cycle. Moreover, the outputs may have momentary false values because of the delay encountered from the time that the inputs change and the time that the flip-flop outputs change. In order to synchronize a Mealy-type circuit, the inputs of the sequential circuit must be synchronized with the clock and the outputs must be sampled immediately before the clock edge. The inputs are changed at the inactive edge of the clock to ensure that the inputs to the flip-flops stabilize before the active edge of the clock occurs. Thus, **the output of the Mealy machine is the value that is present immediately before the active edge of the clock.**

5.6 SYNTHESIZABLE HDL MODELS OF SEQUENTIAL CIRCUITS

The Verilog HDL was introduced in Section 3.9. Combinational circuits were described in Section 4.12, and behavioral modeling with Verilog was introduced in that section as well. Behavioral models are abstract representations of the functionality of digital hardware. That is, they describe how a circuit behaves, but don't specify the internal details of the circuit. Historically, the abstraction has been described by truth tables, state tables, and state diagrams. An HDL describes the functionality differently, by language constructs that represent the operations of registers in a machine. This representation has "added value," i.e., it is important for you to know how to use, because it can be simulated to produce waveforms demonstrating the behavior of the machine.

Behavioral Modeling

There are two kinds of abstract behaviors in the Verilog HDL. Behavior declared by the keyword **initial** is called *single-pass behavior* and specifies a single statement or a block statement (i.e., a list of statements enclosed by either a **begin** . . . **end** or a **fork** . . . **join** keyword pair). A single-pass behavior expires after the associated statement executes. In practice, designers use single-pass behavior primarily to prescribe stimulus signals in a test bench—never to model the behavior of a circuit—because synthesis tools do not accept descriptions that use the **initial** statement. The **always** keyword declares a *cyclic behavior.* Both types of behaviors begin executing when the simulator launches at time $t = 0$. The **initial** behavior expires after its statement executes; the **always** behavior executes and reexecutes indefinitely, until the simulation is stopped. A module may contain an arbitrary number of **initial** or **always** behavioral statements. They execute concurrently with respect to each other, starting at time 0, and may interact through common variables. Here's a word description of how an **always** statement works for a simple model of a D flip-flop: Whenever the rising edge of the clock occurs, if the reset input is asserted, the output q gets 0; otherwise the output Q gets the value of the input D. The execution of statements triggered by the clock is repeated until the simulation ends. We'll see shortly how to write this description in Verilog.

An **initial** behavioral statement executes only once. It begins its execution at the start of simulation and expires after all of its statements have completed execution. As mentioned at the end of Section 4.12, the **initial** statement is useful for generating input signals to simulate a design. In simulating a sequential circuit, it is necessary to generate a clock source for triggering the flip-flops. The following are two possible ways to provide a free-running clock that operates for a specified number of cycles:

```
initial
 begin
  clock = 1'b0;
  repeat (30)
   #10 clock = ~clock;
 end
```

```
initial
 begin
  clock = 1'b0;
 end

initial 300 $finish;
always #10 clock = ~clock;
```

In the first version, the **initial** block contains two statements enclosed within the **begin** and **end** keywords. The first statement sets *clock* to 0 at time $= 0$. The second statement specifies a loop that reexecutes 30 times to wait 10 time units and then complement the value of *clock*. This produces 15 clock cycles, each with a cycle time of 20 time units. In the second version, the first **initial** behavior has a single statement that sets *clock* to 0 at time $= 0$, and it then expires (causes no further simulation activity). The second single-pass behavior declares a stopwatch for the simulation. The system task **finish** causes the simulation to terminate unconditionally after 300 time units have elapsed. Because this behavior has only one statement associated with it, there is no need to write the **begin** . . . **end** keyword pair. After 10 time units, the **always** statement repeatedly complements *clock,* providing a clock generator having a cycle time of 20 time units. The three behavioral statements in the second example can be written in any order.

Here is another way to describe a free-running clock:

```
initial begin clock = 0; forever #10 clock = ~clock; end
```

This version, with two statements in one block statement, initializes the clock and then executes an indefinite loop (**forever**) in which the clock is complemented after a delay of 10 time steps. Note that the single-pass behavior never finishes executing and so does not expire. Another behavior would have to terminate the simulation.

