CHAPTER 5

FSM

5.1 INTRODUCTION

An FSM (finite state machine) is used to model a system that transits among a finite number of internal states. The transitions depend on the current state and external input. Unlike a regular sequential circuit, the state transitions of an FSM do not exhibit a simple, repetitive pattern. Its next-state logic is usually constructed from scratch and is sometimes known as "random" logic. This is different from the next-state logic of a regular sequential circuit, which is composed mostly of "structured" components, such as incrementors and shifters.

In this chapter, we provide an overview of the basic characteristics and representation of FSMs and discuss the derivation of HDL codes. In practice, the main application of an FSM is to act as the controller of a large digital system, which examines the external commands and status and activates proper control signals to control operation of a *data path*, which is usually composed of regular sequential components. This is known as an FSMD (finite state machine with data path) and is discussed in Chapter 6.

5.1.1 Mealy and Moore outputs

The basic block diagram of an FSM is the same as that of a regular sequential circuit and is repeated in Figure 5.1. It consists of a state register, next-state logic, and output logic. An FSM is known as a *Moore machine* if the output is only a function of state, and is known as a *Mealy machine* if the output is a function of state and external input. Both types of output may exist in a complex FSM, and we simply refer to it as containing a Moore output and

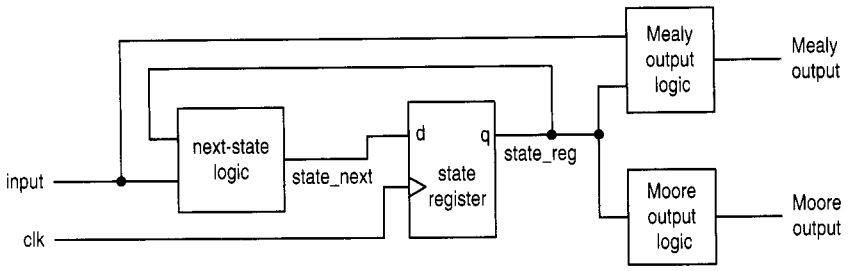


Figure 5.1 Block diagram of a synchronous FSM.

Mealy output. The Moore and Mealy outputs are similar but not identical. Understanding their subtle differences is the key for a controller design. The example in Section 5.3.1 illustrates the behaviors and constructions of the two types of outputs.

5.1.2 FSM representation

An FSM is usually specified by an abstract *state diagram* or *ASM chart* (algorithmic state machine chart), both capturing the FSM's input, output, states, and transitions in a graphical representation. The two representations provide the same information. The FSM representation is more compact and better for simple applications. The ASM chart representation is somewhat like a flowchart and is more descriptive for applications with complex transition conditions and actions.

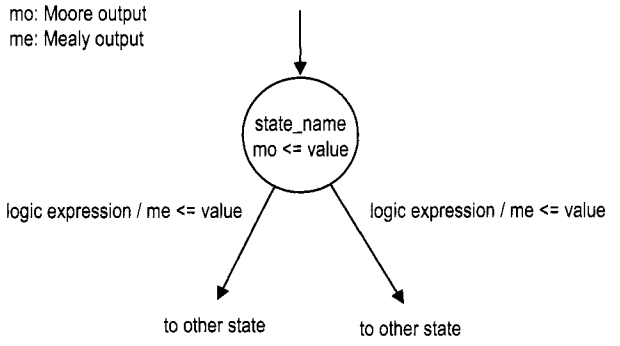
State diagram A state diagram is composed of *nodes*, which represent states and are drawn as circles, and annotated *transitional arcs*. A single node and its transition arcs are shown in Figure 5.2(a). A logic expression expressed in terms of input signals is associated with each transition arc and represents a specific condition. The arc is taken when the corresponding expression is evaluated `true`.

The Moore output values are placed inside the circle since they depend only on the current state. The Mealy output values are associated with the conditions of transition arcs since they depend on the current state and external input. To reduce clutter in the diagram, only asserted output values are listed. The output signal takes the default (i.e., unasserted) value otherwise.

