Lab EE 271: Circuit Theory

Winter 2018

Lab 4

Shawnna Cabanday

Professor Rania Hussein

Lab Section: Tuesday from 1:15 - 3:15

Date submitted: February 11, 2018

Updated 10:35 2/11/2018

Abstract

The purpose of this lab was to create sequential circuits using System Verilog. The first task of this lab was to download a premade design to the FPGA. The second task involved designing a second circuit that simulated the hazard lights with provided patterns.

Introduction

Digital logic circuits handle data encoded in binary form through signals that have only two values, 0 and 1. This binary logic that deals with determining whether or not input conditions evaluate to an output of either "true" and "false" allows scientists and engineers to develop complex digital logic circuits and computers that can be built using a few types of basic circuits called gates, each performing a single elementary logic operation. The purpose of this lab is to practice using System Verilog to create sequential circuits. A System verilog program will be created based on simplified expressions and truth tables, which will be verified using generated waveforms by ModelSim and then downloaded to a Field Programmable Gate Array (FPGA) to demonstrate the physical program. The first task of this lab was to download a premade design to the FPGA that outputted true when the clock received two true inputs in a row. The second task involved designing a second circuit that simulated the hazard lights with provided patterns.

Materials

- Altera Quartus II Lite Program
- ModelSim Lite Program
- (5CSEMA5F31) DE1-SoC FPGA Development Board
- Ethernet to USB Cable
- Power cord for FPGA

Procedures

Part 1: Mapping Sequential Logic to the FPGA

The first task of this lab was to download a premade design to the FPGA that outputted true when the clock received two true inputs in a row. A module titled "simplelogic" was created as the primary driver code that would be instantiated to other testing modules, such as the testbench module and the FPGA module. The code for the "simplelogic" module has 3 inputs labeled "w," "clk," and "reset", and 1 output called "out." The next state logic was developed using an always block with switch case statements with the next state represented as "ns" and the present state represented as "ps." From there, the output logic was assigned to always because to the present state equal to C, and the output was analyzed according to the positive edge clock which was coded in a separate always_ff block.

To simulate the logic, the clock was specified by instantiating the simplelogic module into another module called "testbench". The test bench module for the circuit design sets up the clock by initializing the parameter clock period to 100, creating a separate line of code to represent each clock cylce, and placing those lines of code within an initial begin block. The simulation is ended using $stop. After the System Verilog programs were compiled and the circuit, the design's generated output was then simulated in ModelSim to ensure that the Sequential Logic matched the expected output.

The design was then set up to run on the FPGA using a separate module titled "DE1_SoC". This module provides a clock to the circuit that generates slower clocks so that the changing outputs of the LEDs over time would be more visible. In the code, which clock is selected using "whichClock" and is it set to 25 to yield a clock with a cycle time of a bit over a second, with 24 being twice as fast, and 26 being twice as slow. As a brief overview of the DE1_SoC module code, the inputs include the 50MHz clock named "CLOCK_50," the outputs include the HEX displays [0 - 5], the 10 LEDs to be lit up, the 4 presser keys that when pressed represented false, and the 10 switches. The clk is generated based on CLOCK_50 and is calculated using another separate module named "clock_divider," which adjusts the simple FSM to run properly on the board. FSM inputs and outputs are then determined, instantiating with respect to the "simplelogic" module.

Input and output pins were mapped accordingly. Finally, the ".sof" file created from the Verilog script and pin assignment was downloaded to the FPGA by powering the board, connecting it to computer with a USB to Ethernet cable, and adding the file to the FPGA data platform.

Part 2: Hazard Lights

The second task involved designing a second circuit that simulated the hazard lights with provided patterns. Part 2 implements the separate modules utilized in the previous example and applies them to a design problem that aims at displaying proper landing lights at Sea-Tac. The circuit is given two input (SW[0] and SW[1]) which indicate wind direction and there are three output lights that will display according to the sequence of lights. Table 2.1 illustrates the proper pattern for each inputted switch sequence input (00, 01, and 10). Note that for each situation, the lights must cycle through the given pattern and if the inputs are changed, the pattern must be able to toggle to the next corresponding pattern for the corresponding input. For instance, if the wind is calm, the lights cycle between the outside lights being lit and the center light being lit, repetitively. Additionally, the right to left and left to right crosswind indicators cycle repeatedly through three patterns each, one moving from right to left or left to right, respectively.

First, a state diagram was created to identify the behavior of the circuit (Figure 2.1) Next, a state table was created based on the state diagram (Table 2.2). To implement the next state logic in Verilog, the module hazardlights was created with inputs clk, reset, IN, and OUT; the IN permitting up to 2 bits and the OUT allowing up to 3 bits. State variables were enumerated as A, B, C, and D with each state corresponding to 101, 001, 010, and 100 respectively. Within the module "hazardlights", next state logic was created using case statements in combination with if-else statements, all coded within a "always_comb" block. Resulting outputs for each LEDR were assigned based on the next state variables. In addition the next state logic, a test bench was implemented to simulate the outputs in ModelSim. The resulting waveforms were generated and evaluated based on the state diagram tables.

The design was then set up to run on the FPGA using a separate module titled "DE1_SoC". This module provides a clock to the circuit that generates slower clocks so that the changing outputs of the LEDs over time would be more visible and is similar to the one created in the Systematic Verilog code for part 1.

