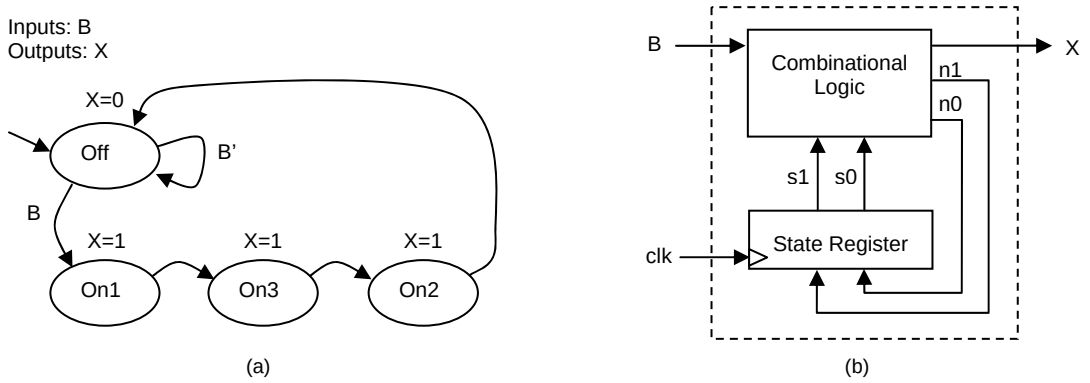
Tutorial: Modeling and Testing Finite State Machines (FSM)

Finite State Machines (FSMs) have been introduced to aid in specifying the behavior of sequential circuits. For example, we want to design part of a laser surgery system such that a surgeon can activate a laser by pressing a button B (B=1). The laser should stay on (X=1) for exactly 3 clock cycles, then shut off (X=0). We want to ensure that the laser will only stay on for 3 clock cycles regardless of whether the button is pressed for multiple clock cycles or if the button is pressed again while the laser is activated. Figure 1(a) illustrates the FSM describing the behavior of the three cycle high laser controller.

To implement the three cycles high laser controller we modify the standard controller architecture for our particular application. Because we have four states, we need a 2-bit state register. Additionally, the FSM accepts input B indicating a button press, and output X controlling the laser. The FSM inputs and outputs are connected to the combinational logic block as shown in Figure 1(b).

Figure 1: Three cycle high laser controller's (a) FSM and (b) corresponding controller architecture.



Modeling FSM in Verilog

We begin modeling the FSM controller architecture by declaring a module, `laser_timer`, and its corresponding interface, as shown in Figure 2. The `laser_timer` module has three inputs (`RST`, `CLK`, `B`) and one output (`X`). It is important to include a reset signal in all sequential circuits. The reset signal is used to indicate the

Figure 2: Verilog Module Declaration

```
module laser_timer(RST, CLK, B, X);

   // user interface
   input RST, CLK;
   input B;
   output X;
   reg X;

   // states
   parameter Off = 2'b00;
   parameter On1 = 2'b01;
   parameter On2 = 2'b10;
   parameter On3 = 2'b11;

   // variables
   reg [1:0] State;
   reg [1:0] NextState;
```

• • •

Figure 3: State Register Process

```
// process modeling state register
always @ (posedge RST or posedge CLK) begin
   if( RST == 1 )
      State <= Off;
   else
      State <= NextState;
end
```

circuit behavior when it is initially powered on. Additionally, we declare several constant values associating state names used in the FSM with their unique user-defined state encoding. The statement `parameter constant_name = value;` assigns the numerical value `value` to a textual name defined by `constant_name`. As we will see later, this step is intended to make it easier for a developer to read and understand the code. In this example, we assign state `Off` the encoding `00`, state `On1` the encoding `01`, state `On2` the encoding `10`, and state `On3` the encoding `11`. Lastly, we declare two variables to keep track of the current state of the FSM and the next state of the FSM, `State` and `NextState` respectively.

The controller architecture is broken down into two subcomponents, a state register and combinational logic. The state register simply keeps track of the current state of the FSM. Figure 3 illustrates the process which models the functionality of the state register. When the circuit first starts up we must specify the initial state. In an FSM, we have a special arrow that indicates the start state. In Verilog we utilize a signal called `RST` indicating that the circuit has just powered on. If `RST` is equal to `1`, we indicate the desired starting behavior. In our case we want to set the initial state to `Off` as indicated by the FSM in Figure 1. Notice, instead of having to set the register to the numerical state encoding value `2'b00` we can simply utilize the state name `Off`. After powering on the circuit, the normal behavior of the state register is to simply hold the current state of the FSM, and update the state on the next clock cycle. We must ensure the state register is only active on reset or on the rising edge of the clock cycle. The sensitivity list in the `always` procedure specifies that the corresponding `if/else` code encompassed in the `always` procedure only occurs on the rising edge of the `RST` signal (`posedge RST`) or the rising edge of the `CLK` signal (`posedge CLK`).

