Bert Zhao
EE 271

Abstract: The purpose of this final project to to put together everything we have learned about digital logic from this course, and we get to choose our own project to demonstrate this fact.

Introduction: For this final project, I implemented the game Mastermind. This is a code breaking game that involves guessing a secret four-digit code, with each digit being a number in base four. The user inputs their guesses, and after each guess, the game will respond with hints. The user will know how many digits they guessed with the correct location and correct value, and how many digits they guessed with the correct value but wrong location. This lab requires many of the components used from previous labs, such as a LFSR for random code generation and finite state machines to input guesses.

Materials:

- Altera Terasic DE1-SoC Board with power and USB cables
- PC with Altera Quartus Prime Lite installed

Procedure: Mastermind

There are four major modules that control the functionality of the game. The first is a 5-bit LFSR to generate the random code. The flip flops in this module are slightly different from those in past labs in that when reset is true, the LFSR is enabled. Once reset is false, the LFSR is paused so that the code can be read from its random sequence. Each digit of the code is generated with two random bits from the LFSR.

The second module controls the inputs of the user's guesses and enables them to hit KEY3 to submit their guess. It is important to understand the difference between the current guess and the submitted guess. The current guess only depends on the state of the switches, which is a continuous and asynchronous input to the system. The submitted guess takes the current guess and saves it to be compared with the secret code. This module will make that distinction for us.

The third module is guess checker. It will first determine if each digit of our guess is in the correct location. If not, then it will determine if the digit exists, but in another location. Once all the digits have been compared, it will return to us how many digits of our guess are in the correct location, or just the correct value. The game is won when all four digits have been guessed in the correct locations.

The fourth module is a game mode selector. The user can choose to pay a practice game, where the have unlimited guesses, or a real game, where they must find the correct code in 10 guesses.

There are also three major modules that control the displays for the game. The first is a state display that tells the user how many digits of their guess were in the correct location, and how many were just the correct value.

The second major display module is the guess counter. It tells the user how many guesses that have made in the current game. In practice mode, the player gets unlimited guesses, so the counter will go from 0 to 99 before looping back to 0. In a real game, the counter only displays 0 to 9. If the player reaches their tenth guess and does not find the code, the display will read "loser."

The third major display module shows the current guesses. Once a game is over, regardless if the user is in practice mode or a real game, the displays will freeze to show the secret code.

There is an option to choose which set of information gets displays. The user can either have the status of the game be shown, which includes the accuracy of the guess and the number of guesses made. The other option is to have the current guess be displayed. The user can switch between these options at any time in during the game, and after a game is complete.

Results:
The overall game uses switches 9 through 0 and keys 3 through 1 as inputs. SW[9] is reset, SW[8] controls the display options, and SW[7:0] are used for the current guess inputs. KEY[3] is used to submit a guess, while KEY[2] is used to select practice mode and KEY[1] is used to select a real game. The user can switch between practice mode and a real game anytime during the game, but if they have exceeded 10 guesses in practice mode, switching to a real game will result in an immediate loss. A new game is set to practice mode by default. The hex displays are all used to show the desired information. On the status page, HEX4 shows the number of digits in the correct location, HEX2 shows the number of digits in the wrong location but correct value, and HEX1 and HEX0 count the total number of guesses. When a game is won, all inputs are locked except for the display toggle, and status display is frozen and the code is shown. On a loss, the only difference are that the status displays "loser" rather than the status of the game. The second option still displays the code. All major modules were tested using ModelSim and the total resource utilization is (150 - 16) + (43 - 16) = 161.

ModelSim Screenshots:

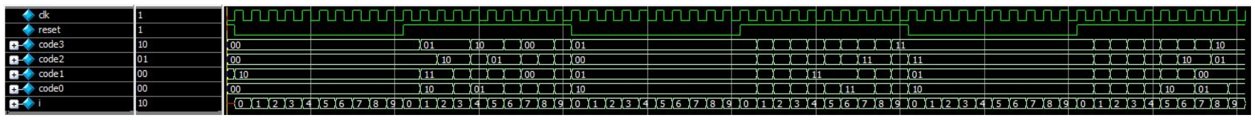


Figure 1: codeGenerator ModelSim screenshot

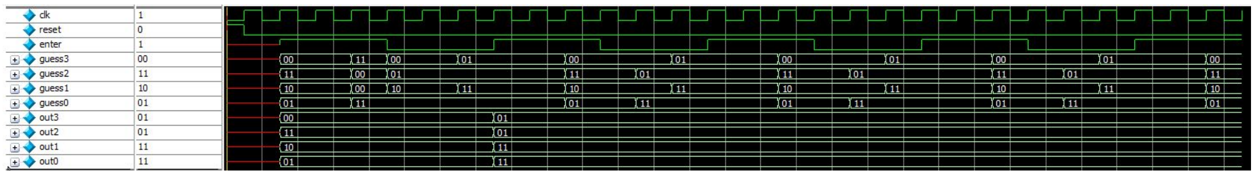


Figure 2: enterGuess ModelSim screenshot

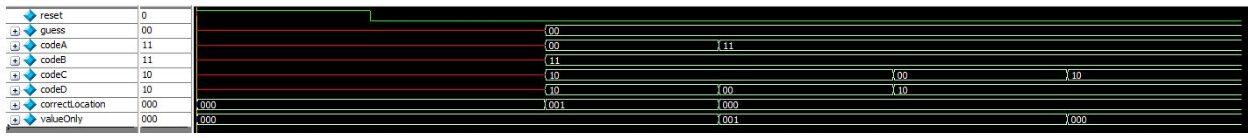


Figure 3: comparator ModelSim screenshot

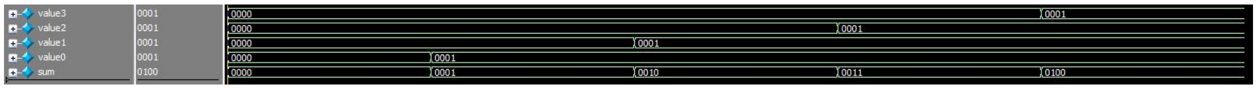


Figure 4: sum ModelSim screenshot

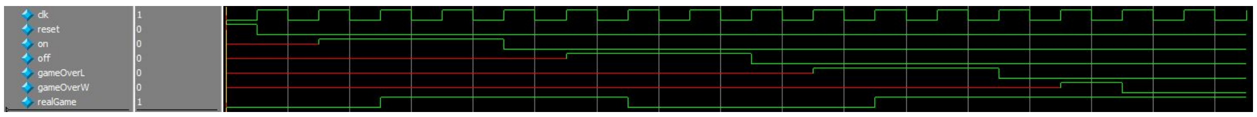


Figure 5: gameMode ModelSim screenshot

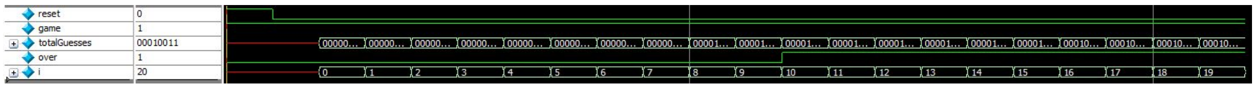


Figure 6: gameLimit ModelSim screenshot

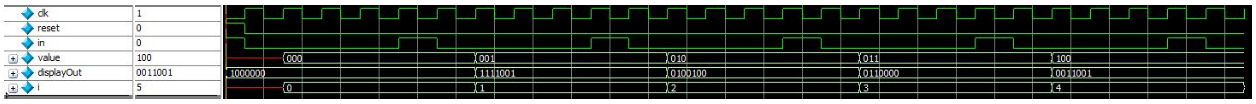


Figure 7: stateDisplay ModelSim screenshot

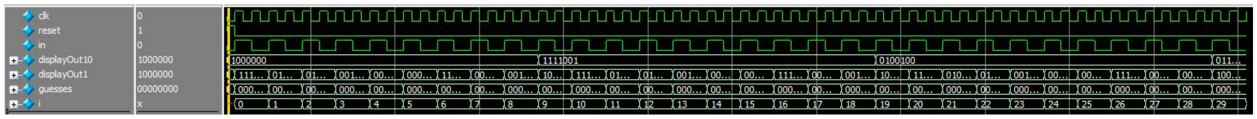
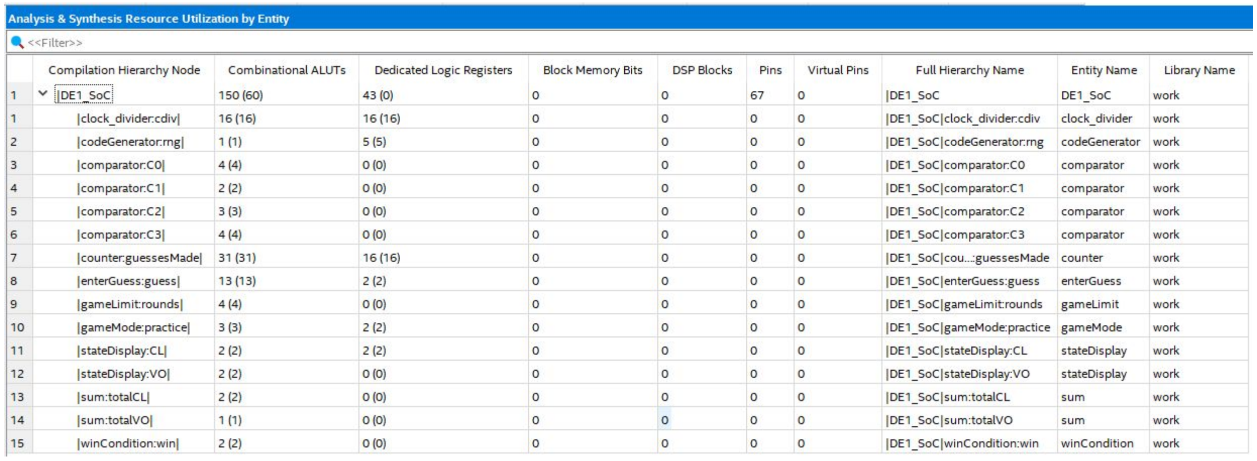