The activity associated with either type of behavioral statement can be controlled by a delay operator that waits for a certain time or by an event control operator that waits for certain conditions to become true or for specified events (changes in signals) to occur. Time delays specified with the # *delay control operator* are commonly used in single-pass behaviors. The delay control operator suspends execution of statements until a specified time has elapsed. We've already seen examples of its use to specify signals in a test bench. Another operator @ is called the *event control operator* and is used to *suspend* activity until an event occurs. An event can be an unconditional change in a signal value (e.g., @ A) or a specified transition of a signal value (e.g., @ (**posedge** clock)). The general form of this type of statement is

```
always @ (event control expression) begin
   // Procedural assignment statements that execute when the condition is met
end
```

The event control expression specifies the condition that must occur to launch execution of the procedural assignment statements. The variables in the left-hand side of the procedural statements must be of the **reg** data type and must be declared as such. The right-hand side can be any expression that produces a value using Verilog-defined operators.

The event control expression (also called the sensitivity list) specifies the events that must occur to initiate execution of the procedural statements associated with the **always** block. Statements within the block execute sequentially from top to bottom. After the last statement executes, the behavior waits for the event control expression to be satisfied. Then the statements are executed again. The sensitivity list can specify level-sensitive events, edge-sensitive events, or a combination of the two. In practice, designers do not make use of the third option, because this third form is not one that synthesis tools are able to translate into physical hardware. Level-sensitive events occur in combinational circuits and in latches. For example, the statement

```
always @ (A or B or C)
```

will initiate execution of the procedural statements in the associated **always** block if a change occurs in *A, B,* or *C*. In synchronous sequential circuits, changes in flip-flops occur only in response to a transition of a clock pulse. The transition may be either a positive edge or a negative edge of the clock, but not both. Verilog HDL takes care of these conditions by providing two keywords: **posedge** and **negedge**. For example, the expression

```
always @(posedge clock or negedge reset)        // Verilog 1995
```

will initiate execution of the associated procedural statements only if the clock goes through a positive transition or if *reset* goes through a negative transition. The 2001 and 2005 revisions to the Verilog language allow a comma-separated list for the event control expression (or sensitivity list):

```
always @(posedge clock, negedge reset)          // Verilog 2001, 2005
```

A procedural assignment is an assignment of a logic value to a variable within an **initial** or **always** statement. This is in contrast to a continuous assignment discussed in Section 4.12 with dataflow modeling. A continuous assignment has an implicit level-sensitive sensitivity list consisting of all of the variables on the right-hand side of its assignment statement. The updating of a continuous assignment is triggered whenever an event occurs in a variable included on the right-hand side of its expression. In contrast, a procedural assignment is made only when an assignment statement is executed and assigns value to it within a behavioral statement. For example, the clock signal in the preceding example was complemented only when the statement *clock = ~clock* executed; the statement did not execute until 10 time units after the simulation began. It is important to remember that a variable having type **reg** remains unchanged until a procedural assignment is made to give it a new value.

There are two kinds of procedural assignments: *blocking* and *nonblocking*. The two are distinguished by the symbols that they use. Blocking assignments use the symbol (=) as the assignment operator, and nonblocking assignments use (< =) as the operator.

Blocking assignment statements are executed sequentially in the order they are listed in a block of statements. Nonblocking assignments are executed concurrently by evaluating the set of expressions on the right-hand side of the list of statements; they do not make assignments to their left-hand sides until all of the expressions are evaluated. The two types of assignments may be better understood by means of an illustration. Consider these two procedural blocking assignments:

```
B = A
C = B + 1
```

The first statement transfers the value of A into B. The second statement increments the value of B and transfers the new value to C. At the completion of the assignments, C contains the value of $A + 1$.

Now consider the two statements as nonblocking assignments:

```
B <= A
C <= B + 1
```

When the statements are executed, the expressions on the right-hand side are evaluated and stored in a temporary location. The value of A is kept in one storage location and the value of $B + 1$ in another. After all the expressions in the block are evaluated and stored, the assignment to the targets on the left-hand side is made. In this case, C will contain the original value of B, plus 1. A general rule is to **use blocking assignments when sequential ordering is imperative and in cyclic behavior that is level sensitive** (i.e., in combinational logic). **Use nonblocking assignments when modeling concurrent execution** (e.g., edge-sensitive behavior such as synchronous, concurrent register transfers) **and when modeling latched behavior.** Nonblocking assignments are imperative in dealing with register transfer level design, as shown in Chapter 8. They model the concurrent operations of physical hardware synchronized by a common clock. Today's designers are expected to know what features of an HDL are useful in a practical way and how to avoid features that are not. Following these rules for using the assignment operators will prevent conditions that lead synthesis tools astray and create mismatches between the behavior of a model and the behavior of physical hardware that is produced by a synthesis tool.

