and testing processes. Our VHDL code description follows this principle and we always use an isolated VHDL segment to describe the memory elements.

Since identifying and separating the memory elements is the key in deriving a sequential circuit, we utilize the following coding practice to emphasize the existence of the memory elements:

- Use an individual VHDL code segment to infer memory elements. The segment should be the standard description of a D FF or register.
- Use the suffix `_reg` to represent the output of a D FF or a register.
- Use the suffix `_next` to indicate the next value (the `d` input) of a D FF or a register.

We examine a few simple, representative sequential circuits in this section and study more sophisticated examples in Chapter 9.

This coding practice may make the code appear to be somewhat cumbersome, especially for a simple circuit. However, its long-term benefits far outweigh the inconvenience. The alternative coding style, which mixes the memory elements and combinational circuit in one VHDL segment, is discussed briefly in Section 8.7.

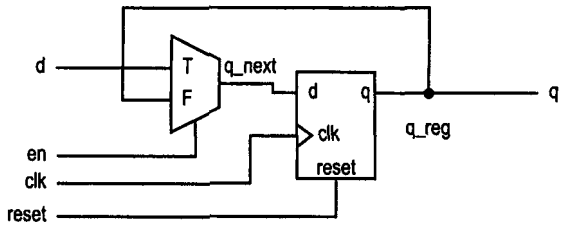
8.5.1 Other types of FFs

There are other types of FFs, such as D FF with an enable signal, JK FF and T FF. They were popular when a digital system was constructed by SSI components because they may reduce the number of IC chips on a printed circuit board. Since all these FFs can be synthesized by a D FF, they are not used today. The following subsections show how to construct them from a D FF.

D FF with enable Consider a D FF with an additional enable signal. The function table is shown in Figure 8.7(a). Note that the enable signal, `en`, has an effect only at the rising edge of the clock. This means that the signal is synchronized to the clock. At the rising edge of the clock, the FF samples both `en` and `d`. If `en` is `'0'`, which means that the FF is not enabled, FF keeps its previous value. On the other hand, if `en` is `'1'`, the FF is enabled and functions as a regular D FF. The VHDL code is shown in Listing 8.7.

reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↓	0	q
0	↓	1	d

(a) Function table



(b) Conceptual diagram

Figure 8.7 D FF with an enable signal.

Listing 8.7 D FF with an enable signal

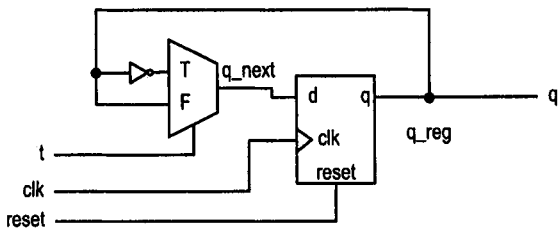
```
library ieee;
use ieee.std_logic_1164.all;
entity dff_en is
   port(
      clk: in std_logic;
      reset: in std_logic;
      en: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end dff_en;

architecture two_seg_arch of dff_en is
   signal q_reg: std_logic;
   signal q_next: std_logic;
begin
   -- D FF
   process(clk,reset)
   begin
      if (reset='1') then
         q_reg <= '0';
      elsif (clk'event and clk='1') then
         q_reg <= q_next;
      end if;
   end process;
   -- next-state logic
   q_next <= d when en ='1' else
             q_reg;
   -- output logic
   q <= q_reg;
end two_seg_arch;
```

The VHDL code follows the basic sequential block diagram and is divided into three segments: a memory element, next-state logic and output logic. The memory element is a regular D FF. The next-state logic is implemented by a conditional signal assignment statement. The q_next signal can be either d or the original content of the FF, q_reg,

reset	clk	t	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↓	0	q
0	↓	1	q'

(a) Function table



(b) Conceptual diagram

Figure 8.8 T FF.

depending on the value of en. At the rising edge of the clock, q_next will be sampled and stored into the memory element. The output logic is simply a wire that connects the output of the register to the q port.

The conceptual diagram is shown in Figure 8.7(b). To obtain the diagram, we first separate and derive the memory element, and then derive the combinational circuit using the procedure described in Chapter 4.

T FF A T FF has a control signal, t, which specifies whether the FF to invert (i.e., *toggle*) its content. The function table of a T FF is shown in Figure 8.8(a). Note that the t signal is sampled at the rising edge of the clock. The VHDL code is shown in Listing 8.8, and the conceptual diagram is shown in Figure 8.8(b).

Listing 8.8 T FF

```
library ieee;
use ieee.std_logic_1164.all;
entity tff is
   port(
      clk: in std_logic;
      reset: in std_logic;
      t: in std_logic;
      q: out std_logic
   );
end tff;

architecture two_seg_arch of tff is
   signal q_reg: std_logic;
   signal q_next: std_logic;
begin
   -- D FF
   process(clk,reset)
   begin
      if (reset='1') then
         q_reg <= '0';
      elsif (clk'event and clk='1') then
         q_reg <= q_next;
      end if;
   end process;
```

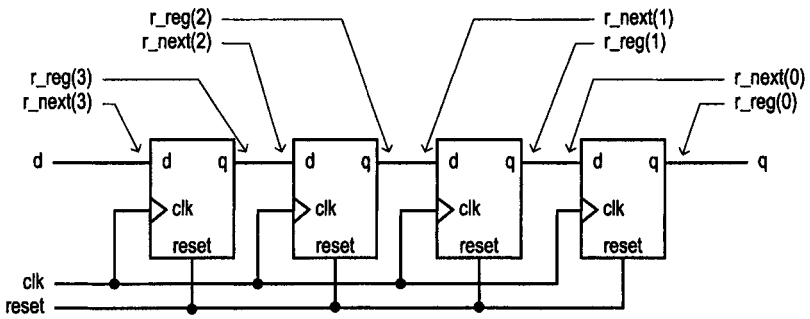


Figure 8.9 4-bit free-running shift-right register.

```
25    -- next-state logic
      q_next <= q_reg when t='0' else
                not(q_reg);
      -- output logic
      q <= q_reg;
30  end two_seg_arch;
```

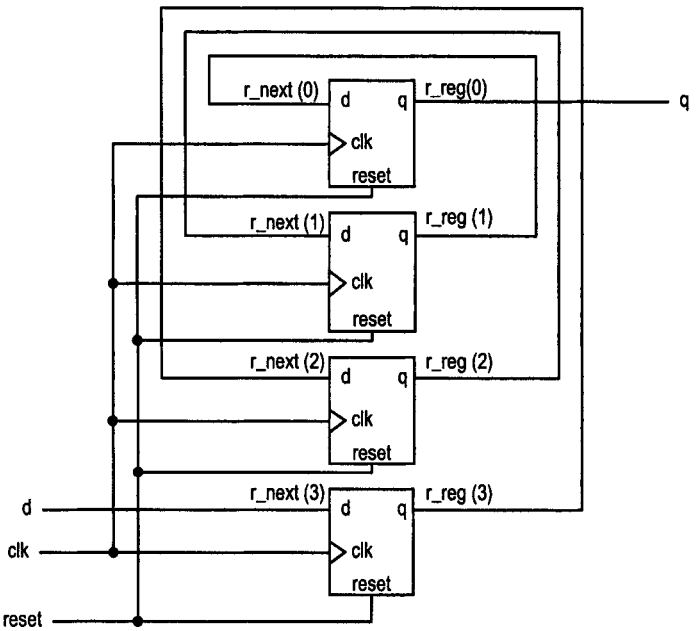
8.5.2 Shift register

A shift register shifts the content of the register left or right 1 bit in each clock cycle. One major application of a shifter register is to send parallel data through a serial line. In the transmitting end, a data word is first loaded to register in parallel and is then shifted out 1 bit at a time. In the receiving end, the data word is shifted in 1 bit at a time and reassembled.