Figure 8: counter ModelSim screenshot

System Cost:



Analysis & Synthesis Resource Utilization by Entity

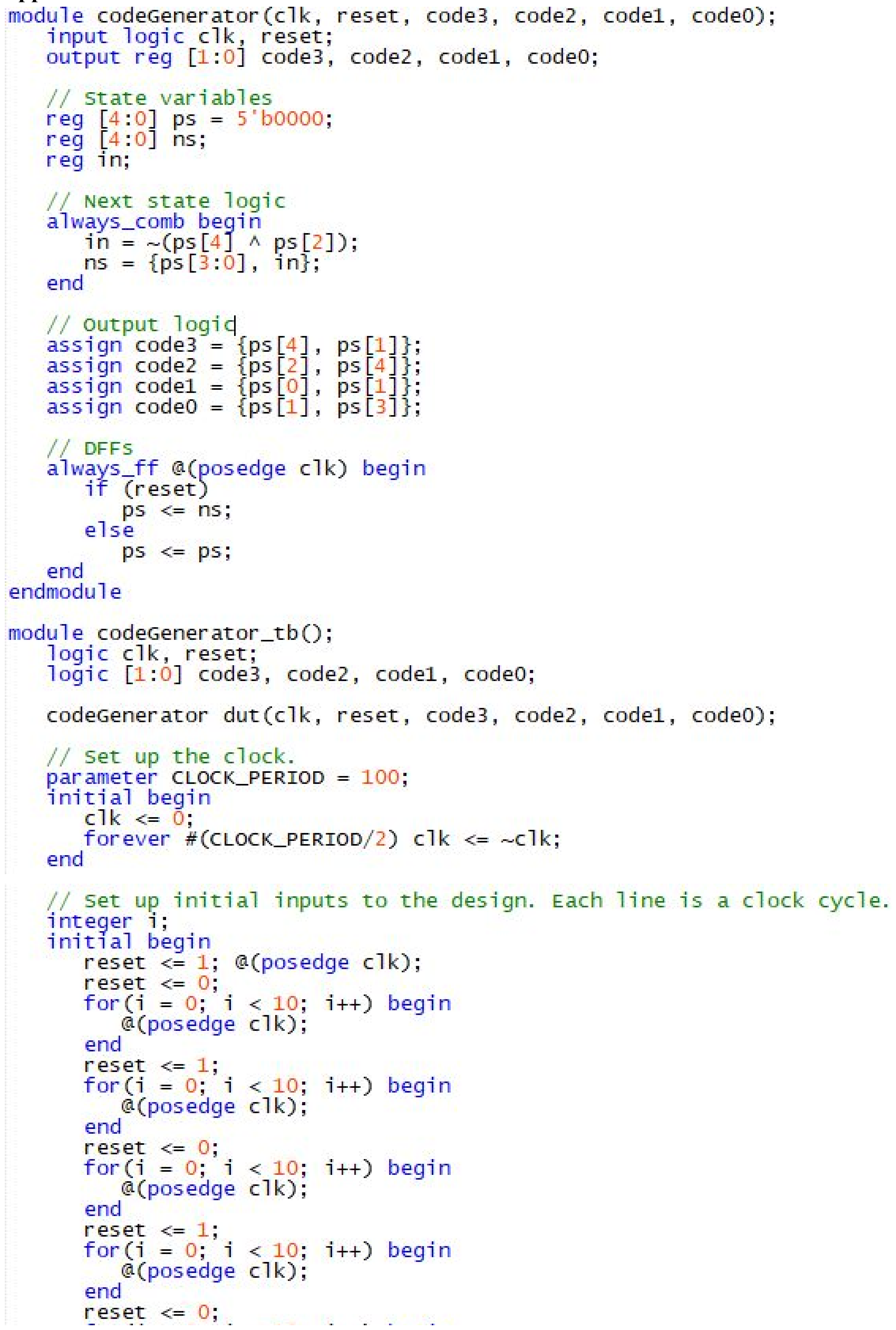
	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins	Full Hierarchy Name	Entity Name	Library Name
1	|DE1_SoC	150 (60)	43 (0)	0	0	67	0	|DE1_SoC	DE1_SoC	work
1	|clock_divider:cdiv|	16 (16)	16 (16)	0	0	0	0	|DE1_SoC|clock_divider:cdiv	clock_divider	work
2	|codeGenerator:rng|	1 (1)	5 (5)	0	0	0	0	|DE1_SoC|codeGenerator:rng	codeGenerator	work
3	|comparator:C0|	4 (4)	0 (0)	0	0	0	0	|DE1_SoC|comparator:C0	comparator	work
4	|comparator:C1|	2 (2)	0 (0)	0	0	0	0	|DE1_SoC|comparator:C1	comparator	work
5	|comparator:C2|	3 (3)	0 (0)	0	0	0	0	|DE1_SoC|comparator:C2	comparator	work
6	|comparator:C3|	4 (4)	0 (0)	0	0	0	0	|DE1_SoC|comparator:C3	comparator	work
7	|counter:guessesMade|	31 (31)	16 (16)	0	0	0	0	|DE1_SoC|cou...:guessesMade	counter	work
8	|enterGuess:guess|	13 (13)	2 (2)	0	0	0	0	|DE1_SoC|enterGuess:guess	enterGuess	work
9	|gameLimit:rounds|	4 (4)	0 (0)	0	0	0	0	|DE1_SoC|gameLimit:rounds	gameLimit	work
10	|gameMode:practice|	3 (3)	2 (2)	0	0	0	0	|DE1_SoC|gameMode:practice	gameMode	work
11	|stateDisplay:CL|	2 (2)	2 (2)	0	0	0	0	|DE1_SoC|stateDisplay:CL	stateDisplay	work
12	|stateDisplay:VO|	2 (2)	0 (0)	0	0	0	0	|DE1_SoC|stateDisplay:VO	stateDisplay	work
13	|sum:totalCL|	2 (2)	0 (0)	0	0	0	0	|DE1_SoC|sum:totalCL	sum	work
14	|sum:totalVO|	1 (1)	0 (0)	0	0	0	0	|DE1_SoC|sum:totalVO	sum	work
15	|winCondition:win|	2 (2)	0 (0)	0	0	0	0	|DE1_SoC|winCondition:win	winCondition	work

Figure 9: Resource utilization by entity screenshot

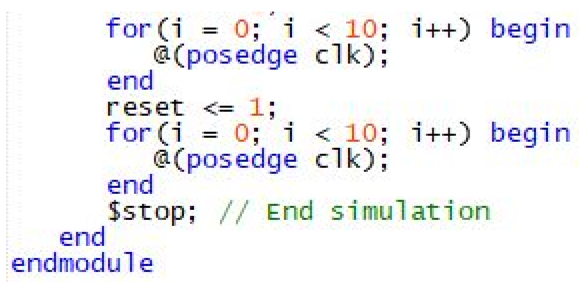
Analysis and Conclusion:

Overall, the final project was an opportunity to combine important concepts of previous labs into one system. Previously topics from both combinational logic and sequential logic were needed to complete this project. There was also a new component learned in this project in that the end was not a completely static system. Rather than having everything frozen at the end, we have a variable display option to check how we did in the game we just completed.