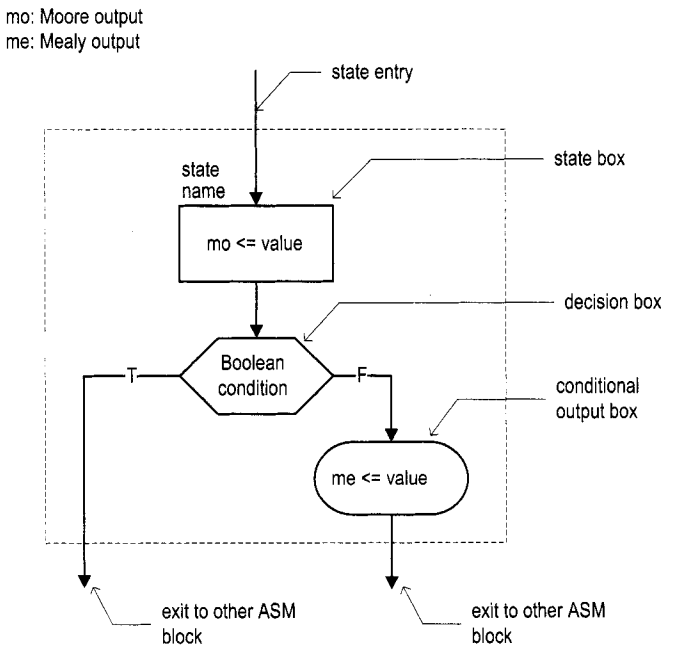
A representative state diagram is shown in Figure 5.3(a). The FSM has four states, two external input signals (i.e., `a` and `b`), one Moore output signal (i.e., `y1`), and one Mealy output signal (i.e., `y0`). The `y1` signal is asserted when the FSM is in the `s2` or `s3` state. The `y0` signal is asserted when the FSM is in the `s0` state and the `a` and `b` signals are "11".

ASM chart An ASM chart is composed 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 representative ASM block is shown in Figure 5.2(b).

A state box represents a state in an FSM, and the asserted Moore output values are listed inside the box. Note that it has only one exit path. A decision box tests the input condition and determines which exit path to take. It has two exit paths, labeled `T` and `F`, which correspond to the `true` and `false` values of the condition. A conditional output box lists asserted Mealy output values and is usually placed after a decision box. It indicates that the listed output signal can be activated only when the corresponding condition in the decision box is met.

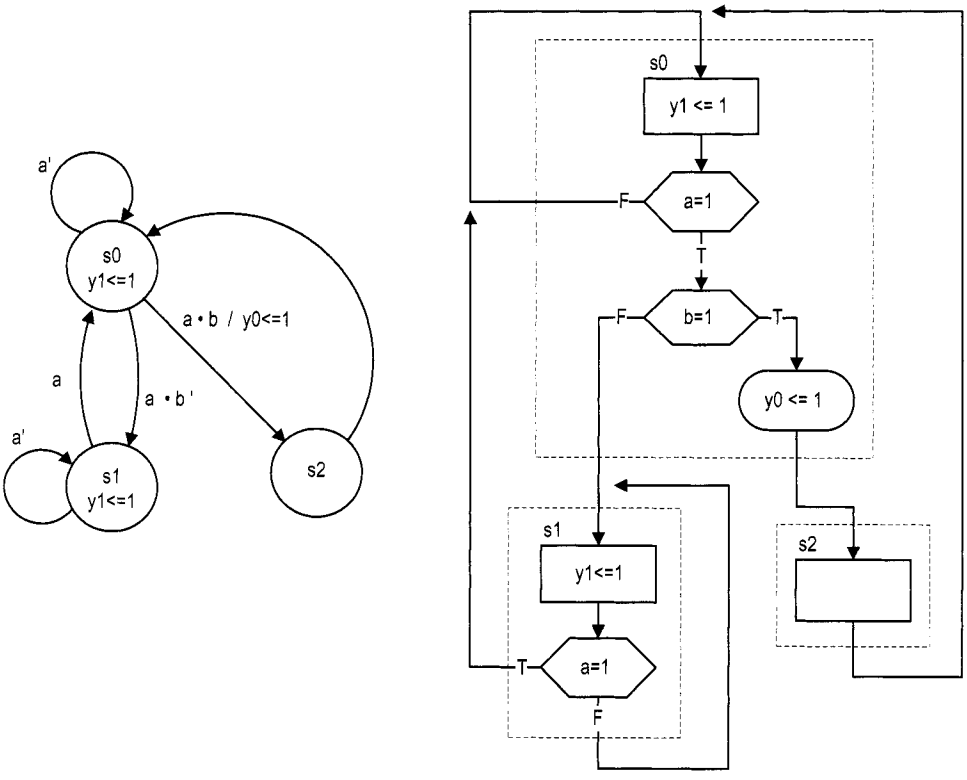


(a) Node



(b) ASM block

Figure 5.2 Symbol of a state.



(a) State diagram

(b) ASM chart

Figure 5.3 Example of an FSM.

A state diagram can easily be converted to an ASM chart, and vice versa. The corresponding ASM chart of the previous FSM state diagram is shown in Figure 5.3(b).

5.2 FSM CODE DEVELOPMENT

The procedure of developing code for an FSM is similar to that of a regular sequential circuit. We first separate the state register and then derive the code for the combinational next-state logic and output logic. The main difference is the next-state logic. For an FSM, the code for the next-state logic follows the flow of a state diagram or ASM chart.

For clarity and flexibility, we use the VHDL's *enumerated data type* to represent the FSM's states. The enumerated data type can best be explained by an example. Consider the FSM of Section 5.1.2, which has three states: s0, s1, and s2. We can introduce a user-defined enumerated data type as follows:

```
type eg_state_type is (s0, s1, s2);
```

The data type simply lists (i.e., *enumerates*) all symbolic values. Once the data type is defined, it can be used for the signals, as in

```
signal state_reg, state_next: eg_state_type;
```

During synthesis, software automatically maps the values in an enumerated data type to binary representations, a process known as *state assignment*. Although there is a mechanism to perform this manually, it is rarely needed.

