Design Problem

To potentially train wildlife to avoid cars, the simulation of a frog crossing a high-traffic road using concepts learned in EE 271 can provide insight. The simulation should include two roads of cars moving in alternate directions with a single animal, such as a frog, starting from one safe end of the road and trying to reach the other side to accurately represent the situation of passing traffic for wildlife.

The cars are represented by orange LEDs on the LED Array attachment to the De1-SoC board, and the uppermost cars move leftward while the cars parallel move rightward with a median in between. The frog is represented by a green LED which can move around the allocated 5x6 upper-left section of the LED Array. The frog starts in the 5th row from the top, and attempts to reach the topmost row while avoiding the passing cars. The frog is controlled by four buttons: KEY[3], KEY[2], KEY[1], KEY[0], which move the frog left, right, forwards, or back respectively. Each key press is only counted once, even when holding it down.

Reaching the topmost row of the LED Array grants one point for a win, updating the HEX0 display with the new value for a maximum of 7 wins. If the frog, however, becomes roadkill instead, then the HEX0 display updates, decreasing the total wins by 1 to a minimum of 0.

The general functionality of the frog and vehicles is illustrated (Fig. 1) to convey how each part of the simulation is generated.

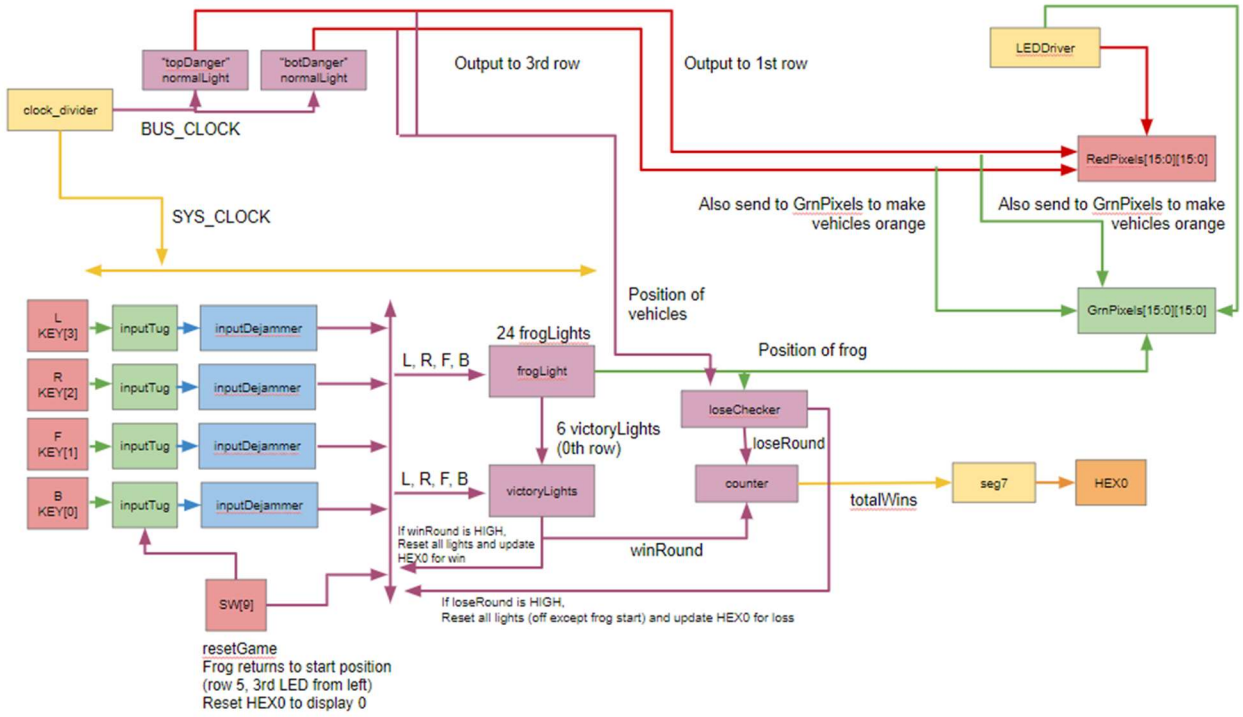

Fig. 1. User Level Block Diagram

The 5 rows * 6 columns upper-left section of the LED Array board will be controlled as demonstrated by the block diagram above.

