CHAPTER 10

FINITE STATE MACHINE: PRINCIPLE AND PRACTICE

A finite state machine (FSM) is a sequential circuit with "random" next-state logic. Unlike the regular sequential circuit discussed in Chapters 8 and 9, the state transitions and event sequence of an FSM do not exhibit a simple pattern. Although the basic block diagram of an FSM is similar to that of a regular sequential circuit, its design procedure is different. The derivation of an FSM starts with a more abstract model, such as a state diagram or an algorithm state machine (ASM) chart. Both show the interactions and transitions between the internal states in graphical formats. In this chapter, we study the representation, timing and implementation issues of an FSM as well as derivation of the VHDL code. Our emphasis is on the application of an FSM as the control circuit for a large, complex system, and our discussion focuses on the issues related to this aspect. As in previous chapters, our discussion is limited to the synchronous FSM, in which the state register is controlled by a single global clock.

10.1 OVERVIEW OF FSMS

As its name indicates, a *finite state machine* (*FSM*) is a circuit with internal states. Unlike the regular sequential circuits discussed in Chapters 8 and 9, state transition of an FSM is more complicated and the sequence exhibits no simple, regular pattern, as in a counter or shift register. The next-state logic has to be constructed from scratch and is sometimes known as "random" logic.

Formally, an FSM is specified by five entities: symbolic states, input signals, output signals, next-state function and output function. A state specifies a unique internal condition

RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability. By Pong P. Chu

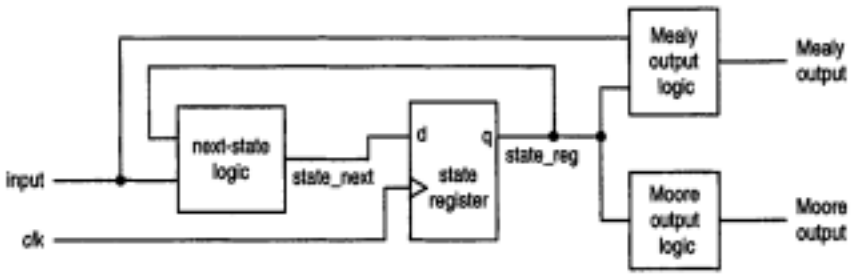


Figure 10.1 Block diagram of an FSM.

of a system. As time progresses, the FSM transits from one state to another. The new state is determined by the next-state function, which is a function of the current state and input signals. In a *synchronous FSM*, the transition is controlled by a clock signal and can occur only at the triggering edge of the clock. As we discussed in Section 8.2, our study strictly follows the synchronous design methodology, and thus coverage is limited to the synchronous FSM.

The output function specifies the value of the output signals. If it is a *function of the state* only, the output is known as a *Moore output*. On the other hand, if it is a *function of the state and input signals*, the output is known as a *Mealy output*. An FSM is called a *Moore machine* or *Mealy machine* if it contains only Moore outputs or Mealy outputs respectively. A complex FSM normally has both types of outputs. The differences and implications of the two types of outputs are discussed in Section 10.4.

The block diagram of an FSM is shown in Figure 10.1. It is similar to the block diagram of a regular sequential circuit. The state register is the memory element that stores the state of the FSM. It is synchronized by a global clock. The next-state logic implements the next-state function, whose input is the current state and input signals. The output logic implements the output function. This diagram includes both Moore output logic, whose input is the current state, and Mealy output logic, whose input is the current state and input signals. The main application of an FSM is to realize operations that are performed in a sequence of steps. A large digital system usually involves complex tasks or algorithms, which can be expressed as a sequence of actions based on system status and external commands. An FSM can function as the control circuit (known as the *control path*) that coordinates and governs the operations of other units (known as the *data path*) of the system. Our coverage of FSM focuses on this aspect. The actual construction of such systems is discussed in the next two chapters. FSMs can also be used in many simple tasks, such as detecting a unique pattern from an input data stream or generating a specific sequence of output values.

10.2 FSM REPRESENTATION

The design of an FSM normally starts with an abstract, graphic description, such as a state diagram or an ASM chart. Both descriptions utilize symbolic state notations, show the transition among the states and indicate the output values under various conditions. A state diagram or an ASM chart can capture all the needed information (i.e., state, input, output, next-state function, and output function) in a single graph.

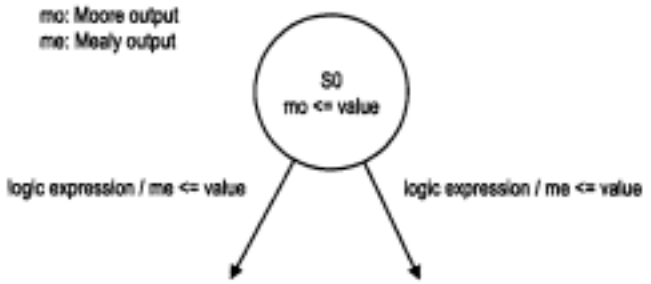


Figure 10.2 Notation for a state.

10.2.1 State diagram

A state diagram consists of nodes, which are drawn as circles (also known as *bubbles*), and one-direction transition arcs. The notation for nodes and arcs is shown in Figure 10.2. A *node* represents a unique state of the FSM and it has a unique symbolic name. An *arc* represents a transition from one state to another and is labeled with the condition that will cause the transition. The condition is expressed as a logic expression composed of input signals. An arc will be taken when the corresponding logic expression is evaluated to be logic '1'.

The output values are also specified on the state diagram. The Moore output is a function of state and thus is naturally placed inside the state bubble. On the other hand, the Mealy output depends on both state and input and thus is placed under the condition expression of the transition arcs. To reduce the clutter, we list only the output signals that are activated or asserted. An output signal will assume the default, unasserted value (*not* don't-care) if it is not listed inside the state bubble or under the logic expression of an arc. We use the following notation for an asserted output value:

```
signal_name <= asserted value; '0'
```

In general, an asserted signal will be logic '1' unless specified otherwise.

The state diagram can best be explained by an example. Figure 10.3 shows the state diagram of a hypothetical memory controller FSM. The controller is between a processor and a memory chip, interpreting commands from the processor and then generating a control sequence accordingly. The commands, *mem*, *rw* and *burst*, from the processor constitute the input signals of the FSM. The *mem* signal is asserted to high when a memory access is required. The *rw* signal indicates the type of memory access, and its value can be either '1' or '0', for memory read and memory write respectively. The *burst* signal is for a special mode of a memory read operation. If it is asserted, four consecutive read operations will be performed. The memory chip has two control signals, *oe* (for output enable) and *we* (for write enable), which need to be asserted during the memory read and memory write respectively. The two output signals of the FSM, *oe* and *we*, are connected to the memory chip's control signals. For comparison purpose, we also add an artificial Mealy output signal, *we_me*, to the state diagram.

Initially, the FSM is in the `idle` state, waiting for the *mem* command from the processor. Once *mem* is asserted, the FSM examines the value of *rw* and moves to either the `read1` state or the `write` state. These input conditions can be formalized to logic expressions, as shown in the transition arcs from the `idle` state:

- mem': represents that no memory operation is required.

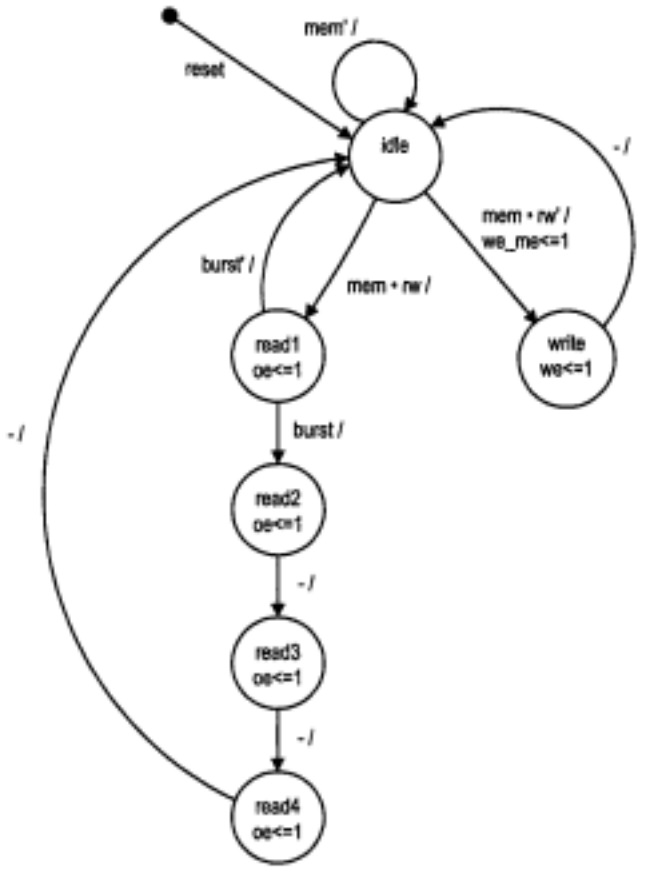


Figure 10.3 State diagram of a memory controller FSM.

- $mem \cdot rw$: represents that a memory read operation is required.
- $mem \cdot rw'$: represents that a memory write operation is required.

The results of these logic expressions are checked at the rising edge of the clock. If the mem' expression is true (i.e., mem is '0'), the FSM stays in the `idle` state. If the $mem \cdot rw$ expression is true (i.e., both mem and rw are '1'), the FSM moves to the `read1` state. Once it is there, the oe signal is activated, as indicated in the state bubble. On the other hand, if the $mem \cdot rw'$ expression is true (i.e., mem is '1' and rw is '0'), the FSM moves to the `write` state and activates the we signal.

After the FSM reaches the `read1` state, the *burst* signal is examined at the next rising edge of the clock. If it is '1', the FSM will go through `read2`, `read3` and `read4` states in the next three clock cycles and then return to the `idle` state. Otherwise, the FSM returns to the `idle` state. We use the notation "–" to represent the "always true" condition. After the FSM reaches the `write` state, it will return to the `idle` state at the next rising edge of the clock.

The *we_me* signal is asserted only when the FSM is in the `idle` state and the $mem \cdot rw'$ expression is true. It will be deactivated when the FSM moves away from the `idle` state (i.e., to the `write` state). It is a Mealy output since its value depends on the state and the input signals (i.e., mem and rw).

In practice, we usually want to force an FSM into an initial state during system initialization. It is frequently done by an asynchronous reset signal, similar to the asynchronous reset signal used in a register of a regular sequential circuit. Sometimes a solid dot is used to indicate this transition, as shown in Figure 10.3. This transition is only for system initialization and has no effect on normal FSM operation.

10.2.2 ASM chart

An *algorithmic state machine* (*ASM*) chart is an alternative method for representing an FSM. Although an ASM chart contains the same amount of information as a state diagram, it is more descriptive. We can use an ASM chart to specify the complex sequencing of events involving commands (input) and actions (output), which is the hallmark of complex algorithms. An ASM chart representation can easily be transformed to VHDL code. It can also be extended to describe FSMD (FSM with a data path), which is discussed in the next two chapters.

An ASM chart is constructed of a network of ASM blocks. An *ASM block* consists of one state box and an optional network of decision boxes and conditional output boxes. A typical ASM block is shown in Figure 10.4. The *state box*, as its name indicates, represents a state in an FSM. It is identified by a symbolic state name on the top left corner of the state box. The action or output listed inside the box describes the desired output signal values when the FSM enters this state. Since the outputs rely on the state only, they correspond to the Moore outputs of the FSM. To reduce the clutter, we list only signals that are activated or asserted. An output signal will assume the default, unasserted value if it is not listed inside the box. We use the same notation for an asserted output signal:

```
signal_name <= asserted value;
```

Again, we assume that an asserted signal will be logic '1' unless specified otherwise.

A *decision box* tests an input condition to determine the exit path of the current ASM block. It contains a Boolean expression composed of input signals and plays a similar role to the logic expression in the transition arc of a state diagram. Because of the flexibility of the Boolean expression, it can describe more complex conditions, such as

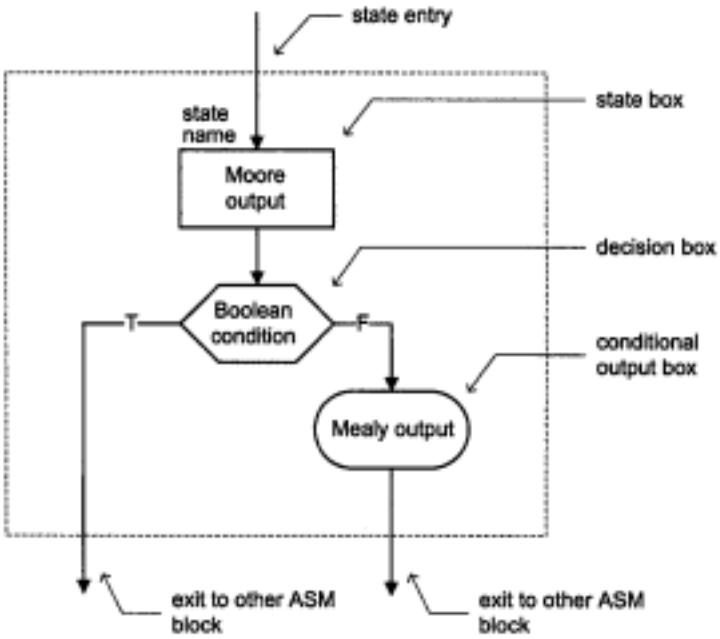


Figure 10.4 ASM block.

`(a > b)` **and** `(c /= 1)`. Depending on the value of the Boolean expression, the FSM can follow either the *true path* or the *false path*, which are labeled as `T` or `F` in the exit paths of the decision box. If necessary, we can cascade multiple decision boxes inside an ASM block to describe a complex condition.

A *conditional output box* also lists asserted output signals. However, it can only be placed after an exit path of a decision box. It implies that these output signals can be asserted only if the condition of the previous decision box is met. Since the condition is composed of a Boolean expression of input signals, these output signals' values depend on the current state and input signals, and thus they are Mealy outputs. Again, to reduce clutter, we place a conditional output box in an ASM block only when the corresponding output signal is asserted. The output signal assumes the default, unasserted value when there is no conditional output box.

Since an ASM chart is another way of representing an FSM, an ASM chart can be converted to a state diagram and vice versa. An ASM block corresponds to a state and its transition arcs of a state diagram. The key for the conversion is the transformation between the logic expressions of the transition arcs in a state diagram and the decision boxes in an ASM chart.

The conversion can best be explained by examining several examples. The first example is shown in Figure 10.5. It is an FSM with no branching arches. The state diagram and the ASM chart are almost identical.

The second example is shown in Figure 10.6. The FSM has two transition arcs from the `s0` state and has a Mealy output, y. The logic expressions a and a' of the transition arches are translated into a decision box with Boolean expression `a = 1`. Note that the two states are transformed into two ASM blocks. The decision and conditional output boxes are not new states, just actions associated with the ASM block `s0`.

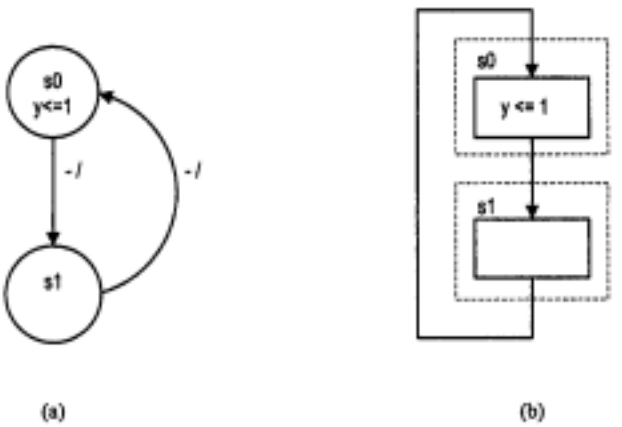


Figure 10.5 Example 1 of state diagram and ASM chart conversion.