Input and output pins were mapped accordingly. Finally, the ".sof" file created from the Verilog script and pin assignment was downloaded to the FPGA by powering the board, connecting it to computer with a USB to Ethernet cable, and adding the file to the FPGA data platform. Finally, the size of the FSM was computed based on "LC Combinationals" and "LC Registers" relative to the clock_divider line.

Table 2.1: Relative Sequence Input and Output

SW[1]	SW[0]	Meaning	Pattern (LEDR[2:0])
0	0	Calm	101 010
0	1	Right to Left	001 010 100
1	0	Left to Right	100 010 001

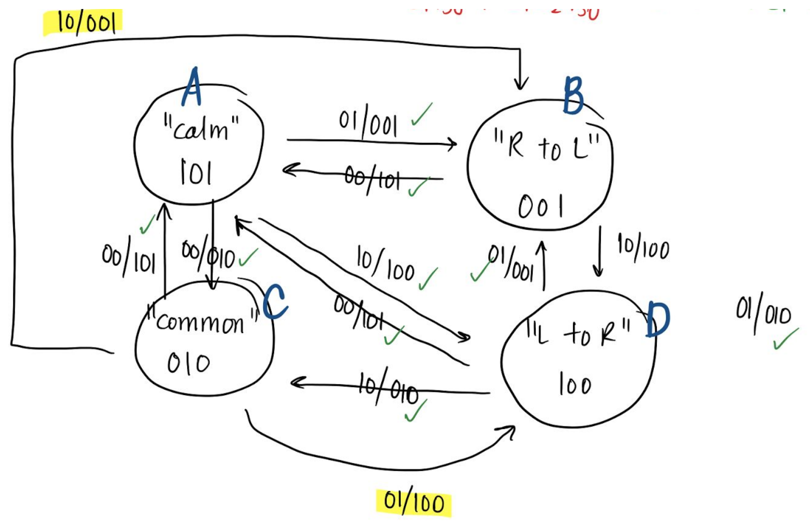


Figure 2.1: State Diagram for Hazard Lights

Table 2.2: State Diagram Table for Hazard Lights

IN[1]	IN[0]	Present State	Present State Letter	Next State	Present State Letter	OUT[2]	OUT[1]	OUT[0]
0	**0**	101	A	010	C	0	1	0
		010	C	101	A	1	0	1
0	**1**	001	B	010	C	0	1	0
		100	D	001	B	0	0	1
		101	A	001	B	0	0	1
1	**0**	100	D	101	A	1	0	1
		100	D	010	C	0	1	0
		010	C	001	B	0	0	1
		001	B	100	D	1	0	0

Analysis

There were several errors that occured while conducting both part 1 and part 2 of this lab. For the first portion, syntax errors in systematic code required intense troubleshooting. File paths to the ModelSim Altera program also had to be fixed in order for the program to run properly in the application. After extensive troubleshooting for procedure 1, I was able to attain results that properly demonstrated the behavior of the logic for the required design circuit. Figure 3.1 illustrates the generated waveforms.

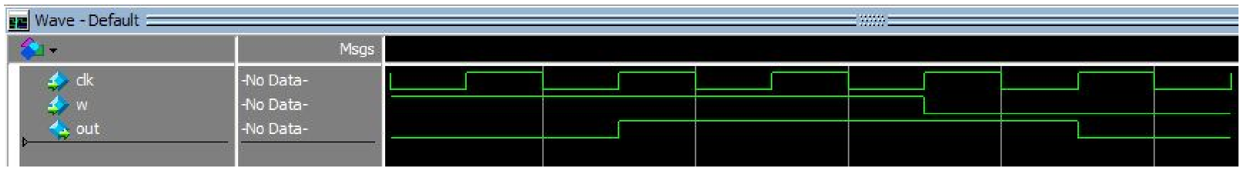


Figure 3.1: Part 1 "Simplelogic" Generated Waveforms

As for the second part of the lab, syntax and logic errors in the systematic were intense. I was able to attain results that properly demonstrated the behavior of the logic for the required design circuit (Figure 3.2), however, these results would not successfully transfer to the FPGA board through instantiation of the De1-SoC code to the top entity module "hazardlights." Figure 3.3 indicates one of the many errors that resulted from improper instantiation. Extensive troubleshooting over several hours was completed to instantiate the De1-SoC module correctly to the top entity module "hazardlights." The lights would flicker in what appeared to be the pattern required, but in a combination that was way too fast for the eyes to determine if it was correct or not. For instance, if both switches were set to an input of 00, it appeared that the pattern flicked from outwards to in, but it was flickering way too fast to completely indicate the pattern. The clock was changed to mitigate the propagation of output, however, there was no significant change in the toggling of lights. Again, with switches set to an input 01, the LEDs would appear to shift in pattern from right to left, but it was too quick to confirm the pattern confidently. I was unable to figure out this issue for the longest time and realized that I had kept my top entity as ***hazardlights*** the entire time. After switching the top entity to DE1-SoC module and reassigning the pin numbers, I was able to successfully load the program onto my FPGA and the lights were correctly toggling. When the input was switched to 00, it would toggle from center to outward lights. When the input was switched to 01, the pattern would toggle from right to left, and when the input was switched to 10, the pattern would toggle to left to right. An input of 11 would provide a solid of pattern of the outermost lights to indicate an issue.

Finally, the size of my FMS was computed. The registers outside the parenthesis was determined to be (31+28)-(26+26) = 7. Ultimately, I used 7 total DFF resources and FPGA logic resources.