Methods and Procedures

Button Input Design

The system inputs and outputs are controlled by the leading edge of a 1526 Hz clock. Button presses are counted only if the button was pressed over a clock edge. The frog light will only ever move on a clock edge. This control over timing also ensures the button cannot be held down over multiple clock edges to count for multiple button taps, but instead just one (Fig. 2). Additionally, player input is run through two flip-flops to ensure stable inputs (and therefore outputs) through instantiated inputDejammer (included in Fig. 7).

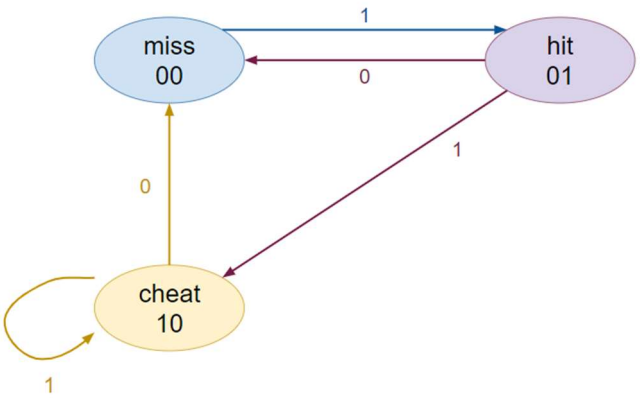

Fig. 2. inputTug State Diagram

The state diagram for the inputTug module, which takes the user input and determines whether it is a miss, a valid hit, or a cheat (pressing button continuously). A 1 represents that the button is pressed, and 0 means unpressed. InputTug was instantiated once for each button for four total instantiations.

When the frog light reaches the uppermost row of the playfield, then the round was won and the win count is updated on the HEX0 display. If the frog is controlled to be in the same position as a car, then the round was lost and the win count is decreased on the HEX0 display. Upon game reset, which is by flipping SW[9], the win count is set back to 0. The win count is only increased or decreased in response to wins or losses on round resets, which occur automatically after a win or a loss. Upon any reset, the frog is set back to its initial position on the 5^{th} row, the third LED from the left on the LED Array and will not move until the player presses a button.

To control which LED is on for the frog, each LED is controlled by module frogLight with inputs of the different keybuttons pressed, which are L, R, F, and B (whether the left, right, forward, or backward button was pressed respectively), and whether certain adjacent lights are currently on. From this, the two states of whether a light is on or off can be determined (Fig. 3).

Frog Movement and Round Win Design

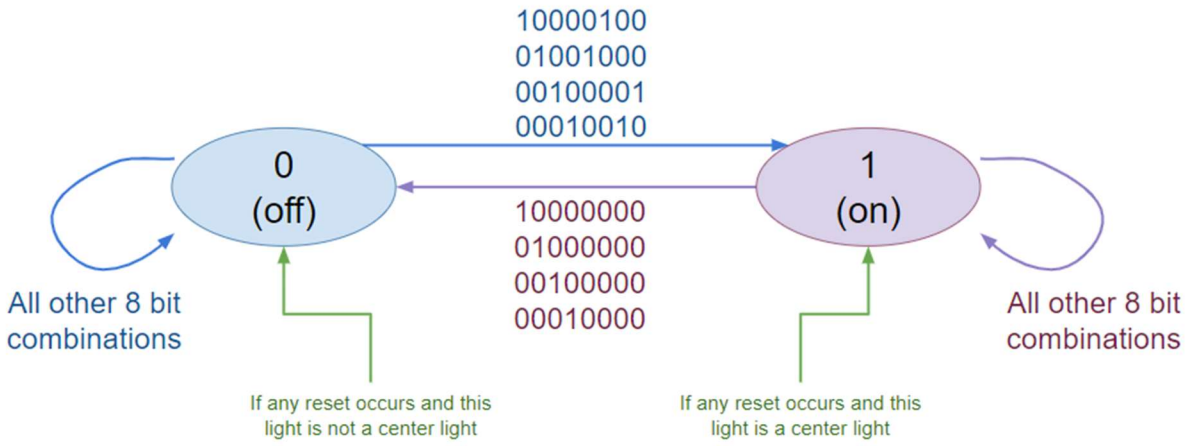

Fig. 3. frogLight State Diagram

8 bit inputs for frogLight are: L, R, F, B, NL, NR, NF, NB in that order, with a 1 representing that the corresponding key was pressed, and a 0 means unpressed.