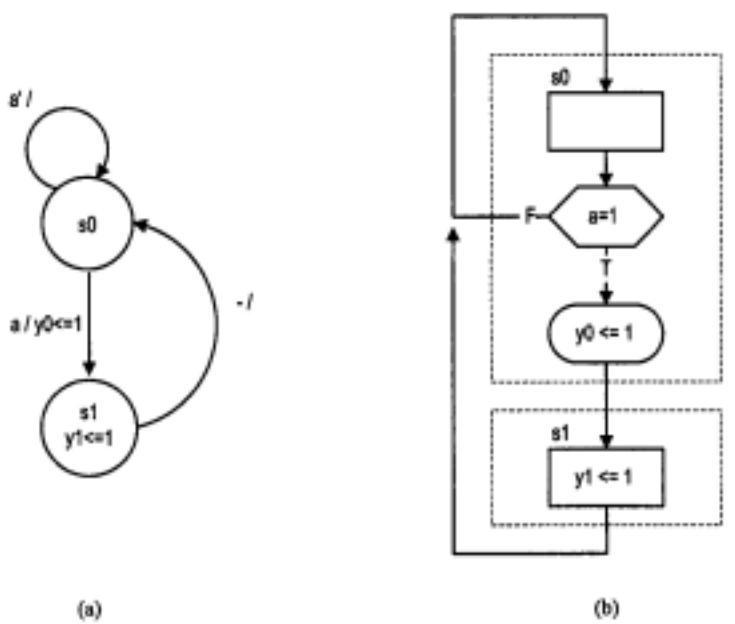


Figure 10.6 Example 2 of state diagram and ASM chart conversion.

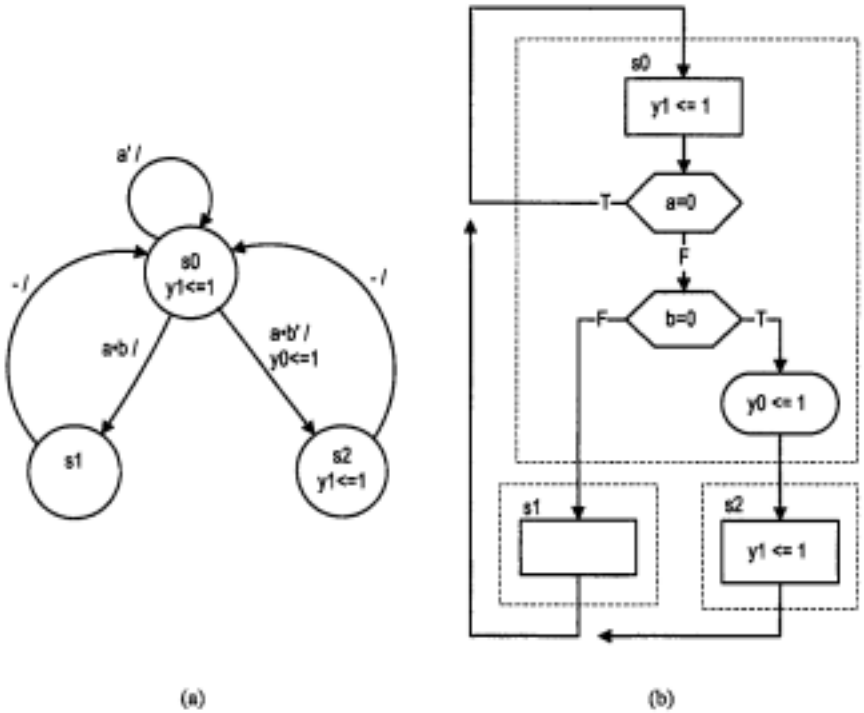


Figure 10.7 Example 3 of state diagram and ASM chart conversion.

The third example is shown in Figure 10.7. The transitions from the s0 state are more involved. We can translate the logic expressions a' and $a \cdot b'$ directly into two decision boxes of conditions a = 0 and (a = 1) **and** (b = 0). However, closer examination shows that the second decision box is on the false path of the first decision box, which implies that a is '1'. Thus, we can eliminate the a = 1 condition from the second decision box and make the decision simpler and more descriptive.

The fourth example is shown in Figure 10.8. The output of the FSM is more complex and depends on various input conditions. The state diagram needs multiple logic expressions in the transition arc to express various input conditions. The ASM chart can accommodate the situation and is more descriptive. Finally, the ASM chart of the previous memory controller FSM, whose state diagram is shown in Figure 10.3, is shown in Figure 10.9.

Since an ASM chart is used to model an FSM, two rules apply:

1. For a given input combination, there is one unique exit path from the current ASM block.
2. The exit path of an ASM block must always lead to a state box. The state box can be the state box of the current ASM block or a state box of another ASM block.

Several common errors are shown in Figure 10.10. The ASM chart of Figure 10.10(a) violates the first rule. There are two exit paths if a and b are both '1', and there is no exit path if a and b are both '0'. The ASM chart of Figure 10.10(b) also violates the first rule since there is no exit path when the condition of the decision box is false. The ASM chart of Figure 10.10(c) violates the second rule because the exit path of the bottom ASM block does not enter the top ASM block via the state box. The second rule essentially states that

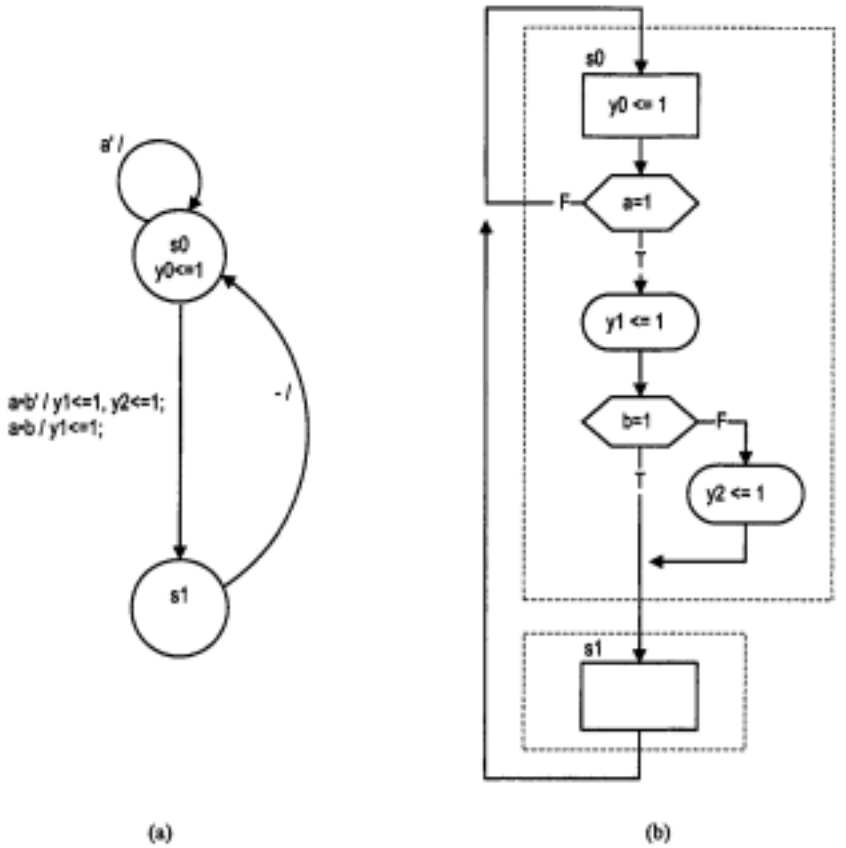


Figure 10.8 Example 4 of state diagram and ASM chart conversion.

the decision boxes and conditional output boxes are associated with a single ASM block and they cannot be shared by other ASM blocks.

An ASM chart and a state diagram contain the same information. Because of the use of decision boxes and flowchart-like graphs, an ASM chart can accommodate the complex conditions involved in state transitions and Mealy outputs, as shown in the third and fourth examples. On the other hand, an ASM chart may be cumbersome for an FSM with simple, straightforward state transitions, and a state diagram is preferred. We use mostly state diagrams in this chapter, but use mainly extended ASM charts while discussing the RT methodology in Chapters 11 and 12.

10.3 TIMING AND PERFORMANCE OF AN FSM

10.3.1 Operation of a synchronous FSM

While a state diagram or an ASM chart shows all the states and transitions, it does not provide information about *when* a transition takes place. In a synchronous FSM, the state transition is controlled by the rising edge of the system clock. Mealy output and Moore output are not directly related to the clock but are responding to input or state change. However, since

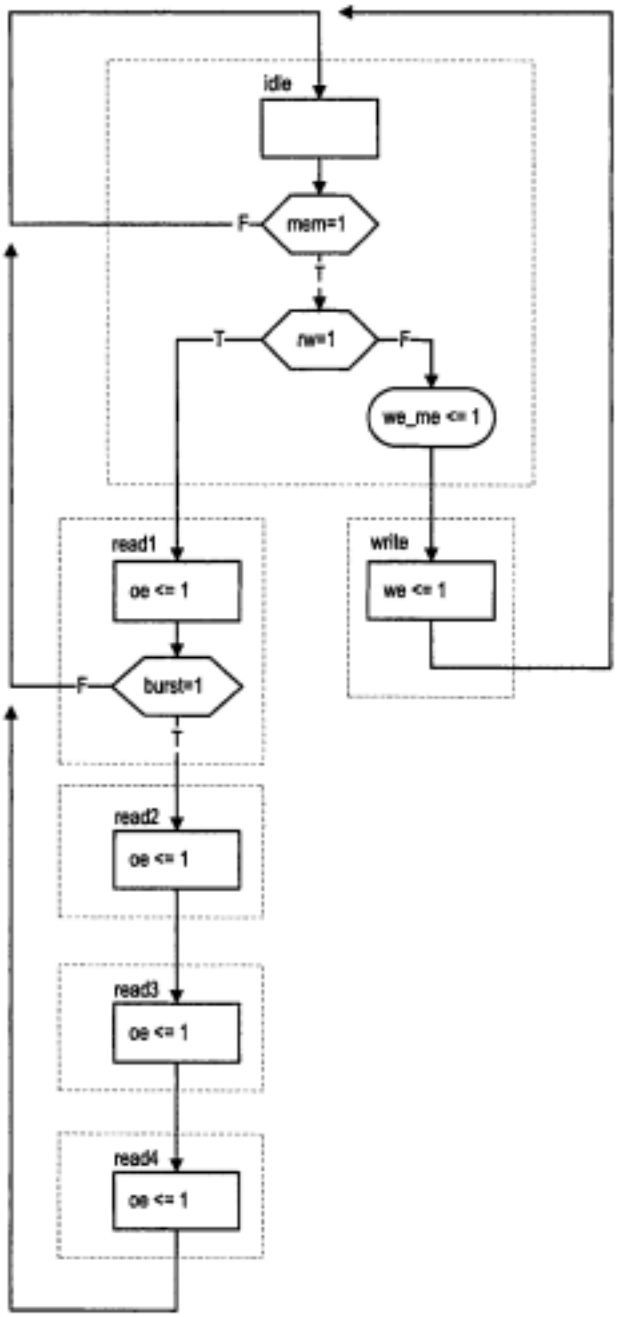


Figure 10.9 ASM chart of a memory controller FSM.

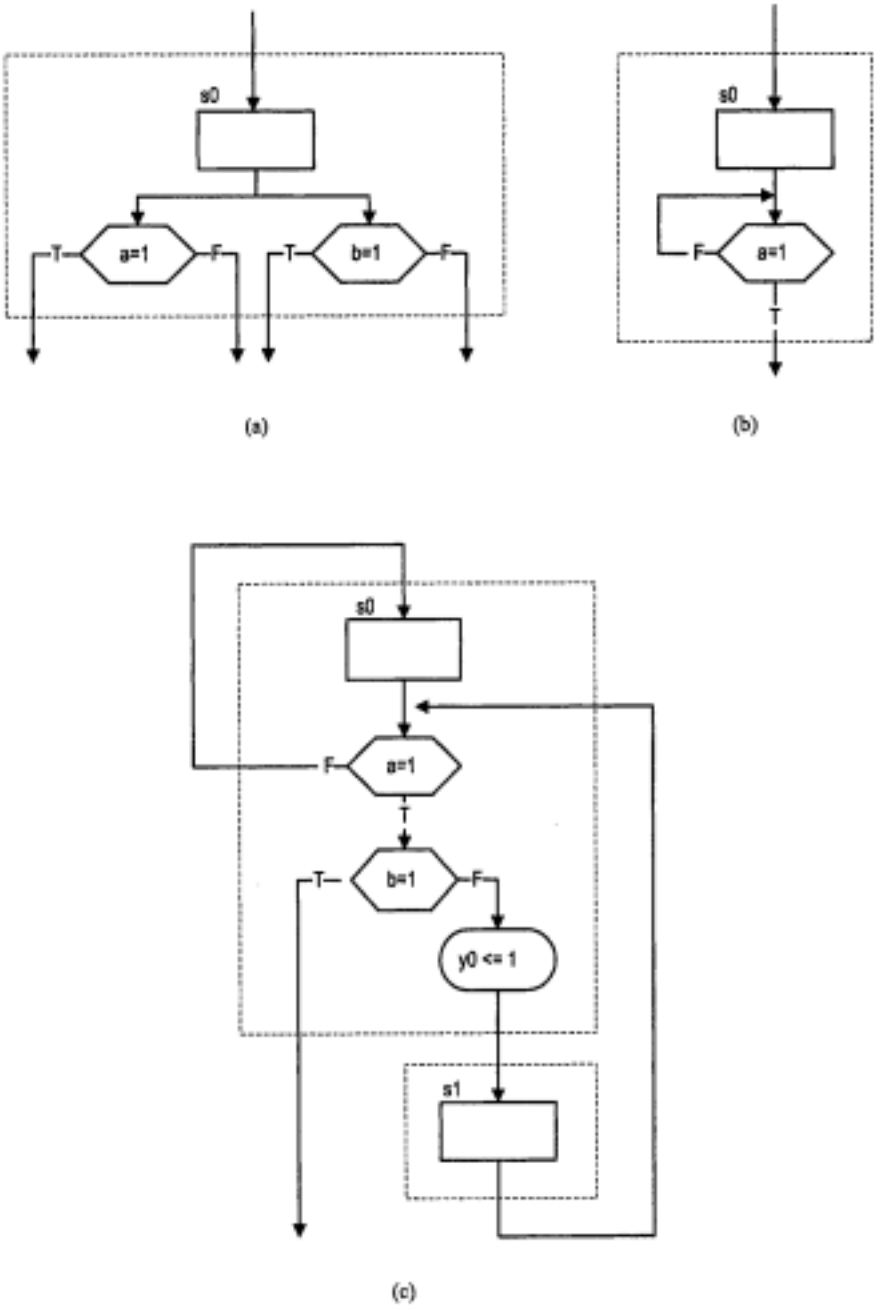


Figure 10.10 Common errors in ASM chart construction.

a Moore output depends only on the state, its transition is indirectly synchronized by the clock.

The timing of a synchronous FSM can best be explained by examining the operation of an ASM block. In an ASM chart, each ASM block represents a state of the FSM. Instead of moving "continuously" from one block to another block, as in a traditional flowchart, the transitions between ASM blocks can occur only at the rising edge of the clock. The operation of an ASM block transition can be interpreted as follows:

1. At the rising edge of the clock, the FSM enters a new state (and thus a new ASM block).
2. During the clock period, the FSM performs several operations. It activates the Moore output signals asserted in this state. It evaluates various Boolean expressions of the decision boxes and activates the Mealy output signals accordingly.
3. At the next rising edge of the clock (which is the end of the current clock period), the results of Boolean expressions are examined simultaneously, an exit path is determined, and the FSM enters the designated new ASM block.

A state and its transitions in a state diagram are interpreted in the same manner.

10.3.2 Performance of an FSM

When an FSM is synthesized, the physical components introduce propagation delays. Since the block diagram of an FSM is almost identical to that of a regular sequential circuit, the timing analysis of an FSM is similar to that of a regular sequential circuit, as discussed in Section 8.6. The main timing parameters associated with the block diagram of Figure 10.1 are:

- T_{cq}, T_{setup}, T_{hold}: the clock-to-q delay, setup time and hold time of the state register.
- $T_{next(max)}$: the maximal propagation delay of the next-state logic.
- $T_{output(mo)}$: the propagation delay of output logic for the Moore output.
- $T_{output(me)}$: the propagation delay of output logic for the Mealy output.

As in a regular sequential circuit, the performance of an FSM is characterized by the maximal clock rate (or minimal clock period). The minimal clock period is

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

and the maximal clock rate is

$$f = \frac{1}{T_{cq} + T_{next(max)} + T_{setup}}$$

Since an FSM is frequently used as the controller, the response of the output signal is also important. A Moore output is characterized by the clock-to-output delay, which is

$$T_{co(mo)} = T_{cq} + T_{output(mo)}$$

A Mealy output may respond to the change of a state or an input signal. The former is characterized by the clock-to-output delay, similar to the Moore output:

$$T_{co(me)} = T_{cq} + T_{output(me)}$$

The latter is just the propagation delay of Mealy output logic, which is $T_{output(me)}$.