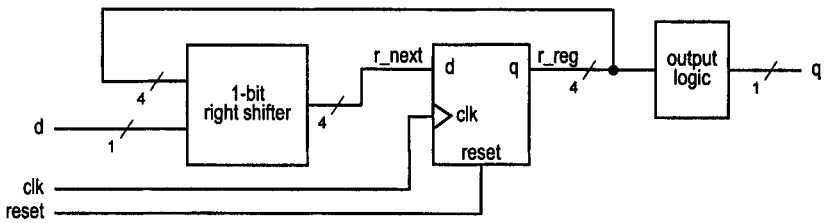
Free-running shift-right register A free-running shift register performs the shifting operation continuously. It has no other control signals. A 4-bit free-running shift-right register is shown in Figure 8.9. We can rearrange the FFs and align them vertically, as in Figure 8.10(a). After grouping the four FFs together and treating them as a single memory block, we transform the circuit into the basic sequential circuit block diagram in Figure 8.10(b). The VHDL code can be derived according to the block diagram, as in Listing 8.9.

Listing 8.9 Free-running shift-right register

```
library ieee;
use ieee.std_logic_1164.all;
entity shift_right_register is
   port(
5     clk, reset: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end shift_right_register;
10
architecture two_seg_arch of shift_right_register is
   signal r_reg: std_logic_vector(3 downto 0);
   signal r_next: std_logic_vector(3 downto 0);
```



(a) Vertical form



(b) "Basic sequential circuit" form

Figure 8.10 Shift register diagram in different forms.

```
  begin
    -- register
    process(clk,reset)
    begin
       if (reset='1') then
          r_reg <= (others=>'0');
       elsif (clk'event and clk='1') then
          r_reg <= r_next;
       end if;
    end process;
    -- next-state logic (shift right 1 bit)
    r_next <= d & r_reg(3 downto 1);
    -- output
    q <= r_reg(0);
  end two_seg_arch;
```

The VHDL code follows the basic sequential circuit block diagram, and the key is the code for the next-state logic. The statement

```
r_next <= d & r_reg(3 downto 1);
```

indicates that the original register content is shifted to the right 1 bit and a new bit, d, is inserted to the left. The memory element part of the code is the standard description of a 4-bit register.

Universal shift register A universal shift register can load a parallel data word and perform shifting in either direction. There are four operations: load, shift right, shift left and pause. A control signal, ctrl, specifies the desired operation. The VHDL code is shown in Listing 8.10. Note that the d(0) input and the q(3) output are used as serial-in and serial-out for the shift-left operation, and the d(3) input and the q(0) output are used as serial-in and serial-out for the shift-right operation. The block diagram is shown in Figure 8.11.

Listing 8.10 Universal shift register

```
library ieee;
use ieee.std_logic_1164.all;
entity shift_register is
   port(
      clk, reset: in std_logic;
      ctrl: in std_logic_vector(1 downto 0);
      d: in std_logic_vector(3 downto 0);
      q: out std_logic_vector(3 downto 0)
   );
end shift_register;

architecture two_seg_arch of shift_register is
   signal r_reg: std_logic_vector(3 downto 0);
   signal r_next: std_logic_vector(3 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
```

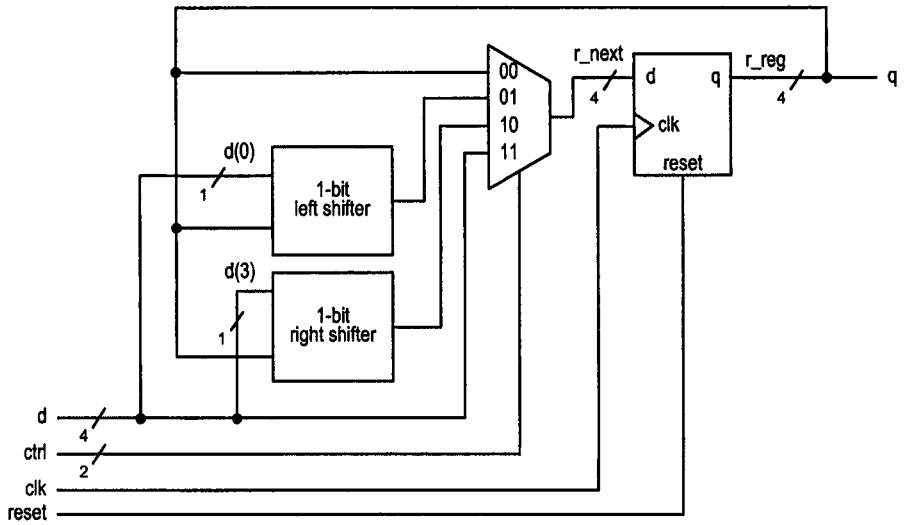


Figure 8.11 4-bit universal register.

```
20          r_reg <= (others=>'0');
         elsif (clk'event and clk='1') then
            r_reg <= r_next;
         end if;
      end process;
25    -- next-state logic
      with ctrl select
         r_next <=
            r_reg                          when "00", --pause
            r_reg(2 downto 0) & d(0)       when "01", --shift left;
30          d(3) & r_reg(3 downto 1)       when "10", --shift right;
            d                              when others; -- load
      -- output logic
      q <= r_reg;
   end two_seg_arch;
```

8.5.3 Arbitrary-sequence counter

A sequential counter circulates a predefined sequence of states. The next-state logic determines the patterns in the sequence. For example, if we need a counter to cycle through the sequence of "000", "011", "110", "101" and "111", we can construct a combinational circuit with a function table that specifies the desired patterns, as in Table 8.1.

The VHDL code is shown in Listing 8.11. Again, the code follows the basic block diagram of Figure 8.5. A conditional signal assignment statement is used to implement the function table.

Listing 8.11 Arbitrary-sequence counter

```
library ieee;
use ieee.std_logic_1164.all;
```

Table 8.1 Patterns of an arbitrary-sequence counter

Input pattern	Next pattern
000	011
011	110
110	101
101	111
111	000

```
entity arbi_seq_counter4 is
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(2 downto 0)
   );
end arbi_seq_counter4;

architecture two_seg_arch of arbi_seq_counter4 is
   signal r_reg: std_logic_vector(2 downto 0);
   signal r_next: std_logic_vector(2 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   r_next <= "011" when r_reg="000" else
             "110" when r_reg="011" else
             "101" when r_reg="110" else
             "111" when r_reg="101" else
             "000";  -- r_reg="111"
   -- output logic
   q <= r_reg;
end two_seg_arch;
```

8.5.4 Binary counter

A binary counter circulates through a sequence that resembles the unsigned binary number. For example, a 3-bit binary counter cycles through "000", "001", "010", "011", "100", "101", "110" and "111", and then repeats.

Free-running binary counter An n-bit binary counter has a register with n FFs, and its output is interpreted as an unsigned integer. A free-running binary counter increments the content of the register every clock cycle, counting from 0 to $2^n - 1$ and then repeating. In addition to the register output, we assume that there is a status signal, `max_pulse`, which

is asserted when the counter is in the all-one state. The VHDL code of a 4-bit binary counter is shown in Listing 8.12.