HDL Models of Flip-Flops and Latches

HDL Examples 5.1 through 5.4 show Verilog descriptions of various flip-flops and a D latch. The D latch is said to be *transparent* because it responds to a change in data input with a change in the output as long as the enable input is asserted—viewing the output is the same as viewing the input. The description of a D latch is shown in HDL Example 5.1 It has two inputs, D and *enable,* and one output, Q. Since Q is assigned value in a behavior, its type must be declared to be **reg.** Hardware latches respond to input signal *levels,* so the two inputs are listed without edge qualifiers in the sensitivity list following the @ symbol in the **always** statement. In this model, there is only one blocking procedural assignment statement, and it specifies the transfer of input D to output Q if enable is true (logic 1).[1] Note that this statement is executed every time there is a change in *D* if *enable* is 1.

A *D*-type flip-flop is the simplest example of a sequential machine. HDL Example 5.2 describes two positive-edge *D* flip-flops in two modules. The first responds only to the clock; the second includes an asynchronous reset input. Output *Q* must be declared as a **reg** data type in addition to being listed as an output. This is because it is a target output of a procedural assignment statement. The keyword **posedge** ensures that the transfer of input *D* into *Q* is synchronized by the positive-edge transition of *Clk*. A change in *D* at any other time does not change *Q*.

HDL Example 5.1 (*D*-Latch)

```
// Description of D latch (See Fig. 5.6)
module D_latch (Q, D, enable);
 output Q;
 input  D, enable;
 reg    Q;
 always @ (enable or D)
  if (enable) Q <= D;              // Same as: if (enable == 1)
endmodule

// Alternative syntax (Verilog 2001, 2005)
module D_latch (output reg Q, input enable, D);
 always @ (enable, D)
  if (enable) Q <= D;              // No action if enable not asserted
endmodule
```

HDL Example 5.2 (*D*-Type Flip-Flop)

```
// D flip-flop without reset
module D_FF (Q, D, Clk);
 output Q;
 input  D, Clk;
 reg    Q;
 always @ (posedge Clk)
   Q <= D;
 endmodule

// D flip-flop with asynchronous reset (V2001, V2005)
module DFF (output reg Q, input D, Clk, rst);
 always @ (posedge Clk, negedge rst)
 if (!rst) Q <= 1'b0;          // Same as: if (rst == 0)
 else Q <= D;
endmodule
```

[1] The statement (single or block) associated with **if** (Boolean expression) executes if the Boolean expression is true.

The second module includes an asynchronous reset input in addition to the synchronous clock. A specific form of an **if** statement is used to describe such a flip-flop so that the model can be synthesized by a software tool. The event expression after the @ symbol in the **always** statement may have any number of edge events, either **posedge** or **negedge**. For modeling hardware, one of the events must be a clock event. The remaining events specify conditions under which asynchronous logic is to be executed. The designer knows which signal is the clock, but *clock* is not an identifier that software tools automatically recognize as the synchronizing signal of a circuit. The tool must be able to infer which signal is the clock, so *you need to write the description in a way that enables the tool to infer the clock correctly*. The rules are simple to follow: (1) Each **if** or **else if** statement in the procedural assignment statements is to correspond to an asynchronous event, (2) the last **else** statement corresponds to the clock event, and (3) the asynchronous events are tested first. There are two edge events in the second module of HDL Example 5.2. The **negedge** rst (reset) event is asynchronous, since it matches the **if** (!rst) statement. As long as *rst* is 0, Q is cleared to 0. If *Clk* has a positive transition, its effect is blocked. Only if $rst = 1$ can the **posedge** clock event synchronously transfer D into Q.

Hardware always has a reset signal. It is strongly recommended that all models of edge-sensitive behavior include a reset (or preset) input signal; otherwise, the initial state of the flip-flops of the sequential circuit cannot be determined. A sequential circuit cannot be tested with HDL simulation unless an initial state can be assigned with an input signal.

HDL Example 5.3 describes the construction of a T or JK flip-flop from a D flip-flop and gates. The circuit is described with the characteristic equations of the flip-flops:

$$Q(t + 1) = Q \oplus T \qquad \text{for a } T \text{ flip-flop}$$
$$Q(t + 1) = JQ' + K'Q \qquad \text{for a } JK \text{ flip-flop}$$

The first module, *TFF,* describes a T flip-flop by instantiating *DFF*. (Instantiation is explained in Section 4.12.) The declared **wire**, *DT,* is assigned the exclusive-OR of Q and T, as is required for building a T flip-flop with a D flip-flop. The instantiation with the value of *DT* replacing D in module *DFF* produces the required T flip-flop. The JK flip-flop is specified in a similar manner by using its characteristic equation to define a replacement for D in the instantiated *DFF*.