10.3.3 Representative timing diagram

The timing diagram helps us to better understand the operation of an FSM and generation of the output signals. It is especially critical when an FSM is used as a control circuit. One tricky part regarding the FSM timing concerns the rising edge of the clock. In an ideal FSM, there is no propagation delay, and thus the state and output signal change at the edge. If the state or output is fed to other synchronous components, which take a sample at the rising edge, it is difficult to determine what the value is. In reality, this will not happen since there is always a clock-to-q delay from the state register. To avoid confusion, this delay should always be included in the timing diagram.

A detailed, representative timing diagram of a state transition is shown in Figure 10.11. It is based on the FSM shown in Figure 10.6. We assume that the next state of the FSM (the `state_next` signal) is `s0` initially. At t_1, the rising edge of the clock, the state register samples the `state_next` signal. After T_{cq} (at t_2), the state register stores the value and reflects the value in its output, the `state_reg` signal. This means that the FSM moves to the `s0` state. At t_3, the `a` input changes from '0' to '1'. According to the ASM chart, the condition of the decision box is met and the true branch is taken. In terms of the circuit, the change of the `a` signal activates both the next-state logic and the Mealy output logic. After the delay of T_{next} (at t_4), the `state_next` signal changes to `s1`. Similarly, the Mealy output, `y0`, changes to '1' after $T_{output(me)}$ (at t_5). At t_6, the `a` signal switches back to '0'. The `state_next` and `y0` signals respond accordingly. Note that the change of the `state_next` signal has no effect on the state register (i.e., the state of the FSM). At t_7, the `a` signal changes to '1' again, and thus the `state_next` and `y0` signals become `s1` and '1' after the delays. At t_8, the current period ends and a new rising edge occurs. The state register samples the `state_next` signal and stores the `s1` value into the register. After T_{cq} (at t_9), the register obtains its new value and the FSM moves to the `s1` state. The change in the `state_reg` signal triggers the next-state logic, Mealy output logic and Moore output logic. After the T_{next} delay (at t_{10}), the next-state logic generates a new value of `s0`. We assume that $T_{output(mo)}$ and $T_{output(me)}$ are similar. After this delay (at t_{11}), the Mealy output, `y0`, is deactivated, and the Moore output, `y1`, is activated. The `y1` signal remains asserted for the entire clock cycle. At t_{12}, a new clock edge arrives, the `state_reg` signal changes to `s0` after the T_{cq} delay (at t_{13}), and the FSM returns to the `s0` state. The `y1` signal is deactivated after the $T_{output(mo)}$ delay (at t_{14}).

The timing diagram illustrates the major difference between an ASM chart and a regular flowchart. In an ASM chart, the state transition (or ASM block transition) occurs only at the rising edge of the clock signal. Within the clock period, the Boolean condition and the next state may change but have no effect on the system state. The new state is determined solely by the values sampled at the rising edge of the clock.

10.4 MOORE MACHINE VERSUS MEALY MACHINE

As we discussed in Section 10.1, an FSM can be classified into a Moore machine or a Mealy machine. In theoretical computer science, a Moore machine and a Mealy machine are considered to have similar computation capability (both can recognize "regular expressions"), although a Mealy machine normally accomplishes the same task with fewer states. When the FSM is used as a control circuit, the control signals generated by a Moore machine and a Mealy machine have different timing characteristics. Understanding the subtle timing difference is critical for the correctness and efficiency of a control circuit. We use a simple

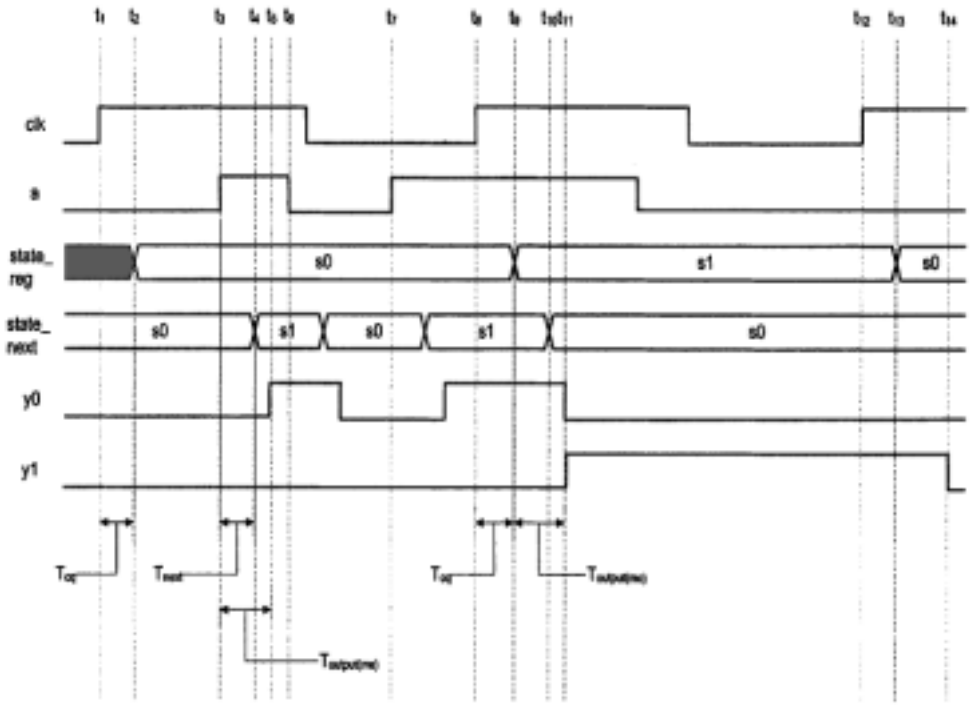


Figure 10.11 FSM timing diagram.

edge detection circuit to illustrate the difference between a Mealy machine and a Moore machine.

10.4.1 Edge detection circuit

We assume that a synchronous system is connected to a slowly varying input signal, `strobe`, which can be asserted to '1' for a long time (much greater than the clock period of the FSM). An edge detection circuit is used to detect the rising edge of the `strobe` signal. It generates a "short" pulse when the `strobe` signal changes from '0' to '1'. The width of the output pulse is about the same or less than a clock period of the FSM. Since the intention is to show the difference between a Mealy machine and a Moore machine, we are deliberately vague about the specification of the width and timing of the output pulse.

The basic design idea is to construct an FSM that has a `zero` state and a `one` state, which represent that the input has been '0' or '1' for a long period of time respectively. The FSM has a single input signal, `strobe`, and a single output signal. The output will be asserted "momentarily" when the FSM transits from the `zero` state to the `one` state.

We first consider a design based on a Moore machine. The state diagram is shown in Figure 10.12(a). There are three states. In addition to the `zero` and `one` states, the FSM also has an `edge` state. When `strobe` becomes '1' in the `zero` state, it implies that `strobe` changes from '0' to '1'. The FSM moves to the `edge` state, in which the output signal, `p1`, is asserted. In normal operation, `strobe` should continue to be '1' and the FSM moves to the `one` state at the next rising edge of the clock and stays there until `strobe` returns to '0'.

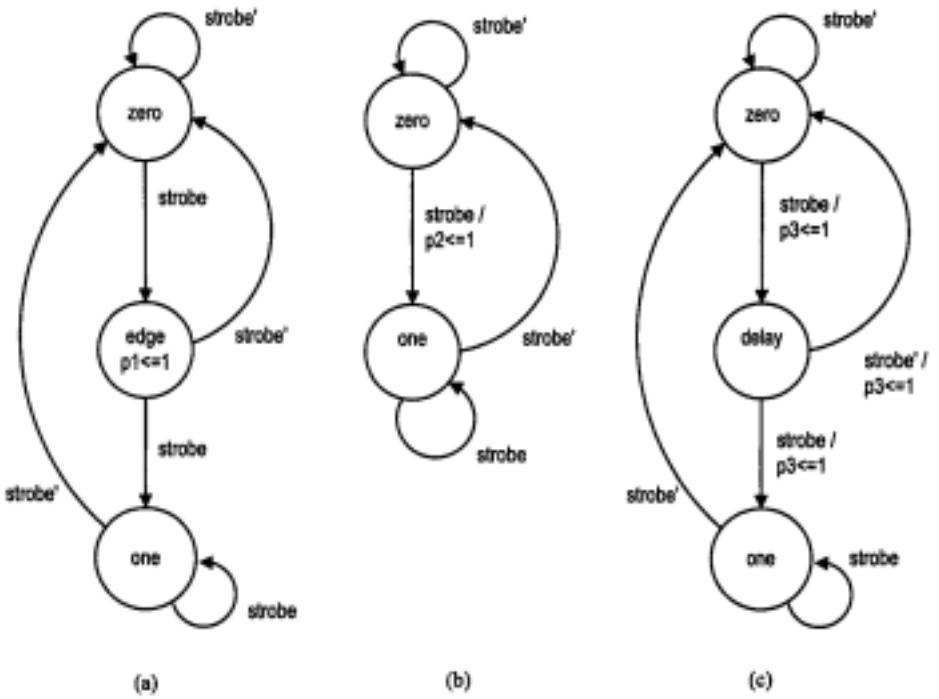


Figure 10.12 Edge detector state diagrams.

If `strobe` is really short and changes to '0' in the `edge` state, the FSM will return to the `zero` state. A representative timing diagram is shown in the top portion of Figure 10.13.

The second design is based on a Mealy machine. The state diagram is shown in Figure 10.12(b). It consists of only the `zero` and `one` states. When `strobe` changes from '0' to '1' in the `zero` state, the FSM moves to the `one` state. From the state diagram, it seems that the output signal, `p2`, is asserted when the FSM transit from the `zero` state to the `one` state. Actually, p2 is asserted in the `zero` state whenever `strobe` is '1'. When the FSM moves to the `one` state, p2 will be deasserted. The timing diagram is shown in the middle portion of Figure 10.13.

For demonstration purposes, we also include a version that combines both types of outputs. The third design inserts a `delay` state into the Mealy machine–based design and prolongs the output pulse for one extra clock cycle. The state diagram is shown in Figure 10.12(c). In this design, the FSM will assert the output, p3, in the `zero` state, as in the second design. However, the FSM moves to the `delay` state afterward and forces p3 to be asserted for another clock cycle by placing the assertion on both transition edges of the `delay` state. Note that since p3 is asserted in the `delay` state under all transition arcs, it implies that p3 will be asserted in the `delay` state regardless of the input condition. The behavior of the FSM in the `delay` state is similar to the `edge` state of the Moore machine–based design, and we can also move the output assertion, `p3<=1`, into the bubble of the `delay` state. The timing diagram is shown in the bottom portion of Figure 10.13.

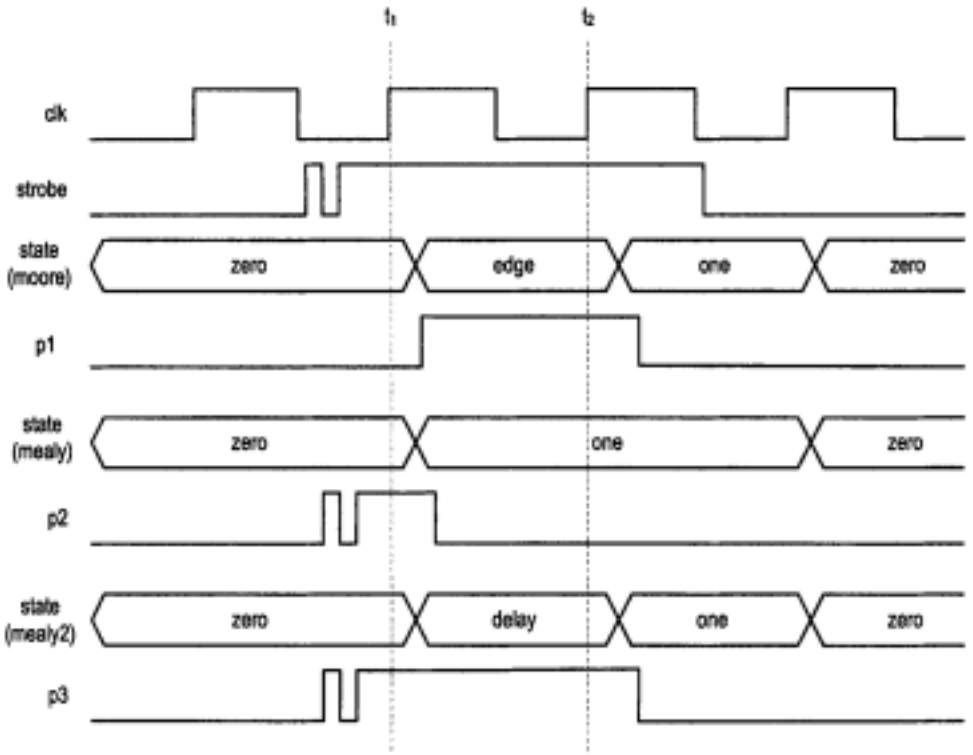


Figure 10.13 Edge detector timing diagram.

10.4.2 Comparison of Moore output and Mealy output

All three edge detector designs can generate a "short" pulse when the input changes from '0' to '1', but there are subtle differences. Understanding the differences is the key to deriving a correct and efficient FSM and an FSM-based control circuit.

There are three major differences between the Moore machine and Mealy machine–based designs. First, a Mealy machine normally requires fewer states to perform the same task. This is due to the fact that its output is a function of states and external inputs, and thus several possible output values can be specified in one state. For example, in the `zero` state of the second design, `p2` can be either '0' or '1', depending on the value of `strobe`. Thus, the Mealy machine–based design requires only two states whereas the Moore machine–based design requires three states.

Second, a Mealy machine can generate a faster response. Since a Mealy output is a function of input, it changes whenever the input meets the designated condition. For example, in Mealy machine–based design, if the FSM is in the `zero` state, `p2` is asserted immediately after `strobe` changes from '0' to '1', as shown in the timing diagram. On the other hand, a Moore machine reacts indirectly to input changes. The Moore machine–based design also senses the changes of `strobe` in the `zero` state. However, it has to wait until the next state (i.e., the `edge` state) to respond. The change causes the FSM to move to the `edge` state. At the next rising edge of the clock, the FSM moves to this state and `p1` responds accordingly, as shown in the timing diagram. In a synchronous system, the distinction

between a Mealy output and a Moore output normally means a delay of one clock cycle. Recall that the input signal of a synchronous system is sampled only at the rising edge of the clock. Let us assume that the output of the edge detection circuit is used by another synchronous system. Consider the first transition edge of `strobe` in Figure 10.13. The p2 signal can be sampled at t_1. However, the p1 signal is not available at that time because of the clock-to-q delay and output logic delay. Its value can be sampled only by the next rising edge at t_2.

The third difference involves the control of the width and timing of the output signal. In a Mealy machine, the width of an output signal is determined by the input signal. The output signal is activated when the input signal meets the designated condition and is normally deactivated when the FSM enters a new state. Thus, its width varies with input and can be very narrow. Also, a Mealy machine is susceptible to glitches in the input signal and passes these undesired disturbances to the output. This is shown in the p2 signal of Figure 10.13. On the other hand, the output of a Moore machine is synchronized with the clock edge and its width is about the same as a clock period. It is not susceptible to glitches from the input signal. Although the output logic can still introduce glitches, this can be overcome by clever output buffering schemes, which are discussed in Section 10.7.

As mentioned earlier, our focus on FSM is primarily on its application as a control circuit. From this perspective, selection between a Mealy machine and a Moore machine depends on the need of control signals. We can divide control signals into two categories: edge-sensitive and level-sensitive. An *edge-sensitive* control signal is used as input for a sequential circuit synchronized by the same clock. A simple example is the enable signal of a counter. Since the signal is sampled only at the rising edge of the clock, the width of the signal and the existence of glitches do not matter as long as it is stable during the setup and hold times of the clock edge. Both the Mealy and the Moore machines can generate output signals that meet this requirement. However, a Mealy machine is preferred since it uses fewer states and responds one clock faster than does a Moore machine. Note that the p3 signal generated by the modified Mealy machine will be active for two clock edges and is actually incorrect for an edge-sensitive control signal.

A *level-sensitive* control signal means that a signal has to be asserted for a certain amount of time. When asserted, it has to be stable and free of glitches. A good example is the write enable signal of an SRAM chip. A Moore machine is preferred since it can accurately control the activation time of its output, and can shield the control signal from input glitches. Because of the potential glitches, the p3 signal is again not desirable.

