```
            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
```

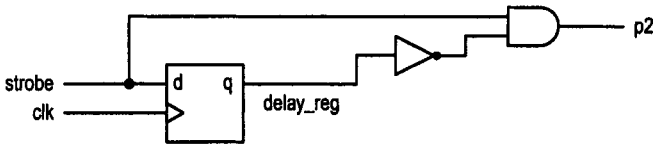


Figure 10.19 Direct implementation of an edge detector.

```
    -- 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

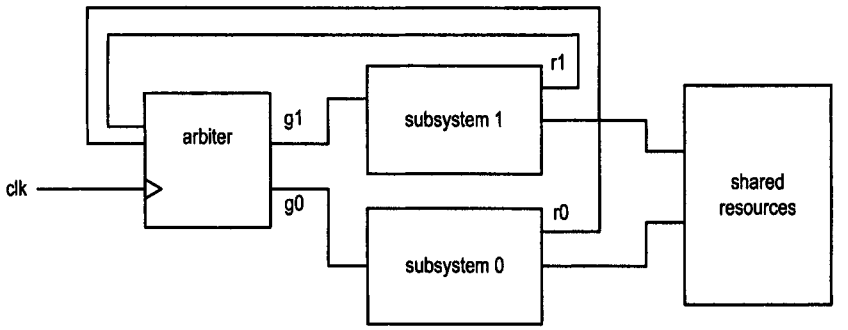


Figure 10.20 Block diagram of an arbiter.

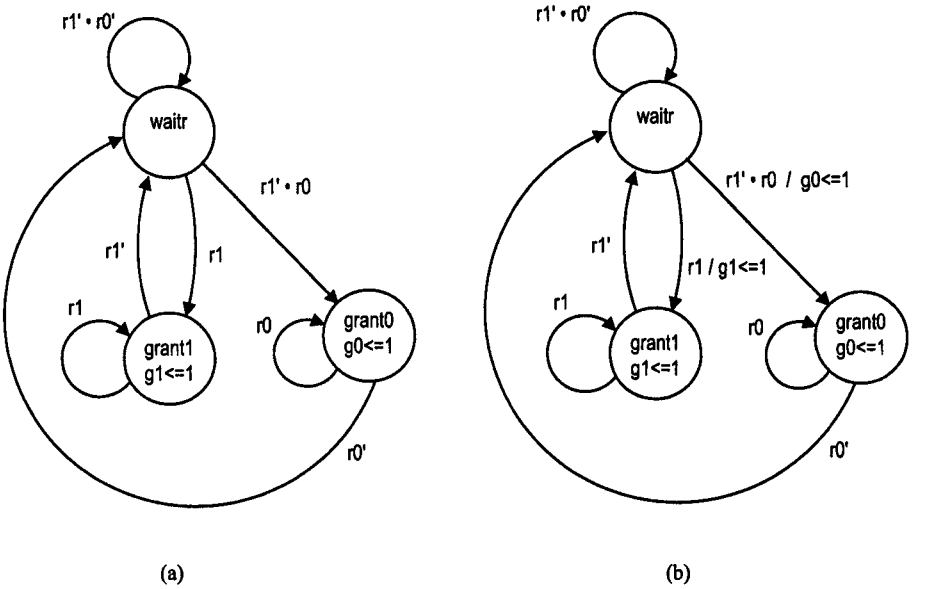


Figure 10.21 State diagrams of a fixed-priority two-request arbiter.

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
```

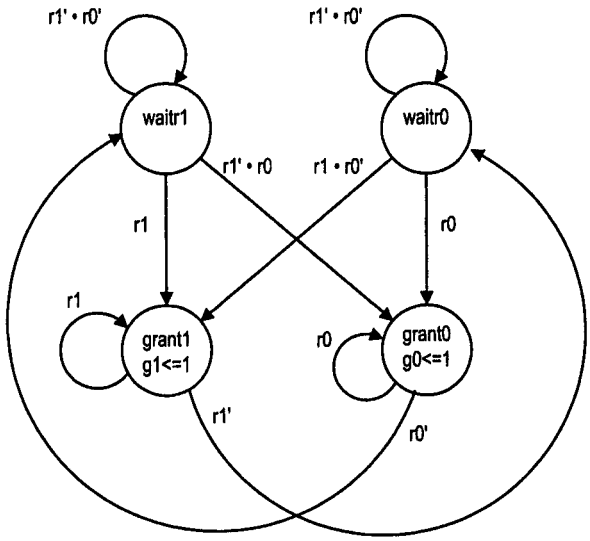


Figure 10.22 State diagram of a fair two-request arbiter.

```
            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;
```