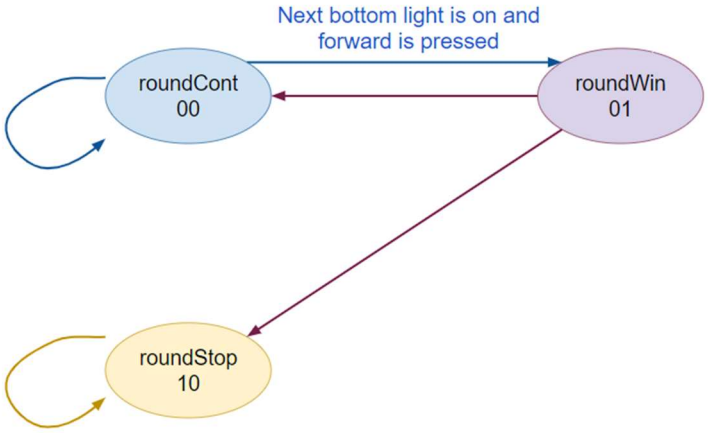

Fig. 4. victoryLight State Diagram

To control the victory case for the frog, the uppermost row of LEDs on the LED Array were each individually controlled by a victoryLight module to determine the state of the game round. The round should continue if it hasn't been won yet, send an output if the round is won, and otherwise should not allow any further changes until the round is reset after a win (Fig. 4). The reset occurs automatically afterwards.

Vehicle Movement and Round Loss Design

The vehicle movement was determined by identifying the different states the LEDs could be in a row, and establishing what the next state for the light should be depending on the inputs. The possible states and what combination of inputs could transition between them was illustrated with a state diagram.

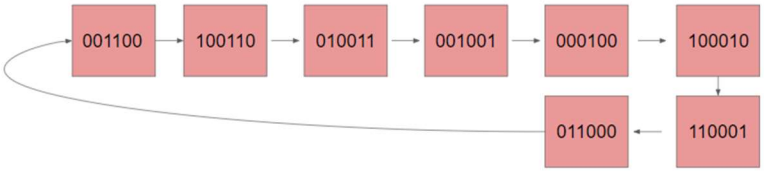

Fig. 5. normalLight State Diagram for Rightward Vehicles

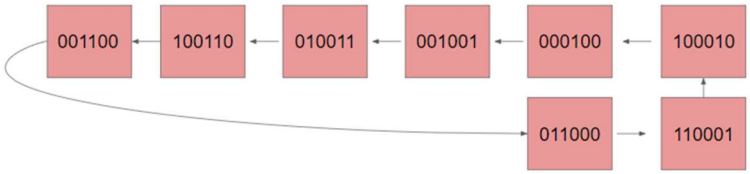

Fig. 6. normalLight State Diagram for Leftward Vehicles

The state diagrams for normalLight are shown in Fig. 7 and 8 with alternating direction. normalLight controls which LEDs are turned on in a row of 6. Each 0 represents an LED off and each 1 represents an LED on. Each state transitions to the next automatically.

All state diagrams were later transferred into Verilog to be simulated and then input into the De1-SoC board. The overall design of the system is illustrated below (Fig. 7).