10.5 VHDL DESCRIPTION OF AN FSM

The block diagram of an FSM shown in Figure 10.1 is similar to that of the regular sequential circuit shown in Figure 8.5. Thus, derivation of VHDL code for an FSM is similar to derivation for a regular sequential circuit. We first identify and separate the memory elements and then derive the next-state logic and output logic. There are two differences in the derivation. The first is that symbolic states are used in an FSM description. To capture this kind of representation, we utilize VHDL's *enumeration data type* for the state registers. The second difference is in the derivation of the next-state logic. Instead of using a regular combinational circuit, such as an incrementor or shifter, we have to construct the code according to a state diagram or ASM chart.

We use the previous memory controller FSM to show the derivation procedure in the following subsections.

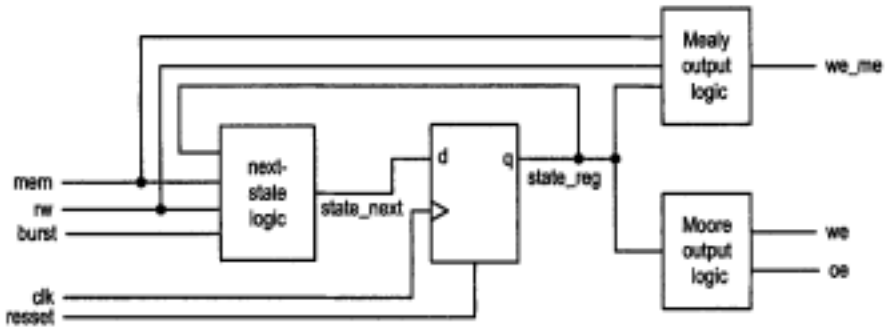


Figure 10.14 Block diagram of a memory controller FSM.

10.5.1 Multi-segment coding style

The first method is to derive the VHDL code according to the blocks of a block diagram, and we call it the multi-segment coding style. The block diagram of the previous memory controller is shown in Figure 10.14. There are four blocks and we use a VHDL code segment for each block. The complete VHDL code is shown in Listing 10.1.

Listing 10.1 Multi-segment memory controller FSM

```
library ieee;
use ieee.std_logic_1164.all;
entity mem_ctrl is
   port(
      clk, reset: in std_logic;
      mem, rw, burst: in std_logic;
      oe, we, we_me: out std_logic
   );
end mem_ctrl ;

architecture mult_seg_arch of mem_ctrl is
   type mc_state_type is
      (idle, read1, read2, read3, read4, write);
   signal state_reg, state_next: mc_state_type;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- next-state logic
   process(state_reg,mem,rw,burst)
   begin
      case state_reg is
         when idle =>
            if mem='1' then
```

```
                if rw='1' then
                   state_next <= read1;
                else
                   state_next <= write;
                end if;
             else
                state_next <= idle;
             end if;
          when write =>
             state_next <= idle;
          when read1 =>
             if (burst='1') then
                state_next <= read2;
             else
                state_next <= idle;
             end if;
          when read2 =>
             state_next <= read3;
          when read3 =>
             state_next <= read4;
          when read4 =>
             state_next <= idle;
       end case;
    end process;
    -- Moore output logic
    process(state_reg)
    begin
       we <= '0'; -- default value
       oe <= '0';  -- default value
       case state_reg is
          when idle =>
          when write =>
             we <= '1';
          when read1 =>
             oe <= '1';
          when read2 =>
             oe <= '1';
          when read3 =>
             oe <= '1';
          when read4 =>
             oe <= '1';
       end case;
    end process;
    -- Mealy output logic
    process(state_reg,mem,rw)
    begin
       we_me <= '0'; -- default value
       case state_reg is
          when idle =>
             if (mem='1') and (rw='0') then
               we_me <= '1';
             end if;
          when write =>
```

```
            when read1 =>
            when read2 =>
            when read3 =>
            when read4 =>
         end case;
      end process;
end mult_seq_arch;
```

Inside the architecture declaration, we use the VHDL's enumeration data type. The data type is declared as

```
type mc_state_type is (idle,read1,read2,read3,read4,write);
```

The syntax of the enumeration data type statement is very simple:

```
type type_name is (list_of_all_possible_values);
```

It simply enumerates all possible values in a list. In this particular example, we list all the symbolic state names. The next statement then uses this newly defined type as the data type for the state register's input and output:

```
signal state_reg, state_next: mc_state_type;
```

The architecture body is divided into four code segments. The first segment is for the state register. Its code is like that of a regular register except that a user-defined data type is used for the signal. We use an asynchronous `reset` signal for initialization. The state register is cleared to the `idle` state when the `reset` signal is asserted.

The second code segment is for the next-state logic and is the key part of the FSM description. It is patterned after the ASM chart of Figure 10.9. We use a case statement with `state_reg` as the selection expression. The `state_reg` signal is the output of the state register and represents the current state of the FSM. Based on its value and input signal, the next state, denoted by the `state_next` signal, can be determined. As shown in the previous segment, the next state will be stored into the state register and becomes the new state at the rising edge of the clock. The `state_next` signal can be derived directly from the ASM block. For a simple ASM block, such as the `read2` block, there is only one exit path and the `state_next` signal is very straightforward:

```
state_next <= idle;
```

For a block with multiple exit paths, we can use if statements to code the decision boxes. The Boolean condition inside a decision box can be directly translated to the Boolean expression of the if statement, and the two exit paths can be expressed as the then branch and the else branch of the if statement. Thus, we can follow the decision boxes and derive the VHDL code for the `state_next` signal accordingly. For example, in the `idle` block, the cascade decision boxes can be translated into a nested if statement:

```
if mem='1' then
   if rw='1' then
      state_next <= read1;
   else
      state_next <= write;
   end if;
else
   state_next <= idle;
end if;
```

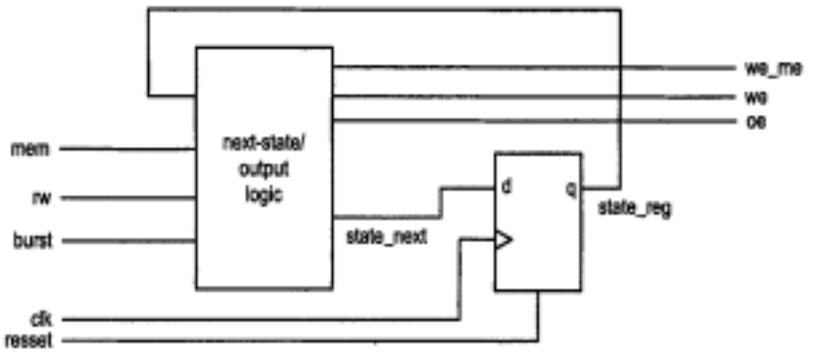


Figure 10.15 Block diagram of a two-segment memory controller FSM.

Note that the ASM has three possible exit paths from the `idle` block, and thus the `state_next` signal has three possible values.

The third code segment is the Moore output logic. Again, we use a case statement with `state_reg` as the selection expression. Note that since the Moore output is a function of state only, no input signal is in the sensitive list. Our code follows the ASM chart. Two sequential signal assignment statements are used to represent the default output value:

```
we <= '0';
oe <= '0';
```

If an output signal is asserted inside a state box, we put a signal assignment statement in the corresponding choice in the VHDL code to overwrite the default value.

The fourth code segment is the Mealy output logic. Note that some input signal is now in the sensitive list. Again, following the ASM chart, we use a case statement with `state_reg` as the selection expression and use an if statement for the decision box. The Mealy output, the `we_me` signal, will be assigned to the designated value according to the input condition.

We intentionally use the case statement to demonstrate the relationship between the code and the ASM chart. It may become somewhat cumbersome. The segment can also be written in a more compact but ad hoc way. For example, the Mealy output logic segment can be rewritten as

```
we_me <= '1' when ((state_reg=idle) and (nen='1') and
                   (rw='0')) else
         '0';
```

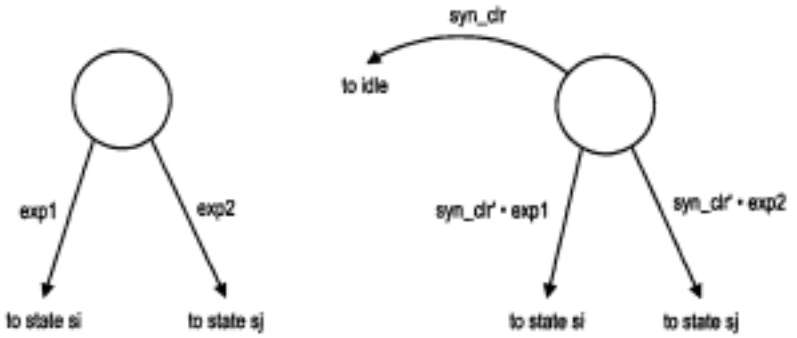
10.5.2 Two-segment coding style

The two-segment coding style divides an FSM into a state register segment and a combinational circuit segment, which integrates the next-state logic, Moore output logic and Mealy output logic. In VHDL code, we need to merge the three segments and move the `state_next`, `oe`, `we` and `we_me` signals into a single process. The block diagram is shown in Figure 10.15. The architecture body of this revised code is shown in Listing 10.2.

Listing 10.2 Two-segment memory controller FSM

```
architecture two_seg_arch of mem_ctrl is
   type mc_state_type is
```

```
        (idle, read1, read2, read3, read4, write);
    signal state_reg, state_next: mc_state_type;
begin
    -- state register
    process(clk,reset)
    begin
       if (reset='1') then
          state_reg <= idle;
       elsif (clk'event and clk='1') then
          state_reg <= state_next;
       end if;
    end process;
    -- next-state logic and output logic
    process(state_reg,mem,rw,burst)
    begin
       oe <= '0';     -- default values
       we <= '0';
       we_me <= '0';
       case state_reg is
          when idle =>
             if mem='1' then
                if rw='1' then
                   state_next <= read1;
                else
                   state_next <= write;
                   we_me <= '1';
                end if;
             else
                state_next <= idle;
             end if;
          when write =>
             state_next <= idle;
             we <= '1';
          when read1 =>
             if (burst='1') then
                state_next <= read2;
             else
                state_next <= idle;
             end if;
             oe <= '1';
          when read2 =>
             state_next <= read3;
             oe <= '1';
          when read3 =>
             state_next <= read4;
             oe <= '1';
          when read4 =>
             state_next <= idle;
             oe <= '1';
       end case;
    end process;
end two_seg_arch;
```



(a) Original state (b) Revised state with synchronous clear

Figure 10.16 Adding synchronous clear to a state diagram.

10.5.3 Synchronous FSM Initialization

An alternative for the asynchronous initialization is to use a synchronous clear signal. To achieve this goal, we have to add an additional transition arc for every state. The logic expression of this arc corresponds to the assertion of the synchronous clear signal and is given preference over other conditions. Assume that the `syn_clr` signal is added to the FSM for this purpose and an FSM will be forced to the `idle` state when the `syn_clr` signal is asserted. The required revision for a state is shown in Figure 10.16.

Although revising a state diagram or an ASM chart introduces a significant amount of clutter, this can be done easily in VHDL. We just add an extra if statement to check the `syn_clr` signal in the next-state logic segment. If the condition `syn_clr='1'` is `true`, the `idle` value will be assigned to the `state_next` signal. Otherwise, the FSM takes the else branch and performs the normal transition. The needed revisions for the memory controller FSM example are shown below.

```
entity mem_ctrl is
port (
   syn_clr: in std_logic;   -- new input
   . . .

architecture mult_seg_arch of mem_ctrl is
. . .
begin
. . .
   -- next-state logic
   process(state_reg,mem,rw,burst,syn_clr)
   begin
      if (syn_clr='1') then  -- synchronous clear
         state_next <= idle;
      else  -- original state_next values
         case state_reg is
            when idle =>
            . . .
            . . .
```

```
        end case;
      end if;
    end process;
  . . .
```

10.5.4 One-segment coding style and its problem

We may be tempted to make the code more compact and describe the FSM in a single segment, as shown in Listing 10.3.

Listing 10.3 One-segment memory controller FSM

```
architecture one_seg_wrong_arch of mem_ctrl is
   type mc_state_type is
        (idle, read1, read2, read3, read4, write);
   signal state_reg: mc_state_type;
begin
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
      elsif (clk'event and clk='1') then
         oe <= '0';      -- default values
         we <= '0';
         we_me <= '0';
         case state_reg is
            when idle =>
               if mem='1' then
                  if rw='1' then
                     state_reg <= read1;
                  else
                     state_reg <= write;
                     we_me <= '1';
                  end if;
               else
                  state_reg <= idle;
               end if;
            when write =>
               state_reg <= idle;
               we <= '1';
            when read1 =>
               if (burst='1') then
                  state_reg <= read2;
               else
                  state_reg <= idle;
               end if;
               oe <= '1';
            when read2 =>
               state_reg <= read3;
               oe <= '1';
            when read3 =>
               state_reg <= read4;
               oe <= '1';
```

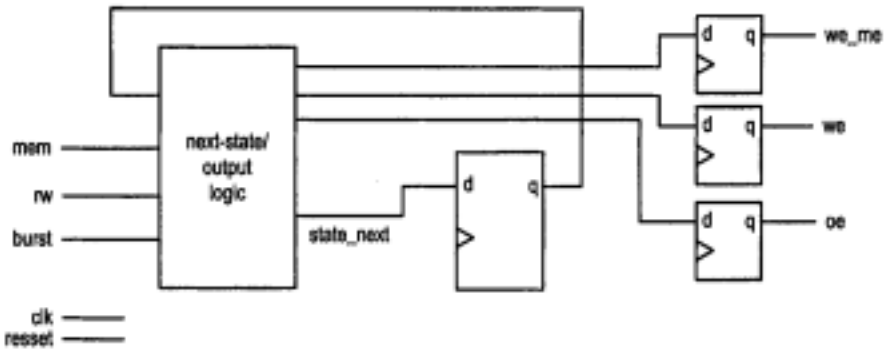


Figure 10.17 FSM with unwanted output buffers.

```
                when read4 =>
                   state_reg <= idle;
                   oe <= '1';
45           end case;
          end if;
       end process;
    end one_seg_wrong_arch;
```

Unfortunately, this code suffers the same problem as that of the similar regular sequential circuit code discussed in Section 8.7. Recall that a left-hand-side signal within the `clk'event and clk='1'` branch infers a register. While this is the desired effect for the `state_reg` signal, three unwanted registers are inferred for the `oe`, `we` and `we_me` signals, as shown in Figure 10.17 (for clarity, the connection lines for the `clk` and `reset` signals are not shown). These signals are delayed by one clock cycle and the code does not meet the specification described by the ASM chart. Although we can fix the problem by using a separate process for the output logic, the resulting code is less clear. We generally refrain from this style of coding.

10.5.5 Synthesis and optimization of FSM

After dividing a sequential circuit into a register and a combinational circuit, we can apply RT-level optimization techniques for the combinational circuit. However, these techniques are mainly for regular combinational circuits. The next-state logic and output logic of the FSMs are normally random in nature since the code includes primarily case and if statements and does not involve complex operators. These circuits are implemented by gate-level components, and there is very little optimization that we can do when writing RT-level VHDL code. Utilizing two-segment coding provides some degree of sharing since the Boolean expressions inside the decision boxes are used by both next-state logic and output logic.

Theoretically, there is a technique to identify the "equivalent states" of an FSM. We can merge these states into one state and thus reduce the number of states of the FSM. However, in a properly designed FSM, the chance of finding a set of equivalent states is very slim, and this technique is not always applied in the design and synthesis process.

There is one other unique opportunity to reduce the complexity of the combinational circuit of the FSM: assigning proper binary representations for the symbolic states. This issue is discussed in the next section.

The multi- and two-segment coding approach of previous subsections is very general and we can use the two VHDL listings as templates. The key to developing good VHDL code for an FSM is the derivation of an efficient and correct state diagram or ASM chart. Once it is completed, obtaining VHDL code becomes more or less a mechanical procedure. Some design entry software can accept a graphical state diagram and convert it to VHDL code automatically.

10.6 STATE ASSIGNMENT

Our discussion of FSM so far utilizes only symbolic states. During synthesis, each symbolic state has to be mapped to a unique binary representation so that the FSM can be realized by physical hardware. *State assignment* is the process of mapping symbolic values to binary representations.