The complete code of the FSM is shown in Listing 5.1. It consists of segments for the state register, next-state logic, Moore output logic, and Mealy output logic.

Listing 5.1 FSM example

```
library ieee;
use ieee.std_logic_1164.all;
entity fsm_eg is
   port(
      clk, reset: in std_logic;
      a, b: in std_logic;
      y0, y1: out std_logic
   );
end fsm_eg;

architecture mult_seg_arch of fsm_eg is
   type eg_state_type is (s0, s1, s2);
   signal state_reg, state_next: eg_state_type;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= s0;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- next-state logic
   process(state_reg,a,b)
```

```
   begin
      case state_reg is
         when s0 =>
            if a='1' then
               if b='1' then
                  state_next <= s2;
               else
                  state_next <= s1;
               end if;
            else
               state_next <= s0;
            end if;
         when s1 =>
            if (a='1') then
               state_next <= s0;
            else
               state_next <= s1;
            end if;
         when s2 =>
            state_next <= s0;
      end case;
   end process;
   -- Moore output logic
   process(state_reg)
   begin
      case state_reg is
         when s0|s2 =>
            y1 <= '0';
         when s1 =>
            y1 <= '1';
      end case;
   end process;
   -- Mealy output logic
   process(state_reg,a,b)
   begin
      case state_reg is
         when s0 =>
            if (a='1') and (b='1') then
              y0 <= '1';
            else
              y0 <= '0';
            end if;
         when s1 | s2 =>
            y0 <= '0';
      end case;
   end process;
end mult_seg_arch;
```

The key part is the next-state logic. It uses a case statement with the `state_reg` signal as the selection expression. The next state (i.e., `state_next` signal) is determined by the current state (i.e., `state_reg`) and external input. The code for each state basically follows the activities inside each ASM block of Figure 5.3(b).

An alternative code is to merge next-state logic and output logic into a single combinational block, as shown in Listing 5.2.

Listing 5.2 FSM with merged combinational logic

```
architecture two_seg_arch of fsm_eg is
   type eg_state_type is (s0, s1, s2);
   signal state_reg, state_next: eg_state_type;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= s0;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- next-state/output logic
   process(state_reg,a,b)
   begin
      state_next <= state_reg;  -- default back to same state
      y0 <= '0';   -- default 0
      y1 <= '0';   -- default 0
      case state_reg is
         when s0 =>
            if a='1' then
               if b='1' then
                  state_next <= s2;
                  y0 <= '1';
               else
                  state_next <= s1;
               end if;
            -- no else branch
            end if;
         when s1 =>
            y1 <= '1';
            if (a='1') then
               state_next <= s0;
            -- no else branch
            end if;
         when s2 =>
            state_next <= s0;
      end case;
   end process;
end two_seg_arch;
```

Note that the default output values are listed at the beginning of the code.

The code for the next-state logic and output logic follows the ASM chart closely. Once a detailed state diagram or ASM chart is derived, converting an FSM to HDL code is almost a mechanical procedure. Listings 5.1 and 5.2 can serve as templates for this purpose.

Xilinx specific

Xilinx ISE includes a utility program called *StateCAD*, which allows a user to draw a state diagram in graphical format. The program then converts the state diagram to HDL code. It is a good idea to try it first with a few simple examples to see whether the generated code and its style are satisfactory, particularly for the output signals.