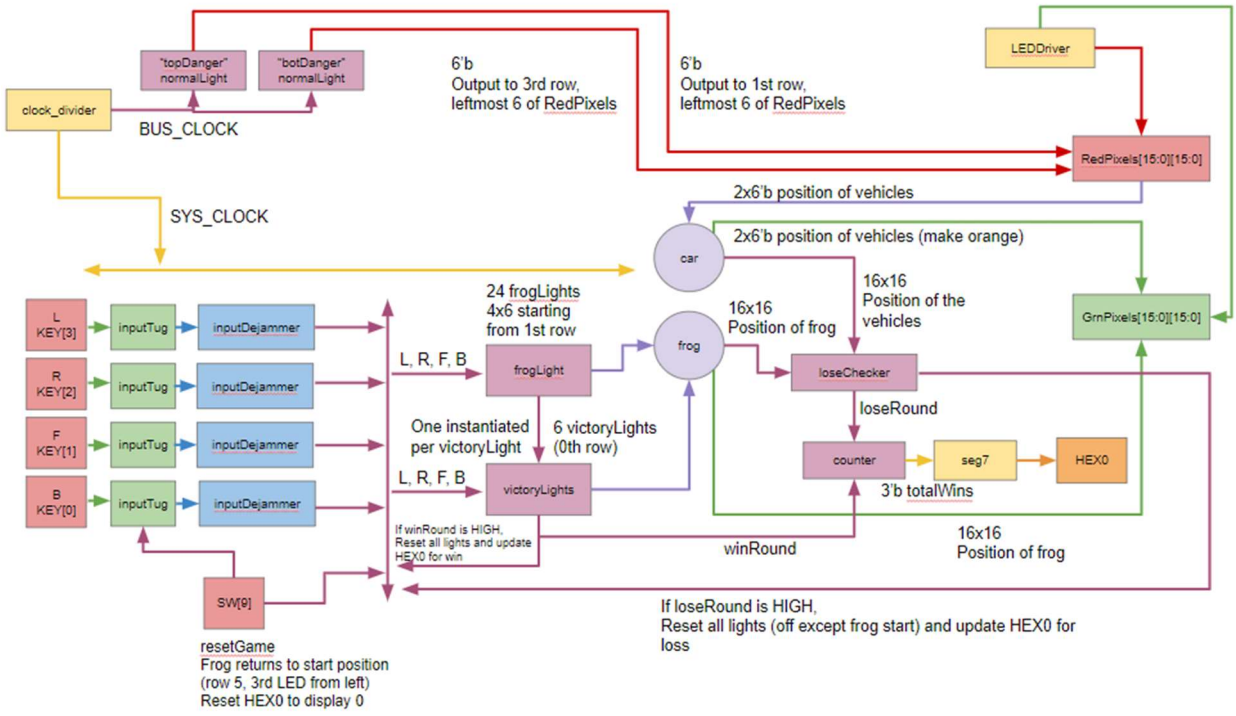

Fig. 7. Designer Level Block Diagram

The 5 rows * 6 columns upper-left section of the LED Array board are controlled by the modules included in the block diagram above. The four keybuttons are transferred through the modules to eventually control the position of the frog in frogLight and victoryLight, which is displayed with GrnPixels. The lights for the vehicles are displayed with both GrnPixels and RedPixels, which are 16x16 arrays from the LEDDriver that control the LED Array. There is detail on the size of outputs or inputs and significant internal variables “car” and “frog” convey the interconnectedness of many of the modules.

Furthermore, the design was minimized for number of gates, which is discussed below in the Generalized Resource Utilization by Entity Report under Results. A more detailed and comprehensive report, including every instantiation of every module is in the Appendix.

Results

Simulations in ModelSim:

Each module was simulated in ModelSim to observe outputs and ensure expected behavior.

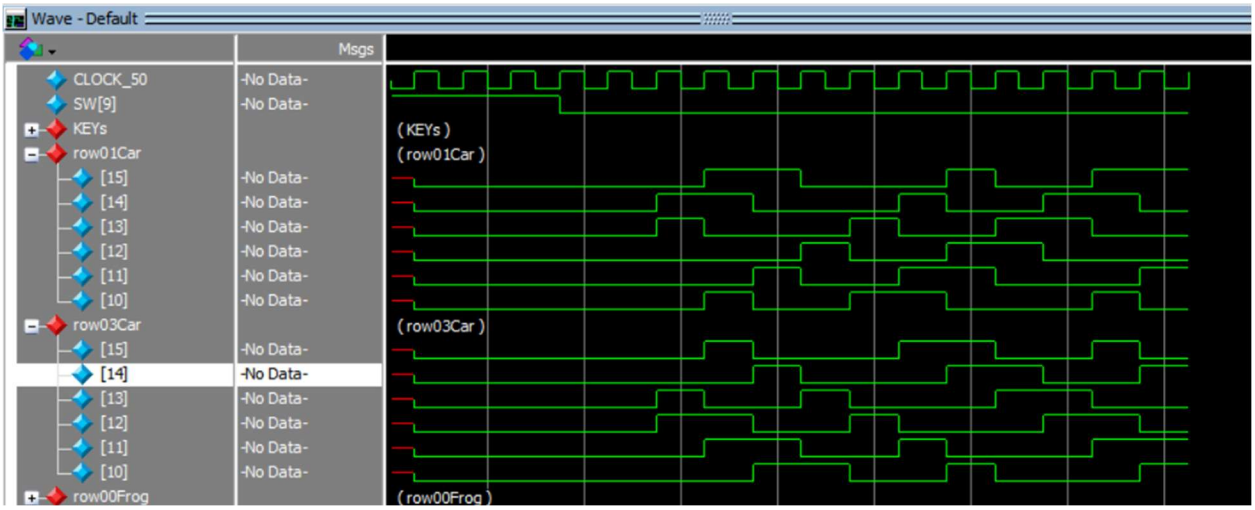

Fig. 8. Vehicle LEDs Clock Transitions in DE1_SoC Testbench

In the DE1_SoC Testbench, the vehicle movement should have HIGH signals moving from one LED to the adjacent one with each clock cycle. Row 1 has leftward movement while Row 3 has rightward movement of vehicles.