Listing 8.12 Free-running binary counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity binary_counter4_pulse is
   port(
      clk, reset: in std_logic;
      max_pulse: out std_logic;
      q: out std_logic_vector(3 downto 0)
   );
end binary_counter4_pulse;

architecture two_seg_arch of binary_counter4_pulse is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic (incrementor)
   r_next <= r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
   max_pulse <= '1' when r_reg="1111" else
                '0';
end two_seg_arch;
```

The next-state logic consists of an incrementor, which calculates the new value for the next state of the register. Note that the definition requests the 4-bit binary counter counts in a wrapped-around fashion; i.e., when the counter reaches the maximal number, "1111", it should return to "0000" and start over again. It seems that we should replace statement

```
r_next <= r_reg + 1;
```

with

```
r_next <= (r_reg + 1) mod 16;
```

However, in the IEEE `numeric_std` package, the definition of + on the `unsigned` data type is modeled after a hardware adder, which behaves like wrapping around when the addition result exceeds the range. Thus, the original statement is fine. While correct, using the **mod** operator is redundant. It may confuse some synthesis software since the **mod** operator cannot be synthesized. The output logic uses a conditional signal assignment statement to implement the desired pulse. The conceptual diagram is shown in Figure 8.12.

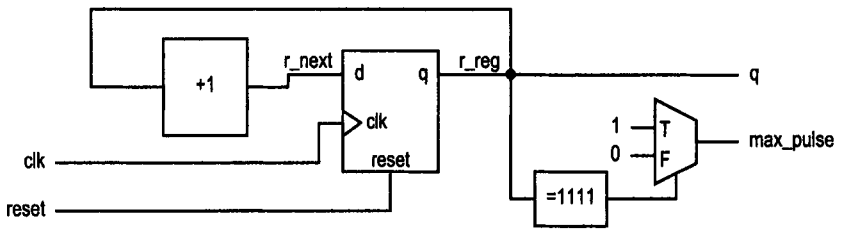


Figure 8.12 Conceptual diagram of a free-running binary counter.

Table 8.2 Function table of a featured binary counter

syn_clr	load	en	q*	Operation
1	–	–	00···00	synchronous clear
0	1	–	d	parallel load
0	0	1	q+1	count
0	0	0	q	pause

Featured binary counter Rather than leaving the counter in the free-running mode, we can exercise more control. The function table in Table 8.2 shows a binary counter with additional features. In the counter, we can synchronously clear the counter to 0, load a specific value, and enable or pause the counting. The VHDL code is shown in Listing 8.13.

Listing 8.13 Featured binary counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity binary_counter4_feature is
   port(
      clk, reset: in std_logic;
      syn_clr, en, load: in std_logic;
      d: in std_logic_vector(3 downto 0);
      q: out std_logic_vector(3 downto 0)
   );
end binary_counter4_feature;

architecture two_seg_arch of binary_counter4_feature is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
```

```
   end process;
   -- next-state logic
   r_next <= (others=>'0') when syn_clr='1' else
             unsigned(d)   when load='1' else
             r_reg + 1     when en ='1' else
             r_reg;
   -- output logic
   q <= std_logic_vector(r_reg);
end two_seg_arch;
```

8.5.5 Decade counter

Instead of utilizing all possible 2^n states of an n-bit binary counter, we sometime only want the counter to circulate through a subset of the states. We define a mod-m counter as a binary counter whose states circulate from 0 to $m-1$ and then repeat. Let us consider the design of a mod-10 counter, also known as a *decade counter*. The counter counts from 0 to 9 and then repeats. We need at least 4 bits ($\lceil \log_2 10 \rceil$) to accommodate the 10 possible states, and the output is 4 bits wide. The VHDL description is shown in Listing 8.14.

Listing 8.14 Decade counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mod10_counter is
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(3 downto 0)
   );
end mod10_counter;

architecture two_seg_arch of mod10_counter is
   constant TEN: integer := 10;
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   r_next <= (others=>'0') when r_reg=(TEN-1) else
             r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
end two_seg_arch;
```

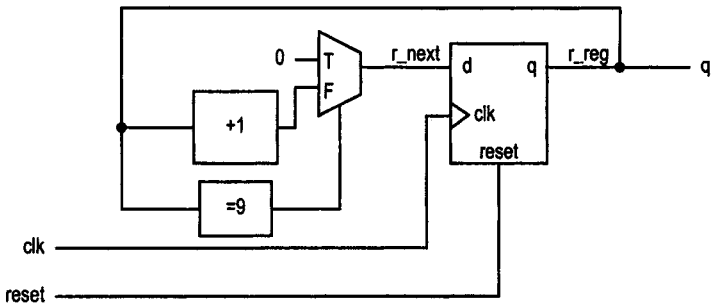


Figure 8.13 Conceptual diagram of a decade counter.

The key to this design is the next-state logic. When the counter reaches 9, as indicated by the condition `r_reg=(TEN-1)`, the next value will be 0. Otherwise, the next value will be incremented by 1. The conceptual diagram is shown in Figure 8.13.

We can rewrite the next-state logic as

```
r_next <= (r_reg + 1) mod 10;
```

Although the code is compact and clean, it cannot be synthesized due to the complexity of the **mod** operator.

8.5.6 Programmable mod-m counter

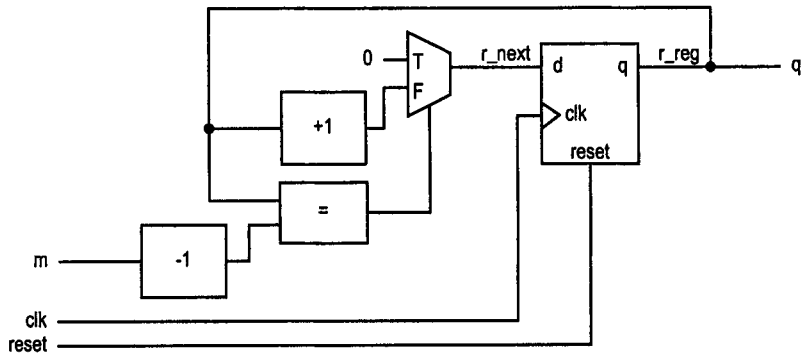
We can easily modify the code of the previous decade counter to a mod-m counter for any m. However, the counter counts a fixed, predefined sequence. In this example, we design a "programmable" 4-bit mod-m counter, in which the value of m is specified by a 4-bit input signal, `m`, which is interpreted as an unsigned number. The range of `m` is from "0010" to "1111", and thus the counter can be programmed as a mod-2, mod-3, ..., or mod-15 counter.

The maximal number in the counting sequence of a mod-m counter is $m - 1$. Thus, when the counter reaches $m - 1$, the next state should be 0. Our first design is based on this observation. The VHDL code is similar to the decade counter except that we need to replace the `r_reg=(TEN-1)` condition of the next-state logic with `r_reg=(unsigned(m)-1)`. The code is shown in Listing 8.15.

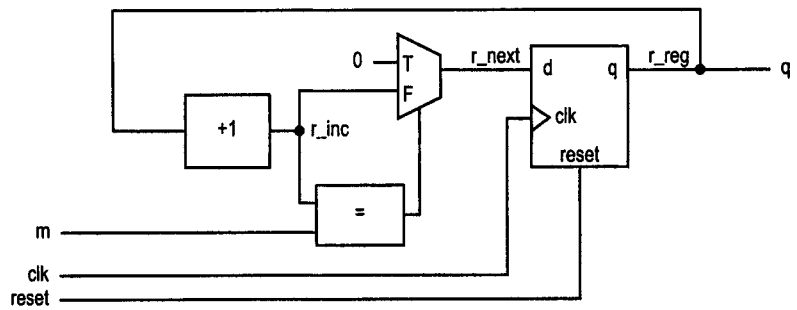
Listing 8.15 Initial description of a programmable mod-m counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity prog_counter is
   port(
      clk, reset: in std_logic;
      m: in std_logic_vector(3 downto 0);
      q: out std_logic_vector(3 downto 0)
   );
end prog_counter;

architecture two_seg_clear_arch of prog_counter is
```