The second sub-component in the controller architecture is the combinational logic. Based on the current state `State` and input `B`, the combinational logic determines the next state `NextState` and the corresponding FSM output `X`. We again specify the `always` procedure's sensitivity list to execute when `State` or `B` changes. Instead

Figure 4: Combinational Logic Process

```
// process modeling combinational circuit
   always @ (State or B) begin
      case( State )
         Off: begin
         // determine next state
            if( B == 1 )
               NextState <= On1;
            else
               NextState <= Off;

            // assign output value
            X <= 0;
         end
         On1: begin
            // determine next state
            NextState <= On2;

            // assign output value
            X <= 1;
         end
                              • • •
```

of having to create a truth table to implement the combinational logic block and the corresponding low-level logic, we instead model the combinational logic functionality behaviorally utilizing a `case` statement. The `case` statement compares the expression specified within the parenthesis of `case (expression)` to a series of cases specified by `case:` and executes the statement or statement group associated with the first matching case. For example, if the value held in `State` is `Off`, then we would check the value of `B`. If `B` is equal to `1`, then we would perform the assignment `NextState <= On1;` indicating that on the next clock cycle the FSM will be in the `On1` state. If `B` is equal to `0`, then we would perform the assignment `NextState <= Off;` indicating that on the next clock cycle the FSM will remain in the `Off` state. In addition to determining the next state, we must also specify an output value. In the `Off` state, output `X` is assigned a value of `0` as shown in Figure 4. Consider instead if the value held in `State` is `On1`, regardless of the input value `B` the next state as indicated by the FSM is `On2`. Thus the within the `On1:` case we perform the assignment `NextState <= On2;`. Again, we must also specify an output value associated with the state and assign `X` a value of `1`. We similarly model the remaining FSM states.

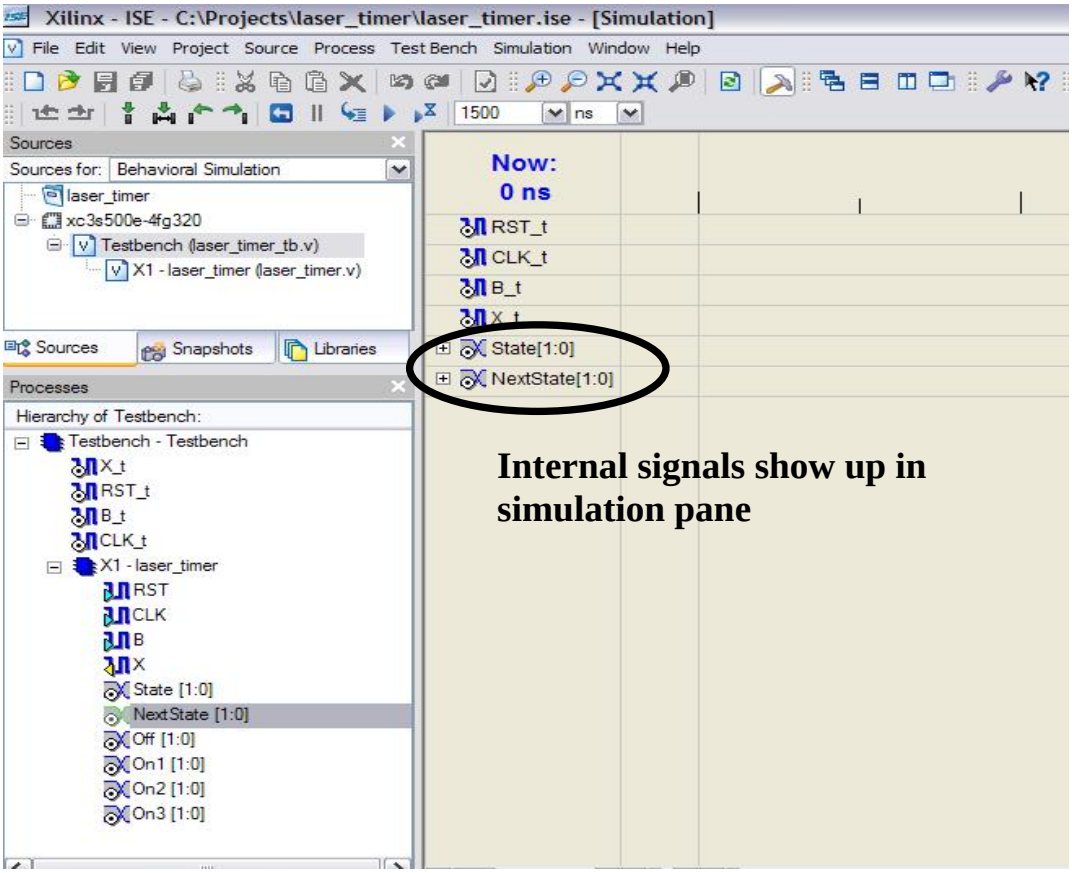
Testing the FSM

To test the behavior of the FSM we construct a testbench. The testbench is a self contained module and contains no input or output ports. We instantiate an instance of the `laser_timer` circuit and map the corresponding inputs and outputs of `laser_timer` to registers and wires declared within the testbench. Unlike the combinational circuits we previously modeled sequential circuits require a clock. Thus, we must also model a clock within the testbench as shown in Figure 5. There is no sensitivity list associated with the `always` procedure, thus the code block associated with this procedure continuously executes during the simulation of the testbench. We first assign the `CLK_t` a value of `1`, wait for 25 time units (where the time unit is specified within the simulator), then assign the `CLK_t` register a value of `0`, then again wait for 25 time units. This procedure creates the clock signal.