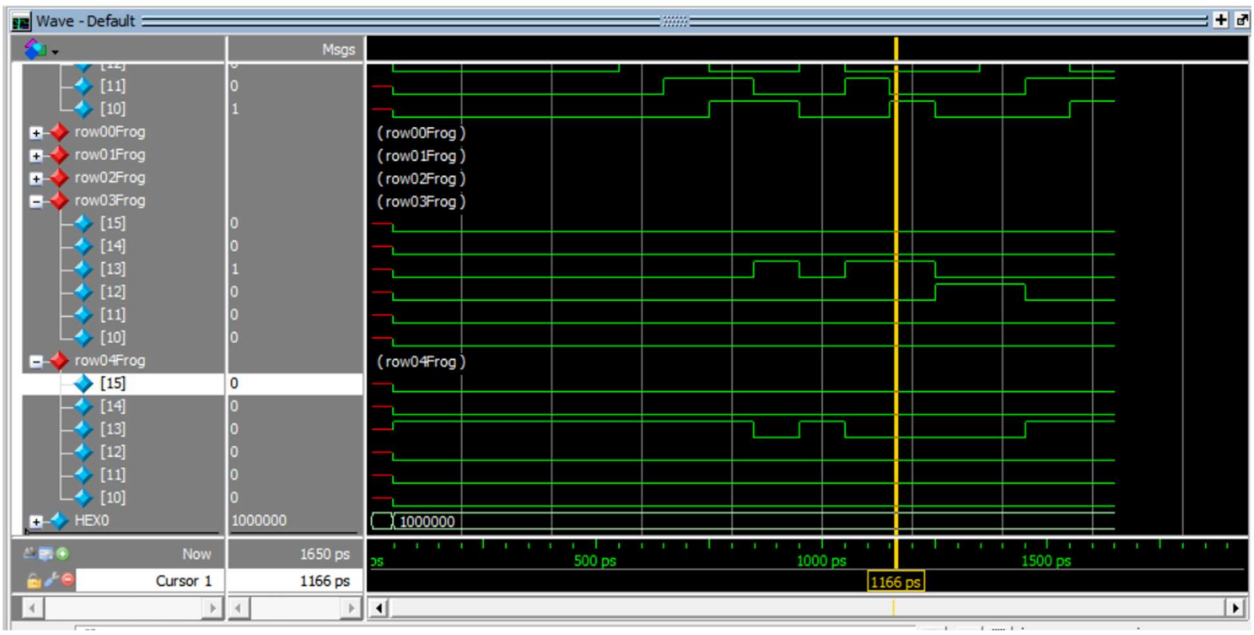

Fig. 9. Frog Movement after Reset in DE1_SoC Testbench

Frog movement is controlled by button presses, which is observed in the frogLight module below (Fig 10), but when encountering the LED of one of the vehicles (shown in the upper portion of the simulation), the frog light resets back to its initial position.

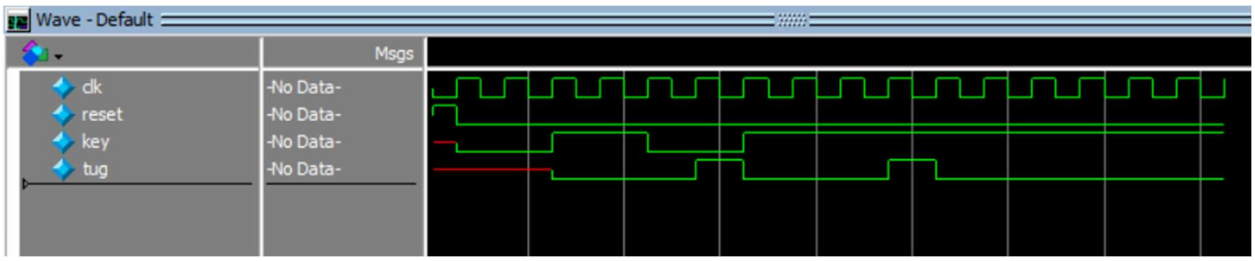

Fig. 9. inputTug

Each pushbutton input is sent to inputTug to ensure each press was only counted once, even if the button was held down. Therefore, each time key is HIGH, no matter how many clock cycles it is HIGH for, it will only be counted for one clock cycle with the output tug.