(a) Block diagram of initial design



(b) Block diagram of more efficient design

Figure 8.14 Block diagrams of a programmable mod-m counter.

```
    signal r_reg: unsigned(3 downto 0);
    signal r_next: unsigned(3 downto 0);
15 begin
    -- register
    process(clk,reset)
    begin
       if (reset='1') then
20        r_reg <= (others=>'0');
       elsif (clk'event and clk='1') then
          r_reg <= r_next;
       end if;
    end process;
25  -- next-state logic
    r_next <= (others=>'0') when r_reg=(unsigned(m)-1) else
              r_reg + 1;
    -- output logic
    q <= std_logic_vector(r_reg);
30 end two_seg_clear_arch;
```

The conceptual diagram of this code is shown in Figure 8.14(a). The next-state logic consists of an incrementor, a decrementor and a comparator. There is an opportunity for sharing. Note that the Boolean expression

```
r_reg=(unsigned(m)-1)
```

can also be written as

```
(r_reg+1)=unsigned(m)
```

Since the r_req+1 operation is needed for incrementing operation, we can use it in comparison and eliminate the decrementor. The revised VHDL code is shown in Listing 8.16.

Listing 8.16 More efficient description of a programmable mod-m counter

```
architecture two_seg_effi_arch of prog_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next, r_inc: unsigned(3 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   r_inc <= r_reg + 1;
   r_next <= (others=>'0') when r_inc=unsigned(m) else
             r_inc;
   -- output logic
   q <= std_logic_vector(r_reg);
end two_seg_effi_arch;
```

Note that we employ a separate statement for the shared expression:

```
r_inc <= r_reg + 1;
```

and use the r_inc signal for both comparison and incrementing. The diagram of the revised code is shown in Figure 8.14(b).

8.6 TIMING ANALYSIS OF A SYNCHRONOUS SEQUENTIAL CIRCUIT

The timing of a combinational circuit is characterized primarily by the propagation delay, which is the time interval required to generate a stable output response from an input change. The timing characteristic of a sequential circuit is different because of the constraints imposed by memory elements. The major timing parameter in a sequential circuit is the *maximal clock rate*, which embeds the effect of the propagation delay of the combination circuit, the clock-to-q delay of the register and the setup time constraint of the register. Other timing issues include the condition to avoid hold time violation and I/O-related timing parameters.

8.6.1 Synchronized versus unsynchronized input

Satisfying the setup and hold time constraints is the most crucial task in designing a sequential circuit. One motivation behind synchronous design methodology is to group all FFs together and control them with the same clock signal. Instead of considering the constraints

of tens or hundreds of FFs, we can treat them as one memory component and deal with the timing constraint of a *single* register.

The conceptual diagram of Figure 8.5 can be considered as a simplified block diagram for all synchronous sequential circuits. In this diagram, FFs and registers are grouped together as the state register. The input of this register is the `state_next` signal. It is generated by next-state logic, which is a combinational logic with two inputs, including the external input and the output of the state register, `state_reg`. To study the timing constraint of the state register, we need to examine the impact of the two inputs of the next-state logic. Our discussion considers the following effects:

- The effect of the `state_reg` signal.
- The effect of *synchronized* external input.
- The effect of *unsynchronized* external input.

Since the `state_reg` signal is the output of the state register, it is synchronized by the same clock. A closed feedback loop is formed in the diagram through this signal. The timing analysis of a synchronous sequential circuit focuses mainly on this loop and is discussed in Section 8.6.2 .

A *synchronized* external input means that the generation of the input signal is controlled by the same clock signal, possibly from a subsystem of the same design. The timing analysis is somewhat similar to the closed-loop analysis describe above, and is discussed in Section 8.6.5.

An *unsynchronized* external input means that the input signal is generated from an external source or an independent subsystem. Since the system has no information about the unsynchronized external input, it cannot prevent timing violations. For this kind of input, we must use an additional *synchronization circuit* to synchronize the signal with the system clock. This issue is be discussed in Chapter 16.

8.6.2 Setup time violation and maximal clock rate

In Figure 8.5, the output of the register is processed via next-state logic, whose output becomes the new input to the register. To analyze the timing, we have to study the operation of this closed feedback loop and examine the `state_reg` and `state_next` signals. The `state_reg` signal is the output of the register, and it also serves as the input to the next-state logic. The `state_next` signal is the input of the register, and it is also the output of the next-state logic.

Maximal clock rate The timing diagram in Figure 8.15 shows the responses of the `state_reg` and `state_next` signals during one clock cycle. At time t_0, the clock changes from '0' to '1'. We assume that the `state_next` signal has stabilized and doesn't change within the setup and hold time periods. After the clock-to-q delay (i.e., T_{cq}), the register's output, `state_reg`, becomes available at time t_1, which is $t_0 + T_{cq}$. Since `state_reg` is the input of the next-state logic, the next-state logic responds accordingly. We define the propagation delays of the fastest and slowest responses as $T_{next(min)}$ and $T_{next(max)}$ respectively. In the timing diagram, the `state_next` signal changes at t_2, which is $t_1 + T_{next(min)}$, and becomes stabilized at t_3, which is $t_1 + T_{next(max)}$. At time t_5, a new rising clock edge arrives and the current clock cycle ends. The `state_next` is sampled at t_5 and the process repeats again. t_5 is determined by the period (T_c) of the clock signals, which is $t_0 + T_c$.

Now let us examine the impact of the setup time constraint. The setup time constraint indicates that the `state_next` signal must be stabilized at least T_{setup} before the next

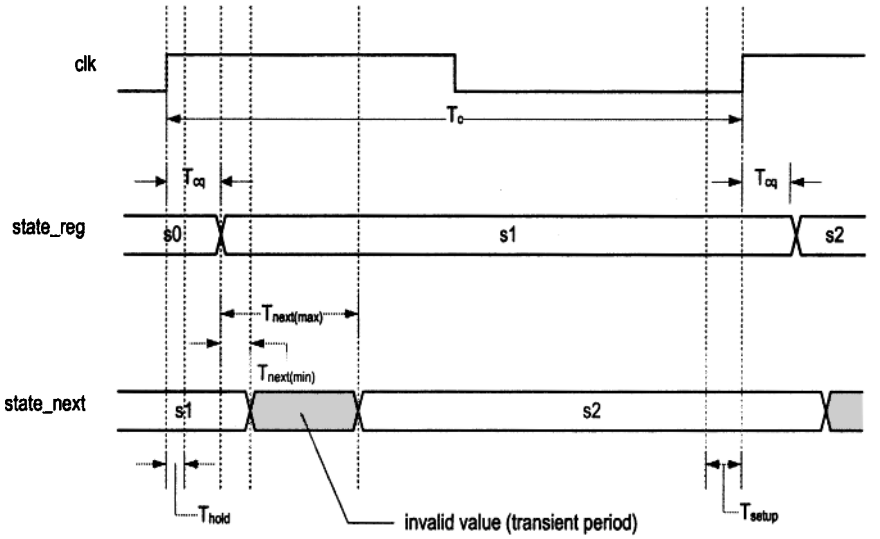


Figure 8.15 Timing analysis of a basic sequential circuit.

sampling edge at t_5. This point is labeled t_4 in the timing diagram. To satisfy the setup time constraint, the `state_next` signal must be stabilized before t_4. This requirement translates into the condition

$$t_3 < t_4$$

From the timing diagram, we see that

$$t_3 = t_0 + T_{cq} + T_{next(max)}$$

and

$$t_4 = t_5 - T_{setup} = t_0 + T_c - T_{setup}$$

We can rewrite the inequality equation as

$$t_0 + T_{cq} + T_{next(max)} < t_0 + T_c - T_{setup}$$

which is simplified to

$$T_{cq} + T_{next(max)} + T_{setup} < T_c$$

This shows the role of the clock period on a sequential circuit. To avoid setup time violation, the minimal clock period must be

$$T_{c(min)} = T_{cq} + T_{next(max)} + T_{setup}$$

The clock period is the main parameter to characterize the timing and performance of a sequential circuit. We commonly use the maximal clock rate or frequency, the reciprocal of the minimal period, to describe the performance of a sequential circuit, as in a 500-MHz counter or 2-GHz processor.