Appendix:

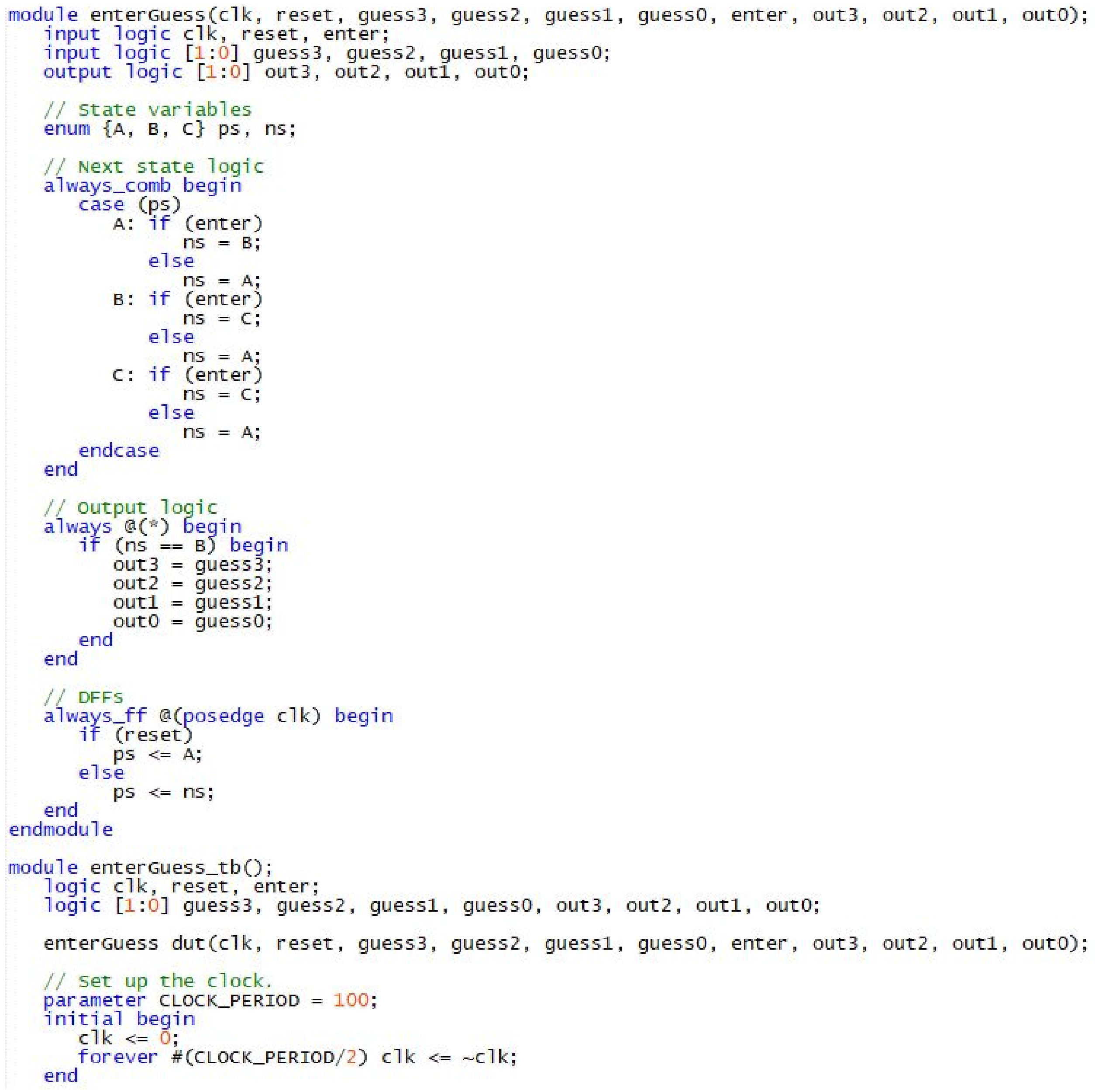


```
module codeGenerator(clk, reset, code3, code2, code1, code0);
   input logic clk, reset;
   output reg [1:0] code3, code2, code1, code0;

   // State variables
   reg [4:0] ps = 5'b0000;
   reg [4:0] ns;
   reg in;

   // Next state logic
   always_comb begin
      in = ~(ps[4] ^ ps[2]);
      ns = {ps[3:0], in};
   end

   // Output logic
   assign code3 = {ps[4], ps[1]};
   assign code2 = {ps[2], ps[4]};
   assign code1 = {ps[0], ps[1]};
   assign code0 = {ps[1], ps[3]};

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

module codeGenerator_tb();
   logic clk, reset;
   logic [1:0] code3, code2, code1, code0;

   codeGenerator dut(clk, reset, code3, code2, code1, code0);

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

   // Set up initial inputs to the design. Each line is a clock cycle.
   integer i;
   initial begin
      reset <= 1; @(posedge clk);
      reset <= 0;
      for(i = 0; i < 10; i++) begin
         @(posedge clk);
      end
      reset <= 1;
      for(i = 0; i < 10; i++) begin
         @(posedge clk);
      end
      reset <= 0;
      for(i = 0; i < 10; i++) begin
         @(posedge clk);
      end
      reset <= 1;
      for(i = 0; i < 10; i++) begin
         @(posedge clk);
      end
      reset <= 0;
```

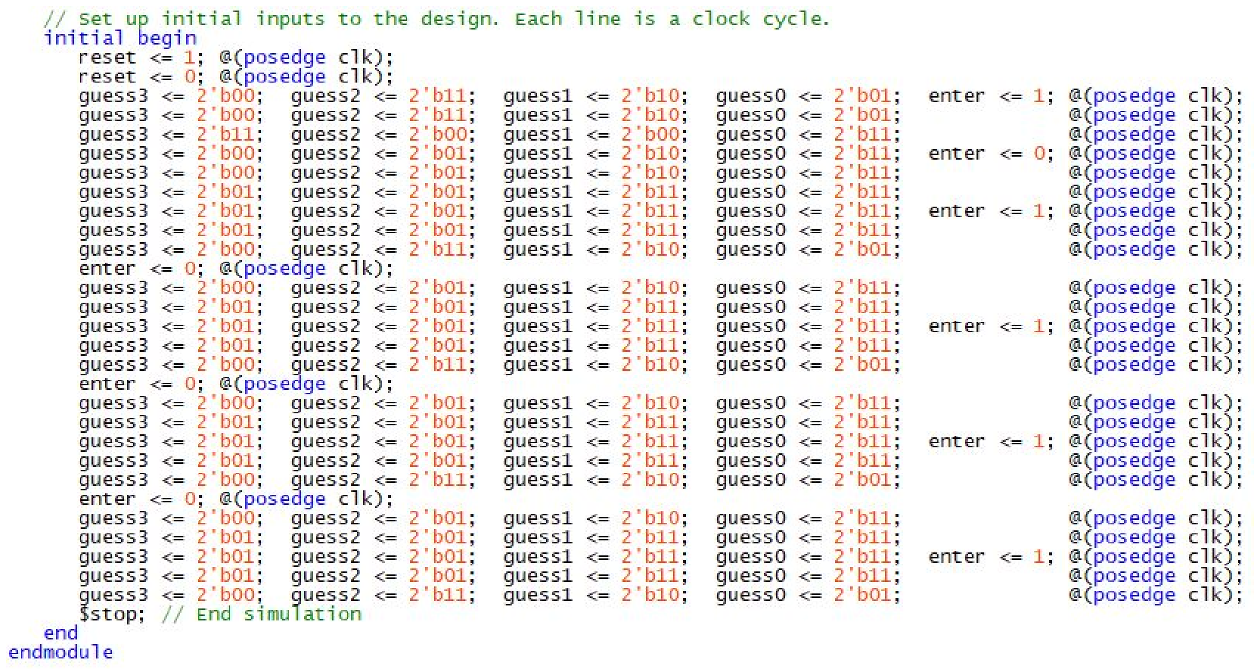


```
      for(i = 0; i < 10; i++) begin
         @(posedge clk);
      end
      reset <= 1;
      for(i = 0; i < 10; i++) begin
         @(posedge clk);
      end
      $stop; // End simulation
   end
endmodule
```