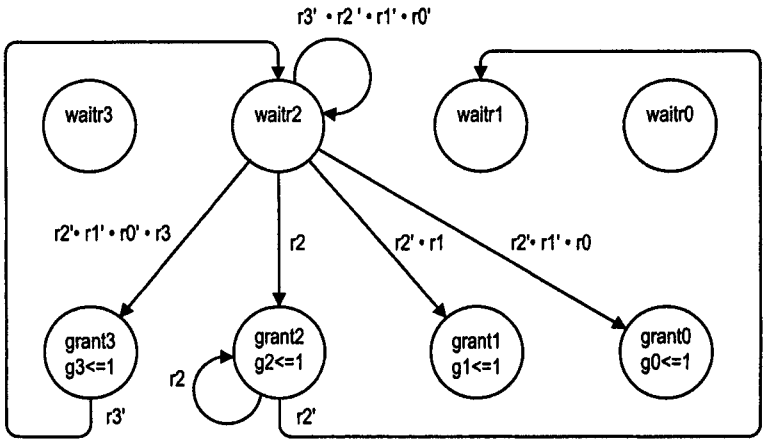


Figure 10.23 Partial state diagram of a four-request arbiter.

```
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.

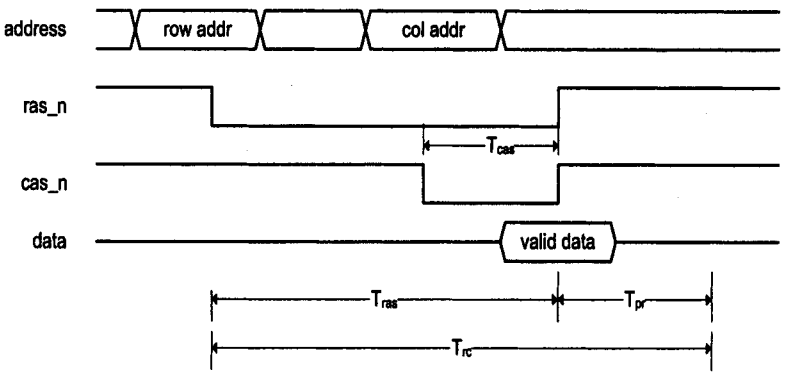
10.8.3 DRAM strobe generation circuit

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:

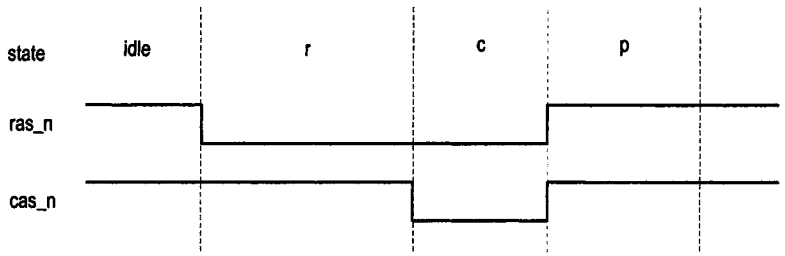
- T_{ras}: `ras` access time, the time required to obtain output data after `ras_n` is asserted (i.e., `ras_n` goes to '0').
- T_{cas}: `cas` access time, the time required to obtain output data after `cas_n` is asserted (i.e., `cas_n` goes to '0').
- T_{pr}: precharge time, the time to recharge the 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.

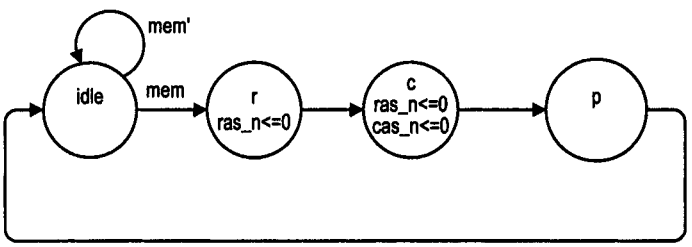