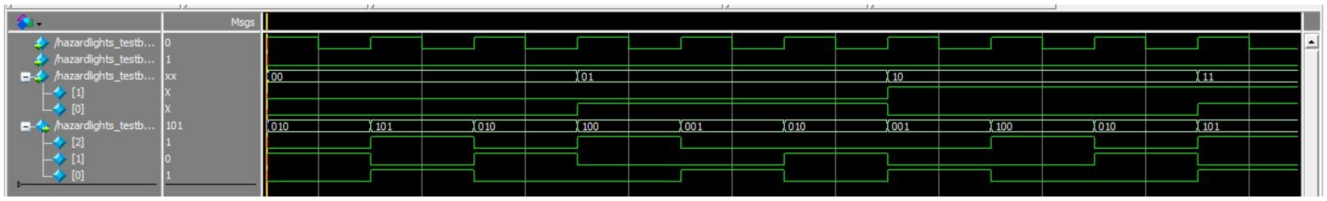
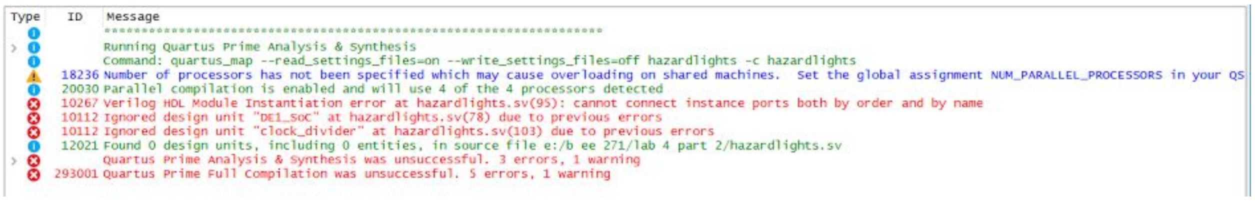
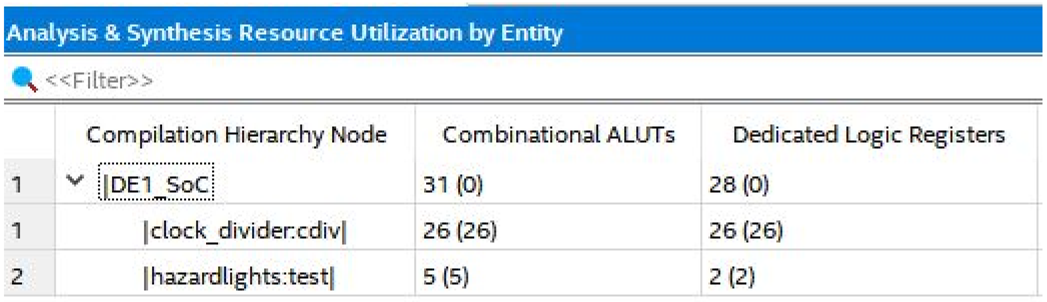


Figure 3.2: Part 2 "Hazardlights" Generated Waveforms



```
Type  ID     Message
             ************************************************************************
             Running Quartus Prime Analysis & Synthesis
             Command: quartus_map --read_settings_files=on --write_settings_files=off hazardlights -c hazardlights
      18236  Number of processors has not been specified which may cause overloading on shared machines.  Set the global assignment NUM_PARALLEL_PROCESSORS in your QS
      20030  Parallel compilation is enabled and will use 4 of the 4 processors detected
      10267  Verilog HDL Module Instantiation error at hazardlights.sv(95): cannot connect instance ports both by order and by name
      10112  Ignored design unit "DE1_SoC" at hazardlights.sv(78) due to previous errors
      10112  Ignored design unit "clock_divider" at hazardlights.sv(103) due to previous errors
      12021  Found 0 design units, including 0 entities, in source file e:/b ee 271/lab 4 part 2/hazardlights.sv
             Quartus Prime Analysis & Synthesis was unsuccessful. 3 errors, 1 warning
     293001  Quartus Prime Full Compilation was unsuccessful. 5 errors, 1 warning
```

Figure 3.3: Part 2 "Hazardlights" Error Message



Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers
1	|DE1_SoC|	31 (0)	28 (0)
1	|clock_divider:cdiv|	26 (26)	26 (26)
2	|hazardlights:test|	5 (5)	2 (2)