5.3 DESIGN EXAMPLES

5.3.1 Rising-edge detector

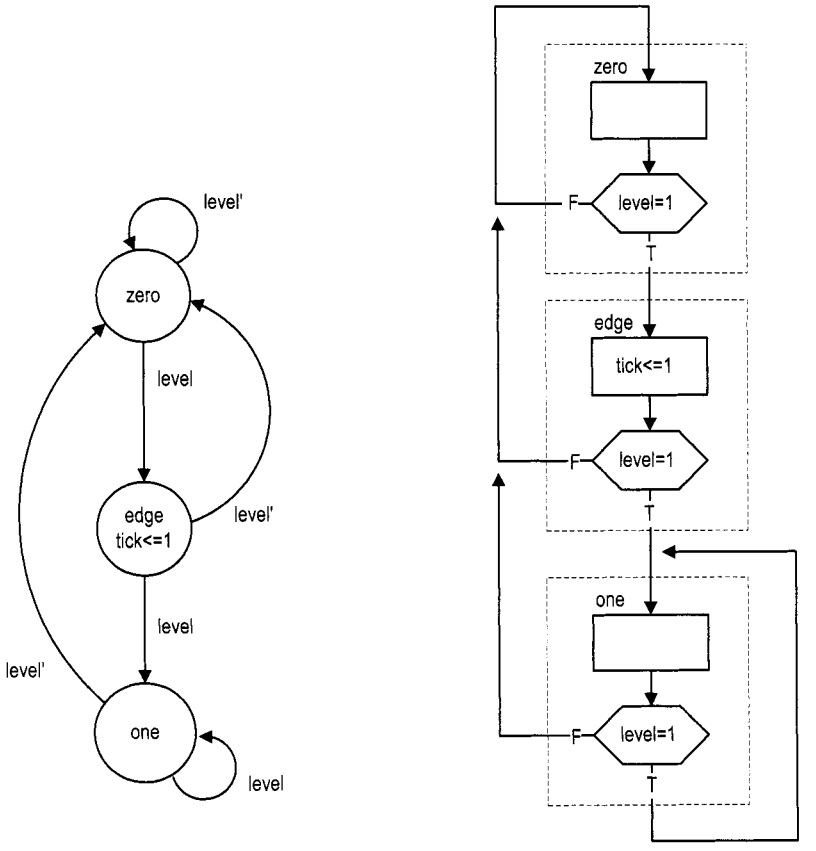
The rising-edge detector is a circuit that generates a short, one-clock-cycle pulse (we call it a *tick*) when the input signal changes from '0' to '1'. It is usually used to indicate the onset of a slow time-varying input signal. We design the circuit using both Moore and Mealy machines, and compare their differences.

Moore-based design The state diagram and ASM chart of a Moore machine–based edge detector are shown in Figure 5.4. The `zero` and `one` states indicate that the input signal has been '0' and '1' for awhile. The rising edge occurs when the input changes to '1' in the `zero` state. The FSM moves to the `edge` state and the output, `tick`, is asserted in this state. A representative timing diagram is shown at the middle of Figure 5.5. The code is shown in Listing 5.3.

Listing 5.3 Moore machine–based edge detector

```
library ieee;
use ieee.std_logic_1164.all;
entity edge_detect is
   port(
      clk, reset: in std_logic;
      level: in std_logic;
      tick: out std_logic
   );
end edge_detect;

architecture moore_arch of edge_detect 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;
   end process;
   -- next-state/output logic
   process(state_reg,level)
   begin
      state_next <= state_reg;
      tick <= '0';
      case state_reg is
```



(a) State diagram (b) ASM chart

Figure 5.4 Edge detector based on a Moore machine.

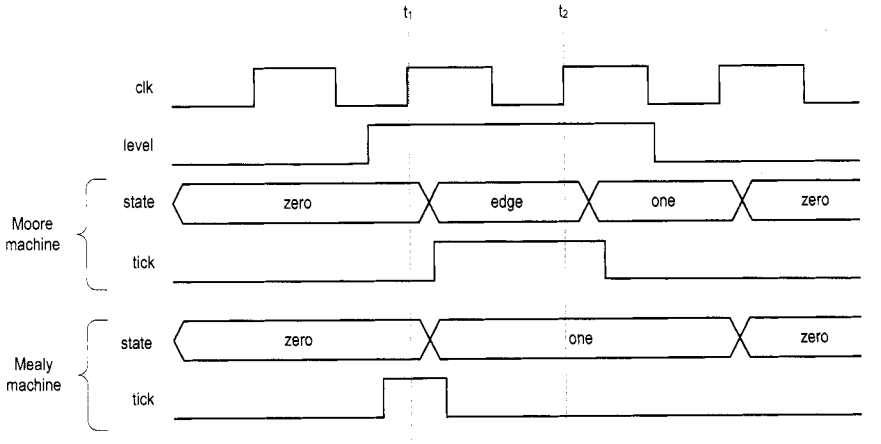
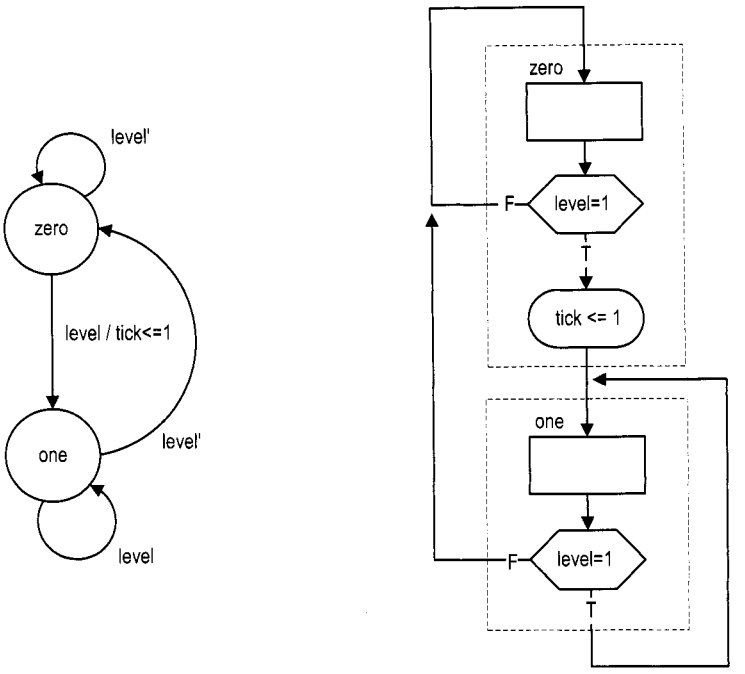


Figure 5.5 Timing diagram of two edge detectors.



