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

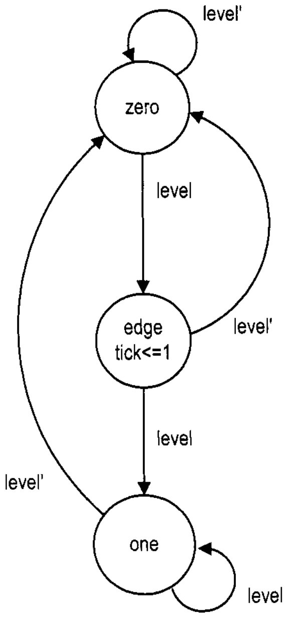
The rising-edge detector is a circuit that generates a short, one-clock-cycle pulse (we call it a *tick*) when the input signal changes from '0' to '1'. It is usually used to indicate the onset of a slow time-varying input signal. We design the circuit using both Moore and Mealy machines, and compare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

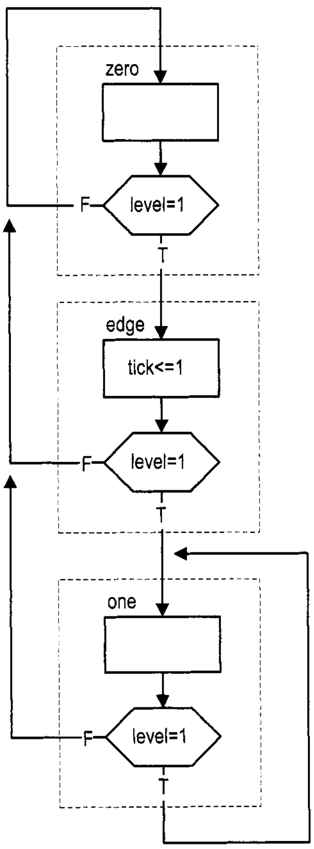
Listing 5.3 Moore machine–based edge detector

```
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.

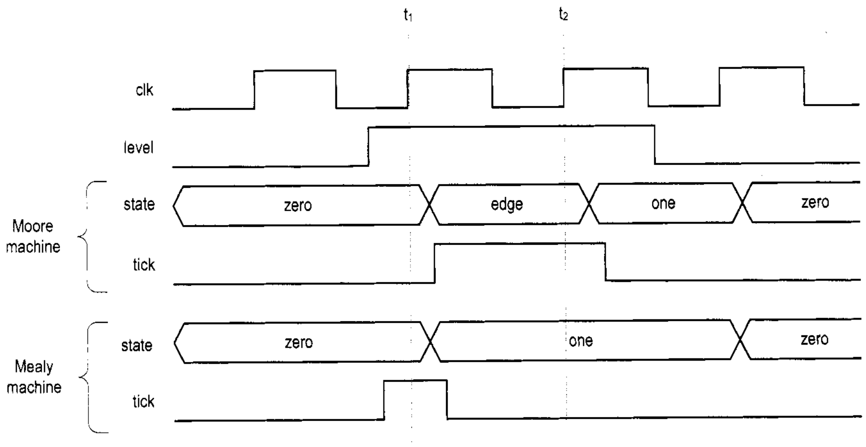
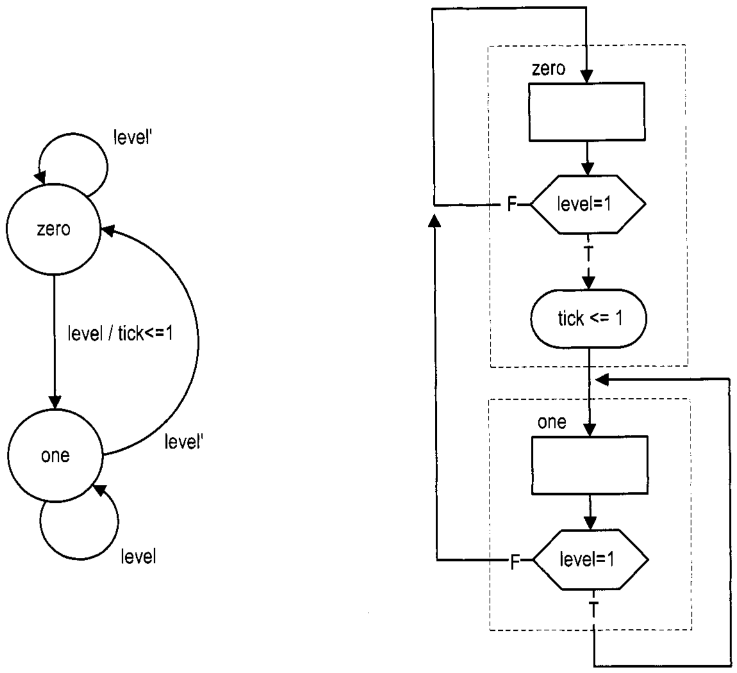


Figure 5.5 Timing diagram of two edge detectors.



(a) State diagram (b) ASM chart

Figure 5.6 Edge detector based on a Mealy machine.