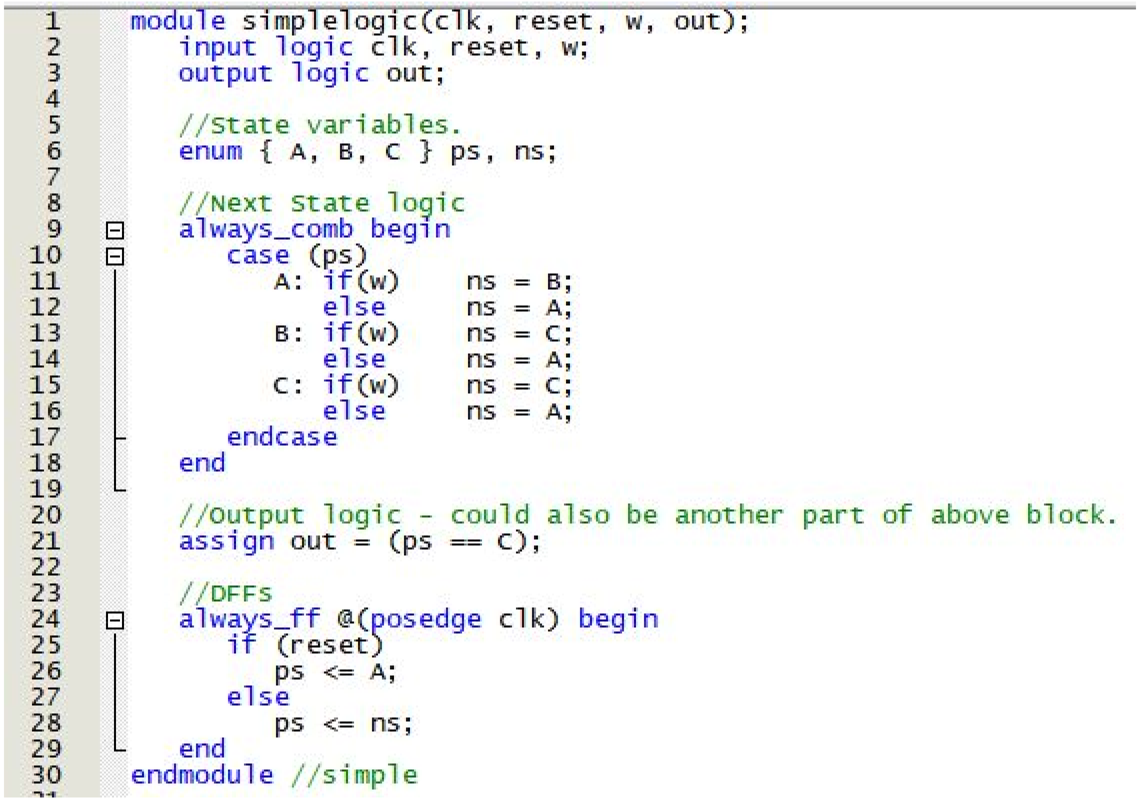
Figure 3.4: Part 2 "DE1-Soc" FMS Screenshot

Conclusion

The use of computer aided simulations in Quartus, ModelSim, and Systematic Verilog programming allows for increased efficiency in the steps and processes needed to develop and verify digital logic circuits on the FPGA. Debugging and determining the major sources for issues proved to be an extremely long process for all simulations and Systematic Verilog programming. It should be noted that future labs should be attempted and designed way ahead of time. Additionally, state diagrams proved to helpful with understanding the changing behavior of circuit through logical steps.

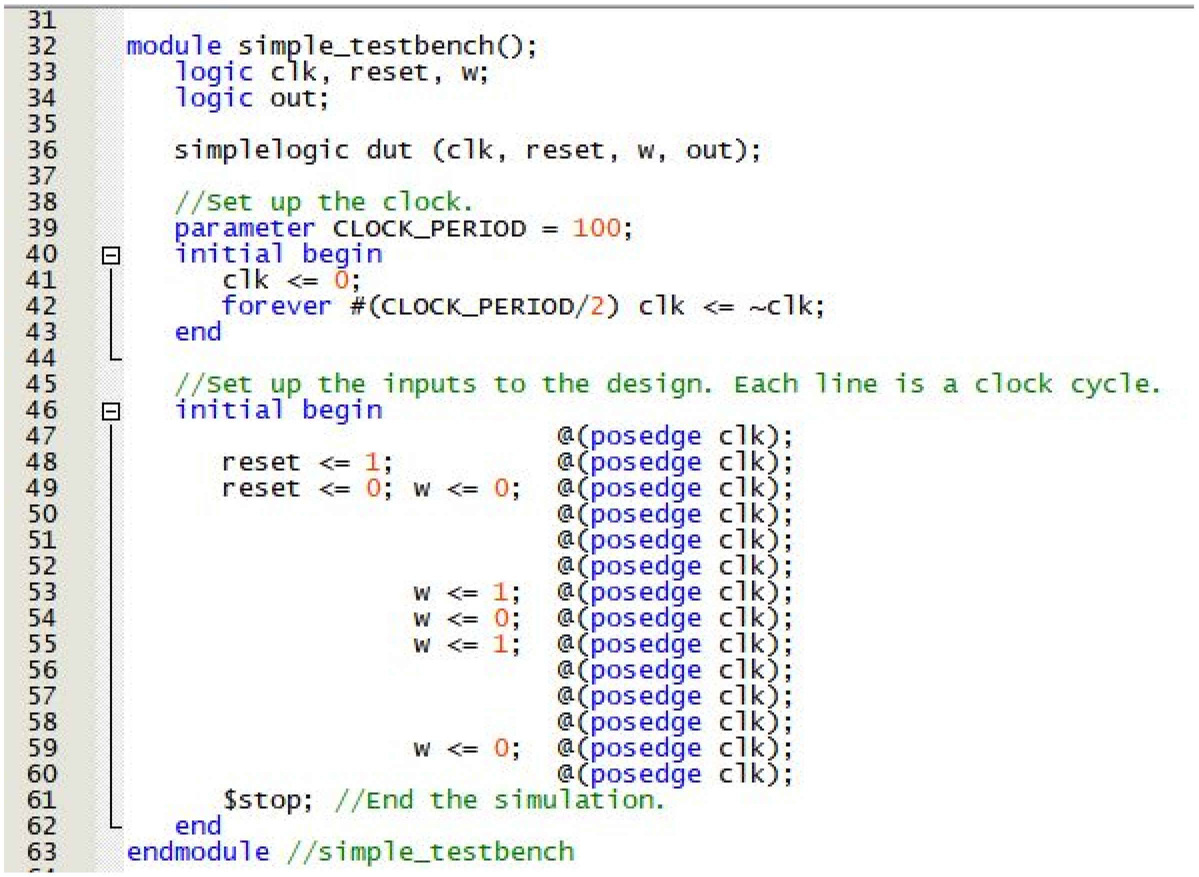
Appendix

1. Example of Code 1: Driver Code for “Simple Logic”
2. Example of Code 2: Testbench for “Simple Logic”
3. Example of Code 3: DE1_SoC FPGA Code for “Simple Logic”
4. Example of Code 4: Driver Code for “Hazardlights”
5. Example of Code 5: Testbench for “Hazardlights”
6. Example of Code 6: DE1_SoC FPGA Code for “Hazardlights”