HDL Example 5.3 (Alternative Flip-Flop Models)

```
// T flip-flop from D flip-flop and gates
module TFF (Q, T, Clk, rst);
 output Q;
 input   T, Clk, rst;
 wire     DT;
 assign DT = Q ^ T ;                    // Continuous assignment
// Instantiate the D flip-flop
 DFF TF1 (Q, DT, Clk, rst);
endmodule
```

```
// JK flip-flop from D flip-flop and gates (V2001, 2005)
module JKFF (output reg Q, input J, K, Clk, rst);
 wire JK;
 assign JK = (J & ~Q) | (~K & Q);
// Instantiate D flip-flop
 DFF JK1 (Q, JK, Clk, rst);
endmodule

// D flip-flop (V2001, V2005)
module DFF (output reg Q, input D, Clk, rst);
 always @ (posedge Clk, negedge rst)
  if (!rst) Q <= 1'b0;
  else Q <= D;
endmodule
```

HDL Example 5.4 shows another way to describe a *JK* flip-flop. Here, we describe the flip-flop by using the characteristic table rather than the characteristic equation. The **case** multiway branch condition checks the two-bit number obtained by concatenating the bits of *J* and *K*. The **case** expression ({*J*, *K*}) is evaluated and compared with the values in the list of statements that follows. The first value that matches the true condition is executed. Since the concatenation of *J* and *K* produces a two-bit number, it can be equal to 00, 01, 10, or 11. The first bit gives the value of *J* and the second the value of *K*. The four possible conditions specify the value of the next state of *Q* after the application of a positive-edge clock.

HDL Example 5.4 (*JK* Flip-Flop)

```
// Functional description of JK flip-flop (V2001, 2005)
module JK_FF (input J, K, Clk, output reg Q, output Q_b);
 assign Q_b = ~ Q ;
 always @ (posedge Clk)
 case ({J,K})
  2'b00: Q <= Q;
  2'b01: Q <= 1'b0;
  2'b10: Q <= 1'b1;
  2'b11: Q <= !Q;
 endcase
endmodule
```

State diagram-Based HDL Models

An HDL model of the operation of a sequential circuit can be based on the format of the circuit's state diagram. A Mealy HDL model is presented in HDL Example 5.5 for the zero-detector machine described by the sequential circuit in Fig. 5.15 and its state diagram shown in Fig. 5.16. The input, output, clock, and reset are declared in the usual manner.

The state of the flip-flops is declared with identifiers *state* and *next_state*. These variables hold the values of the present state and the next value of the sequential circuit. The state's binary assignment is done with a **parameter** statement. (Verilog allows constants to be defined in a module by the keyword **parameter.**) The four states *S0* through *S3* are assigned binary 00 through 11. The notation $S2 = 2'$b10 is preferable to the alternative $S2 = 2$. The former uses only two bits to store the constant, whereas the latter results in a binary number with 32 (or 64) bits because an unsized number is interpreted and sized as an integer.

HDL Example 5.5 (Mealy Machine: Zero Detector)

```
// Mealy FSM zero detector (See Fig. 5.15 and Fig. 5.16)        Verilog 2001, 2005 syntax
module Mealy_Zero_Detector (
 output reg y_out,
 input    x_in, clock, reset
);
 reg [1: 0]           state, next_state;
 parameter            S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
 always @ (posedge clock, negedge reset)    Verilog 2001, 2005 syntax
  if (reset == 0) state <= S0;
  else state <= next_state;

 always @ (state, x_in)                      // Form the next state
  case (state)
    S0:      if (x_in)  next_state = S1; else next_state = S0;
    S1:      if (x_in)  next_state = S3; else next_state = S0;
    S2:      if (~x_in)  next_state = S0; else next_state = S2;
    S3:      if (x_in)  next_state = S2; else next_state = S0;
  endcase

 always @ (state, x_in)                      // Form the Mealy output
  case (state)
    S0:        y_out = 0;
    S1, S2, S3: y_out = ~x_in;
  endcase
endmodule

module t_Mealy_Zero_Detector;
 wire     t_y_out;
 reg      t_x_in, t_clock, t_reset;

Mealy_Zero_Detector M0 (t_y_out, t_x_in, t_clock, t_reset);
initial #200 $finish;
initial begin t_clock = 0; forever #5 t_clock = ~t_clock; end

initial fork
    t_reset = 0;
 #2 t_reset = 1;
 #87 t_reset = 0;
 #89 t_reset = 1;
 #10 t_x_in = 1;
 #30 t_x_in = 0;
 #40 t_x_in = 1;
 #50 t_x_in = 0;
 #52 t_x_in = 1;
 #54 t_x_in = 0;
 #70 t_x_in = 1;
 #80 t_x_in = 1;
 #70 t_x_in = 0;
 #90 t_x_in = 1;
 #100 t_x_in = 0;
 #120 t_x_in = 1;
 #160 t_x_in = 0;
 #170 t_x_in = 1;
 join
endmodule
```