Code 1: codeGenerator.sv

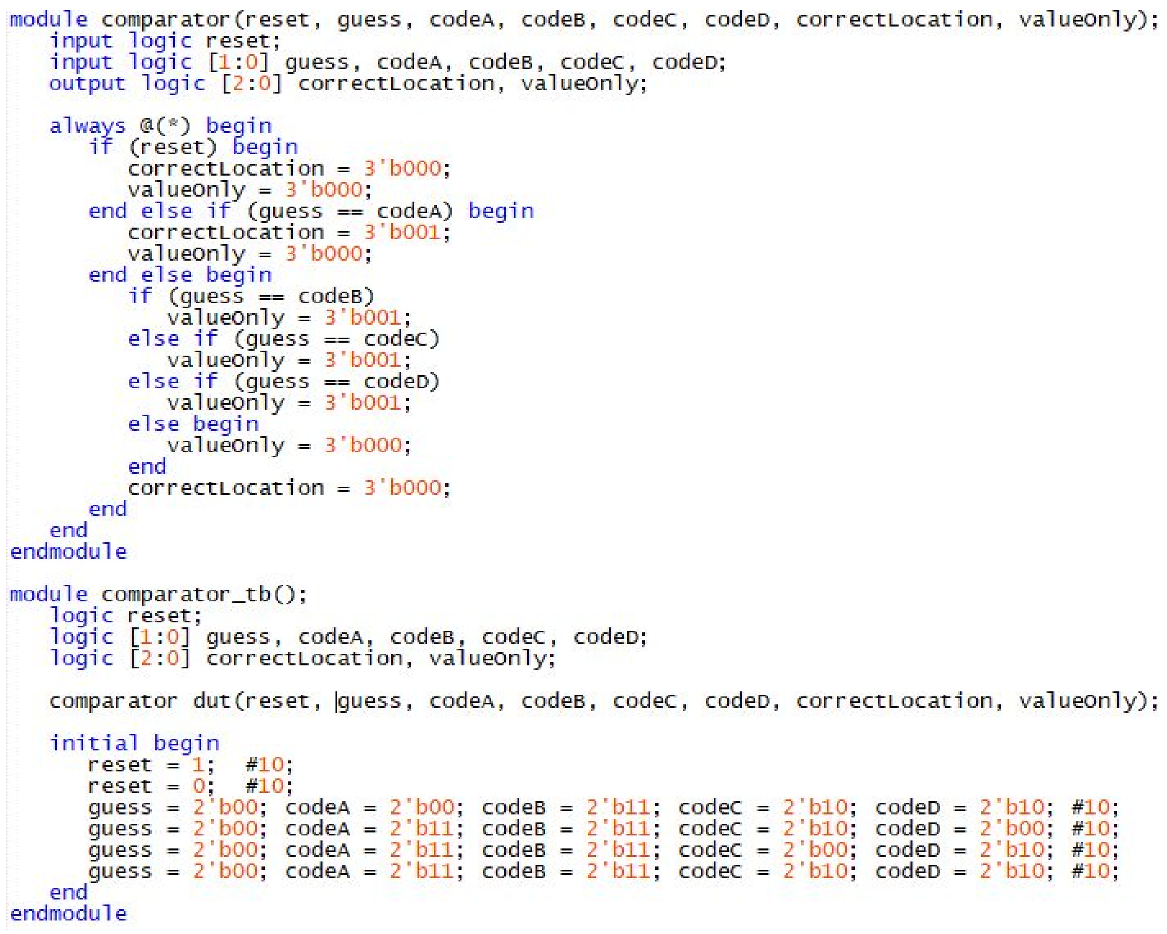


```
module enterGuess(clk, reset, guess3, guess2, guess1, guess0, enter, out3, out2, out1, out0);
   input logic clk, reset, enter;
   input logic [1:0] guess3, guess2, guess1, guess0;
   output logic [1:0] out3, out2, out1, out0;

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

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

   // Output logic
   always @(*) begin
      if (ns == B) begin
         out3 = guess3;
         out2 = guess2;
         out1 = guess1;
         out0 = guess0;
      end
   end

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

module enterGuess_tb();
   logic clk, reset, enter;
   logic [1:0] guess3, guess2, guess1, guess0, out3, out2, out1, out0;

   enterGuess dut(clk, reset, guess3, guess2, guess1, guess0, enter, out3, out2, out1, out0);

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



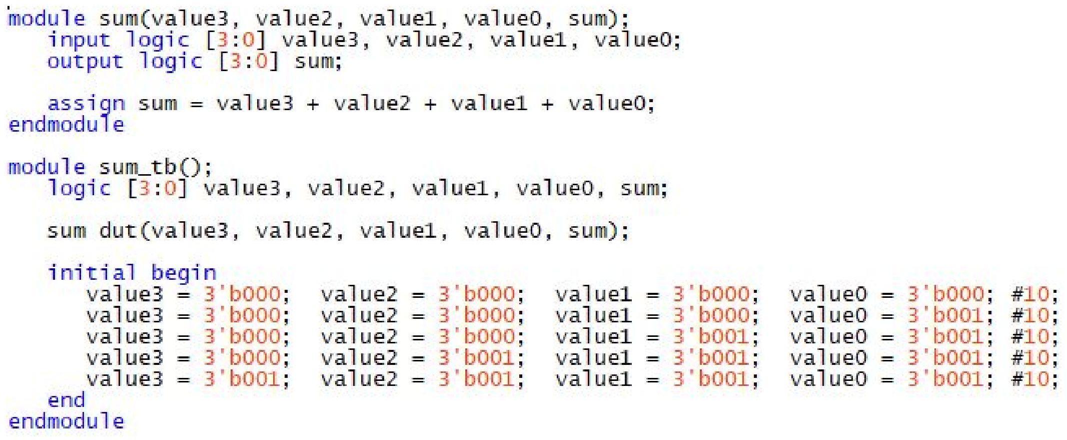
```
   // Set up initial inputs to the design. Each line is a clock cycle.
   initial begin
      reset <= 1; @(posedge clk);
      reset <= 0; @(posedge clk);
      guess3 <= 2'b00;  guess2 <= 2'b11;  guess1 <= 2'b10;  guess0 <= 2'b01;  enter <= 1; @(posedge clk);
      guess3 <= 2'b00;  guess2 <= 2'b11;  guess1 <= 2'b10;  guess0 <= 2'b01;               @(posedge clk);
      guess3 <= 2'b11;  guess2 <= 2'b00;  guess1 <= 2'b00;  guess0 <= 2'b11;               @(posedge clk);
      guess3 <= 2'b00;  guess2 <= 2'b01;  guess1 <= 2'b10;  guess0 <= 2'b11;  enter <= 0; @(posedge clk);
      guess3 <= 2'b00;  guess2 <= 2'b01;  guess1 <= 2'b10;  guess0 <= 2'b11;               @(posedge clk);
      guess3 <= 2'b01;  guess2 <= 2'b01;  guess1 <= 2'b11;  guess0 <= 2'b11;               @(posedge clk);
      guess3 <= 2'b01;  guess2 <= 2'b01;  guess1 <= 2'b11;  guess0 <= 2'b11;  enter <= 1; @(posedge clk);
      guess3 <= 2'b01;  guess2 <= 2'b01;  guess1 <= 2'b11;  guess0 <= 2'b11;               @(posedge clk);
      guess3 <= 2'b00;  guess2 <= 2'b11;  guess1 <= 2'b10;  guess0 <= 2'b01;               @(posedge clk);
      enter <= 0; @(posedge clk);
      guess3 <= 2'b00;  guess2 <= 2'b01;  guess1 <= 2'b10;  guess0 <= 2'b11;               @(posedge clk);
      guess3 <= 2'b01;  guess2 <= 2'b01;  guess1 <= 2'b11;  guess0 <= 2'b11;               @(posedge clk);
      guess3 <= 2'b01;  guess2 <= 2'b01;  guess1 <= 2'b11;  guess0 <= 2'b11;  enter <= 1; @(posedge clk);
      guess3 <= 2'b01;  guess2 <= 2'b01;  guess1 <= 2'b11;  guess0 <= 2'b11;               @(posedge clk);
      guess3 <= 2'b00;  guess2 <= 2'b11;  guess1 <= 2'b10;  guess0 <= 2'b01;               @(posedge clk);
      enter <= 0; @(posedge clk);
      guess3 <= 2'b00;  guess2 <= 2'b01;  guess1 <= 2'b10;  guess0 <= 2'b11;               @(posedge clk);
      guess3 <= 2'b01;  guess2 <= 2'b01;  guess1 <= 2'b11;  guess0 <= 2'b11;               @(posedge clk);
      guess3 <= 2'b01;  guess2 <= 2'b01;  guess1 <= 2'b11;  guess0 <= 2'b11;  enter <= 1; @(posedge clk);
      guess3 <= 2'b01;  guess2 <= 2'b01;  guess1 <= 2'b11;  guess0 <= 2'b11;               @(posedge clk);
      guess3 <= 2'b00;  guess2 <= 2'b11;  guess1 <= 2'b10;  guess0 <= 2'b01;               @(posedge clk);
      enter <= 0; @(posedge clk);
      guess3 <= 2'b00;  guess2 <= 2'b01;  guess1 <= 2'b10;  guess0 <= 2'b11;               @(posedge clk);
      guess3 <= 2'b01;  guess2 <= 2'b01;  guess1 <= 2'b11;  guess0 <= 2'b11;               @(posedge clk);
      guess3 <= 2'b01;  guess2 <= 2'b01;  guess1 <= 2'b11;  guess0 <= 2'b11;  enter <= 1; @(posedge clk);
      guess3 <= 2'b01;  guess2 <= 2'b01;  guess1 <= 2'b11;  guess0 <= 2'b11;               @(posedge clk);
      guess3 <= 2'b00;  guess2 <= 2'b11;  guess1 <= 2'b10;  guess0 <= 2'b01;               @(posedge clk);
      $stop; // End simulation
   end
endmodule
```

Code 2: enterGuess.sv



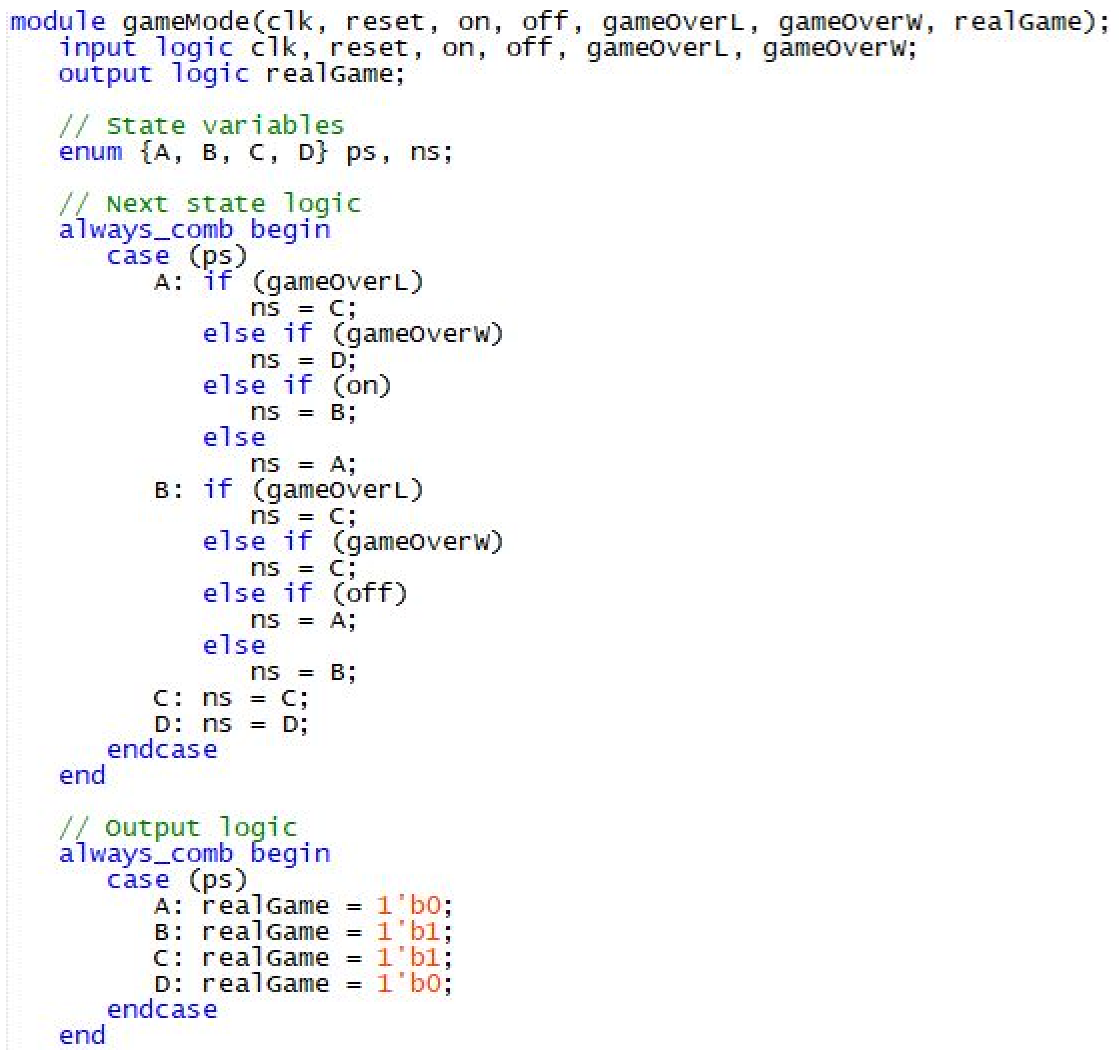
```
module comparator(reset, guess, codeA, codeB, codeC, codeD, correctLocation, valueOnly);
   input logic reset;
   input logic [1:0] guess, codeA, codeB, codeC, codeD;
   output logic [2:0] correctLocation, valueOnly;