(a) State diagram (b) ASM chart

Figure 5.6 Edge detector based on a Mealy machine.

```
30          when zero=>
               if level= '1' then
                  state_next <= edge;
               end if;
            when edge =>
35             tick <= '1';
               if level= '1' then
                  state_next <= one;
               else
                  state_next <= zero;
40             end if;
            when one =>
               if level= '0' then
                  state_next <= zero;
               end if;
45       end case;
     end process;
  end moore_arch;
```

Mealy-based design The state diagram and ASM chart of a Mealy machine–based edge detector are shown in Figure 5.6. The zero and one states have similar meaning. When the FSM is in the zero state and the input changes to '1', the output is asserted

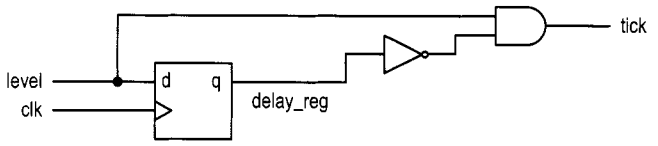


Figure 5.7 Gate-level implementation of an edge detector.

immediately. The FSM moves to the one state at the rising edge of the next clock and the output is deasserted. A representative timing diagram is shown at the bottom of Figure 5.5. Note that due to the propagation delay, the output signal is still asserted at the rising edge of the next clock (i.e., at t_1). The code is shown in Listing 5.4.

Listing 5.4 Mealy machine–based edge detector

```
architecture mealy_arch of edge_detect 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/output logic
   process(state_reg,level)
   begin
      state_next <= state_reg;
      tick <= '0';
      case state_reg is
         when zero=>
            if level= '1' then
               state_next <= one;
               tick <= '1';
            end if;
         when one =>
            if level= '0' then
               state_next <= zero;
            end if;
      end case;
   end process;
end mealy_arch;
```

Direct implementation Since the transitions of the edge detector circuit are very simple, it can be implemented without using an FSM. We include this implementation for comparison purposes. The circuit diagram is shown in Figure 5.7. It can be interpreted that the output is asserted only when the current input is '1' and the previous input, which is stored in the register, is '0'. The corresponding code is shown in Listing 5.5.

Listing 5.5 Gate-level implementation of an edge detector

```
architecture gate_level_arch of edge_detect is
   signal delay_reg: std_logic;
begin
   -- delay register
   process(clk,reset)
   begin
      if (reset='1') then
         delay_reg <= '0';
      elsif (clk'event and clk='1') then
         delay_reg <= level;
      end if;
   end process;
   -- decoding logic
   tick <= (not delay_reg) and level;
end gate_level_arch;
```

Although the descriptions in Listings 5.4 and 5.5 appear to be very different, they describe the same circuit. The circuit diagram can be derived from the FSM if we assign '0' and '1' to the zero and one states.

Comparison Whereas both Moore machine– and Mealy machine–based designs can generate a short tick at the rising edge of the input signal, there are several subtle differences. The Mealy machine–based design requires fewer states and responds faster, but the width of its output may vary and input glitches may be passed to the output.

The choice between the two designs depends on the subsystem that uses the output signal. Most of the time the subsystem is a synchronous system that shares the same clock signal. Since the FSM's output is sampled only at the rising edge of the clock, the width and glitches do not matter as long as the output signal is stable around the edge. Note that the Mealy output signal is available for sampling at t_1, which is one clock cycle faster than the Moore output, which is available at t_2. Therefore, the Mealy machine–based circuit is preferred for this type of application.

5.3.2 Debouncing circuit

The slide and pushbutton switches on the prototyping board are mechanical devices. When pressed, the switch may bounce back and forth a few times before settling down. The bounces lead to glitches in the signal, as shown at the top of Figure 5.8. The bounces usually settle within 20 ms. The purpose of a debouncing circuit is to filter out the glitches associated with switch transitions. The debounced output signals from two FSM-based design schemes are shown in the two bottom parts of Figure 5.8. The first design scheme is discussed in this subsection and the second scheme is left as an exercise in Experiment 5.5.2. A better alternative FSMD-based scheme is discussed in Section 6.2.1.

An FSM-based design uses a free-running 10-ms timer and an FSM. The timer generates a one-clock-cycle enable tick (the m_tick signal) every 10 ms and the FSM uses this information to keep track of whether the input value is stabilized. In the first design scheme, the FSM ignores the short bounces and changes the value of the debounced output only after the input is stabilized for 20 ms. The output timing diagram is shown at the middle of Figure 5.8. The state diagram of this FSM is shown in Figure 5.9. The zero and one states indicate that the switch input signal, sw, has been stabilized with '0' and '1' values.