The circuit I HDL Example 5.5 detects a 0 following a sequence of 1s in a serial bit stream. Its Verilog model uses three **always** blocks that execute concurrently and interact through common variables. The first **always** statement resets the circuit to the initial state $S0 = 00$ and specifies the synchronous clocked operation. The statement *state* <= *next_state* is synchronized to a positive-edge transition of the clock. This means that any change in the value of *next_state* in the second **always** block can affect the value of *state* only as a result of a **posedge** event of *clock*. The second **always** block determines the value of the next state transition as a function of the present state and input. The value assigned to *state* by the nonblocking assignment is the value of *next_state* immediately before the rising edge of *clock*. Notice how the multiway branch condition implements the state transitions specified by the annotated edges in the state diagram of Fig. 5.16. The third **always** block specifies the output as a function of the present state and the input. Although this block is listed as a separate behavior for clarity, it could be combined with the second block. Note that the value of output *y_out* may change if the value of input *x_in* changes while the circuit is in any given state.

So let's summarize how the model describes the behavior of the machine: At every rising edge of *clock*, if *reset* is not asserted, the state of the machine is updated by the first **always** block; when *state* is updated by the first **always** block, the change in *state* is detected by the sensitivity list mechanism of the second **always** block; then the second **always** block updates the value of *next_state* (it will be used by the first **always** block at the next tick of the clock); the third **always** block also detects the change in *state* and updates the value of the output. In addition, the second and third **always** blocks detect changes in *x_in* and update *next_state* and *y_out* accordingly. The test bench provided with *Mealy_Zero_Detector* provides some waveforms to stimulate the model, producing the results shown in Fig. 5.22. Notice how *t_y_out* responds to changes in both the state and the input, and has a glitch (a transient logic value). We display both to *state*[1:0] and *next_state*[1:0] to illustrate how changes in *t_x_in* influence the value of next_state and *t_y_out*. The Mealy glitch in *t_y_out* is due to the (intentional) dynamic behavior of *t_x_in*. The input, *t_x_in,* settles

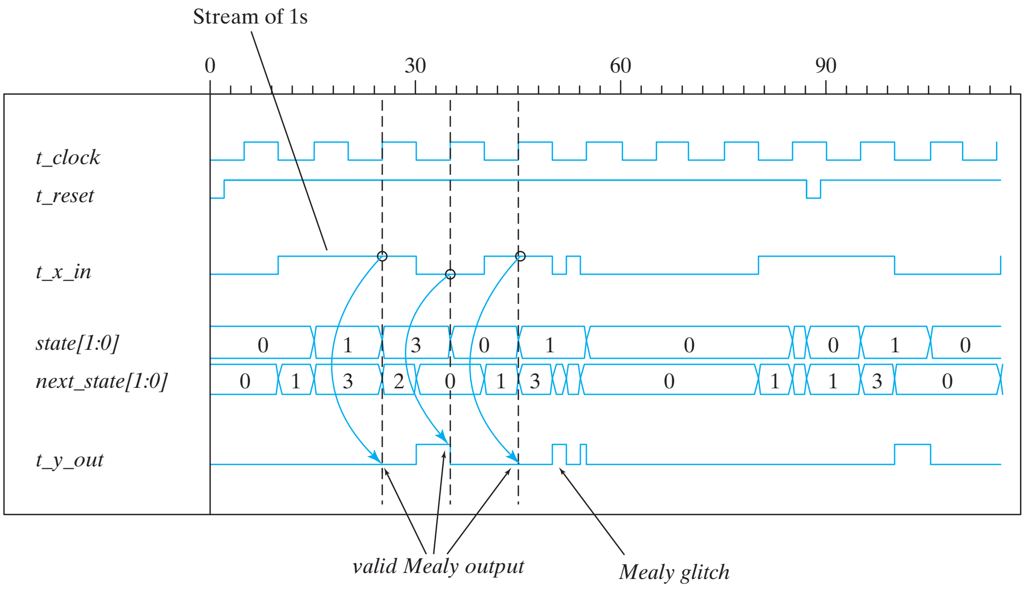

FIGURE 5.22
Simulation output of *Mealy_Zero_Detector*

to a value of 0 immediately before the clock, and at the clock, the state makes a transition from 0 to 1, which is consistent with Fig. 5.16. The output is 1 in state *S*1 immediately before the clock, and changes to 0 as the state enters *S*0.