10.6.1 Overview of state assignment

For a synchronous FSM, the circuit is not delay sensitive and is immune to hazards. As long as the clock period is large enough, the synthesized circuit will function properly for any state assignment. However, physical implementation of next-state logic and output logic is different for each assignment. A good assignment can reduce the circuit size and decrease the propagation delays, which in turn, increases the clock rate of the FSM.

An FSM with n symbolic states requires a state register of at least $\lceil \log_2 n \rceil$ bits to encode all possible symbolic values. We sometimes utilize more bits for other purposes. There are several commonly used state assignment schemes:

- *Binary* (or *sequential*) *assignment*: assigns states according to a binary sequence. This scheme uses a minimal number of bits and needs only a $\lceil \log_2 n \rceil$-bit register.
- *Gray code assignment*: assigns states according to a Gray code sequence. This scheme also uses a minimal number of bits. Because only one bit changes between the successive code words in the sequence, we may reduce the complexity of next-state logic if assigning successive code words to neighboring states.
- *One-hot assignment*: assigns one bit for each state, and thus only a single bit is '1' (or "hot") at a time. For an FSM with n states, this scheme needs an n-bit register.
- *Almost one-hot assignment*: is similar to the one-hot assignment except that the all-zero representation ("$0 \cdots 0$") is also included. The all-zero state is frequently used as the initial state since it can easily be reached by asserting the asynchronous reset signal of D FFs. This scheme needs an $(n-1)$-bit register for n states.

Although one-hot and almost one-hot assignments need more register bits, empirical data from various studies show that these assignments may reduce the circuit size of next-state logic and output logic. Table 10.1 illustrates these schemes used for the previous memory controller FSM.

Obtaining the optimal assignment is very difficult. For example, if we choose the one-hot scheme for an FSM with n states, there are $n!$ (which is worse than 2^n) possible assignments. It is not practical to obtain the optimal assignment by examining all possible combinations. However, there exists special software that utilizes heuristic algorithms that can obtain a good, suboptimal assignment.

Table 10.1 State assignment example

	Binary assignment	Gray code assignment	One-hot assignment	Almost one-hot assignment
idle	000	000	000001	00000
read1	001	001	000010	00001
read2	010	011	000100	00010
read3	011	010	001000	00100
read4	100	110	010000	01000
write	101	111	100000	10000

10.6.2 State assignment in VHDL

In some situations, we may want to specify the state assignment for an FSM manually. This can be done implicitly or explicitly. In *implicit state assignment*, we keep the original enumeration data type but pass the desired assignment by other mechanisms. The VHDL standard does not define any rule for mapping the values of an enumeration data type to a set of binary representations. It is performed during synthesis. One way to pass the desired statement assignment to software is to use a VHDL feature, known as a *user attribute*, to set a "directive" to guide operation of the software. A user attribute has no effect on the semantics of VHDL code and is recognized only by the software that defines it. The IEEE 1076.6 RTL synthesis standard defines an attribute named `enum_encoding` for encoding the values of an enumeration data type. This attribute can be used for state assignment. For example, if we wish to assign the binary representations "0000", "0100", "1000", "1001", "1010" and "1011" to the `idle`, `write`, `read1`, `read2`, `read3` and `read4` states of the memory controller FSM, we can add the following VHDL segment to the original code:

```
type mc_state_type is (idle,write,read1,read2,read3,read4);
attribute enum_encoding: string;
attribute enum_encoding of mc_state_type:
          type is "0000 0100 1000 1001 1010 1011";
```

This user attribute is very common and should be accepted by most synthesis software.

Synthesis software normally provides several simple state assignment schemes similar to the ones discussed in the previous subsection. If we don't utilize a user attribute, we can specify the desired scheme as a parameter while invoking the software. If nothing is specified, the software will perform the state assignment automatically. It normally selects between binary assignment and one-hot assignment, depending on the characteristics of the targeting device technology. We can also use specialized FSM optimization software to obtain a good, suboptimal assignment.

We can *explicitly* specify the desired state assignment by replacing the symbolic values with the actual binary representations, and use the `std_logic_vector` data type for this purpose. To demonstrate this scheme, we incorporate the previous state assignment into the memory controller FSM. The revised multi-segment VHDL code is shown in Listing 10.4.

Listing 10.4 Explicit user-defined state assignment

```
architecture state_assign_arch of mem_ctrl is
   constant idle:  std_logic_vector(3 downto 0):="0000";
   constant write: std_logic_vector(3 downto 0):="0100";
   constant read1: std_logic_vector(3 downto 0):="1000";
```

```
    constant read2: std_logic_vector(3 downto 0):="1001";
    constant read3: std_logic_vector(3 downto 0):="1010";
    constant read4: std_logic_vector(3 downto 0):="1011";
    signal state_reg,state_next: std_logic_vector(3 downto 0);
begin
   -- state register
    process(clk,reset)
    begin
       if (reset='1') then
          state_reg <= idle;
       elsif (clk'event and clk='1') then
          state_reg <= state_next;
       end if;
    end process;
    -- next-state logic
    process(state_reg,mem,rw,burst)
    begin
       case state_reg is
          when idle =>
             if mem='1' then
                if rw='1' then
                   state_next <= read1;
                else
                   state_next <= write;
                end if;
             else
                state_next <= idle;
             end if;
          when write =>
             state_next <= idle;
          when read1 =>
             if (burst='1') then
                state_next <= read2;
             else
                state_next <= idle;
             end if;
          when read2 =>
             state_next <= read3;
          when read3 =>
             state_next <= read4;
          when read4 =>
             state_next <= idle;
          when others =>
             state_next <= idle;
       end case;
    end process;
    -- Moore output logic
    process(state_reg)
    begin
       we <= '0'; -- default value
       oe <= '0';  -- default value
       case state_reg is
          when idle =>
```

```
            when write =>
               we <= '1';
            when read1 =>
               oe <= '1';
            when read2 =>
               oe <= '1';
            when read3 =>
               oe <= '1';
            when read4 =>
              oe <= '1';
            when others =>
         end case;
   end process;
   -- Mealy output logic
   we_ne <= '1' when ((state_reg=idle) and (mem='1') and
                      (rw='0')) else
            '0';
end state_assign_arch;
```

In this code, we use `std_logic_vector(3 downto 0)` as the state register's data type. Six constants are declared to represent the six symbolic state names. Because of the choice of the constant names, the appearance of the code is very similar to that of the original code. However, the name here is just an alias of a binary representation, but the name in the original code is a value of the enumeration data type. One difference in the next-state logic code segment is an extra when clause:

```
when others =>
   state_next <= idle;
```

This revision is necessary since the selection expression of the case statement, `state_reg`, now is with the `std_logic_vector(3 downto 0)` data type, and thus has 9^4 possible combinations. The **when others** clause is used to cover all the unused combinations. This mean that when the FSM reaches an unused binary representation (e.g., "1111"), it will return to the `idle` state in the next clock cycle. We can also use

```
when others =>
   state_next <= "----";
```

if the software accepts the don't-care expression. A **when others** clause is also added for the Moore output code segment.

The explicit state assignment allows us to have more control over the FSM but makes the code more difficult to maintain and prevents the use of FSM optimization software. Unless there is a special need, using an enumeration data type for state representation is preferred.

10.6.3 Handling the unused states

When we map the symbolic states of an FSM to binary representations, there frequently exist unused binary representations (or states). For example, there are six states in the memory controller FSM. If the binary assignment is used, a 3-bit (i.e., $\lceil \log_2 6 \rceil$) register is needed. Since there are 2^3 possible combinations from 3 bits, two binary states are not used in the mapping. If one-hot state assignment is used, there are 58 (i.e., $2^6 - 6$) unused states.

During the normal operation, the FSM will not reach these states; however, it may accidentally enter an unused state due to noise or an external disturbance. One question is what we should do if the FSM reaches an unused state.

In certain applications, we can simply ignore the situation. It is because we assume that the error will never happen, or, if it happens, the system can never recover. In the latter case, there is nothing we can do with the error.

On the other hand, some applications can resume from a short period of anomaly and continue to run. In this case we have to design an FSM that can recover from the unused states. It is known as a *fault-tolerant* or *safe FSM*. For an FSM coded with an explicit state assignment, incorporating this feature is straightforward. We just specify the desired action in the **when others** clause of the case statement. For example, the `state_assign_arch` architecture in Listing 10.4 is a safe FSM. The code specifies that the FSM returns to the `idle` state if it enters an unused state:

```
when others =>
   state_next <= idle;
```

If desired, we can revise the code to add an `error` state for special error handling:

```
when others =>
   state_next <= error;
```

There is no easy way to specify a safe FSM if the enumeration data type is used. Since all possible values of the enumeration data type are used in the case statement of the next-state logic, there is no unused state in VHDL code. The unused states emerge only later during synthesis, and thus they cannot be handled in VHDL code. Some software accepts an artificially added **when others** clause for the unused states. However, by VHDL definition, this clause is redundant and may not be interpreted consistently by different synthesis software.

10.7 MOORE OUTPUT BUFFERING

We can add a buffer by inserting a register or a D FF to any output signal. The purpose of an output buffer is to remove glitches and minimize the clock-to-output delay (T_{co}). The disadvantage of this approach is that the output signal is delayed by one clock cycle.

Since the output of an FSM is frequently used for control purposes, we sometimes need a fast, glitch-free signal. We can apply the regular output buffering scheme to a Mealy or Moore output signal. The buffered signal, of course, is delayed by one clock cycle. For a Moore output, it is possible to obtain a buffered signal without the delay penalty. The following subsections discuss how to design an FSM to achieve this goal.

10.7.1 Buffering by clever state assignment

In a typical Moore machine, we need combinational output logic to implement the output function, as shown in Figure 10.1. Since the Moore output is not a function of input signals, it is shielded from the glitches of the input signals. However, the state transition and output logic may still introduce glitches to the output signals. There are two sources of glitches. The first is the possible simultaneous multiple-bit transitions of the state register, as from the "111" state to the "000" state. Even the register bits are controlled by the same clock, the clock-to-q delay of each D FF may be slightly different, and thus a glitch may show up in the output signal. The second source is the possible hazards inside the output logic.

Table 10.2 State assignment for the memory controller FSM output buffering

	q_3q_2 (oe)(we)	q_1q_0	$q_3q_2q_1q_0$
idle	00	00	0000
read1	10	00	1000
read2	10	01	1001
read3	10	10	1010
read4	10	11	1011
write	01	00	0100

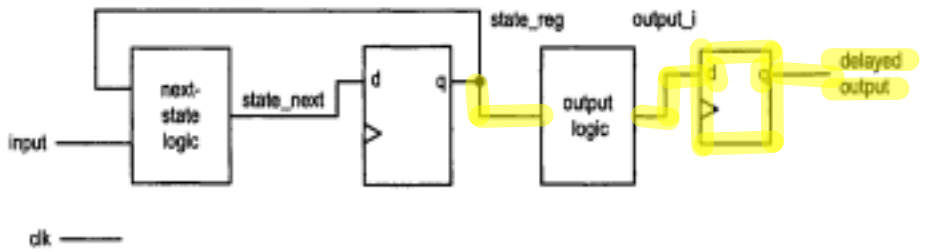
Recall that the clock-to-output delay (T_{co}) is the sum of the clock-to-q delay (T_{cq}) of the register and the propagation delay of the output logic. The existence of the output logic clearly increases the clock-to-output delay.

One way to reduce the effect of the output logic is to eliminate it completely by clever state assignment. In this approach, we first allocate a register bit to each Moore output signal and specify its value according to the output function. Again, let us consider the memory controller FSM. We can assign two register bits according to the output values of the `oe` and `we` signals, as shown in the first column of Table 10.2. Since some states may have the same output patterns, such as the `read1`, `read2`, `read3` and `read4` states of the memory controller, we need to add additional register bits to ensure that each state is mapped to a unique binary representation. In this example, we need at least two extra bits to distinguish the four read states, as shown in the second column of Table 10.2. We then can complete the state assignment by filling the necessary values for the `idle` and `write` states, as shown in the third column of Table 10.2. In this state assignment, the value of `oe` is identical to the value of `state_reg(3)`, and the value of `we` is identical to the value of `state_reg(2)`. In other words, the output function can be realized by connecting the output signals to the two register bits, and the output logic is reduced to wires. This implementation removes the sources of glitches and reduces T_{co} to T_{cq}.

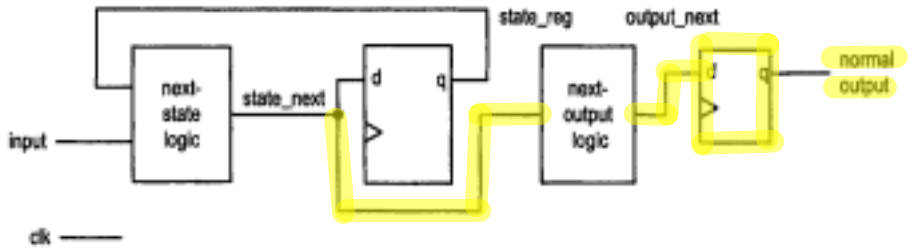
This design requires manual state assignment and access to individual register bits. Only the explicit state assignment can satisfy the requirement. The `state_assign_arch` architecture in Listing 10.4 actually uses the state assignment from Table 10.2. We can replace the Moore output logic code segment by connecting the output signals directly to the register's output:

```
-- Moore output logic
oe <= state_reg(3);
we <= state_reg(2);
```

Because the state register is also used as an output buffer, this approach potentially uses fewer register bits for certain output patterns. The disadvantage of this method is the manual manipulation of the state assignment. It becomes tedious as the number of states or output signals grows larger. Furthermore, the assignment has to be modified whenever the number of output signals is changed, the number of states is changed, or the output function is modified. This makes the code error-prone and difficult to maintain.



(a) Moore output with a regular output buffer



(b) Moore output with a look-ahead output buffer

Figure 10.18 Block diagrams of output buffering schemes.

10.7.2 Look-ahead output circuit for Moore output

A more systematic approach to Moore output buffering is to use a *look-ahead output circuit*. The basic idea is to buffer the *next output value* to cancel the one-clock delay introduced by the output buffer. In most systems, we don't know a signal's next or future value. However, in an FSM, the next value of the state register is generated by next-state logic and is always available.

This scheme can best be explained by examining the basic FSM block diagram. The block diagram of an FSM with a regular output buffering structure is shown in Figure 10.18(a). The output signals, of course, are delayed by one clock cycle. To cancel the effect of the delay, we can feed the output buffer with the next output value. After being delayed by one clock cycle, the next output value becomes the current output value, which is the desired output. Obtaining the next output is very straightforward. Recall that the current output is a function of the current state, which is the output of the state register, labeled `state_reg` in the diagram. The next output should be a function of the next state, which is the output of next-state logic, labeled `state_next` in the diagram. To obtain the next output, we need only disconnect the input of the output logic from the `state_reg` signal and reconnect it to the `state_next` signal, as shown in Figure 10.18(b).

Once understanding the block diagrams of Figure 10.18, we can develop the VHDL code accordingly. Again, we use the memory controller FSM as an example. The `we_ne` output will be ignored since it is irrelevant to the Moore output buffering. Note that the state register and next-state logic are the same as in the original block diagram, and only the Moore output logic part is modified. For comparison purposes, we show the VHDL codes for both diagrams. The codes are based on the `mutli_seg_arch` architecture of Section 10.5.1. The code of the memory controller FSM with a regular output buffer is shown in Listing 10.5.

Listing 10.5 FSM with a regular output buffer

```
architecture plain_buffer_arch of mem_ctrl is
   type mc_state_type is
      (idle, read1, read2, read3, read4, write);
   signal state_reg, state_next: mc_state_type;
   signal oe_i, we_i, oe_buf_reg, we_buf_reg: std_logic;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- output buffer
   process(clk,reset)
   begin
      if (reset='1') then
         oe_buf_reg <= '0';
         we_buf_reg <= '0';
      elsif (clk'event and clk='1') then
         oe_buf_reg <= oe_i;
         we_buf_reg <= we_i;
      end if;
   end process;
   -- next-state logic
   process(state_reg,mem,rw,burst)
   begin
      case state_reg is
         when idle =>
            if mem='1' then
               if rw='1' then
                  state_next <= read1;
               else
                  state_next <= write;
               end if;
            else
               state_next <= idle;
            end if;
         when write =>
            state_next <= idle;
         when read1 =>
            if (burst='1') then
               state_next <= read2;
            else
               state_next <= idle;
            end if;
         when read2 =>
            state_next <= read3;
         when read3 =>
            state_next <= read4;
```

```
            when read4 =>
               state_next <= idle;
      end case;
   end process;
   -- Moore output logic
   process(state_reg)
   begin
      we_i <= '0'; -- default value
      oe_i <= '0';  -- default value
      case state_reg is
         when idle =>
         when write =>
            we_i <= '1';
         when read1 =>
            oe_i <= '1';
         when read2 =>
            oe_i <= '1';
         when read3 =>
            oe_i <= '1';
         when read4 =>
            oe_i <= '1';
      end case;
   end process;
   -- output
   we <= we_buf_reg;
   oe <= oe_buf_reg;
end plain_buffer_arch;
```

In this code, we rename the original output signals of the output logic with a post-fix "_i" (for intermediate output signals). These signals are then connected to the output buffers.