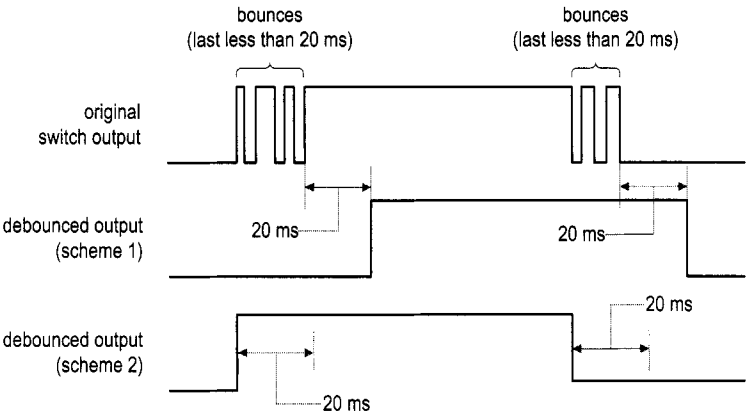


Figure 5.8 Original and debounced waveforms.

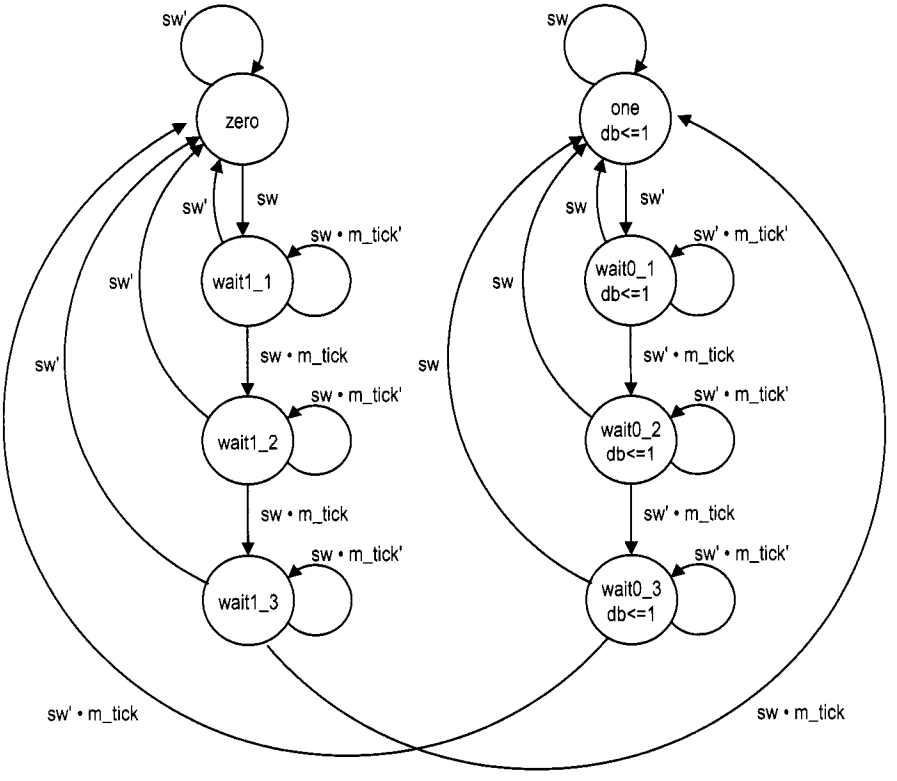


Figure 5.9 State diagram of a debouncing circuit.

Assume that the FSM is initially in the zero state. It moves to the wait1_1 state when sw changes to '1'. At the wait1_1 state, the FSM waits for the assertion of m_tick. If sw becomes '0' in this state, it implies that the width of the '1' value does not last long enough and the FSM returns to the zero state. This action repeats two more times for the wait1_2 and wait1_3 states. The operation from the one state is similar except that the sw signal must be '0'.

Since the 10-ms timer is free-running and the m_tick tick can be asserted at any time, the FSM checks the assertion three times to ensure that the sw signal is stabilized for at least 20 ms (it is actually between 20 and 30 ms). The code is shown in Listing 5.6. It includes a 10-ms timer and the FSM.