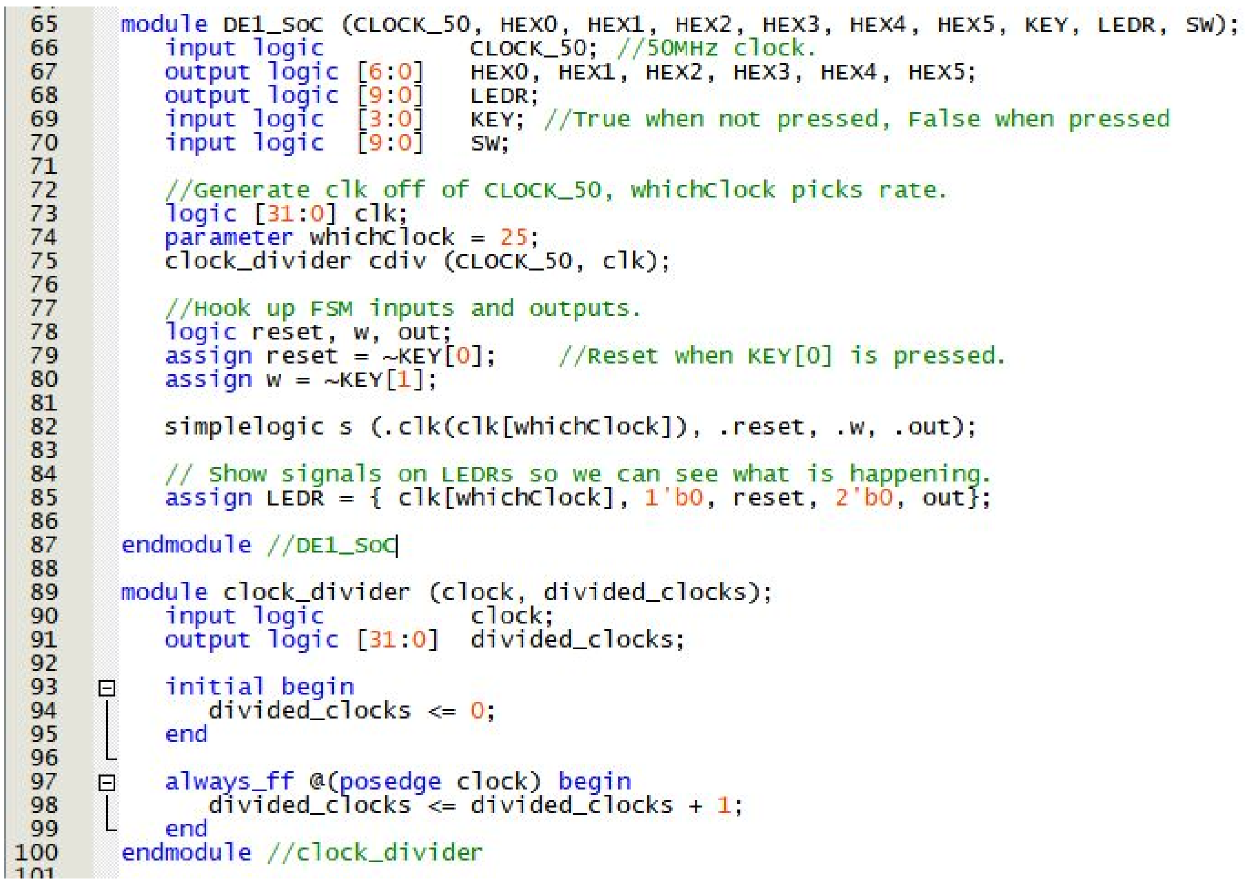
```
module simplelogic(clk, reset, w, out);
   input logic clk, reset, w;
   output logic out;

   //State variables.
   enum { A, B, C } ps, ns;

   //Next State logic
   always_comb begin
      case (ps)
         A: if(w)    ns = B;
            else     ns = A;
         B: if(w)    ns = C;
            else     ns = A;
         C: if(w)    ns = C;
            else     ns = A;
      endcase
   end

   //Output logic - could also be another part of above block.
   assign out = (ps == C);

   //DFFs
   always_ff @(posedge clk) begin
      if (reset)
         ps <= A;
      else
         ps <= ns;
   end
endmodule //simple
```

Example of Code 1: Driver Code for “Simple Logic”