To obtain the VHDL code for the look-ahead output buffer, we change the input of the output logic. This can be done by substituting the `state_reg` signal with the `state_next` signal in the case statement and the sensitivity list of the process. To make the code more descriptive, we use the post-fix "_next" for the next output signals. The modified code is shown in Listing 10.6.

Listing 10.6 FSM with a look-ahead output buffer

```
architecture look_ahead_buffer_arch of mem_ctrl is
   type mc_state_type is
      (idle, read1, read2, read3, read4, write);
   signal state_reg, state_next: mc_state_type;
   signal oe_next,we_next,oe_buf_reg,we_buf_reg: std_logic;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- output buffer
```

```
process(clk,reset)
begin
   if (reset='1') then
      oe_buf_reg <= '0';
      we_buf_reg <= '0';
   elsif (clk'event and clk='1') then
      oe_buf_reg <= oe_next;
      we_buf_reg <= we_next;
   end if;
end process;
-- next-state logic
process(state_reg,mem,rw,burst)
begin
   case state_reg is
      when idle =>
         if mem='1' then
            if rw='1' then
               state_next <= read1;
            else
               state_next <= write;
            end if;
         else
            state_next <= idle;
         end if;
       when write =>
          state_next <= idle;
       when read1 =>
          if (burst='1') then
             state_next <= read2;
          else
             state_next <= idle;
          end if;
       when read2 =>
          state_next <= read3;
       when read3 =>
          state_next <= read4;
       when read4 =>
          state_next <= idle;
    end case;
end process;
-- look-ahead output logic
process(state_next)
begin
   we_next <= '0'; -- default value
   oe_next <= '0'; -- default value
   case state_next is
      when idle =>
      when write =>
         we_next <= '1';
      when read1 =>
         oe_next <= '1';
      when read2 =>
         oe_next <= '1';
```

```
            when read3 =>
               oe_next <= '1';
            when read4 =>
               oe_next <= '1';
         end case;
      end process;
      -- output
      we <= we_buf_reg;
      oe <= oe_buf_reg;
   end look_ahead_buffer_arch;
```

The look-ahead buffer is a very effective scheme for buffering Moore output. It provides a glitch-free output signal and reduces T_{co} to T_{cq}. Furthermore, this scheme has no effect on the next-state logic or state assignment and needs only minimal modification over the original code.

10.8 FSM DESIGN EXAMPLES

Our focus on the FSM is to use it as the control circuit in large systems. Such systems involve a data path that is composed of regular sequential circuits, and are discussed in Chapters 11 and 12. This section shows several simple stand-alone FSM applications.

10.8.1 Edge detection circuit

The VHDL code for the Moore machine–based edge detection design of Section 10.4.1 is shown in Listing 10.7. The code is based on the state diagram of Figure 10.12(a) and is done in multi-segment style.

Listing 10.7 Edge detector with regular Moore output

```
library ieee;
use ieee.std_logic_1164.all;
entity edge_detector1 is
   port(
      clk, reset: in std_logic;
      strobe: in std_logic;
      p1: out std_logic
   );
end edge_detector1;

architecture moore_arch of edge_detector1 is
   type state_type is (zero, edge, one);
   signal state_reg, state_next: state_type;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= zero;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
```

Table 10.3 State assignment for edge detector output buffering

State	state_reg(1) (p1)	state_reg(0)
zero	0	0
edge	1	0
one	0	1

```
    end process;
    -- next-state logic
    process(state_reg,strobe)
    begin
       case state_reg is
          when zero=>
             if strobe= '1' then
                state_next <= edge;
             else
                state_next <= zero;
             end if;
          when edge =>
             if strobe= '1' then
                state_next <= one;
             else
                state_next <= zero;
             end if;
          when one =>
             if strobe= '1' then
                state_next <= one;
             else
                state_next <= zero;
             end if;
       end case;
    end process;
    -- Moore output logic
    p1 <= '1' when state_reg=edge else
          '0';
end moore_arch;
```

Assume that we want the output signal to be glitch-free. We can do it by using the clever state assignment or look-ahead output buffer scheme. One possible state assignment is shown in Table 10.3, and the VHDL code is shown in Listing 10.8.

Listing 10.8 Edge detector with clever state assignment

```
architecture clever_assign_buf_arch of edge_detector1 is
   constant zero: std_logic_vector(1 downto 0):= "00";
   constant edge: std_logic_vector(1 downto 0):= "10";
   constant one: std_logic_vector(1 downto 0) := "01";
   signal state_reg,state_next: std_logic_vector(1 downto 0);
begin
   -- state register
   process(clk,reset)
```

```
    begin
10     if (reset='1') then
          state_reg <= zero;
       elsif (clk'event and clk='1') then
          state_reg <= state_next;
       end if;
15  end process;
    -- next-state logic
    process(state_reg,strobe)
    begin
       case state_reg is
20        when zero=>
             if strobe= '1' then
                state_next <= edge;
             else
                state_next <= zero;
25           end if;
          when edge =>
             if strobe= '1' then
                state_next <= one;
             else
30              state_next <= zero;
             end if;
          when others =>
             if strobe= '1' then
                state_next <= one;
35           else
                state_next <= zero;
             end if;
       end case;
    end process;
40  -- Moore output logic
    p1 <= state_reg(1);
end clever_assign_buf_arch;
```

The VHDL code for the look-ahead output circuit scheme is given in Listing 10.9.

Listing 10.9 Edge detector with a look-ahead output buffer

```
architecture look_ahead_arch of edge_detector1 is
   type state_type is (zero, edge, one);
   signal state_reg, state_next: state_type;
   signal p1_reg, p1_next: std_logic;
5 begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
10       state_reg <= zero;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
15 -- output buffer
```

```
   process(clk,reset)
   begin
      if (reset='1') then
         p1_reg <= '0';
      elsif (clk'event and clk='1') then
         p1_reg <= p1_next;
      end if;
   end process;
   -- next-state logic
   process(state_reg,strobe)
   begin
      case state_reg is
         when zero=>
            if strobe= '1' then
               state_next <= edge;
            else
               state_next <= zero;
            end if;
         when edge =>
            if strobe= '1' then
               state_next <= one;
            else
               state_next <= zero;
            end if;
         when one =>
            if strobe= '1' then
               state_next <= one;
            else
               state_next <= zero;
            end if;
      end case;
   end process;
   -- look-ahead output logic
   p1_next <= '1' when state_next=edge else
              '0';
   -- output
   p1 <= p1_reg;
end look_ahead_arch;
```

Note that in this particular example the clever statement assignment scheme can be implemented by using 2 bits (i.e., two D FFs) but the look-ahead output circuit scheme needs at least three D FFs (2 bits for the state register and 1 bit for the output buffer).

The VHDL code for the Mealy output–based design is shown in Listing 10.10. The code is based on the state diagram of Figure 10.12(b).

Listing 10.10 Edge detector with Mealy output

```
library ieee;
use ieee.std_logic_1164.all;
entity edge_detector2 is
   port(
      clk, reset: in std_logic;
      strobe: in std_logic;
      p2: out std_logic
```

```
    );
end edge_detector2;

architecture mealy_arch of edge_detector2 is
   type state_type is (zero, one);
   signal state_reg, state_next: state_type;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= zero;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- next-state logic
   process(state_reg,strobe)
   begin
      case state_reg is
         when zero=>
            if strobe= '1' then
               state_next <= one;
            else
               state_next <= zero;
            end if;
         when one =>
            if strobe= '1' then
               state_next <= one;
            else
               state_next <= zero;
            end if;
      end case;
   end process;
   -- Mealy output logic
   p2 <= '1' when (state_reg=zero) and (strobe='1') else
         '0';
end mealy_arch;
```

An alternative to deriving an edge detector is to treat it as a regular sequential circuit and design it in an ad hoc manner. One possible implementation is shown in Figure 10.19. The D FF in this circuit delays the `strobe` signal for one clock cycle and its output is the "previous value" of the `strobe` signal. The output of the and cell is asserted when the previous value of the `strobe` signal is '0' and the current value of the `strobe` signal is '1', which implies a positive transition edge of the `strobe` signal. The output signal is like a Mealy output since its value depends on the register's state and input signal. The VHDL code is shown in Listing 10.11. The entity declaration is identical to the Mealy machine–based edge detector in Listing 10.10.

Listing 10.11 Edge detector using direct implementation

```
architecture direct_arch of edge_detector2 is
   signal delay_reg: std_logic;
begin
```

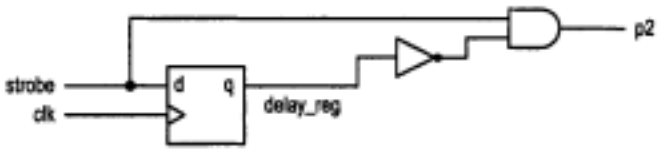


Figure 10.19 Direct implementation of an edge detector.

```
    -- delay register
    process(clk,reset)
    begin
       if (reset='1') then
          delay_reg <= '0';
       elsif (clk'event and clk='1') then
          delay_reg <= strobe;
       end if;
    end process;
    -- decoding logic
    p2 <= (not delay_reg) and strobe;
end direct_arch;
```

Although the code is compact for this particular case, this ad hoc approach can only be applied to simple designs. For example, if the requirement specifies a glitch-free Moore output, it is very difficult to derive the circuit this way. Actually, we can easily verify that this ad hoc design is actually Mealy machine–based design with binary state assignment (i.e., 0 to the `zero` state and 1 to the `one` state).

10.8.2 Arbiter

In a large system, some resources are shared by many subsystems. For example, several processors may share the same block of memory, and many peripheral devices may be connected to the same bus. An arbiter is a circuit that resolves any conflict and coordinates the access to the shared resource. This example considers an arbiter with two subsystems, as shown in Figure 10.20. The subsystems communicate with the arbiter by a pair of request and grant signals, which are labeled as `r(1)` and `g(1)` for subsystem 1, and as `r(0)` and `g(0)` for subsystem 0. When a subsystem needs the resources, it activates the request signal. The arbiter monitors use of the resources and the requests, and grants access to a subsystem by activating the corresponding grant signal. Once its grant signal is activated, a subsystem has permission to access the resources. After the task has been completed, the subsystem releases the resources and deactivates the request signal. Since an arbiter's decision is based partially on the events that occurred earlier (i.e., previous request and grant status), it needs internal states to record what happened in the past. An FSM can meet this requirement.

One critical issue in designing an arbiter is the handling of simultaneous requests. Our first design gives priority to subsystem 1. The state diagram of the FSM is shown in Figure 10.21(a). It consists of three states, `waitr`, `grant1` and `grant0`. The `waitr` state indicates that the resources is available and the arbiter is waiting for a request. The `grant1` and `grant0` states indicate that the resource is granted to subsystem 1 and subsystem 0 respectively. Initially, the arbiter is in the `waitr` state. If the `r(1)` input (the request from subsystem 1) is activated at the rising edge of the clock, it grants the resources to subsystem 1 by moving to the `grant1` state. The `g(1)` signal is asserted in this state to

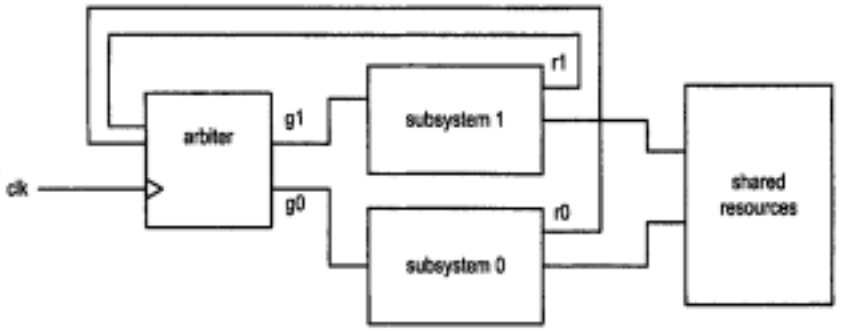


Figure 10.20 Block diagram of an arbiter.

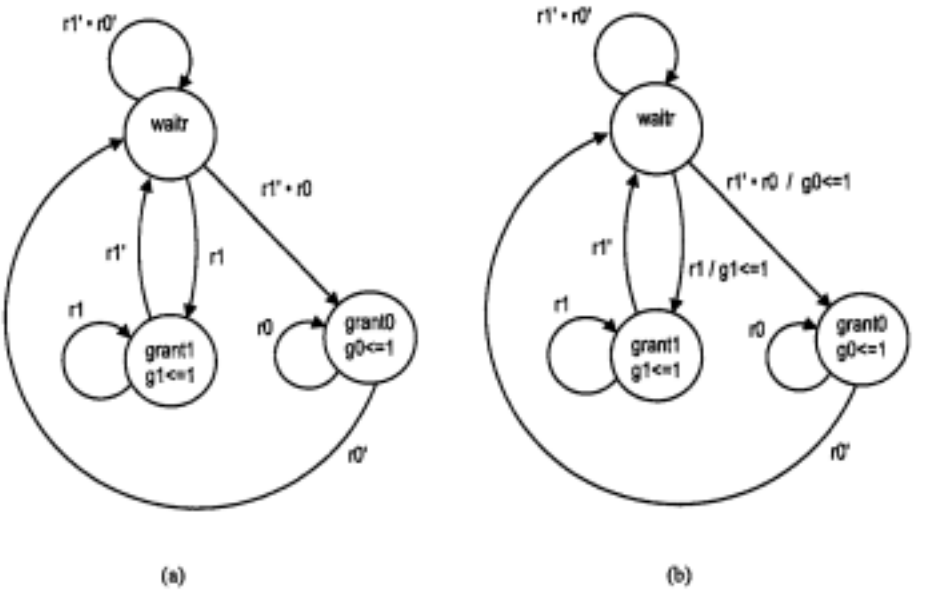


Figure 10.21 State diagrams of a fixed-priority two-request arbiter.

inform subsystem 1 of the availability of the resources. After subsystem 1 completes its usage, it signals the release of the resources by deactivating the `r(1)` signal. The arbiter returns to the `waitr` state accordingly.

In the `waitr` state, if `r(1)` is not activated and `r(0)` is activated at the rising edge, the arbiter grants the resources to subsystem 0 by moving to the `grant0` state and activates the `g(0)` signal. Subsystem 0 can then have the resources until it releases them. The VHDL code for this design is shown in Listing 10.12.