Clock rate examples For a given technology, the T_{cq} and T_{setup} of a D FF are obtained from the data sheet. We can determine the maximal clock rate of a sequential circuit once the propagation delay of the next-state logic is known. This information can only be determined after synthesis and placement and routing. However, we can calculate and estimate the rate of some simple examples.

Assume that we use the technology discussed in Section 6.2.6, and T_{cq} and T_{setup} of its D FF cell are 1 and 0.5 ns respectively. The delay information of combinational components can be obtained from Table 6.2. Let us first consider the free-running shift register of Section 8.5.2. The next-state logic of the shift register only involves the routing of the input and output signals. If we assume that the wiring delay is negligible, its propagation delay is 0. The minimal clock period and maximal clock rate become

$$T_{c(min)} = T_{cq} + T_{setup} = 1.5 \text{ ns}$$

$$f_{max} = \frac{1}{T_{cq} + T_{setup}} = \frac{1}{1.5 \text{ ns}} \approx 666.7 \text{ MHz}$$

Clearly, this is the maximal clock rate that can be achieved with this particular technology.

The second example is an 8-bit free-running binary counter, similar to the 4-bit version of Section 8.5.4. The next-state logic of this circuit is the incrementor, as shown in Figure 8.12. If we choose the incrementor that is optimized for area, the clock rate for this 8-bit binary counter is

$$f_{max} = \frac{1}{T_{cq} + T_{8\_bit\_inc(area)} + T_{setup}} = \frac{1}{1 \text{ ns} + 2.4 \text{ ns} + 0.5 \text{ ns}} \approx 256.4 \text{ MHz}$$

If we increase the size of the counter, a wider incrementor must be utilized, and the propagation delay of the incrementor is increased accordingly. The clock rate of a 16-bit binary counter is reduced to

$$f_{max} = \frac{1}{T_{cq} + T_{16\_bit\_inc(area)} + T_{setup}} = \frac{1}{1 \text{ ns} + 5.5 \text{ ns} + 0.5 \text{ ns}} \approx 142.9 \text{ MHz}$$

and the clock rate of a 32-bit counter is reduced to

$$f_{max} = \frac{1}{T_{cq} + T_{32\_bit\_inc(area)} + T_{setup}} = \frac{1}{1 \text{ ns} + 11.6 \text{ ns} + 0.5 \text{ ns}} \approx 76.3 \text{ MHz}$$

To increase the performance of a binary counter, we must reduce the value of $T_{cq} + T_{next(max)} + T_{setup}$. Since T_{cq} and T_{setup} are determined by the intrinsic characteristics of FFs, they cannot be altered unless we switch to a different device technology. The only way to increase performance is to reduce the propagation delay of the incrementor. If we replace the incrementors that are optimized for delay, the clock rates of the 8-, 16- and 32-bit binary counters are increased to

$$f_{max} = \frac{1}{T_{cq} + T_{8\_bit\_inc(delay)} + T_{setup}} = \frac{1}{1 \text{ ns} + 1.5 \text{ ns} + 0.5 \text{ ns}} \approx 333.3 \text{ MHz}$$

$$f_{max} = \frac{1}{T_{cq} + T_{16\_bit\_inc(delay)} + T_{setup}} = \frac{1}{1 \text{ ns} + 3.3 \text{ ns} + 0.5 \text{ ns}} \approx 208.3 \text{ MHz}$$

and

$$f_{max} = \frac{1}{T_{cq} + T_{32\_bit\_inc(delay)} + T_{setup}} = \frac{1}{1 \text{ ns} + 7.5 \text{ ns} + 0.5 \text{ ns}} \approx 111.1 \text{ MHz}$$

respectively.

8.6.3 Hold time violation

The impact of the hold time constraint is somewhat different from the setup time constraint. Hold time, T_{hold}, is the time period that the input signal must be stabilized after the sampling edge. In the timing diagram of Figure 8.15, it means that the `state_next` must be stable between t_0 and t_h, which is $t_0 + T_{hold}$. Note that the earliest time that `state_next` changes is at time t_2. To satisfy the hold time constraint, we must ensure that

$$t_h < t_2$$

From the timing diagram, we see that

$$t_2 = t_0 + T_{cq} + T_{next(min)}$$

and

$$t_h = t_0 + T_{hold}$$

The inequality becomes

$$t_0 + T_{hold} < t_0 + T_{cq} + T_{next(min)}$$

which is simplified to:

$$T_{hold} < T_{cq} + T_{next(min)}$$

$T_{next(min)}$ depends on the complexity of next-state logic. In some applications, such as the shift register, the output of one FF is connected to the input of another FF, and the propagation delay of the next-state logic is the wire delay, which can be close to 0. Thus, in the worst-case scenario, the inequality becomes

$$T_{hold} < T_{cq}$$

Note that both parameters are the intrinsic timing parameters of the FF, and the inequality has nothing to do with the next-state logic. Manufacturers usually guarantee that their devices satisfy this condition. Thus, we need not worry about the hold time constraint unless the clock edge cannot arrive at all FFs at the same time. We discuss this issue in Chapter 16.

8.6.4 Output-related timing considerations

The closed feedback diagram in Figure 8.5 is the core of a sequential system. In addition, there are also external inputs and outputs. Let us first consider the output part of the circuit. The output signal of a sequential circuit can be divided into the *Moore-typed output* (or just *Moore output*) and *Mealy-typed output* (or just *Mealy output*). For Moore output, the output signal is a function of *system state* (i.e., the output of the register) only. On the other hand, for Mealy output, the output signal is a function of *system state and the external input.* The two types of output can coexist, as shown in Figure 8.16. The main timing parameter for both types of outputs is T_{co}, the time required to obtain a valid output signal after the rising edge of the clock. The value of T_{co} is the summation of T_{cq} and T_{output} (the propagation delay of the output logic); that is,

$$T_{co} = T_{cq} + T_{output}$$

For Mealy output, there exists a path in which the input can affect the output directly. The propagation delay from input to output is simply the combinational propagation delay of output logic.

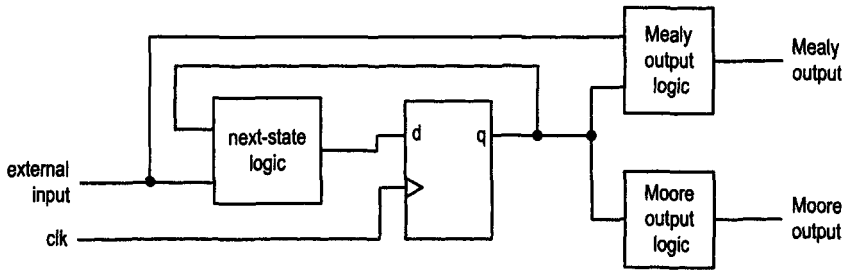


Figure 8.16 Output circuits of a sequential circuit.