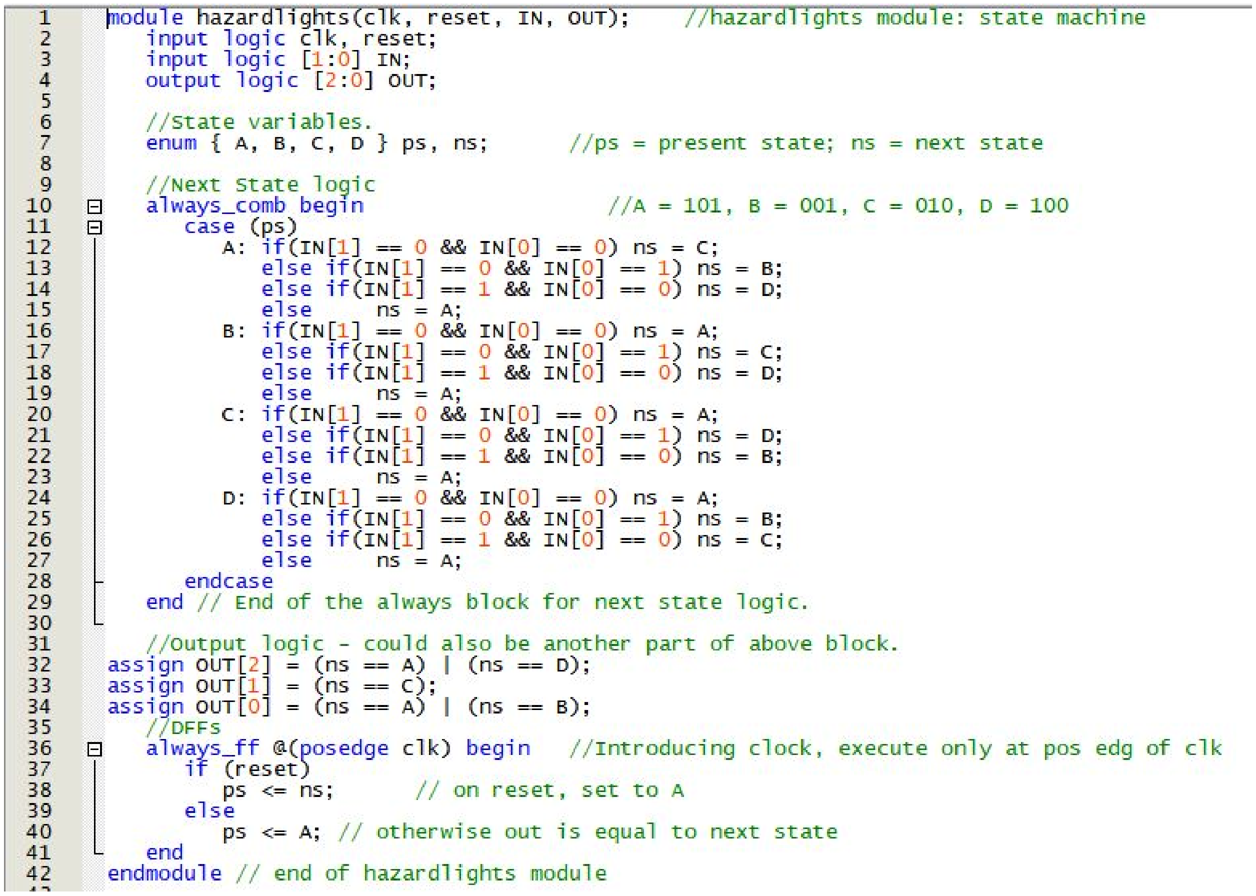
```
module simple_testbench();
   logic clk, reset, w;
   logic out;

   simplelogic dut (clk, reset, w, out);

   //Set up the clock.
   parameter CLOCK_PERIOD = 100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   //Set up the inputs to the design. Each line is a clock cycle.
   initial begin
                                @(posedge clk);
      reset <= 1;               @(posedge clk);
      reset <= 0; w <= 0;       @(posedge clk);
                                @(posedge clk);
                                @(posedge clk);
                                @(posedge clk);
                  w <= 1;       @(posedge clk);
                  w <= 0;       @(posedge clk);
                  w <= 1;       @(posedge clk);
                                @(posedge clk);
                                @(posedge clk);
                                @(posedge clk);
                  w <= 0;       @(posedge clk);
                                @(posedge clk);
      $stop; //End the simulation.
   end
endmodule //simple_testbench
```

Example of Code 2: Testbench for “Simple Logic”