Listing 10.12 Arbiter with fixed priority

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity arbiter2 is
```

```
    port(
       clk: in std_logic;
       reset: in std_logic;
       r: in std_logic_vector(1 downto 0);
       g: out std_logic_vector(1 downto 0)
    );
end arbiter2;

architecture fixed_prio_arch of arbiter2 is
    type mc_state_type is (waitr, grant1, grant0);
    signal state_reg, state_next: mc_state_type;
begin
    -- state register
    process(clk,reset)
    begin
       if (reset='1') then
          state_reg <= waitr;
       elsif (clk'event and clk='1') then
          state_reg <= state_next;
       end if;
    end process;
    -- next-state and output logic
    process(state_reg,r)
    begin
       g <= "00";     -- default values
       case state_reg is
          when waitr =>
             if r(1)='1' then
                state_next <= grant1;
             elsif r(0)='1' then
                state_next <= grant0;
             else
                state_next <= waitr;
             end if;
          when grant1 =>
             if (r(1)='1') then
                state_next <= grant1;
             else
                state_next <= waitr;
             end if;
             g(1) <= '1';
          when grant0 =>
             if (r(0)='1') then
                state_next <= grant0;
             else
                state_next <= waitr;
             end if;
             g(0) <= '1';
       end case;
    end process;
end fixed_prio_arch;
```

If the subsystems are synchronized by the same clock, we can make g(1) and g(0) be Mealy output. The revised state diagram is shown in Figure 10.21(b). This allows the subsystems to obtain the resources one clock cycle earlier. In VHDL code, we modify the code under the waitr segment of the case statement to reflect the change. The revised portion becomes

```
when waitr =>
   if r(1)='1' then
      state_next <= grant1;
      g(1) <= '1';  -- newly added line
   elsif r(0)='1' then
      state_next <= grant0;
      g(0) <= '1';  -- newly added line
   else
      state_next <= waitr;
   end if;
```

The resource allocation of the previous design gives priority to subsystem 1. The preferential treatment may cause a problem if subsystem 1 requests the resources continuously. We can revise the state diagram to enforce a fairer arbitration policy. The new policy keeps track of which subsystem had the resources last time and gives preference to the other subsystem if the two request signals are activated simultaneously. The new design has to distinguish two kinds of wait conditions. The first condition is that the resources were last used by subsystem 1 so preference should be given to subsystem 0. The other condition is the reverse of the first. To accommodate the two conditions, we split the original waitr state into the waitr1 and waitr0 states, in which subsystem 1 and subsystem 0 will be given preferential treatment respectively. The revised state diagram is shown in Figure 10.22. Note that FSM moves from the grant0 state to the waitr1 state after subsystem 0 deactivates the request signal, and moves from the grant1 state to the waitr0 state after subsystem 1 deactivates the request signal. The revised VHDL code is shown in Listing 10.13.

Listing 10.13 Arbiter with alternating priority

```
architecture rotated_prio_arch of arbiter2 is
   type mc_state_type is (waitr1, waitr0, grant1, grant0);
   signal state_reg, state_next: mc_state_type;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= waitr1;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- next-state and output logic
   process(state_reg,r)
   begin
      g <= "00";    -- default values
      case state_reg is
         when waitr1 =>
            if r(1)='1' then
```

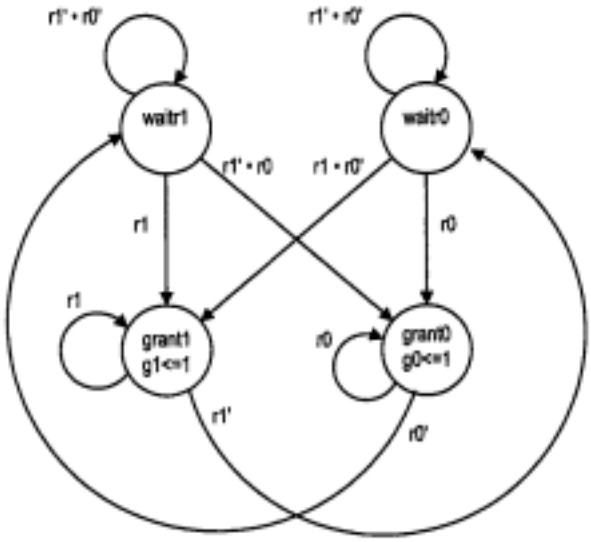


Figure 10.22 State diagram of a fair two-request arbiter.

```
            state_next <= grant1;
         elsif r(0)='1' then
            state_next <= grant0;
         else
            state_next <= waitr1;
         end if;
      when waitr0 =>
         if r(0)='1' then
            state_next <= grant0;
         elsif r(1)='1' then
            state_next <= grant1;
         else
            state_next <= waitr0;
         end if;
      when grant1 =>
         if (r(1)='1') then
            state_next <= grant1;
         else
            state_next <= waitr0;
         end if;
         g(1) <= '1';
      when grant0 =>
         if (r(0)='1') then
            state_next <= grant0;
         else
            state_next <= waitr1;
         end if;
         g(0) <= '1';
   end case;
end process;
```

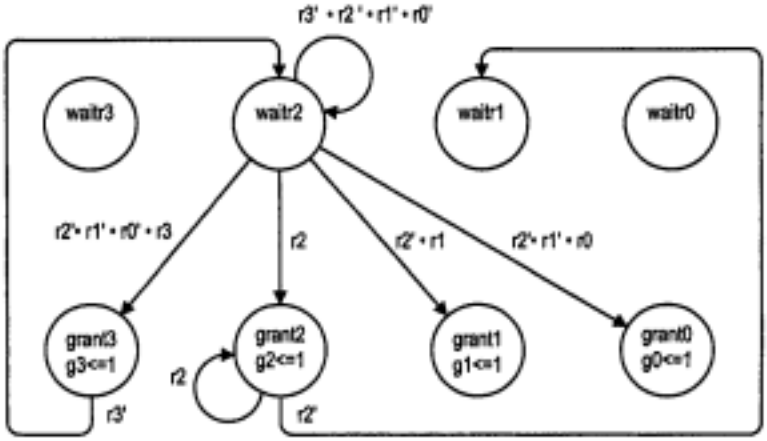


Figure 10.23 Partial state diagram of a four-request arbiter.

```
end rotated_prio_arch;
```

We can apply the same idea and expand the arbiter to handle more than two requests. The partial state diagram of an arbiter with four requests is shown in Figure 10.23. It assigns priority in round-robin fashion (i.e., subsystem 3, subsystem 2, subsystem 1, subsystem 0, then wrapping around), and the subsystem that obtains the resources will be assigned to the lowest priority next.

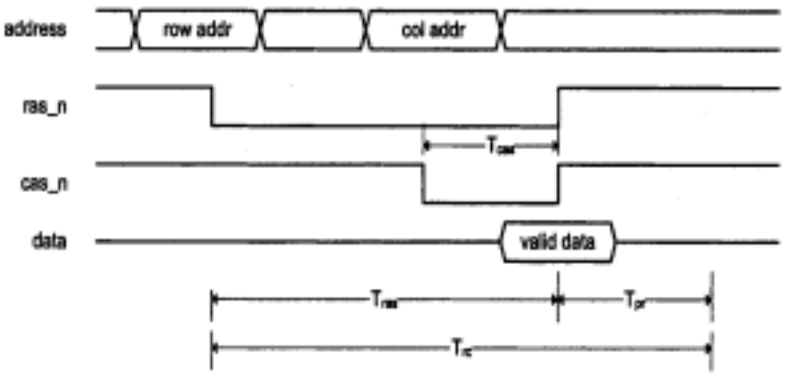
10.8.3 DRAM strobe generation circuit

Because of the large number of memory cells, the address signals of a dynamic RAM (DRAM) device are split into two parts, known as row and column. They are sent to the DRAM's address line in a time-multiplexed manner. Two control signals, `ras_n` (row address strobe) and `cas_n` (column address strobe), are strobe signals used to store the address into the DRAM's internal latches. The post-fix "_n" indicates active-low output, the convention used in most memory chips. The simplified timing diagram of a DRAM read cycle is shown in Figure 10.24(a). It is characterized by the following parameters:

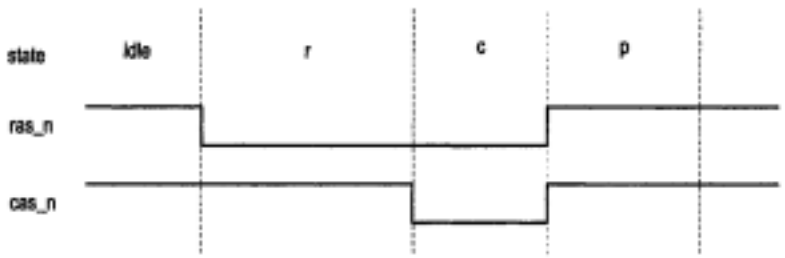
- T_{ras}: `ras` access time, the time required to obtain output data after `ras_n` is asserted (i.e., `ras_n` goes to '0').
- T_{cas}: `cas` access time, the time required to obtain output data after `cas_n` is asserted (i.e., `cas_n` goes to '0').
- T_{pr}: precharge time, the time to recharge the DRAM cell to restore the original value (since the cell's content is destroyed by the read operation).
- T_{rc}: read cycle, the minimum elapsed time between two read operations.

The operation of a conventional DRAM device is asynchronous and the device does not have a clock signal. The strobe signals have to be asserted in proper sequence and last long enough to provide the necessary time for decoding, multiplexing and memory cell recharging.

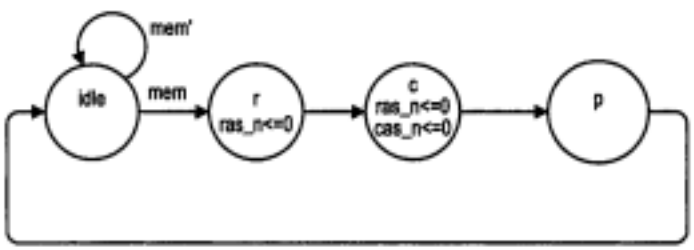
A memory controller is the interface between a DRAM device and a synchronous system. One function of the memory controller is to generate proper strobe signals. This example shows how to use an FSM to accomplish this task. A real memory controller should also



(a) Simplified timing of a DRAM read cycle



(b) State of the strobe signals



(c) State diagram of slow strobe generation

Figure 10.24 Read strobe generation FSM.

contain register and buffer to store address and data and should have extra control signals to coordinate the address bus and data bus operation. A complete memory controller example is discussed in Section 12.3.

Suppose that a DRAM has a read cycle of 120 ns, and T_{ras}, T_{cas} and T_{pr} are 85, 20 and 35 ns respectively. We want to design an FSM that generates the strobe signals, `ras_n` and `cas_n`, after the input command signal `mem` is asserted. The timing diagram of Figure 10.24(a) shows that the `ras_n` and `cas_n` signals have to be asserted and deasserted following a specific sequence:

- The `ras_n` signal is asserted first for at least 65 ns. The output pattern of the FSM is "01" in this interval.
- The `cas_n` signal is then asserted first for at least 20 ns. The output pattern of the FSM is "00" in this interval.
- The `ras_n` and `cas_n` signals are de-asserted first for at least 35 ns. The output pattern of the FSM is "11" in this interval.

Our first design uses a state for a pattern in the sequence and divides a read cycle into three states, namely the `r`, `c` and `p` states, as shown in Figure 10.24(b). The state diagram is shown in Figure 10.24(c). An extra `idle` state is added to accommodate the no-operation condition. We use a Moore machine since it has better control over the width of the intervals and can be modified to generate glitch-free output. In this design, each pattern lasts for one clock cycle. To satisfy the timing requirement for the three intervals, the clock period has to be at least 65 ns, and it takes 195 ns (i.e., 3*65 ns) to complete a read operation. The VHDL code is shown in Listing 10.14.

Listing 10.14 Slow DRAM read strobe generation FSM with regular output

```
library ieee;
use ieee.std_logic_1164.all;
entity dram_strobe is
   port(
      clk, reset: in std_logic;
      mem: in std_logic;
      cas_n, ras_n: out std_logic
   );
end dram_strobe;

architecture fsm_slow_clk_arch of dram_strobe is
   type fsm_state_type is (idle, r, c, p);
   signal state_reg, state_next: fsm_state_type;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- next-state logic
   process(state_reg,mem)
   begin
      case state_reg is
```

```
            when idle =>
               if mem='1' then
                  state_next <= r;
               else
                  state_next <= idle;
               end if;
            when r =>
               state_next <=c;
            when c =>
               state_next <=p;
            when p =>
               state_next <=idle;
         end case;
      end process;
      -- output logic
      process(state_reg)
      begin
         ras_n <= '1';
         cas_n <= '1';
         case state_reg is
            when idle =>
            when r =>
               ras_n <= '0';
            when c =>
               ras_n <= '0';
               cas_n <= '0';
            when p =>
         end case;
      end process;
   end fsm_slow_clk_arch;
```

Since the strobe signals are level-sensitive, we have to ensure that these signals are glitch-free. We can revise the previous code to add the look-ahead output buffer, as shown in Listing 10.15.

Listing 10.15 Slow DRAM read strobe generation FSM with a look-ahead output buffer

```
architecture fsm_slow_clk_buf_arch of dram_strobe is
   type fsm_state_type is (idle,r,c,p);
   signal state_reg, state_next: fsm_state_type;
   signal ras_n_reg, cas_n_reg: std_logic;
   signal ras_n_next, cas_n_next: std_logic;
begin
   -- state register and output buffer
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
         ras_n_reg <= '1';
         cas_n_reg <= '1';
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         ras_n_reg <= ras_n_next;
         cas_n_reg <= cas_n_next;
```

```
      end if;
   end process;
   -- next-state
   process(state_reg,mem)
   begin
      case state_reg is
         when idle =>
            if mem='1' then
               state_next <= r;
            else
               state_next <= idle;
            end if;
         when r =>
            state_next <=c;
         when c =>
            state_next <=p;
         when p =>
            state_next <=idle;
      end case;
   end process;
   -- look-ahead output logic
   process(state_next)
   begin
      ras_n_next <= '1';
      cas_n_next <= '1';
      case state_next is
         when idle =>
         when r =>
            ras_n_next <= '0';
         when c =>
            ras_n_next <= '0';
            cas_n_next <= '0';
         when p =>
       end case;
   end process;
   --output
   ras_n <= ras_n_reg;
   cas_n <= cas_n_reg;
end fsm_slow_clk_buf_arch;
```

To improve the performance of the memory operation, we can use a smaller clock period to accommodate the differences between the three intervals. For example, we can use a clock with a period of 20 ns and use multiple states for each output pattern. The three output patterns need 4 (i.e., $\lceil \frac{85}{20} \rceil$) states, 1 (i.e., $\lceil \frac{20}{20} \rceil$) state and 2 (i.e., $\lceil \frac{35}{20} \rceil$) states respectively. The revised state diagram is shown in Figure 10.25, in which the original `r` state is split into `r1`, `r2`, `r3` and `r4` states, and the original `p` state is split into `p1` and `p2` states. It now takes seven states, which amounts to 140 ns (i.e., 7*20 ns), to complete a read operation. We can further improve the performance by using a 5-ns clock signal (assuming that the next-state logic and register are fast enough to support it). The three output patterns need 13, 4 and 7 states respectively, and a read operation can be done in 120 ns, the fastest operation speed of this DRAM chip. While still simple, the state diagram becomes tedious to draw. RT methodology (to be discussed in Chapters 11 and 12) can combine counters with FSM and

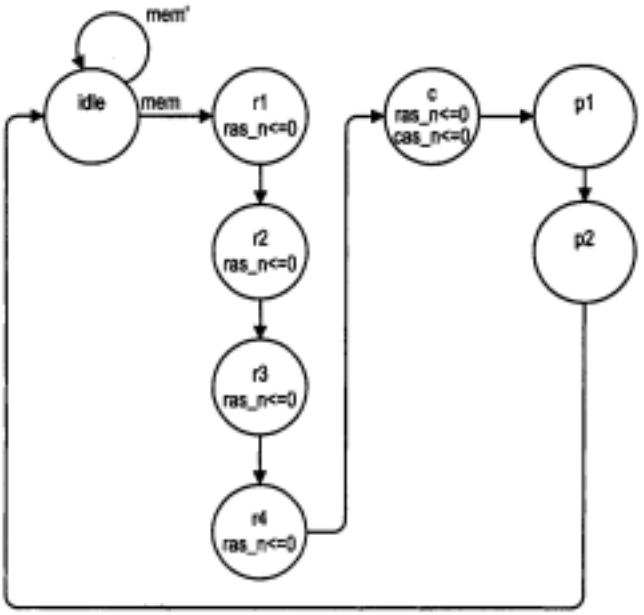


Figure 10.25 State diagram of fast read strobe generation.

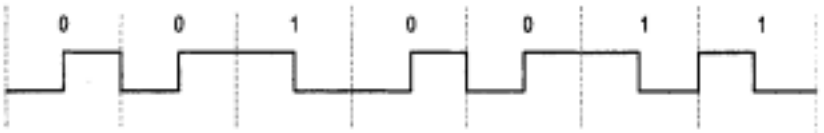


Figure 10.26 Sample waveform of Manchester encoding.

provide a better alternative to implement this type of circuit. In a more realistic scenario, the strobe generation circuit should be part of a large system, and it cannot use an independent clock. The design has to accommodate the clock rate of the main system and adjust the number of states in each pattern accordingly.

10.8.4 Manchester encoding circuit

Manchester code is a coding scheme used to represent a bit in a data stream. A '0' value of a bit is represented as a 0-to-1 transition, in which the lead half is '0' and the remaining half is '1'. Similarly, a '1' value of a bit is represented as a 1-to-0 transition, in which the lead half is '1' and the remaining half is '0'. A sample data stream in Manchester code is shown in Figure 10.26. The Manchester code is frequently used in a serial communication line. Since there is a transition in each bit, the receiving system can use the transitions to recover the clock information.

The Manchester encoder transforms a regular data stream into a Manchester-coded data stream. Because an encoded bit includes a sequence of "01" or "10", two clock cycles are needed. Thus, the maximal data rate is only half of the clock rate. There are two input signals. The d signal is the input data stream, and the v signal indicates whether the d

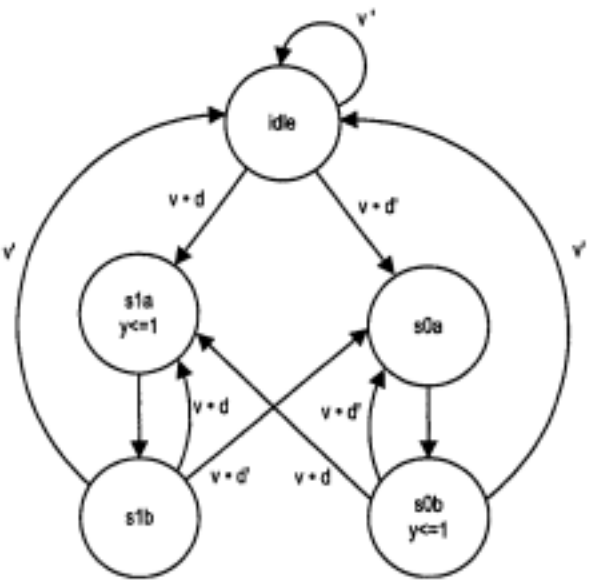


Figure 10.27 State diagram of a Manchester encoder.

signal is valid (i.e., whether there is data to transmit). The d signal should be converted to Manchester code if the v signal is asserted. The output remains '0' otherwise. The state diagram is shown in Figure 10.27. While v is asserted, the FSM starts the encoding process. If d is '0', it travels through the s0a and s0b states. If d is '1', the FSM travels through the s1a and s1b states. Once the FSM reaches the s1b or s0b state, it checks the v signal. If the v signal is still asserted, the FSM skips the idle state and continuously encodes the next input data. The Moore output is used because we have to generate two equal intervals for each bit. The VHDL code is shown in Listing 10.16.

Listing 10.16 Manchester encoder with regular output