A memory controller is the interface between a DRAM device and a synchronous system. One function of the memory controller is to generate proper strobe signals. This example shows how to use an FSM to accomplish this task. A real memory controller should also



(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., $\lceil \frac{65}{20} \rceil$) 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

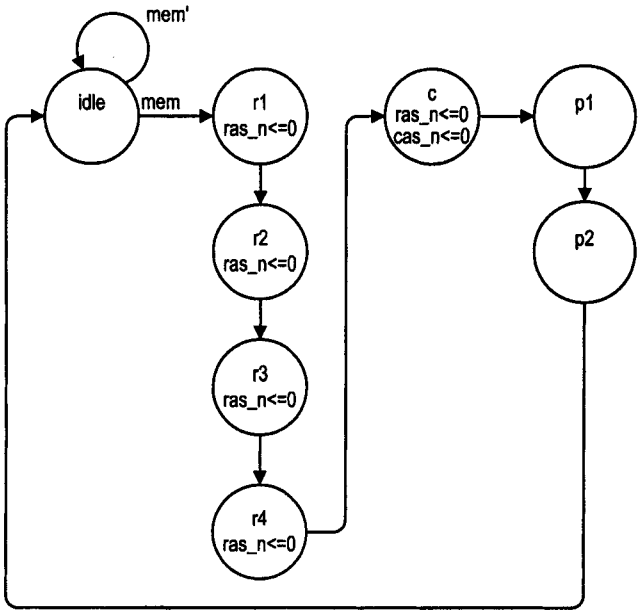


Figure 10.25 State diagram of fast read strobe generation.

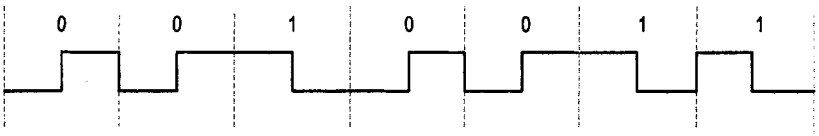


Figure 10.26 Sample waveform of Manchester encoding.

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

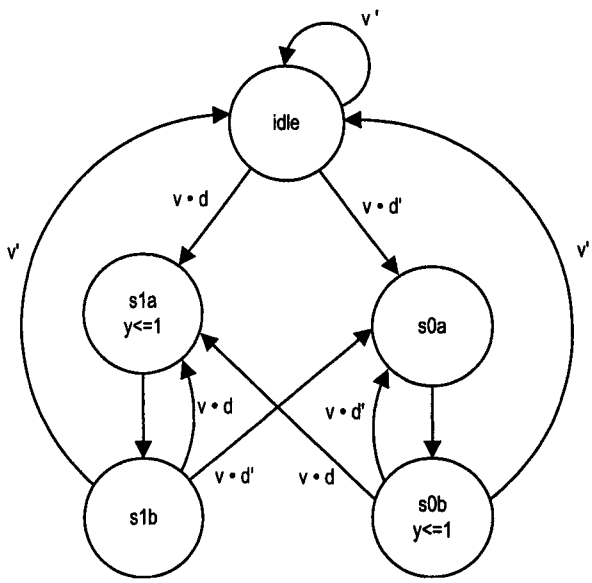


Figure 10.27 State diagram of a Manchester encoder.

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;
```