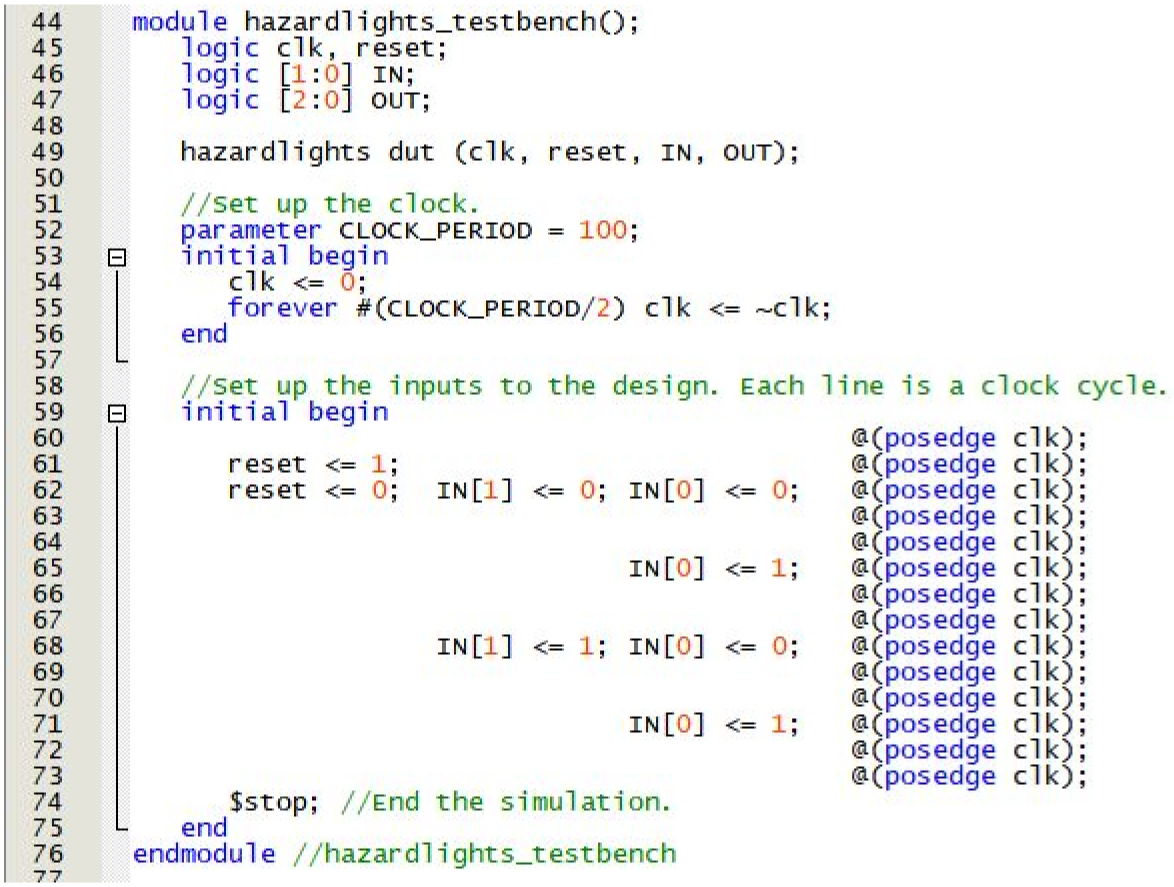
```
module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);
   input logic          CLOCK_50; //50MHz clock.
   output logic [6:0]   HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   output logic [9:0]   LEDR;
   input logic  [3:0]   KEY; //True when not pressed, False when pressed
   input logic  [9:0]   SW;

   //Generate clk off of CLOCK_50, whichClock picks rate.
   logic [31:0] clk;
   parameter whichClock = 25;
   clock_divider cdiv (CLOCK_50, clk);

   //Hook up FSM inputs and outputs.
   logic reset, w, out;
   assign reset = ~KEY[0];    //Reset when KEY[0] is pressed.
   assign w = ~KEY[1];

   simplelogic s (.clk(clk[whichClock]), .reset, .w, .out);

   // Show signals on LEDRs so we can see what is happening.
   assign LEDR = { clk[whichClock], 1'b0, reset, 2'b0, out};

endmodule //DE1_SoC

module clock_divider (clock, divided_clocks);
   input logic          clock;
   output logic [31:0]  divided_clocks;

   initial begin
      divided_clocks <= 0;
   end

   always_ff @(posedge clock) begin
      divided_clocks <= divided_clocks + 1;
   end
endmodule //clock_divider
```

Example of Code 3: DE1_SoC FPGA Code for "Simple Logic"