```
library ieee;
use ieee.std_logic_1164.all;
entity manchester_encoder is
   port(
      clk, reset: in std_logic;
      v,d: in std_logic;
      y: out std_logic
   );
end manchester_encoder;

architecture moore_arch of manchester_encoder is
   type state_type is (idle, s0a, s0b, s1a, s1b);
   signal state_reg, state_next: state_type;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
```

```
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- next-state logic
   process(state_reg,v,d)
   begin
      case state_reg is
         when idle=>
            if v= '0' then
               state_next <= idle;
            else
               if d= '0' then
                  state_next <= s0a;
               else
                  state_next <= s1a;
               end if;
            end if;
         when s0a =>
            state_next <= s0b;
         when s1a =>
            state_next <= s1b;
         when s0b =>
            if v= '0' then
               state_next <= idle;
            else
               if d= '0' then
                  state_next <= s0a;
               else
                  state_next <= s1a;
               end if;
            end if;
         when s1b =>
            if v= '0' then
               state_next <= idle;
            else
               if d= '0' then
                  state_next <= s0a;
               else
                  state_next <= s1a;
               end if;
            end if;
      end case;
   end process;
   -- Moore output logic
   y <= '1' when state_reg=s1a or state_reg=s0b else
        '0';
end moore_arch;
```

Because the transition edge of the Manchester code is frequently used by the receiver to recover the clock signal, we should make the output data stream glitch-free. This can be achieved by using the look-ahead output buffer. The revised VHDL code is shown in Listing 10.17.

Listing 10.17 Manchester encoder with a look-ahead output buffer

```
architecture out_buf_arch of manchester_encoder is
    type state_type is (idle, s0a, s0b, s1a, s1b);
    signal state_reg, state_next: state_type;
    signal y_next, y_buf_reg: std_logic;
begin
   -- state register and output buffer
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
         y_buf_reg <= '0';
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         y_buf_reg <= y_next;
      end if;
   end process;
   -- next-state logic
   process(state_reg,v,d)
   begin
      case state_reg is
         when idle=>
            if v='0' then
               state_next <= idle;
            else
               if d= '0' then
                  state_next <= s0a;
               else
                  state_next <= s1a;
               end if;
            end if;
         when s0a =>
            state_next <= s0b;
         when s1a =>
            state_next <= s1b;
         when s0b =>
            if v='0' then
               state_next <= idle;
            else
               if d='0' then
                  state_next <= s0a;
               else
                  state_next <= s1a;
               end if;
            end if;
         when s1b =>
            if v= '0' then
               state_next <= idle;
            else
               if d= '0' then
                  state_next <= s0a;
               else
                  state_next <= s1a;
```

Figure 10.28 State diagram of a free-running mod-16 counter.

```
                end if;
              end if;
      end case;
    end process;
    -- look-ahead output logic
    y_next <= '1' when state_next=s1a or state_next=s0b else
              '0';
    -- output
    y <= y_buf_reg;
  end out_buf_arch;
```

10.8.5 FSM-based binary counter

As discussed in Section 8.2.3, our classification of regular sequential circuits and FSMs (random sequential circuits) is for "design practicality." In theory, all sequential circuits with finite memory can be modeled by FSMs and derived accordingly. This example demonstrates the derivation of an FSM-based binary counter. Let us first consider a free-running 4-bit counter, similar to the one in Section 8.5.4. A 4-bit counter has to traverse 16 (2^4) distinctive states, and thus the FSM should have 16 states. The state diagram is shown in Figure 10.28. Note the regular pattern of transitions.

The FSM can be modified to add more features to this counter and gradually transform it to the featured binary counter of Section 8.5.4. To avoid clutter in the diagram, we use a single generic `si` state (the ith state of the counter) to illustrate the required modifications. The process is shown in Figure 10.29. We first add the synchronous clear signal, `syn_clr`, which clears the counter to 0, as in Figure 10.29(b). In the FSM, it corresponds to forcing the FSM to return to the initial state, `s0`. Note that the logic expressions give priority to the synchronous clear operation. The next step is to add the load operation. This actually involves five input bits, which include the 1-bit control signal, `load`, and the 4-bit data signal, `d`. The `d` signal is the value to be loaded into the counter and it is composed of four individual bits, `d3`, `d2`, `d1` and `d0`. The load operation changes the content of the register according to the value of `d`. In terms of FSM operation, 16 transitions are needed to express the possible 16 next states. The revised diagram is shown in Figure 10.29(c). Finally, we can add the enable signal, `en`, which can suspend the counting. In terms of FSM operation, it corresponds to staying in the same state. The final diagram is shown in Figure 10.29(d). Note that the logic expressions of the transition arches set the priority of the control signals in the order `syn_clr`, `load` and `en`. Although this design process is theoretically doable, it is very tedious. The diagram will become extremely involved for a larger, say, a 16- or 32-bit, counter. This example shows the distinction between a regular sequential circuit and a random sequential circuit. In Section 12.2, we present a more comprehensive comparison

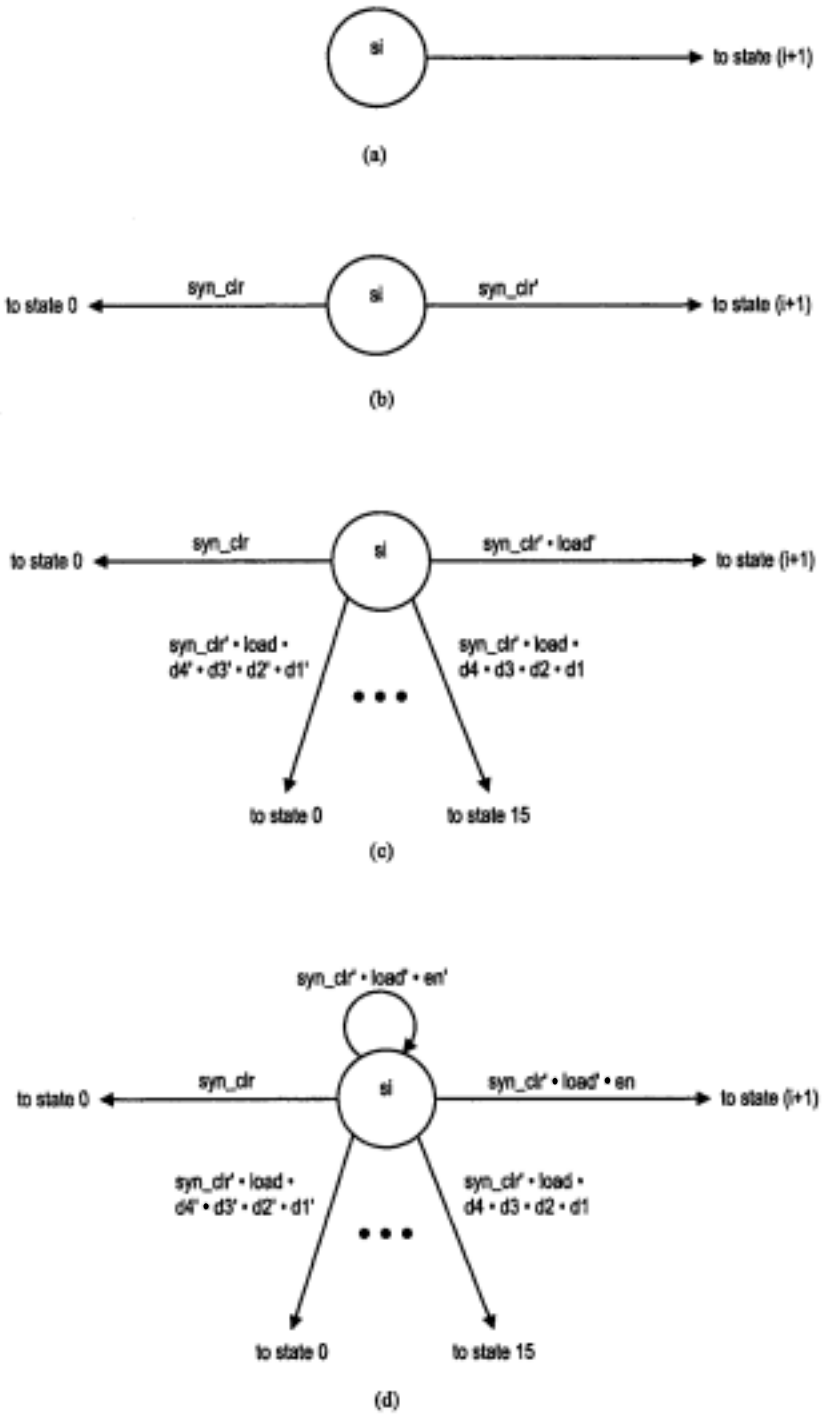


Figure 10.29 State diagram development of a featured mod-16 counter.

between regular sequential circuits, random sequential circuits and combined sequential circuits, which consist of both regular and random sequential circuits.

10.9 BIBLIOGRAPHIC NOTES

FSM is a standard topic in an introductory digital systems course. Typical digital systems texts, such as *Digital Design Principles and Practices* by J. F. Wakerly and *Contemporary Logic Design* by R. H. Katz, provide comprehensive coverage of the derivation of state diagrams and ASM charts as well as a procedure to realize them manually in hardware. They also show the techniques for state reduction. On the other hand, obtaining optimal state assignment for an FSM is a much more difficult problem. For example, it takes two theoretical texts, *Synthesis of Finite State Machines: Logic Optimization* by T. Villa et al. and *Synthesis of Finite State Machines: Functional Optimization* by T. Kam, to discuss the optimization algorithms.

Problems

10.1 For the "burst" read operation, the memory controller FSM of Section 10.2.1 implicitly specifies that the main system has to activate the rw and mem signals in the first clock cycle and then activate the $burst$ signal in the next clock cycle. We wish to simplify the timing requirement for the main system so that it only needs to issue the command in the first clock cycle (i.e., activates the $burst$ signal at the same time as the rw and mem signals).

(a) Revise the state diagram to achieve this goal.

(b) Convert the state diagram to an ASM chart.

(c) Derive VHDL code according to the ASM chart.

10.2 The memory controller FSM of Section 10.2.1 has to return to the `idle` state for each memory operation. To achieve better performance, revise the design so that the controller can support "back-to-back" operations; i.e., the FSM can initiate a new memory operation after completing the current operation without first returning to the `idle` state.

(a) Derive the revised state diagram.

(b) Convert the state diagram to an ASM chart.

(c) Derive VHDL code according to the ASM chart.

10.3 Revise the edge detection circuit of Section 10.4.1 to detect both 0-to-1 and 1-to-0 transitions; i.e., the circuit will generate a short pulse whenever the `strobe` signal changes state. Use a Moore machine with a minimal number of states to realize this circuit.

(a) Derive the state diagram.

(b) Convert the state diagram to an ASM chart.

(c) Derive VHDL code according to the ASM chart.

10.4 Repeat Problem 10.3, but use a Mealy machine to realize the circuit. The Mealy machine needs only two states.

10.5 In digital communication, a special synchronization pattern, known as a *preamble*, is used to indicate the beginning of a packet. For example, the Ethernet II preamble includes eight repeating octets of "10101010". We wish to design an FSM that generates the "10101010" pattern. The circuit has an input signal, `start`, and an output, `data_out`. When `start` is '1', the "10101010" will be generated in the next eight clock cycles.

(a) Derive the state diagram.
(b) Convert the state diagram to an ASM chart.
(c) Derive VHDL code according to the ASM chart.
(d) Use a clever state assignment to obtain glitch-free output signal. Derive the revised VHDL code.
(e) Use a look-ahead output buffer for the output signal. Derive the revised VHDL code

10.6 Now we wish to design an FSM to detect the "10101010" pattern in the receiving end. The circuit has an input signal, `data_in`, and an output signal, `match`. The `match` signal will be asserted as '1' for one clock period when the input pattern "10101010" is detected.

(a) Derive the state diagram.
(b) Convert the state diagram to an ASM chart.
(c) Derive VHDL code according to the ASM chart.

10.7 Can we apply look-ahead output buffer for Mealy output? Explain.

10.8 The first arbiter of Section 10.8.2 has to return to the `wait` state before it can grant the resources to another request. Revise the design so that the arbiter can move from one grant state to another grant state when there is an active request.

(a) Derive the state diagram.
(b) Convert the state diagram to an ASM chart.
(c) Derive VHDL code according to the ASM chart.

10.9 Consider the fair arbiter of Section 10.8.2. Its design is based on the assumption that a subsystem will release the resources voluntarily. An alternative is to use a timeout signal to prevent a subsystem from exhausting the resource. When the timeout signal is asserted, the arbiter will return to a wait state regardless of whether the corresponding request signal is still active.

(a) Derive the revised state diagram.
(b) Convert the state diagram to an ASM chart.
(c) Derive VHDL code according to the ASM chart.

10.10 Redesign the DRAM strobe generation circuit of Section 10.8.3 for a system with a different clock period. Derive the state diagram and determine the required time to complete a read cycle for the following clock periods:

(a) A clock period of 10 ns.
(b) A clock period of 40 ns.
(c) A clock period of 200 ns.

10.11 A Manchester decoder transforms a Manchester-coded data stream back to a regular binary data stream. There are two output signals. The `data` signal is the recovered data bit, which can be '0' or '1'. The `valid` signal indicates whether a transition occurs. The `valid` signal is used to distinguish whether the '0' of the `data` signal is due to the 0-to-1 transition or inactivity of the data stream.

(a) Derive the state diagram.
(b) Convert the state diagram to an ASM chart.
(c) Derive VHDL code according to the ASM chart.

10.12 Non-return to-zero invert-to ones (NRZI) code is another code used in serial transmission. The output of an NRZI encoder is '0' if the current input value is different from the previous value and is '1' otherwise. Design an NRZI encoder using an FSM and derive the VHDL code accordingly.

10.13 Repeat Problem 10.12, but design an NRZI decoder, which converts a NRZI-coded stream back to a regular binary stream.

10.14 Derive the VHDL code for the FSM-based free-running mod-16 counter of Section 10.8.5.

(a) Synthesize the code using an ASIC technology. Compare the area and performance (in term of maximal clock rate) of the code in Section 10.8.5.

(b) Synthesize the code using an FPGA technology. Compare the area and performance of the code in Section 10.8.5.

10.15 Repeat Problem 10.14 for the featured mod-16 counter of Section 10.8.5.