```
      when zero=>
         if level= '1' then
            state_next <= edge;
         end if;
      when edge =>
         tick <= '1';
         if level= '1' then
            state_next <= one;
         else
            state_next <= zero;
         end if;
      when one =>
         if level= '0' then
            state_next <= zero;
         end if;
    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

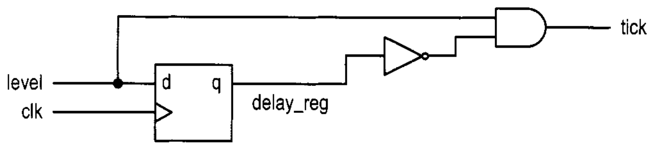


Figure 5.7 Gate-level implementation of an edge detector.

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.

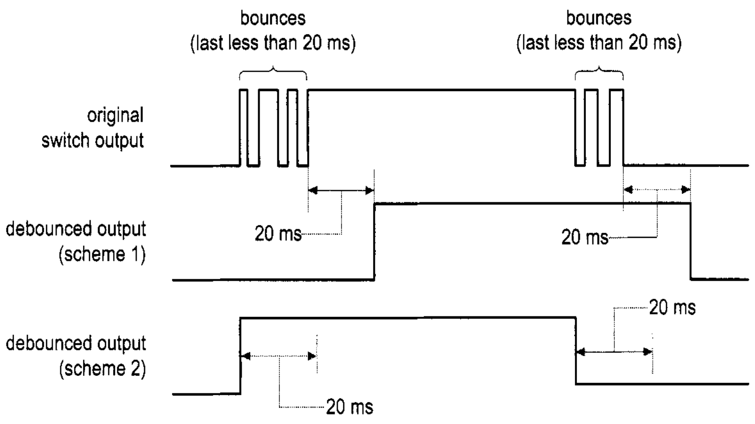


Figure 5.8 Original and debounced waveforms.

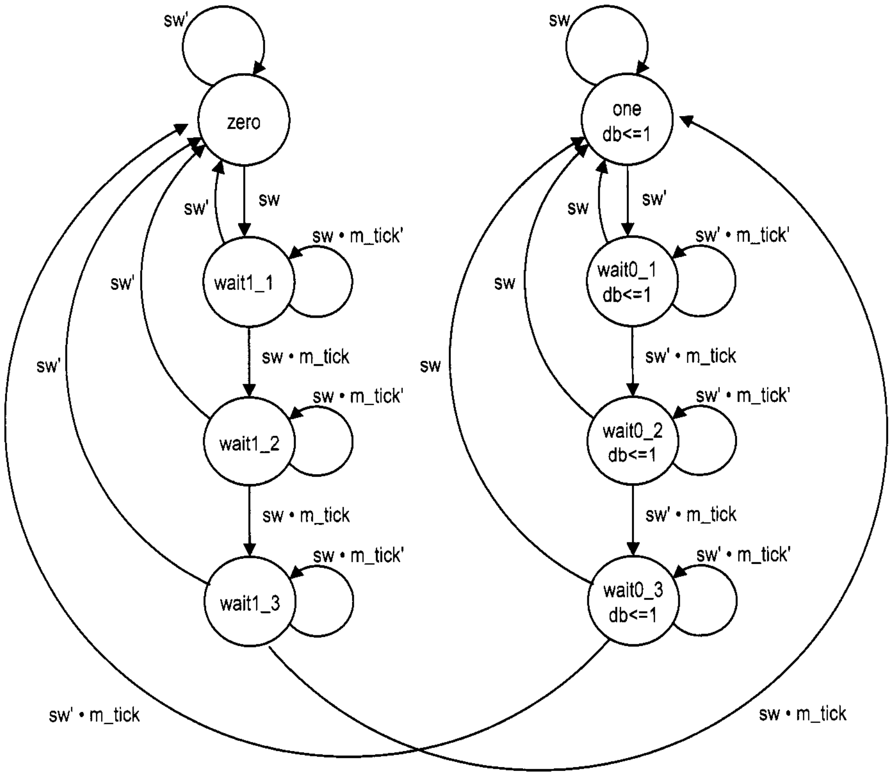


Figure 5.9 State diagram of a debouncing circuit.

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;
```

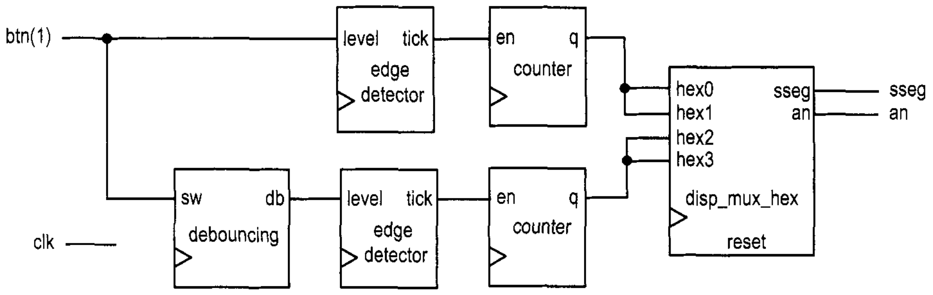


Figure 5.10 Debouncing testing circuit.

```
            end if;
         when wait0_2 =>
            db <='1';
            if sw='1' then
               state_next <= one;
            else
               if m_tick='1' then
                  state_next <= wait0_3;
               end if;
            end if;
         when wait0_3 =>
            db <='1';
            if sw='1' then
               state_next <= one;
            else
               if m_tick='1' then
                  state_next <= zero;
               end if;
            end if;
      end case;
   end process;
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

```vhdl
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;

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