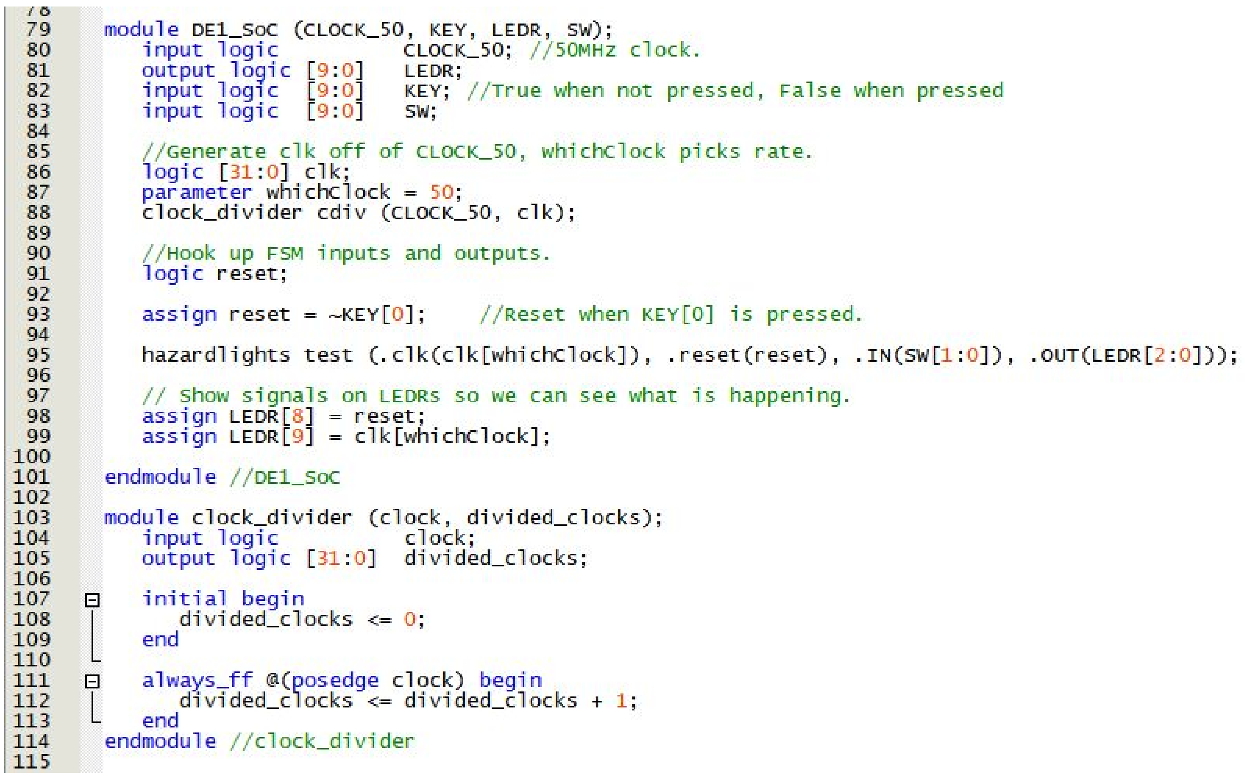
```
module hazardlights(clk, reset, IN, OUT);    //hazardlights module: state machine
   input logic clk, reset;
   input logic [1:0] IN;
   output logic [2:0] OUT;

   //State variables.
   enum { A, B, C, D } ps, ns;      //ps = present state; ns = next state

   //Next State logic
   always_comb begin                   //A = 101, B = 001, C = 010, D = 100
      case (ps)
         A: if(IN[1] == 0 && IN[0] == 0) ns = C;
            else if(IN[1] == 0 && IN[0] == 1) ns = B;
            else if(IN[1] == 1 && IN[0] == 0) ns = D;
            else     ns = A;
         B: if(IN[1] == 0 && IN[0] == 0) ns = A;
            else if(IN[1] == 0 && IN[0] == 1) ns = C;
            else if(IN[1] == 1 && IN[0] == 0) ns = D;
            else     ns = A;
         C: if(IN[1] == 0 && IN[0] == 0) ns = A;
            else if(IN[1] == 0 && IN[0] == 1) ns = D;
            else if(IN[1] == 1 && IN[0] == 0) ns = B;
            else     ns = A;
         D: if(IN[1] == 0 && IN[0] == 0) ns = A;
            else if(IN[1] == 0 && IN[0] == 1) ns = B;
            else if(IN[1] == 1 && IN[0] == 0) ns = C;
            else     ns = A;
      endcase
   end // End of the always block for next state logic.

   //Output logic - could also be another part of above block.
assign OUT[2] = (ns == A) | (ns == D);
assign OUT[1] = (ns == C);
assign OUT[0] = (ns == A) | (ns == B);
   //DFFs
   always_ff @(posedge clk) begin   //Introducing clock, execute only at pos edg of clk
      if (reset)
         ps <= ns;      // on reset, set to A
      else
         ps <= A; // otherwise out is equal to next state
   end
endmodule // end of hazardlights module
```

Example of Code 4: Driver Code for "Hazardlights"



```
module hazardlights_testbench();
   logic clk, reset;
   logic [1:0] IN;
   logic [2:0] OUT;

   hazardlights dut (clk, reset, IN, OUT);

   //Set up the clock.
   parameter CLOCK_PERIOD = 100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   //Set up the inputs to the design. Each line is a clock cycle.
   initial begin
                                                    @(posedge clk);
      reset <= 1;                                   @(posedge clk);
      reset <= 0;  IN[1] <= 0; IN[0] <= 0;          @(posedge clk);
                                                    @(posedge clk);
                                                    @(posedge clk);
                               IN[0] <= 1;          @(posedge clk);
                                                    @(posedge clk);
                                                    @(posedge clk);
                   IN[1] <= 1; IN[0] <= 0;          @(posedge clk);
                                                    @(posedge clk);
                                                    @(posedge clk);
                               IN[0] <= 1;          @(posedge clk);
                                                    @(posedge clk);
                                                    @(posedge clk);
      $stop; //End the simulation.
   end
endmodule //hazardlights_testbench
```

Example of Code 5: Testbench for "Hazardlights"



```
module DE1_SoC (CLOCK_50, KEY, LEDR, SW);
   input logic          CLOCK_50; //50MHz clock.
   output logic [9:0]   LEDR;
   input logic  [9:0]   KEY; //True when not pressed, False when pressed
   input logic  [9:0]   SW;

   //Generate clk off of CLOCK_50, whichClock picks rate.
   logic [31:0] clk;
   parameter whichClock = 50;
   clock_divider cdiv (CLOCK_50, clk);

   //Hook up FSM inputs and outputs.
   logic reset;

   assign reset = ~KEY[0];    //Reset when KEY[0] is pressed.

   hazardlights test (.clk(clk[whichClock]), .reset(reset), .IN(SW[1:0]), .OUT(LEDR[2:0]));

   // Show signals on LEDRs so we can see what is happening.
   assign LEDR[8] = reset;
   assign LEDR[9] = clk[whichClock];

endmodule //DE1_SoC

module clock_divider (clock, divided_clocks);
   input logic          clock;
   output logic [31:0]  divided_clocks;

   initial begin
      divided_clocks <= 0;
   end

   always_ff @(posedge clock) begin
      divided_clocks <= divided_clocks + 1;
   end
endmodule //clock_divider
```

Example of Code 6: DE1_SoC FPGA Code for "Hazardlights"

***Note:** Most of the code has been revised since last updated photos.