Listing 5.6 FSM implementation of a debouncing circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity db_fsm is
   port(
      clk, reset: in std_logic;
      sw: in std_logic;
      db: out std_logic
   );
end db_fsm;

architecture arch of db_fsm is
   constant N: integer:=19;  -- 2^N * 20ns = 10ms
   signal q_reg, q_next: unsigned(N-1 downto 0);
   signal m_tick: std_logic;
   type eg_state_type is (zero,wait1_1,wait1_2,wait1_3,
                          one,wait0_1,wait0_2,wait0_3);
   signal state_reg, state_next: eg_state_type;
begin
   --=================================
   -- counter to generate 10ms tick
   -- (2^19 * 20ns)
   --=================================
   process(clk,reset)
   begin
      if (clk'event and clk='1') then
         q_reg <= q_next;
      end if;
   end process;
   -- next-state logic
   q_next <= q_reg + 1;
   --output tick
   m_tick <= '1' when q_reg=0 else
             '0';
   --=================================
   -- debouncing FSM
   --=================================
   -- 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/output logic
process(state_reg,sw,m_tick)
begin
   state_next <= state_reg; --default: back to same state
   db <= '0';   -- default 0
   case state_reg is
      when zero =>
         if sw='1' then
            state_next <= wait1_1;
         end if;
      when wait1_1 =>
         if sw='0' then
            state_next <= zero;
         else
            if m_tick='1' then
               state_next <= wait1_2;
            end if;
         end if;
      when wait1_2 =>
         if sw='0' then
            state_next <= zero;
         else
            if m_tick='1' then
               state_next <= wait1_3;
            end if;
         end if;
      when wait1_3 =>
         if sw='0' then
            state_next <= zero;
         else
            if m_tick='1' then
               state_next <= one;
            end if;
         end if;
      when one =>
         db <='1';
         if sw='0' then
            state_next <= wait0_1;
         end if;
      when wait0_1 =>
         db <='1';
         if sw='1' then
            state_next <= one;
         else
            if m_tick='1' then
               state_next <= wait0_2;
            end if;
```

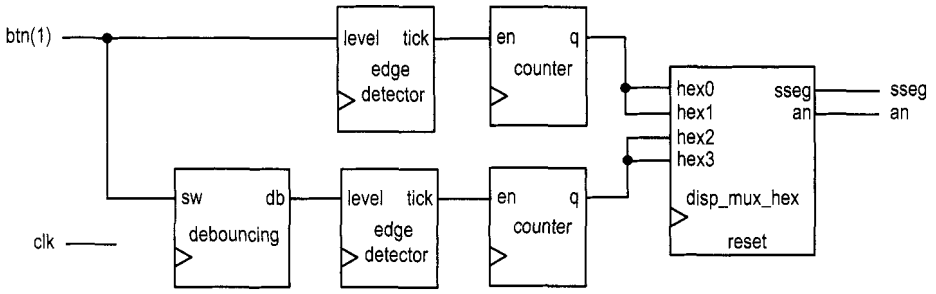


Figure 5.10 Debouncing testing circuit.

```
            end if;
95        when wait0_2 =>
            db <='1';
            if sw='1' then
               state_next <= one;
            else
100            if m_tick='1' then
                  state_next <= wait0_3;
               end if;
            end if;
          when wait0_3 =>
105         db <='1';
            if sw='1' then
               state_next <= one;
            else
               if m_tick='1' then
110               state_next <= zero;
               end if;
            end if;
       end case;
    end process;
115 end arch;
```

5.3.3 Testing circuit

We use a bounce counting circuit to verify operation of the rising-edge detector and the debouncing circuit. The block diagram is shown in Figure 5.10. The input of the verification circuit is from a pushbutton switch. In the lower part, the signal is first fed to the debouncing circuit and then to the rising-edge detector. Therefore, a one-clock-cycle tick is generated each time the button is pressed and released. The tick in turn controls the enable input of an 8-bit counter, whose content is passed to the LED time-multiplexing circuit and shown on the left two digits of the prototyping board's seven-segment LED display. In the upper part, the input signal is fed directly to the edge detector without the debouncing circuit, and the number is shown on the right two digits of the prototyping board's seven-segment LED display. The bottom counter thus counts one desired 0-to-1 transition as well as the bounces.

The code is shown in Listing 5.7. It basically uses component instantiation to realize the block diagram.

Listing 5.7 Verification circuit for a debouncing circuit and rising-edge detector

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity debounce_test is
   port(
      clk: in std_logic;
      btn: in std_logic_vector(3 downto 0);
      an: out std_logic_vector(3 downto 0);
      sseg: out std_logic_vector(7 downto 0)
   );
end debounce_test;

architecture arch of debounce_test is
   signal q1_reg, q1_next: unsigned(7 downto 0);
   signal q0_reg, q0_next: unsigned(7 downto 0);
   signal b_count, d_count: std_logic_vector(7 downto 0);
   signal btn_reg, db_reg: std_logic;
   signal db_level, db_tick, btn_tick, clr: std_logic;
begin
   --================================================
   -- component instantiation
   --================================================
   -- instantiate hex display time-multiplexing circuit
   disp_unit: entity work.disp_hex_mux
      port map(
         clk=>clk, reset=>'0',
         hex3=>b_count(7 downto 4), hex2=>b_count(3 downto 0),
         hex1=>d_count(7 downto 4), hex0=>d_count(3 downto 0),
         dp_in=>"1011", an=>an, sseg=>sseg);
   -- instantiate debouncing circuit
   db_unit: entity work.db_fsm(arch)
      port map(
         clk=>clk, reset=>'0',
         sw=>btn(1), db=>db_level);

   --================================================
   -- edge detection circuits
   --================================================
   process(clk)
   begin
      if (clk'event and clk='1') then
         btn_reg <= btn(1);
         db_reg <= db_level;
      end if;
   end process;
   btn_tick <= (not btn_reg) and btn(1);
   db_tick <= (not db_reg) and db_level;

   --================================================
```

```
   -- two counters
   --=================================================
   clr <= btn(0);
   process(clk)
   begin
      if (clk'event and clk='1') then
         q1_reg <= q1_next;
         q0_reg <= q0_next;
      end if;
   end process;
   -- next-state logic for the counter
   q1_next <= (others=>'0') when clr='1' else
              q1_reg + 1 when btn_tick='1' else
              q1_reg;
   q0_next <= (others=>'0') when clr='1' else
              q0_reg + 1 when db_tick='1' else
              q0_reg;
   --output
   b_count <= std_logic_vector(q1_reg);
   d_count <= std_logic_vector(q0_reg);
end arch;
```