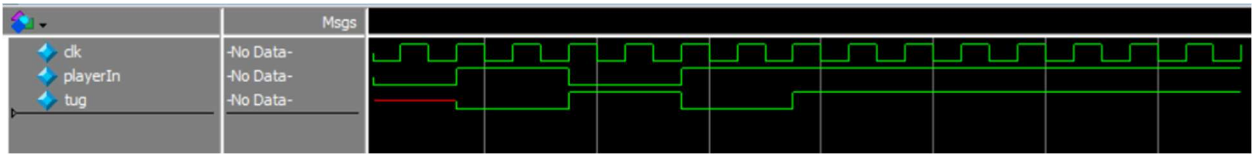

Fig. 10. inputDejammer

inputDejammer ensures stability by delaying the output by two clock cycles. Output tug follows playerIn after two clock cycles.

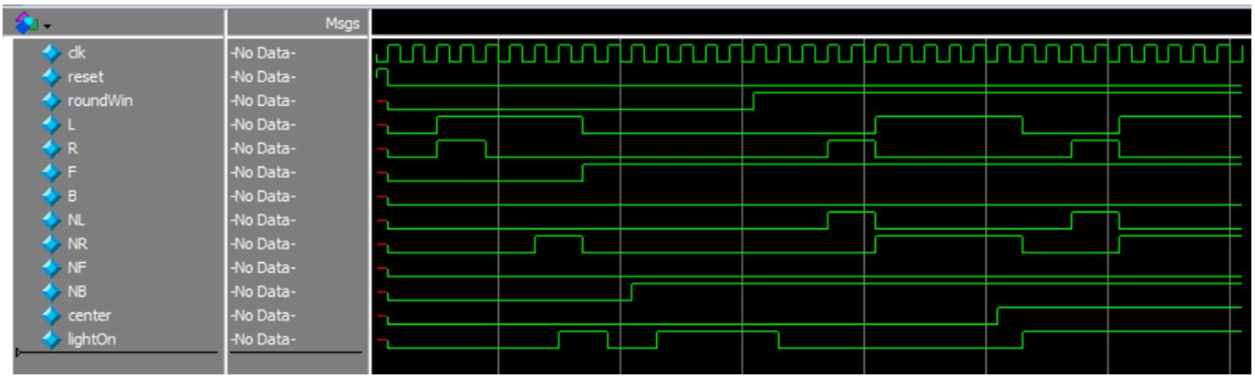

Fig. 10. frogLight

frogLight demonstrates the functionality of each individual LED that the frog travels to in the 5x6 playfield on the LED Array. L, R, F, and B are the buttons pressed (Left, Right, Forward, Back), and NL, NR, NF, and NB represent the adjacent LEDs to the current frogLight. If a button is pressed but no adjacent lights are on, the light does not turn on (lightOn is LOW). If a button is pressed and the opposing direction adjacent light is on, then lightOn goes HIGH. If the round is won (roundWin is HIGH), but the light is not a center light, then the light should turn off. If it is a center light, then the light should turn on.

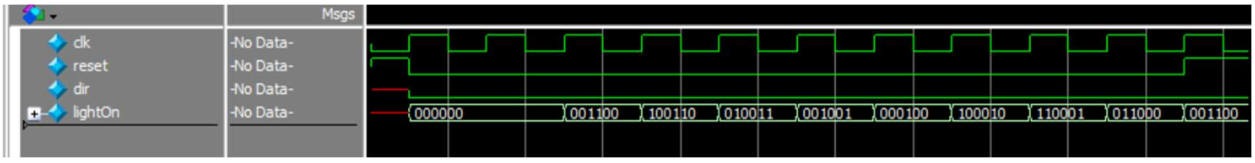

Fig. 12. normalLight

normalLight illustrates the animated movement of the cars. The first few transitions are shown for when dir is LOW, meaning rightward movement. lightOn represents a single row on the LED Array where the cars travel. With each clock cycle, the LEDs that are active shift to the right, with any LED on the right edge appearing on the left in the next transition for cyclical behavior.