   always @(*) begin
      if (reset) begin
         correctLocation = 3'b000;
         valueOnly = 3'b000;
      end else if (guess == codeA) begin
         correctLocation = 3'b001;
         valueOnly = 3'b000;
      end else begin
         if (guess == codeB)
            valueOnly = 3'b001;
         else if (guess == codeC)
            valueOnly = 3'b001;
         else if (guess == codeD)
            valueOnly = 3'b001;
         else begin
            valueOnly = 3'b000;
         end
         correctLocation = 3'b000;
      end
   end
endmodule

module comparator_tb();
   logic reset;
   logic [1:0] guess, codeA, codeB, codeC, codeD;
   logic [2:0] correctLocation, valueOnly;

   comparator dut(reset, guess, codeA, codeB, codeC, codeD, correctLocation, valueOnly);

   initial begin
      reset = 1;   #10;
      reset = 0;   #10;
      guess = 2'b00; codeA = 2'b00; codeB = 2'b11; codeC = 2'b10; codeD = 2'b10; #10;
      guess = 2'b00; codeA = 2'b11; codeB = 2'b11; codeC = 2'b10; codeD = 2'b00; #10;
      guess = 2'b00; codeA = 2'b11; codeB = 2'b11; codeC = 2'b00; codeD = 2'b10; #10;
      guess = 2'b00; codeA = 2'b11; codeB = 2'b11; codeC = 2'b10; codeD = 2'b10; #10;
   end
endmodule
```

Code 3: comparator.sv

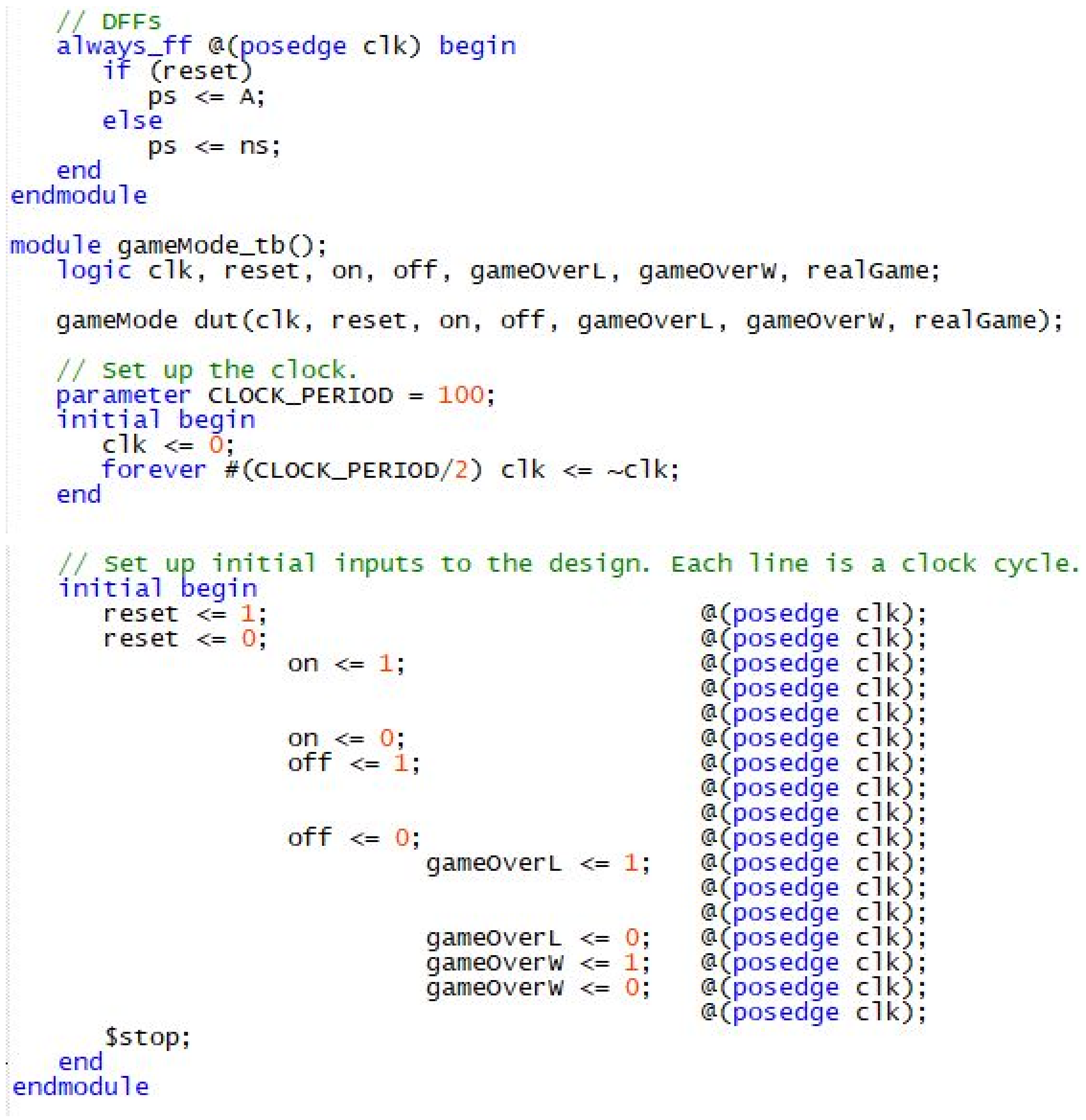