The description of waveforms in the test bench uses the **fork** . . . **join** construct. Statements with the **fork** . . . **join** block execute in parallel, so the time delays are relative to a common reference of $t = 0$, the time at which the block begins execution.[2] It is usually more convenient to use the **fork** . . . **join** block instead of the **begin** . . . **end** block in describing waveforms. Notice that the waveform of reset is triggered "on the fly" to demonstrate that the machine recovers from an unexpected (asynchronous) reset condition during any state.

How does our Verilog model *Mealy_Zero_Detector* correspond to hardware? The first **always** block corresponds to a *D* flip-flop implementation of the state register in Fig. 5.21; the second **always** block is the combinational logic block describing the next state; the third **always** block describes the output combinational logic of the zero-detecting Mealy machine. The register operation of the state transition uses the nonblocking assignment operator (< =) because the (edge-sensitive) flip-flops of a sequential machine are updated concurrently by a common clock. The second and third **always** blocks describe combinational logic, which is level sensitive, so they use the blocking (=) assignment operator.

[2]A **fork** . . . **join** block completes execution when the last executing statement within it completes its execution.

Their sensitivity lists include both the state and the input because their logic must respond to a change in either or both of them.

Note: The modeling style illustrated by *Mealy_Zero_Detector* is commonly used by designers because it has a close relationship to the state diagram of the machine that is being described. Notice that the reset signal is associated with the **always** block that synchronizes the state transitions. In this example, it is modeled as an active-low reset. Because the reset condition is included in the description of the state transitions, there is no need to include it in the combinational logic that specifies the next state and the output, and the resulting description is less verbose, simpler, and more readable.

HDL Example 5.6 presents the Verilog behavioral model of the Moore FSM shown in Fig. 5.18 and having the state diagram given in Fig. 5.19. The model illustrates an alternative style in which the state transitions of the machine are described by a single clocked (i.e., edge-sensitive) cyclic behavior, i.e., by one **always** block. The present state of the circuit is identified by the variable state, and its transitions are triggered by the rising edge of the clock according to the conditions listed in the **case** statement. The combinational logic that determines the next state is included in the nonblocking assignment to state. In this example, the output of the circuits is independent of the input and is taken directly from the outputs of the flip-flops. The two-bit output *y_out* is specified with a continuous assignment statement and is equal to the value of the present state vector. Figure 5.23 shows some simulation results for *Moore_Model_Fig_5_19*. Here are some important observations: (1) the output depends on only the state, (2) reset "on-the-fly" forces the state of the machine back to S0 (00), and (3) the state transitions are consistent with Fig. 5.19.

HDL Example 5.6 (Moore Machine: Zero Detector)

```
// Moore model FSM (see Fig. 5.19)               Verilog 2001, 2005 syntax
module Moore_Model_Fig_5_19 (
  output [1: 0]         y_out,
  input                 x_in, clock, reset
);
  reg [1: 0]            state;
  parameter             S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

  always @ (posedge clock, negedge reset)
    if (reset == 0) state <= S0;                    // Initialize to state S0
    else case (state)
      S0:      if (~x_in) state <= S1; else state <= S0;
      S1:      if (x_in)   state <= S2; else state <= S3;
      S2:      if (~x_in) state <= S3; else state <= S2;
      S3:      if (~x_in) state <= S0; else state <= S3;
    endcase

  assign y_out = state;        // Output of flip-flops
endmodule
```