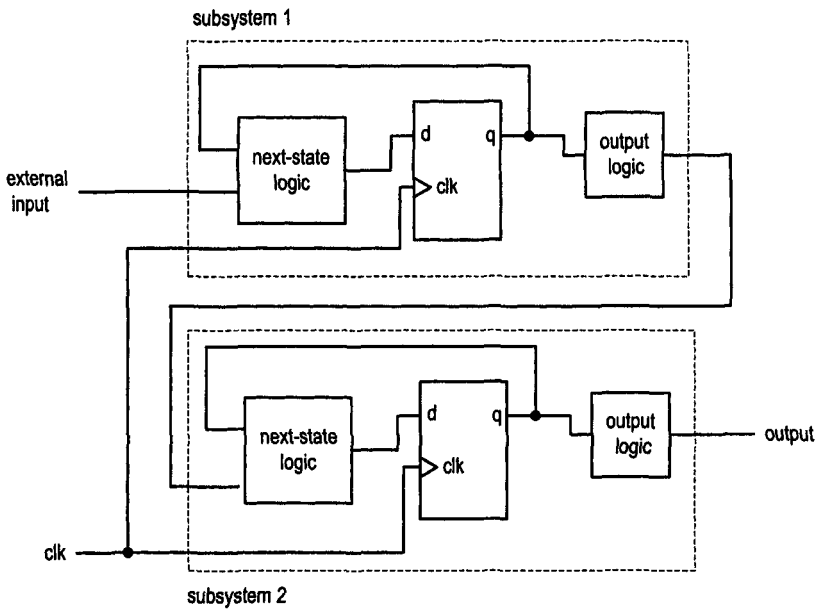


Figure 8.17 Input timing of two synchronous subsystems.

8.6.5 Input-related timing considerations

In a large design, a system may contain several synchronous subsystems. Thus, it is possible that an input comes from a subsystem that is controlled and synchronized by the same clock. The block diagram of this situation is shown in Figure 8.17. Note that the two subsystems are controlled by the same clock and thus are synchronous. At the rising edge of the clock, the register of subsystem 1 samples a new input value. After $T_{co(system1)}$, its new output, which is the input for the next-state logic of subsystem 2, becomes available. At this point the timing analysis is identical to that in Section 8.6.2. To avoid setup time violation, the timing of the two circuits must satisfy the following condition:

$$T_{co(system1)} + T_{next(max)} + T_{setup} < T_c$$

Note that $T_{next(max)}$, the propagation delay of next-state logic, is somewhat different from the calculation used in Section 8.6.2. The $T_{next(max)}$ here is the propagation delay

from the external input to `state_next`, whereas $T_{next(max)}$ used in earlier minimal clock period calculation in Section 8.6.2 is the propagation delay from the internal register output (i.e., `state_reg`) to `state_next`. To be more accurate, we should separate the two constraints. The constraint for the closed loop is

$$T_{cq} + T_{next(max\ of\ state\_reg-to-state\_next)} + T_{setup} < T_{c1}$$

and the constraint for the external input is

$$T_{co(system1)} + T_{next(max\ of\ ext\_input-to-state\_next)} + T_{setup} < T_{c2}$$

We usually determine the clock period based on the calculation of T_{c1}. If T_{c2} turns out to be greater than T_{c1}, we normally redesign the I/O buffer rather than slowing down the clock rate of the entire system. For example, we can employ an extra input buffer for the external input of subsystem 2. Although this approach delays the external input by one clock cycle, it reduces the $T_{co(system1)}$ to T_{cq} in the second constraint.

8.7 ALTERNATIVE ONE-SEGMENT CODING STYLE

So far, all VHDL coding follows the basic block diagram of Figure 8.5 and separates the memory elements from the rest of the logic. Alternatively, we can describe the memory elements and the next-state logic in a single process segment. For a simple circuit, this style appears to be more compact. However, it becomes involved and error-prone for more complex circuits. In this section, we use some earlier examples to illustrate the one-segment VHDL description and the problems associated with this style.

8.7.1 Examples of one-segment code

D FF with enable Consider the D FF with an enable signal in Listing 8.7. It can be rewritten in one-segment style, as in Listing 8.17.

Listing 8.17 One-segment description of a D FF with enable

```
architecture one_seg_arch of dff_en is
begin
   process(clk,reset)
   begin
      if (reset='1') then
         q <='0';
      elsif (clk'event and clk='1') then
         if (en='1') then
            q <= d;
         end if;
      end if;
   end process;
end one_seg_arch;
```

The code is similar to a regular D FF except that there is an if statement inside the elsif branch:

```
if (en='1') then
   q <= d;
end if;
```

The interpretation the code is that at the rising edge of `clk`, if `en` is `'1'`, `q` gets the value of the `d` input. Note that there is no else branch in the previous statement. It implies that if `en` is not `'1'`, `q` will keep its previous value, which should be the value of the register's output. Thus, the code correctly describes the function of the `en` signal. In the actual implementation, "keep its previous value" is achieved by sampling the FF's output and again stores the value back to the FF. This point is elaborated in the next example.

T FF Consider the T FF in Listing 8.8. It can be rewritten in one-segment style, as in Listing 8.18.

Listing 8.18 One-segment description of a T FF

```
architecture one_seg_arch of tff is
   signal q_reg: std_logic;
begin
   process(clk, reset)
   begin
      if reset='1' then
          q_reg <= '0';
      elsif (clk'event and clk='1') then
         if (t='1') then
            q_reg <= not q_reg;
         end if;
      end if;
   end process;
   q <= q_reg;
end one_seg_arch;
```

We use an internal signal, `q_reg`, to represent the content and the output of an FF. The statement

```
q_reg <= not q_reg;
```

may appear strange at first glance. So let us examine it in more detail. The `q_reg` signal on the right-hand side represents the output value of the FF, and the **not** `q_reg` expression forms the new value of `q_reg`. This value has no effect on the FF until the process is activated and the `clk'event` **and** `clk='1'` condition is `true`, which specified the occurrence of the rising edge of the `clk` signal. At this point the value is assigned to `q_reg` (actually, stored into the FF named `q_reg`). Thus, the code correctly describes the desired function. Note that if this statement is an isolated concurrent signal assignment statement, a closed combinational feedback loop is formed, in which the output and input of an inverter are tied together.

As in the previous example, the inner if statement has no else branch, and thus `q_reg` will keep its previous value if the `t='1'` condition is `false`. In actual implementation, "keep its previous value" is achieved by sampling the FF's output and storing the value back to the FF. Thus, the more descriptive if statement can be written as

```
if (t='1') then
   q_reg <= not(q_reg);
else
   q_reg <= q_reg;
end if;
```

Featured binary counter Consider the featured binary counter in Listing 8.13. We can convert it into one-segment code, as in Listing 8.19.

Listing 8.19 One-segment description of a featured binary counter

```
architecture one_seg_arch of binary_counter4_feature is
   signal r_reg: unsigned(3 downto 0);
begin
   -- register & next-state logic
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         if syn_clr='1' then
            r_reg <= (others=>'0');
         elsif load='1' then
            r_reg <= unsigned(d);
         elsif en ='1' then
            r_reg <= r_reg + 1;
         end if;
      end if;
   end process;
   -- output logic
   q <= std_logic_vector(r_reg);
end one_seg_arch;
```

The key to this code is the incrementing part, which is done using the statement

```
r_reg <= r_reg + 1;
```

The interpretation of `r_reg` in this statement is similar to that in T FF except that the **not** operation is replaced by incrementing.