```
module sum(value3, value2, value1, value0, sum);
   input logic [3:0] value3, value2, value1, value0;
   output logic [3:0] sum;

   assign sum = value3 + value2 + value1 + value0;
endmodule

module sum_tb();
   logic [3:0] value3, value2, value1, value0, sum;

   sum dut(value3, value2, value1, value0, sum);

   initial begin
      value3 = 3'b000;  value2 = 3'b000;  value1 = 3'b000;  value0 = 3'b000; #10;
      value3 = 3'b000;  value2 = 3'b000;  value1 = 3'b000;  value0 = 3'b001; #10;
      value3 = 3'b000;  value2 = 3'b000;  value1 = 3'b001;  value0 = 3'b001; #10;
      value3 = 3'b000;  value2 = 3'b001;  value1 = 3'b001;  value0 = 3'b001; #10;
      value3 = 3'b001;  value2 = 3'b001;  value1 = 3'b001;  value0 = 3'b001; #10;
   end
endmodule
```

Code 4: sum.sv

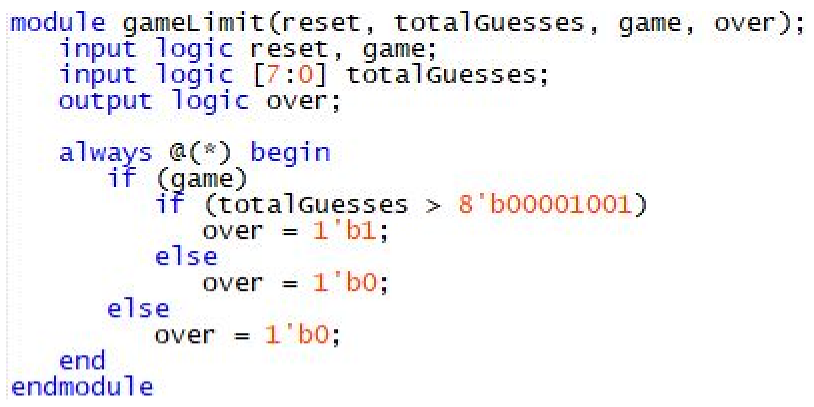


```
module gameMode(clk, reset, on, off, gameOverL, gameOverW, realGame);
   input logic clk, reset, on, off, gameOverL, gameOverW;
   output logic realGame;

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

   // Next state logic
   always_comb begin
      case (ps)
         A: if (gameOverL)
               ns = C;
            else if (gameOverW)
               ns = D;
            else if (on)
               ns = B;
            else
               ns = A;
         B: if (gameOverL)
               ns = C;
            else if (gameOverW)
               ns = C;
            else if (off)
               ns = A;
            else
               ns = B;
         C: ns = C;
         D: ns = D;
      endcase
   end

   // Output logic
   always_comb begin
      case (ps)
         A: realGame = 1'b0;
         B: realGame = 1'b1;
         C: realGame = 1'b1;
         D: realGame = 1'b0;
      endcase
   end
```

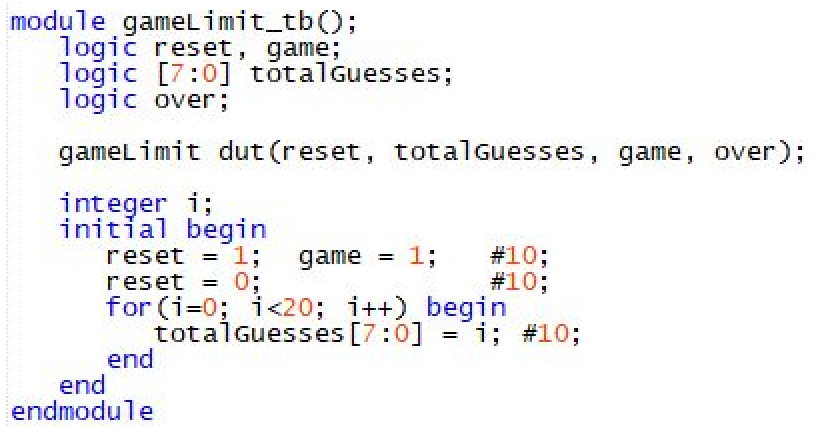


```
   // DFFs
   always_ff @(posedge clk) begin
      if (reset)
         ps <= A;
      else
         ps <= ns;
   end
endmodule

module gameMode_tb();
   logic clk, reset, on, off, gameOverL, gameOverW, realGame;

   gameMode dut(clk, reset, on, off, gameOverL, gameOverW, realGame);

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

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

Code 5: gameMode.sv

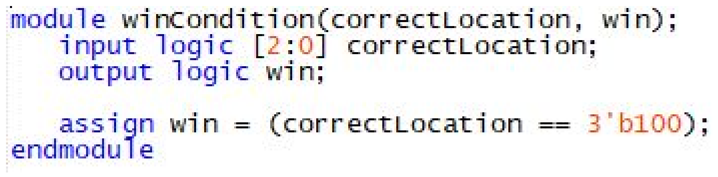


```
module gameLimit(reset, totalGuesses, game, over);
   input logic reset, game;
   input logic [7:0] totalGuesses;
   output logic over;

   always @(*) begin
      if (game)
         if (totalGuesses > 8'b00001001)
            over = 1'b1;
         else
            over = 1'b0;
      else
         over = 1'b0;
   end
endmodule
```



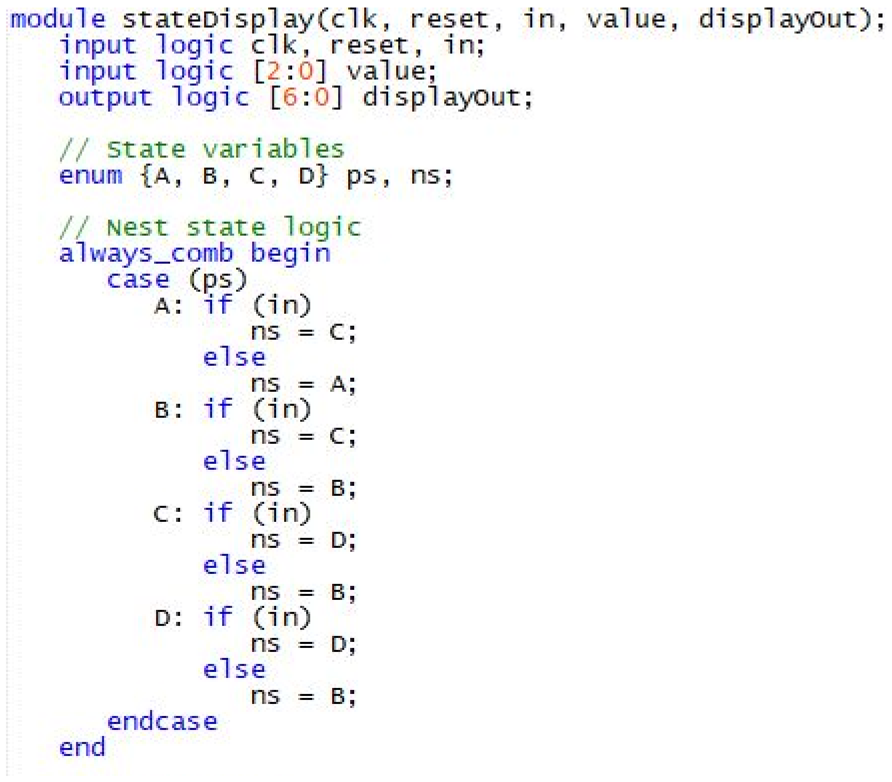
```
module gameLimit_tb();
   logic reset, game;
   logic [7:0] totalGuesses;
   logic over;

   gameLimit dut(reset, totalGuesses, game, over);

   integer i;
   initial begin
      reset = 1;  game = 1;    #10;
      reset = 0;               #10;
      for(i=0; i<20; i++) begin
         totalGuesses[7:0] = i; #10;
      end
   end
endmodule
```