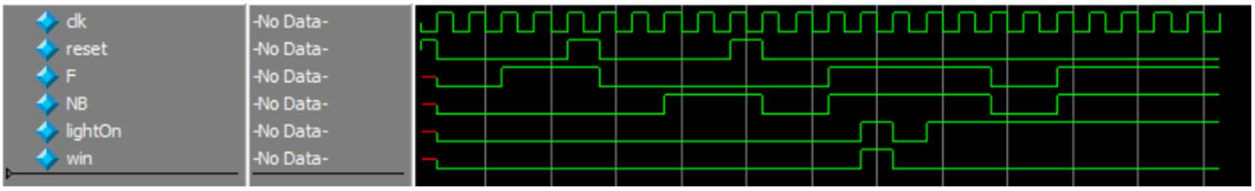

Fig. 13. victoryLight

victoryLight is controlled mostly by the forward button and the light just below it. There is an individual victoryLight for each LED in the uppermost row of the LED Array. Whenever the light below (NB) is on and the forward button is pressed (F), the victoryLight LED turns on and outputs a win. Only one win should occur between each reset if conditions are satisfied.

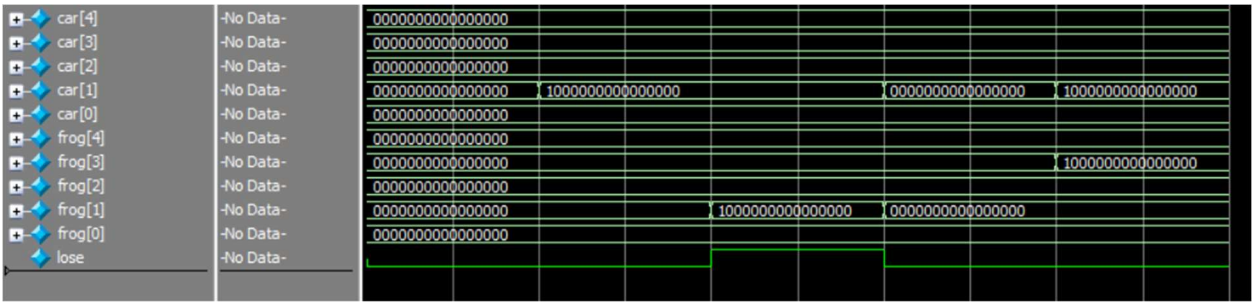

Fig. 14. loseChecker

loseChecker checks for when the position of the frog is the same as the position of a car by comparing each row of the LED Array. When the positions do not match, the round should continue. However, if both match (such as with 10000…. And 10000….) within the same row, then a loss should be output. If both are HIGH but are in different positions, such as in different rows with the last testcase, then a loss should not be recorded.

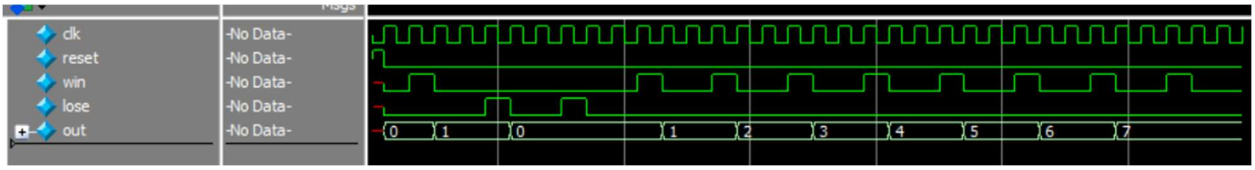

Fig. 15. counter

counter increments by 1 every time win is HIGH and decrements by 1 every time lose is HIGH. The value of out has a maximum of 7 and a minimum of 0, meaning that if a loss is recorded when out is 0, out will remain 0. If a win is recorded when out is 7, out will remain 7.

Generalized Resource Utilization by Entity Report:

The generalized resources utilized to implement the Frogger design are shown in Table 1. The most amount of gates in the overall design are in LEDDriver, which is necessary to use the LED Array attachment with the De1-SoC board. Following LEDDriver is clock_divider, necessary to divide the clock into a smaller frequency to slow the speed of the vehicles in comparison to the frog (so that the frog may have a chance to cross the road). Otherwise, most gates are used in normalLight, which controls an entire row and has multiple states to specify the LEDs. An individual victoryLight required the fewest, as they had very few inputs to determine logic and states. To minimize the design's usage of resources, minimal flip flops were added, as well as gate-level implementations of some of the system's elements, to ensure minimal logic gates being used for certain functions.