Free-running binary counter Consider the 4-bit free-running binary counter in Listing 8.12. The first attempt to convert it to a single-segment style is shown in Listing 8.20.

Listing 8.20 Incorrect one-segment description of a free-running binary counter

```
architecture not_work_one_seg_glitch_arch
                    of binary_counter4_pulse is
   signal r_reg: unsigned(3 downto 0);
begin
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_reg + 1;
         if r_reg="1111" then
            max_pulse <= '1';
         else
            max_pulse <= '0';
         end if;
      end if;
   end process;
```

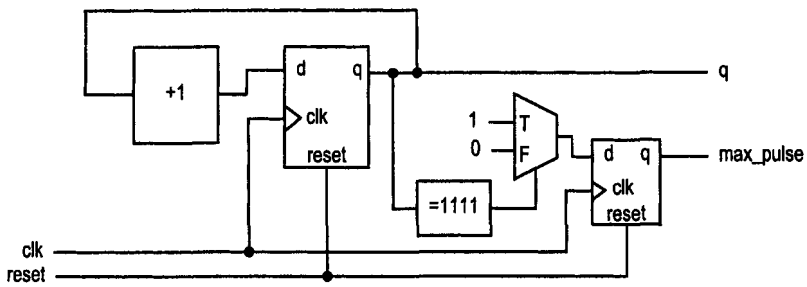


Figure 8.18 Free-running binary counter with an unintended output buffer.

```
   q <= std_logic_vector(r_reg);
end not_work_one_seg_glitch_arch;
```

The output logic does not function as we expected. Because the statement

```
if r_reg="1111" then
   max_pulse <= '1';
else
   max_pulse <= '0';
end if;
```

is inside the `clk'event and clk='1'` branch, a 1-bit register is inferred for the `max_pulse` signal. The register works as a buffer and delays the output by one clock cycle, and thus the `max_pulse` signal will be asserted when `r_reg="0000"`. The block diagram of this code is shown in Figure 8.18.

To correct the problem, we have to move the output logic outside the process, as in Listing 8.21.

Listing 8.21 Correct one-segment description of a free-running binary counter

```
architecture work_one_seg_glitch_arch
                    of binary_counter4_pulse is
   signal r_reg: unsigned(3 downto 0);
begin
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_reg + 1;
      end if;
   end process;
   q <= std_logic_vector(r_reg);
   max_pulse <= '1' when r_reg="1111" else
                '0';
end work_one_seg_glitch_arch;
```

Programmable counter Consider the programmable mod-m counter in Listing 8.16. The first attempt to reconstruct the `two_seg_effi_arch` architecture in one-segment coding style is shown in Listing 8.22.

Listing 8.22 Incorrect one-segment description of a programmable counter

```
architecture not_work_one_arch of prog_counter is
   signal r_reg: unsigned(3 downto 0);
begin
   process(clk,reset)
   begin
      if reset='1' then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_reg+1;
         if (r_reg=unsigned(m)) then
            r_reg<= (others=>'0');
         end if;
      end if;
   end process;
   q <= std_logic_vector(r_reg);
end not_work_one_arch;
```

The code does not work as specified. Recall that a signal will not be updated until the end of the process. Thus, `r_reg` is updated to `r_reg+1` in the end. When the comparison `r_reg=unsigned(m)` is performed, the old value of `r_reg` is used. Because the correct `r_reg` value is late for one clock, the counter counts one extra value. The code actually specified a mod-$(m+1)$ counter instead.

To correct the problem, we must move the incrementing operation outside the process so that it can be performed concurrently with the process. The modified VHDL code is shown in Listing 8.23.

Listing 8.23 Correct one-segment description of a programmable counter

```
architecture work_one_arch of prog_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_inc: unsigned(3 downto 0);
begin
   process(clk,reset)
   begin
      if reset='1' then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         if (r_inc=unsigned(m)) then
            r_reg <= (others=>'0');
         else
            r_reg <= r_inc;
         end if;
      end if;
   end process;
   r_inc <= r_reg + 1;
   q <= std_logic_vector(r_reg);
end work_one_arch;
```

8.7.2 Summary

When we combine the memory elements and next-state logic in the same process, it is much harder to "visualize" the circuit and to map the VHDL statements into hardware components. This style may make code more compact for a few simple circuits, as in the first three examples. However, when a slightly more involved feature is needed, as the `max_pulse` output or the incrementor sharing of the last two examples, the one-segment style makes the code difficult to understand and error-prone. Although we can correct the problems, the resulting code contains extra statements and is far worse than the codes in Section 8.5. Furthermore, since the combinational logic and memory elements are mixed in the same process, it is more difficult to perform optimization and to fine-tune the combinational circuit. In summary, although the two-segment code may occasionally appear cumbersome, its benefits far outweigh the inconvenience, and we generally use this style in this book.

8.8 USE OF VARIABLES IN SEQUENTIAL CIRCUIT DESCRIPTION

We have learned how to infer an FF or a register from a signal. It is done by using the `clk'event` **and** `clk='1'` condition to indicate the rising edge of the clock signal. Any signal assigned under this condition is required to keep its previous value, and thus an FF or a register is inferred accordingly.

A variable can also be assigned under the `clk'event` **and** `clk='1'` condition, but its implication is different because a variable is local to the process and its value is not needed outside the process. If a variable is *assigned a value before it is used*, it will get a value every time when the process is invoked and there is no need to keep its previous value. Thus, no memory element is inferred. On the other hand, if a variable is *used before it is assigned a value*, it will use the value from the previous process execution. The variable has to memorize the value between the process invocations, and thus an FF or a register will be inferred.

Since using a variable to infer memory is more error-prone, we generally prefer to use a signal for this task. The major use of variables is to obtain an intermediate value inside the `clk'event` **and** `clk='1'` branch without introducing an unintended register. This can best be explained by an example. Let us consider a simple circuit that performs an operation `a` **and** `b` and stores the result into an FF at the rising edge of the clock. We use three outputs to illustrate the effect of different coding attempts. The VHDL code is shown in Listing 8.24.

Listing 8.24 Using a variable to infer an FF

```
library ieee;
use ieee.std_logic_1164.all;
entity varaible_ff_demo is
   port(
      a,b,clk: in std_logic;
      q1,q2,q3: out std_logic
   );
end varaible_ff_demo;

architecture arch of varaible_ff_demo is
   signal tmp_sig1: std_logic;
begin
```

```
    -- attempt 1
    process(clk)
    begin
       if (clk'event and clk='1') then
          tmp_sig1 <= a and b;
          q1 <= tmp_sig1;
       end if;
    end process;
    -- attempt 2
    process(clk)
       variable  tmp_var2: std_logic;
    begin
       if (clk'event and clk='1') then
          tmp_var2 := a and b;
          q2 <= tmp_var2;
       end if;
    end process;
    -- attempt 3
    process(clk)
       variable  tmp_var3: std_logic;
    begin
       if (clk'event and clk='1') then
          q3 <= tmp_var3;
          tmp_var3 := a and b;
       end if;
    end process;
end arch;
```

In the first attempt, we try to use the `tmp_sig1` signal for the temporary result. However, since the `tmp_sig1` signal is inside the `clk'event and clk='1'` branch, an unintended D FF is inferred. The two statements

```
tmp_sig1 <= a and b;
q1 <= tmp_sig1;
```

are interpreted as follows. At the rising edge of the `clk` signal, the value of `a and b` will be sampled and stored into an FF named `tmp_sig1`, and the old value (not current value of `a and b`) from the `tmp_sig1` signal will be stored into an FF named `q1`. The diagram is shown in Figure 8.19(a).