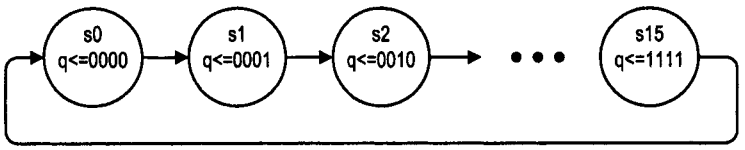


Figure 10.28 State diagram of a free-running mod-16 counter.

```
                end if;
            end if;
55      end case;
    end process;
    -- look-ahead output logic
    y_next <= '1' when state_next=s1a or state_next=s0b else
              '0';
60  -- 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

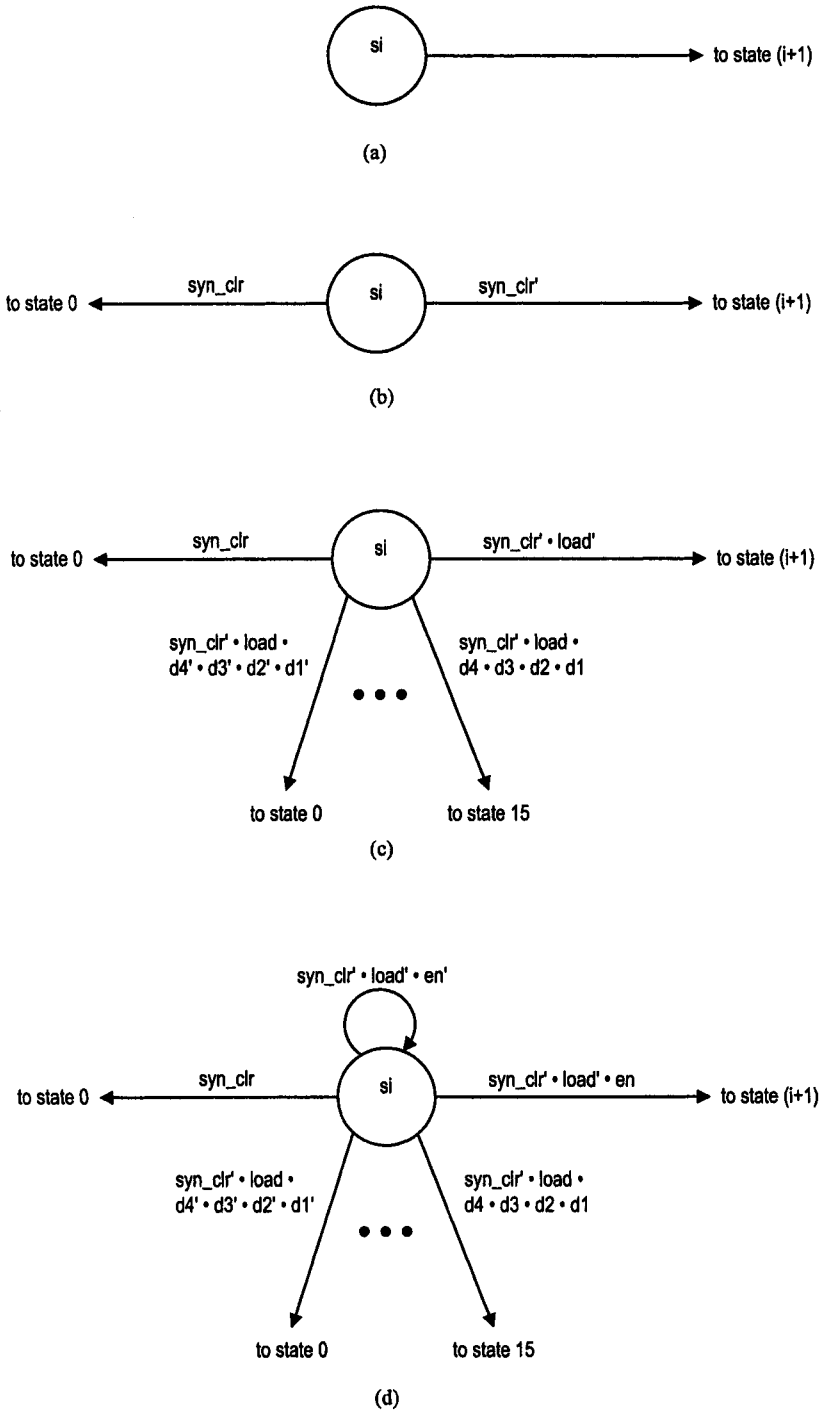


Figure 10.29 State diagram development of a featured mod-16 counter.

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.