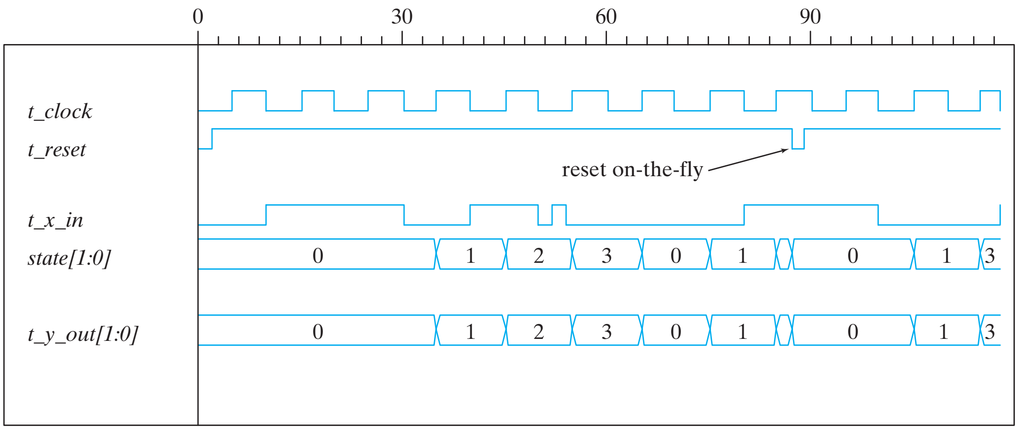

FIGURE 5.23
Simulation output of HDL Example 5.6

Structural Description of Clocked Sequential Circuits

Combinational logic circuits can be described in Verilog by a connection of gates (primitives and UDPs), by dataflow statements (continuous assignments), or by level-sensitive cyclic behaviors (**always** blocks). Sequential circuits are composed of combinational logic and flip-flops, and their HDL models use sequential UDPs and behavioral statements (edge-sensitive cyclic behaviors) to describe the operation of flip-flops. One way to describe a sequential circuit uses a combination of dataflow and behavioral statements. The flip-flops are described with an **always** statement. The combinational part can be described with **assign** statements and Boolean equations. The separate modules can be combined to form a structural model by instantiation within a **module**.

The structural description of a Moore-type zero detector sequential circuit is shown in HDL Example 5.7. We want to encourage the reader to consider alternative ways to model a circuit, so as a point of comparison, we first present *Moore_Model_Fig_5_20,* a Verilog behavioral description of a binary counter having the state diagram examined earlier shown in Fig. 5.20(b). This style of modeling follows directly from the state diagram. An alternative style, used in *Moore_Model_STR_Fig_5_20*, represents the structure shown in Fig. 5.20(a). This style uses two modules. The first describes the circuit of Fig. 5.20(a). The second describes the *T* flip-flop that will be used by the circuit. We also show two ways to model the *T* flip-flop. The first asserts that, at every clock tick, the value of the output of the flip-flop toggles if the toggle input is asserted. The second model describes the behavior of the toggle flip-flop in terms of its characteristic equation. The first style is attractive because it does not require the reader to remember the characteristic equation. Nonetheless, the models are interchangeable and will synthesize to the same hardware circuit. A test bench module provides a stimulus for verifying the functionality of the circuit. The sequential circuit is a two-bit binary counter controlled by input *x_in*. The output, *y_out*, is enabled when the count reaches binary 11. Flip-flops

A and *B* are included as outputs in order to check their operation. The flip-flop input equations and the output equation are evaluated with continuous assignment (**assign**) statements having the corresponding Boolean expressions. The instantiated *T* flip-flops use *TA* and *TB* as defined by the input equations.

The second module describes the *T* flip-flop. The *reset* input resets the flip-flop to 0 with an active-low signal. The operation of the flip-flop is specified by its characteristic equation, $Q(t + 1) = Q \oplus T$.

The test bench includes both models of the machine. The stimulus module provides common inputs to the circuits to simultaneously display their output responses. The first **initial** block provides eight clock cycles with a period of 10 ns. The second **initial** block specifies a toggling of input *x_in* that occurs at the negative edge transition of the clock. The result of the simulation is shown in Fig. 5.24. The pair (*A, B*) goes through the binary sequence 00, 01, 10, 11, and back to 00. The change in the count is triggered by a positive edge of the clock, provided that $x\_in = 1$. If $x\_in = 0$, the count does not change. *y_out* is equal to 1 when both *A* and *B* are equal to 1. This verifies the main functionality of the circuit, but not a recovery from an unexpected reset event.