The seven-segment display shows the accumulated numbers of 0-to-1 edges of bounced and debounced switch input. After pressing and releasing the pushbutton switch several times, we can determine the average number of bounces for each transition.

5.4 BIBLIOGRAPHIC NOTES

The bibliographic information for this chapter is similar to that for Chapter 3.

5.5 SUGGESTED EXPERIMENTS

5.5.1 Dual-edge detector

A dual-edge detector is similar to a rising-edge detector except that the output is asserted for one clock cycle when the input changes from 0 to 1 (i.e., rising edge) and 1 to 0 (i.e., falling edge).

1. Design the circuit based on the Moore machine and draw the state diagram and ASM chart.
2. Derive the HDL code based on the state diagram of the ASM chart.
3. Derive a testbench and use simulation to verify operation of the code.
4. Replace the rising detectors in Section 5.3.3 with dual-edge detectors and verify their operations.
5. Repeat steps 1 to 4 for a Mealy machine–based design.

5.5.2 Alternative debouncing circuit

One problem with the debouncing design in Section 5.3.2 is the delayed response of the onset of a switch transition. An alternative is to react to the first edge in the transition and

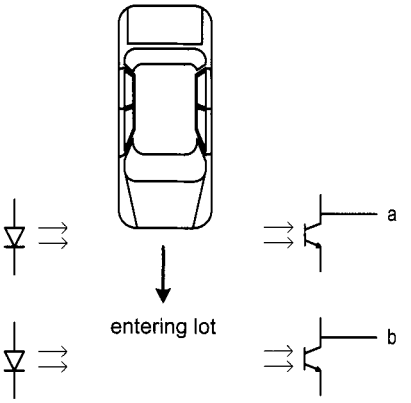


Figure 5.11 Conceptual diagram of gate sensors.

then wait for a small amount of time (at least 20 ms) to have the input signal settled. The output timing diagram is shown at the bottom of Figure 5.8. When the input changes from '0' to '1', the FSM responds immediately. The FSM then ignores the input for about 20 ms to avoid glitches. After this amount of time, the FSM starts to check the input for the falling edge. Follow the design procedure in Section 5.3.2 to design the alternative circuit.

1. Derive the state diagram and ASM chart for the circuit.
2. Derive the HDL code.
3. Derive the HDL code based on the state diagram and ASM chart.
4. Derive a testbench and use simulation to verify operation of the code.
5. Replace the debouncing circuit in Section 5.3.3 with the alternative design and verify its operation.

5.5.3 Parking lot occupancy counter

Consider a parking lot with a single entry and exit gate. Two pairs of photo sensors are used to monitor the activity of cars, as shown in Figure 5.11. When an object is between the photo transmitter and the photo receiver, the light is blocked and the corresponding output is asserted to '1'. By monitoring the events of two sensors, we can determine whether a car is entering or exiting or a pedestrian is passing through. For example, the following sequence indicates that a car enters the lot:

- Initially, both sensors are unblocked (i.e., the `a` and `b` signals are "00").
- Sensor a is blocked (i.e., the `a` and `b` signals are "10").
- Both sensors are blocked (i.e., the `a` and `b` signals are "11").
- Sensor a is unblocked (i.e., the `a` and `b` signals are "01").
- Both sensors becomes unblocked (i.e., the `a` and `b` signals are "00").

Design a parking lot occupancy counter as follows:

1. Design an FSM with two input signals, `a` and `b`, and two output signals, `enter` and `exit`. The `enter` and `exit` signals assert one clock cycle when a car enters and one clock cycle when a car exits the lot, respectively.
2. Derive the HDL code for the FSM.

3. Design a counter with two control signals, `inc` and `dec`, which increment and decrement the counter when asserted. Derive the HDL code.
4. Combine the counter and the FSM and LED multiplexing circuit. Use two debounced pushbuttons to mimic operation of the two sensor outputs. Verify operation of the occupancy counter.