Code 6: gameLimit.sv

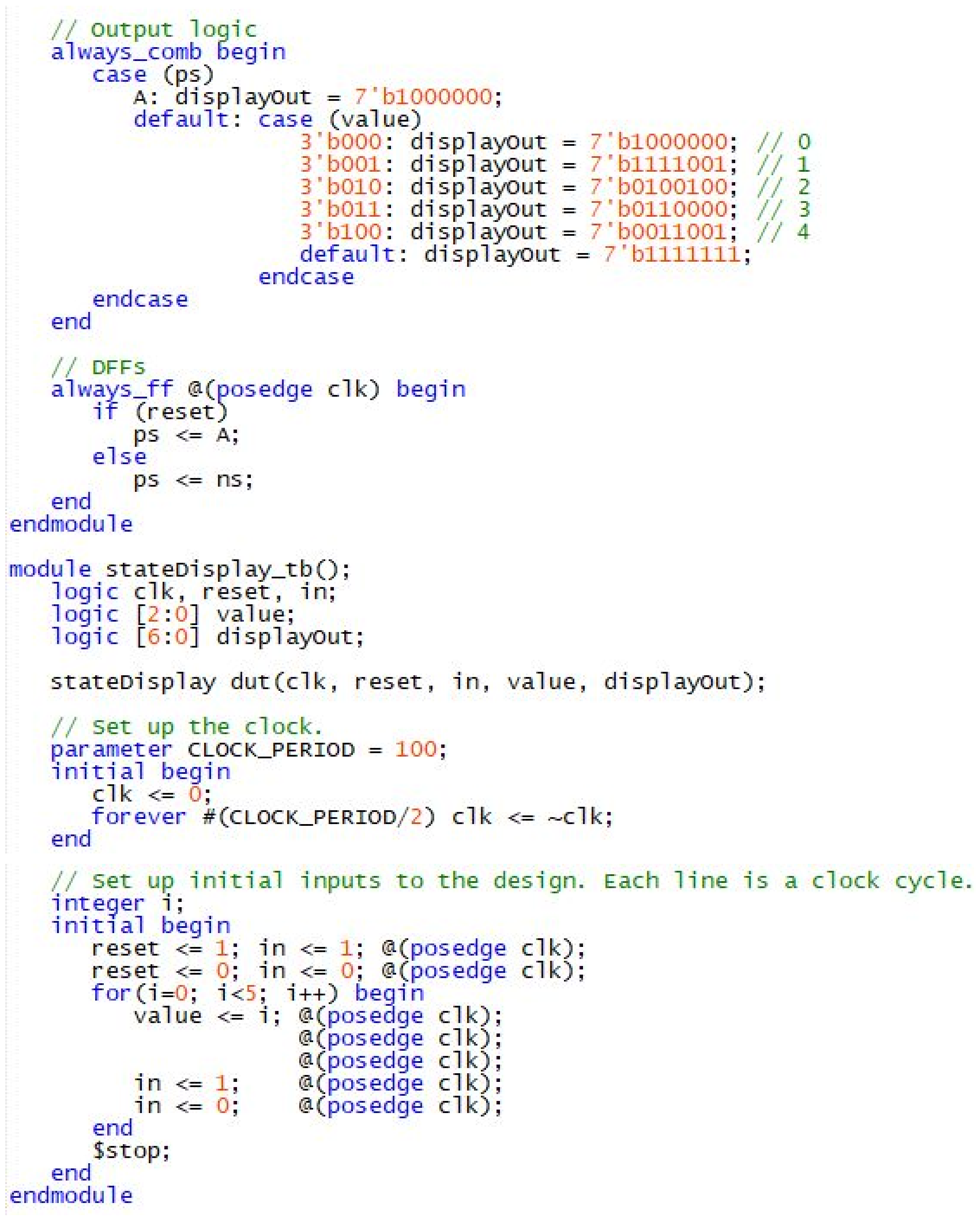


```
module winCondition(correctLocation, win);
   input logic [2:0] correctLocation;
   output logic win;

   assign win = (correctLocation == 3'b100);
endmodule
```

Code 7: winCondition.sv



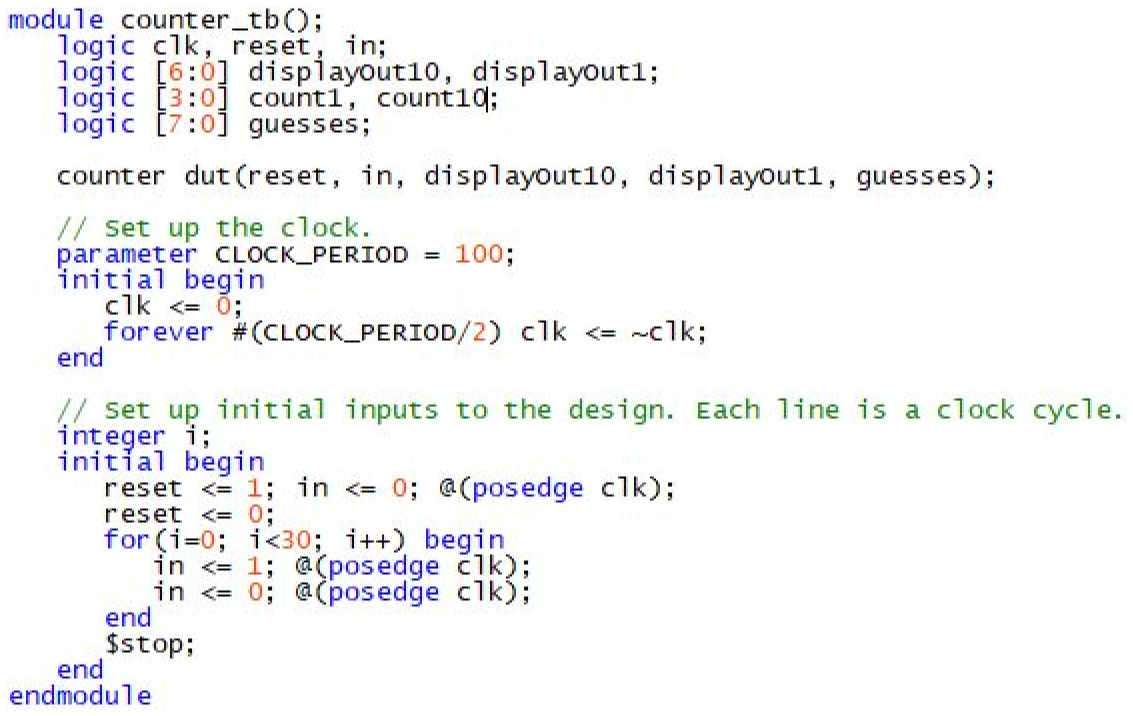
```
module stateDisplay(clk, reset, in, value, displayOut);
   input logic clk, reset, in;
   input logic [2:0] value;
   output logic [6:0] displayOut;

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

   // Nest state logic
   always_comb begin
      case (ps)
         A: if (in)
               ns = C;
            else
               ns = A;
         B: if (in)
               ns = C;
            else
               ns = B;
         C: if (in)
               ns = D;
            else
               ns = B;
         D: if (in)
               ns = D;
            else
               ns = B;
      endcase
   end
```



```
   // Output logic
   always_comb begin
      case (ps)
         A: displayOut = 7'b1000000;
         default: case (value)
                     3'b000: displayOut = 7'b1000000; // 0
                     3'b001: displayOut = 7'b1111001; // 1
                     3'b010: displayOut = 7'b0100100; // 2
                     3'b011: displayOut = 7'b0110000; // 3
                     3'b100: displayOut = 7'b0011001; // 4
                     default: displayOut = 7'b1111111;
                  endcase
      endcase
   end

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

module stateDisplay_tb();
   logic clk, reset, in;
   logic [2:0] value;
   logic [6:0] displayOut;

   stateDisplay dut(clk, reset, in, value, displayOut);

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

   // Set up initial inputs to the design. Each line is a clock cycle.
   integer i;
   initial begin
      reset <= 1; in <= 1; @(posedge clk);
      reset <= 0; in <= 0; @(posedge clk);
      for(i=0; i<5; i++) begin
         value <= i; @(posedge clk);
                     @(posedge clk);
                     @(posedge clk);
         in <= 1;    @(posedge clk);
         in <= 0;    @(posedge clk);
      end
      $stop;
   end
endmodule
```

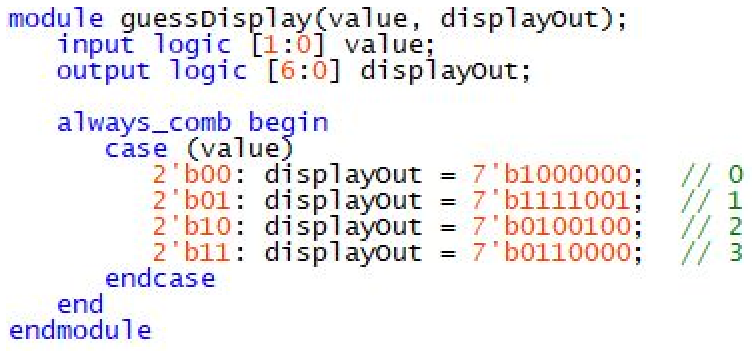
Code 8: stateDisplay.sv

```
module counter(reset, in, displayOut10, displayOut1, guesses);
   input logic reset, in;
   output logic [6:0] displayOut10, displayOut1;
   output logic [7:0] guesses;

   reg [3:0] count1, count10;

   // Counters
   always @(posedge in or posedge reset) begin
      if (reset) begin
         count1 <= 4'b0000;
         count10 <= 4'b0000;
         guesses <= 0;
      end else if (in) begin
         guesses = guesses + 1;
         if (count1 < 9)
            count1 <= count1 + 4'b0001;
         else begin
            if (count10 < 9)
               count10 <= count10 + 1;
            else
               count10 <= 0;
            count1 = 0;
         end
      end
   end

   // Ones place display
   always @(count1) begin
      case (count1)
         4'b0000: displayOut1 = 7'b1000000;  // 0
         4'b0001: displayOut1 = 7'b1111001;  // 1
         4'b0010: displayOut1 = 7'b0100100;  // 2
         4'b0011: displayOut1 = 7'b0110000;  // 3
         4'b0100: displayOut1 = 7'b0011001;  // 4
         4'b0101: displayOut1 = 7'b0010010;  // 5
         4'b0110: displayOut1 = 7'b0000010;  // 6
         4'b0111: displayOut1 = 7'b1111000;  // 7
         4'b1000: displayOut1 = 7'b0000000;  // 8
         4'b1001: displayOut1 = 7'b0010000;  // 9
         default: displayOut1 = 7'b1111111;
      endcase
   end

   // Tens place display
   always @(count10) begin
      case (count10)
         4'b0000: displayOut10 = 7'b1000000; // 0
         4'b0001: displayOut10 = 7'b1111001; // 1
         4'b0010: displayOut10 = 7'b0100100; // 2
         4'b0011: displayOut10 = 7'b0110000; // 3
         4'b0100: displayOut10 = 7'b0011001; // 4
         4'b0101: displayOut10 = 7'b0010010; // 5
         4'b0110: displayOut10 = 7'b0000010; // 6
         4'b0111: displayOut10 = 7'b1111000; // 7
         4'b1000: displayOut10 = 7'b0000000; // 8
         4'b1001: displayOut10 = 7'b0010000; // 9
         default: displayOut10 = 7'b1111111;
      endcase
   end
endmodule
```



```
module counter_tb();
   logic clk, reset, in;
   logic [6:0] displayOut10, displayOut1;
   logic [3:0] count1, count10;
   logic [7:0] guesses;

   counter dut(reset, in, displayOut10, displayOut1, guesses);

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

   // Set up initial inputs to the design. Each line is a clock cycle.
   integer i;
   initial begin
      reset <= 1; in <= 0; @(posedge clk);
      reset <= 0;
      for(i=0; i<30; i++) begin
         in <= 1; @(posedge clk);
         in <= 0; @(posedge clk);
      end
      $stop;
   end
endmodule
```