The value of `a and b` is delayed by the unintended buffer, and thus this description fails to meet the specification. Since both statements are signal assignment statements, we will obtain the same result if we switch the order of the two statements.

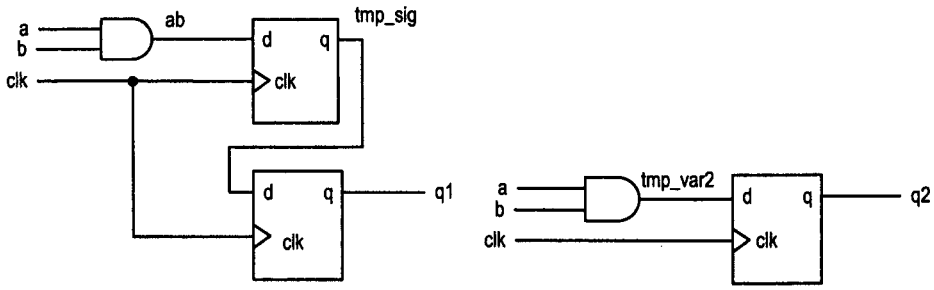
The second attempt uses a variable, `tmp_var2`, for the temporary result and the statements become

```
tmp_var2 := a and b;
q2 <= tmp_var2;
```

Note that the `tmp_var2` variable is first assigned a value and then used in the next statement. Thus, no memory element is inferred and the circuit meets the specification. The diagram is shown in Figure 8.19(b).

The third attempt uses a variable, `tmp_var3`, for the temporary result. It is similar to the second process except that the order of the two statements is reversed:

```
q3 <= tmp_var3;
```



(a) Block diagram of first attempt

(b) Block diagram of second attempt

Figure 8.19 Register inference with a variable.

```
tmp_var3 := a and b;
```

In this code, the `tmp_var3` variable is first used before it is assigned a value. According to the VHDL definition, the value of `tmp_var3` from the previous process invocation will be used. An FF will be inferred to store the previous value. Thus, the circuit described by the third attempt is the same as that of the first attempt, which contains an unwanted buffer.

We can use a variable to overcome the problem of the one-segment programmable mod-m counter in Listing 8.22. The revised code is shown in Listing 8.25.

Listing 8.25 Variable description of a programmable counter

```
architecture variable_arch of prog_counter is
   signal r_reg: unsigned(3 downto 0);
begin
   process(clk,reset)
      variable q_tmp: unsigned(3 downto 0);
   begin
      if reset='1' then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         q_tmp := r_reg + 1;
         if (q_tmp=unsigned(m)) then
            r_reg <= (others=>'0');
         else
            r_reg <= q_tmp;
         end if;
      end if;
    end process;
   q <= std_logic_vector(r_reg);
end variable_arch;
```

Instead of using the `r_reg` signal, we create a variable, `q_tmp`, to store the intermediate result of the incrementing operation. Unlike the signal assignment, the variable assignment takes effect immediately, and thus the code functions as intended.

8.9 SYNTHESIS OF SEQUENTIAL CIRCUITS

In Chapter 6, we examined the synthesis procedure for a combinational circuit. The synthesis of a sequential circuit is identical to this procedure but has two extra steps:

1. Identify and separate the memory elements from the circuit.
2. Select the proper leaf cells from the device library to realize the memory elements.
3. Synthesize the remaining combinational circuit.

If we follow the recommended coding style, the memory elements are specified in individual VHDL segments and thus can be easily inferred and properly instantiated by synthesis software. Once this is done, the remaining process is identical to the synthesis of a combinational circuit.

While synthesizing a combinational circuit, we can include a timing constraint to specify the desired maximal propagation delay, and the synthesis software will try to obtain a circuit to meet this constraint. For a sequential circuit, we can specify the desired maximal clock rate. In a synchronous design, this constraint can easily be translated into the maximal propagation delay of the combinational next-state logic, as indicated by the minimal clock period equation. Thus, all the optimization schemes used in combinational circuits can also be applied to sequential circuit synthesis.

In summary, when we design and code a sequential circuit in a disciplined way, synthesizing it is just like the synthesis of a combinational circuit. We can apply the analysis and optimization schemes developed for combinational circuits to sequential circuit design.

8.10 SYNTHESIS GUIDELINES

- Strictly follow the synchronous design methodology; i.e., all registers in a system should be synchronized by a common global clock signal.
- Isolate the memory components from the VHDL description and code them in a separate segment. One-segment coding style is not advisable.
- The memory components should be coded clearly so that a predesigned cell can be inferred from the device library.
- Avoid synthesizing a memory component from scratch.
- Asynchronous reset, if used, should be only for system initialization. It should not be used to clear the registers during regular operation.
- Unless there is a compelling reason, a variable should not be used to infer a memory component.

8.11 BIBLIOGRAPHIC NOTES

Design and analysis of intermediate-sized synchronous sequential circuits are covered by standard digital systems texts, such as *Digital Design Principles and Practices* by J. F. Wakerly and *Contemporary Logic Design* by R. H. Katz. The former also has a section on the derivation and analysis of asynchronous sequential circuits.

Problems

8.1 Repeat the timing analysis of Section 8.3 for the circuit shown in Figure 8.6 with the following assumptions and examine the q output.

- The propagation delay of the inverter is T and the propagation delays of and and or gates are $2T$.
- The propagation delay of the inverter is $2T$ and the propagation delays of and and or gates are T.

8.2 The SR latch is defined in the left table below. Some device library does not have an SR-latch cell. Instead of synthesizing it from scratch using combinational gates, we want to do this by using a D latch. Derive the VHDL code for this design. The code should contain a standard VHDL description to infer a D latch and a combinational segment that maps the `s` and `r` signals to the `d` and `c` ports of the D latch to achieve the desired function.

s	r	q*
0	0	q
0	1	0
1	0	1
1	1	not allowed

SR latch

j	k	clk	q*
-	-	0	q
-	-	1	q
0	0	↓	q
0	1	↓	0
1	0	↓	1
1	1	↓	q'

JK FF

8.3 A JK FF is defined as in the right table above. Use a D FF and a combinational circuit to design the circuit. Derive the VHDL code and draw the conceptual diagram for this circuit.

8.4 If we replace the D FFs of the free-running shift register of Section 8.5.2 with D latches and connect the external clock signal to the c ports of all D latches, discuss what will happen to the circuit.

8.5 Expand the design of the universal shift register of Section 8.5.2 to include rotate-right and rotate-left operations. To accommodate the revision, the `ctrl` signal has to be extended to 3 bits. Derive the VHDL code for this circuit.

8.6 Consider an 8-bit free-running up–down binary counter. It has a control signal, up. The counter counts up when the up signal is `'1'` and counts down otherwise. Derive the VHDL code for this circuit and draw the conceptual top-level diagram.

8.7 Consider a 4-bit counter that counts from 3 ("0011") to 12 ("1100") and then wraps around. If the counter enters an unused state (such as "0000") because of noise, it will restart from "0011" at the next rising edge of the clock. Derive the VHDL code for this circuit and draw the conceptual top-level diagram.

8.8 Redesign the arbitrary counter of Section 8.5.3 using a mod-5 counter and special output decoding logic. Derive the VHDL code for this design.