HDL Example 5.7 (Binary Counter_Moore Model)

```
// State-diagram-based model (V2001, 2005)
module Moore_Model_Fig_5_20 (
 output  y_out,
 input    x_in, clock, reset
);
 reg [1: 0]              state;
 parameter              S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

 always @ (posedge clock, negedge reset)
  if (reset == 0) state <= S0;                    // Initialize to state S0
  else case (state)
   S0:      if (x_in)  state <= S1; else state <= S0;
   S1:      if (x_in)  state <= S2; else state <= S1;
   S2:      if (x_in)  state <= S3; else state <= S2;
   S3:      if (x_in)  state <= S0; else state <= S3;
  endcase

 assign y_out = (state == S3);            // Output of flip-flops
endmodule

// Structural model
module Moore_Model_STR_Fig_5_20 (
 output      y_out, A, B,
 input        x_in, clock, reset
);
 wire         TA, TB;

// Flip-flop input equations
 assign TA = x_in & B;
 assign TB = x_in;
// Output equation
 assign y_out = A & B;

// Instantiate Toggle flip-flops
 Toggle_flip_flop_3 M_A (A, TA, clock, reset);
 Toggle_flip_flop_3 M_B (B, TB, clock, reset);
endmodule

module Toggle_flip_flop (Q, T, CLK, RST_b);
 output      Q;
 input       T, CLK, RST_b;
 reg         Q;

 always @ (posedge CLK, negedge RST_b)
  if (RST_b == 0) Q <= 1'b0;
  else if (T) Q <= ~Q;
endmodule

// Alternative model using characteristic equation
//  module Toggle_flip_flop (Q, T, CLK, RST_b);
//  output   Q;
//  input    T, CLK, RST_b;
//  reg      Q;

//  always @ (posedge CLK, negedge RST)
//   if (RST_b == 0) Q <= 1'b0;
//   else Q <= Q ^ T;
//  endmodule

module t_Moore_Fig_5_20;
 wire        t_y_out_2, t_y_out_1;
 reg         t_x_in, t_clock, t_reset;

Moore_Model_Fig_5_20           M1(t_y_out_1, t_x_in, t_clock, t_reset);
Moore_Model_STR_Fig_5_20       M2 (t_y_out_2, A, B, t_x_in, t_clock, t_reset);

initial #200 $finish;
initial begin
   t_reset = 0;
   t_clock = 0;
   #5 t_reset = 1;
 repeat (16)
   #5 t_clock = ~t_clock;
end
initial begin
     t_x_in = 0;
 #15 t_x_in = 1;
 repeat (8)
  #10 t_x_in = ~t_x_in;
 end
endmodule
```

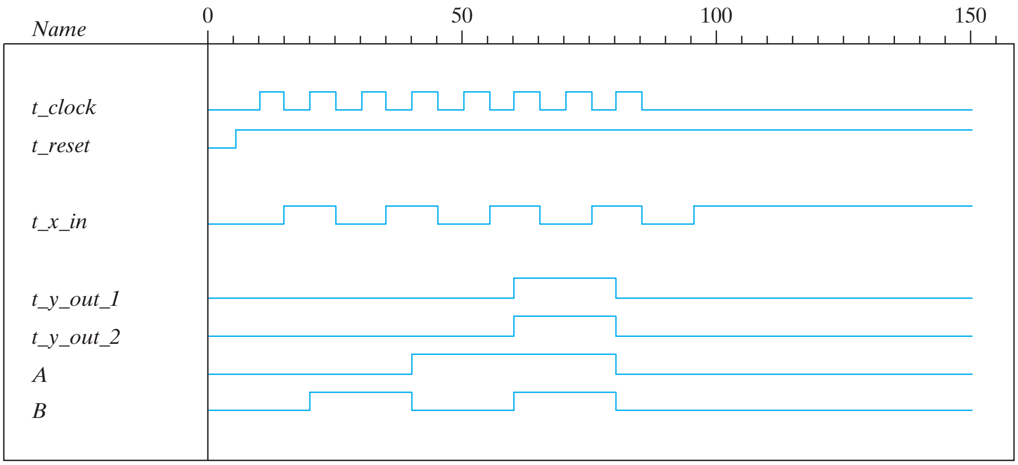

FIGURE 5.24
Simulation output of HDL Example 5.7

5.7 STATE REDUCTION AND ASSIGNMENT

The *analysis* of sequential circuits starts from a circuit diagram and culminates in a state table or diagram. The *design* (synthesis) of a sequential circuit starts from a set of specifications and culminates in a logic diagram. Design procedures are presented in Section 5.8. Two sequential circuits may exhibit the same input–output behavior, but have a different number of internal states in their state diagram. The current section discusses certain properties of sequential circuits that may simplify a design by reducing the number of gates and flip-flops it uses. In general, reducing the number of flip-flops reduces the cost of a circuit.

State Reduction

The reduction in the number of flip-flops in a sequential circuit is referred to as the *state-reduction* problem. State-reduction algorithms are concerned with procedures for reducing the number of states in a state table, while keeping the external input–output requirements unchanged. Since m flip-flops produce 2^m states, a reduction in the number of states may (or may not) result in a reduction in the number of flip-flops. An unpredictable effect in reducing the number of flip-flops is that sometimes the equivalent circuit (with fewer flip-flops) may require more combinational gates to realize its next state and output logic.

We will illustrate the state-reduction procedure with an example. We start with a sequential circuit whose specification is given in the state diagram of Fig. 5.25. In our example, only the input–output sequences are important; the internal states are used merely to provide the required sequences. For that reason, the states marked inside the circles are denoted