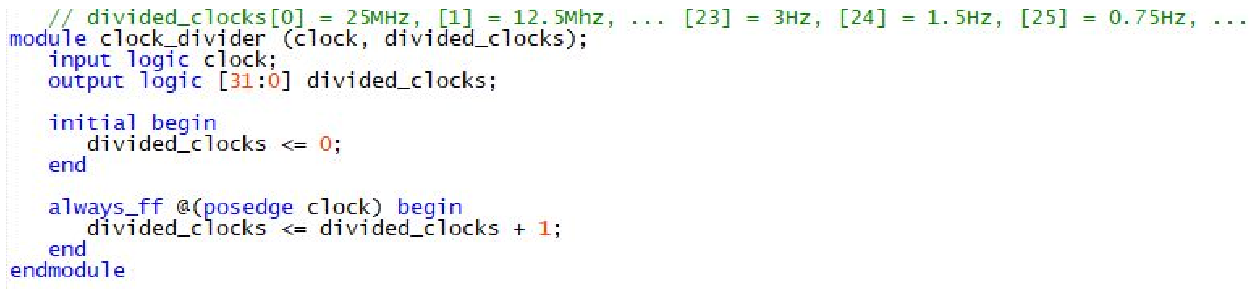
Code 9: counter.sv



```
module guessDisplay(value, displayOut);
   input logic [1:0] value;
   output logic [6:0] displayOut;

   always_comb begin
      case (value)
         2'b00: displayOut = 7'b1000000;   // 0
         2'b01: displayOut = 7'b1111001;   // 1
         2'b10: displayOut = 7'b0100100;   // 2
         2'b11: displayOut = 7'b0110000;   // 3
      endcase
   end
endmodule
```

Code 10: guessDisplay.sv

```
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 = 15;
   clock_divider cdiv (CLOCK_50, clk);

   // Hook up FSM inputs and outputs.
   assign reset = SW[9];                                        // reset
   assign enterGuess = ~KEY[3];                                 // push to submit current guess
   assign LEDR[0] = wonGame;                                    // true if won game
   assign LEDR[9] = lostGame;                                   // true if lost game
   assign LEDR[7] = gameMode;                                   // true if not practice game
   logic [1:0] digit3, digit2, digit1, digit0;                  // four digit code
   logic [1:0] guessD3, guessD2, guessD1, guessD0;              // selected guesses
   logic [7:0] totalGuesses;                                    // number of guesses made
   logic [3:0] CL3, CL2, CL1, CL0;                              // number of correct values and correct locations
   logic [3:0] VO3, VO2, VO1, VO0;                              // number of correct values but wrong locations
   logic [3:0] totalCorrect, totalValuesOnly;                   // total number of CL and VO
   logic [6:0] guessDisp3, guessDisp2, guessDisp1, guessDisp0;  // hex displays for guesses
   logic [6:0] codeDisp3, codeDisp2, codeDisp1, codeDisp0;      // hex displays for code
   logic [6:0] locationDisp, valueDisp, counter10, counter1;    // hex displays for game states


   // Random code generator
   codeGenerator rng(.clk(clk[whichClock]), .reset, .code3(digit3), .code2(digit2), .code1(digit1), .code0(digit0));

   // Make guess
   enterGuess guess(.clk(clk[whichClock]), .reset, .guess3({SW[7], SW[6]}), .guess2({SW[5], SW[4]}), .guess1({SW[3], SW[2]}), .guess0({SW[1], SW[0]}),
                    |.enter(enterGuess || wonGame || lostGame), .out3(guessD3), .out2(guessD2), .out1(guessD1), .out0(guessD0));

   // Compare guess with code
   comparator C3(.reset, .guess(guessD3), .codeA(digit3), .codeB(digit2), .codeC(digit1), .codeD(digit0), .correctLocation(CL3), .valueOnly(VO3));
   comparator C2(.reset, .guess(guessD2), .codeA(digit2), .codeB(digit3), .codeC(digit1), .codeD(digit0), .correctLocation(CL2), .valueOnly(VO2));
   comparator C1(.reset, .guess(guessD1), .codeA(digit1), .codeB(digit3), .codeC(digit2), .codeD(digit0), .correctLocation(CL1), .valueOnly(VO1));
   comparator C0(.reset, .guess(guessD0), .codeA(digit0), .codeB(digit3), .codeC(digit2), .codeD(digit1), .correctLocation(CL0), .valueOnly(VO0));

   // Finds total correct locations and total values only
   sum totalCL(.value3(CL3), .value2(CL2), .value1(CL1), .value0(CL0), .sum(totalCorrect));
   sum totalVO(.value3(VO3), .value2(VO2), .value1(VO1), .value0(VO0), .sum(totalValuesOnly));

   // Win condition
   winCondition win(.correctLocation(totalCorrect), .win(wonGame));

   // Select game mode
   gameMode practice(.clk(clk[whichClock]), .reset, .on(~KEY[1]), .off(~KEY[2]), .gameOverL(lostGame), .gameOverW(wonGame), .realGame(gameMode));
   gameLimit rounds(.reset, .totalGuesses(totalGuesses), .game(gameMode), .over(lostGame));

   // Display options
   // Display each guess
   guessDisplay G3(.value({SW[7], SW[6]}), .displayOut(guessDisp3));
   guessDisplay G2(.value({SW[5], SW[4]}), .displayOut(guessDisp2));
   guessDisplay G1(.value({SW[3], SW[2]}), .displayOut(guessDisp1));
   guessDisplay G0(.value({SW[1], SW[0]}), .displayOut(guessDisp0));

   // Display secret code
   guessDisplay Code3(.value(digit3), .displayOut(codeDisp3));
   guessDisplay Code2(.value(digit2), .displayOut(codeDisp2));
   guessDisplay Code1(.value(digit1), .displayOut(codeDisp1));
   guessDisplay Code0(.value(digit0), .displayOut(codeDisp0));

   // Display status of game
   stateDisplay CL(.clk(clk[whichClock]), .reset, .in(enterGuess), .value(totalCorrect), .displayOut(locationDisp));
   stateDisplay VO(.clk(clk[whichClock]), .reset, .in(enterGuess), .value(totalValuesOnly), .displayOut(valueDisp));
   counter guessesMade(.reset, .in(enterGuess || wonGame), .displayOut10(counter10), .displayOut1(counter1), .guesses(totalGuesses));

   // Control displayed game information
   always @(*) begin
      if (wonGame && ~lostGame)
         if (SW[8]) begin
            HEX5 = 7'b1111111;
            HEX4 = 7'b1111111;
            HEX3 = codeDisp3;
            HEX2 = codeDisp2;
            HEX1 = codeDisp1;
            HEX0 = codeDisp0;
         end else begin
            HEX5 = 7'b1111111;
            HEX4 = locationDisp;
            HEX3 = 7'b1111111;
            HEX2 = valueDisp;
            HEX1 = counter10;
            HEX0 = counter1;
         end
      else if (lostGame)
         if (SW[8]) begin
            HEX5 = 7'b1111111;
            HEX4 = 7'b1111111;
            HEX3 = codeDisp3;
            HEX2 = codeDisp2;
            HEX1 = codeDisp1;
            HEX0 = codeDisp0;
         end else begin
            HEX5 = 7'b1000111; // L
            HEX4 = 7'b0100011; // o
            HEX3 = 7'b0010010; // S
            HEX2 = 7'b0000110; // E
            HEX1 = 7'b0101111; // r
            HEX0 = 7'b1111111;
         end
      else
         if (SW[8]) begin
            HEX5 = 7'b1111111;
            HEX4 = 7'b1111111;
            HEX3 = guessDisp3;
            HEX2 = guessDisp2;
            HEX1 = guessDisp1;
            HEX0 = guessDisp0;
         end else begin
            HEX5 = 7'b1111111;
            HEX4 = locationDisp;
            HEX3 = 7'b1111111;
            HEX2 = valueDisp;
            HEX1 = counter10;
            HEX0 = counter1;
         end
   end
endmodule
```



```
   // divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz, ...
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
```

Code 11: DE1_SoC.sv