Figure 5: FSM Testbench

```
module Testbench;
   reg RST_t, CLK_t, B_t;
   wire X_t;

   laser_timer X1(RST_t, CLK_t, B_t, X_t);

   // model clk
   always
   begin
      CLK_t <= 0;
      #25;
      CLK_t <= 1;
      #25;
   end

   // specify input to laser_timer
   initial
   begin
      // reset
      RST_t <= 1;
      B_t <= 0;
      #50;

      // don't forget to turn reset off
      RST_t <= 0;
      #50;

      // vary input value of button

      • • •
```

Figure 6: Simulating internal signals.

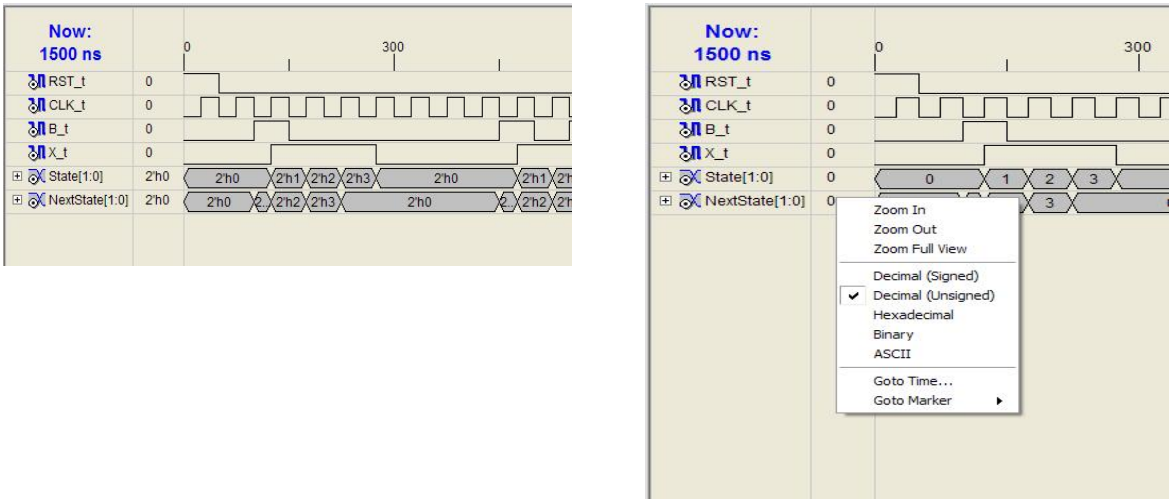


The `initial` procedure executes the code block associated with this procedure once upon execution of the testbench. Similar to previous labs, the input values are assigned various values to determine the corresponding FSM behavior. However, before the FSM's functionality can be tested the FSM needs to have an initial state. The reset signal is utilized to indicate the circuit has been powered on, thus we first set the reset signal to `1`. If reset if left as `1`, the circuit will remain in the reset state and will not perform its "normal" operation, thus after a short delay we then set the reset signal to `0`. Afterwards, we can test the corresponding FSM behavior for different input scenarios by varying the value and duration of the button input `B`.

When simulating FSMs it can be useful to trace internal signals such as the current and next state values. To trace internals signals,

- Expand the testbench entity in the "Processes" window (on the left) by clicking on the "+" symbol.
- Expand the `laser_timer` module in the "Processes" window by clicking on the "+" symbol. You should now be able to see all signals declared in the `laser_timer` module.
- Find the `State` and `NextState` signals. Click on each signal and drag it to the simulation pane. These signals should now appear in the simulation pane as shown in Figure 6.
- You can now run the simulation and the corresponding values for these internal signals should appear, as shown in Figure 7.
- By default these values will appear in hexadecimal. To change the display settings, right-click on the signal and choose the "Decimal (Unsigned)" option. The values should now appear in decimal (0 is the `Off` state, 1 is `On1` state, 2 is `On2` state, and 3 is `On3` state as specified by the state encodings).

Figure 7: Testbench simulation with internal state signals in default display setting changed to display as unsigned decimal values.



Try it out

The corresponding FSM and testbench code is available. Download and simulate the code to make sure you understand how the FSM model works. Consider the following scenarios:

- What happens when we vary the input values of B without toggling the reset signal?
- What happens when we vary the inputs values of B while reset remains at a value of 1?
- What happens to output X when the button is pressed for multiple cycles?
- What happens to output X when the button is pressed again before the three high cycles are finished?