Table 1. Number of Gates Used by Module (Generalized)

Module	LC Combinationals	LC Registers
DE1_SoC	328(58)	106 (0)
LEDDriver	49(49)	4(4)
clock_divider	24(24)	24(24)
counter	3(3)	3(3)
seg7	7(7)	0(0)
normalLight (top Row)	9(9)	9(9)
normalLight (bottom Row)	8(8)	8(8)
frogLight (single)	3(3)	1(1)
inputTug (single)	2(2)	4(4)
loseChecker	5(5)	0(0)
victoryLight (single row)	5(3)	3(2)
victoryLight (single)	2(2)	1(1)

Reflection

In the future for embedded systems classes, I will be thinking a lot more about resource utilization. Since the first few labs in EE 271, I gradually learned how to try to minimize the size of my circuits and methods to check and reduce them. This included finding alternative ways to implement certain behavior and switching to gate-level implementation to enforce a certain amount of gates. In future embedded systems classes, I'm sure I'll be learning more about how to reduce resources used and am eager to improve my understanding of programming these circuits.

I believe that through a lot of troubleshooting, with lots of trial and error, I improved my understanding of how programming in Verilog translates to hardware. This is most seen with specifying which pin goes

where for each module. With this lab especially, I struggled to map connections between modules and internal logic to prevent linking one input from two different assignments, especially within generate statements. By focusing on interconnectedness through previous labs and this one, I believe I've come to understand it much better than initially.

In labs, I struggle most with making progress in the very beginning. Afterwards, I find that once I have some ideas of what to do, I can move forward quickly, with necessary troubleshooting afterwards. However, drafting what to do first has generally been difficult. I'm unsure of which implementation is the most efficient, and usually I cannot foresee the complications I may have with my initial implementation, which may lead to even longer periods of troubleshooting. I'm hoping that in the future when I become more practiced with designing circuits that I may be able to approach this stage of starting designs more confidently. Otherwise, I think I have learned a good process for programming large projects in Verilog. Following my difficulty with the initial stage, I am usually able to build smaller parts before working to the larger project successfully after verifying each step of the way. This helps reduce troubleshooting time when possible.

As I progressed through the EE 271 labs and encountered more than one finite state machine or combinational logic block, I learned that it was very important to understand and map out where the inputs and outputs for each module (or machine/combinational logic) was going. For each lab, I made sure to draft out my ideas through paper and pencil before doing the block diagram. I wanted to ensure that I had a good idea of what to do, since working with more than one finite state machine generally made the process more confusing. I will likely continue my method of drafting before the block diagram in future embedded systems classes to try to understand the lab better before proceeding.

Appendix

Resource Utilization by Entity

Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins	Full Hierarchy Name	Entity Name
1	|DE1_SoC	328 (58)	106 (0)	0	0	103	0	|DE1_SoC	DE1_SoC
1	|LEDDriver:Driver|	49 (49)	4 (4)	0	0	0	0	|DE1_SoC|LEDDriver:Driver	LEDDriver
2	|clock_divider:divider|	24 (24)	24 (24)	0	0	0	0	|DE1_SoC|clock_divider:divider	clock_divider
3	|counter:wins|	3 (3)	3 (3)	0	0	0	0	|DE1_SoC|counter:wins	counter
4	|frogLight:botLeft|	3 (3)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:botLeft	frogLight
5	|frogLight:botRight|	3 (3)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:botRight	frogLight
6	|frogLight:botRow[11].led|	3 (3)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:botRow[11].led	frogLight
7	|frogLight:botRow[12].led|	3 (3)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:botRow[12].led	frogLight
8	|frogLight:botRow[13].led|	3 (3)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:botRow[13].led	frogLight
9	|frogLight:botRow[14].led|	3 (3)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:botRow[14].led	frogLight
10	|frogLight:leftCol[1].led|	9 (9)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:leftCol[1].led	frogLight
11	|frogLight:leftCol[2].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:leftCol[2].led	frogLight
12	|frogLight:leftCol[3].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:leftCol[3].led	frogLight
13	|frogLight:m...Col[11].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:...w[1].midCol[11].led	frogLight
14	|frogLight:m...Col[12].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:...w[1].midCol[12].led	frogLight
15	|frogLight:m...Col[13].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:...w[1].midCol[13].led	frogLight
16	|frogLight:m...Col[14].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:...w[1].midCol[14].led	frogLight
17	|frogLight:m...Col[11].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:...w[2].midCol[11].led	frogLight
18	|frogLight:m...Col[12].led|	5 (5)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:...w[2].midCol[12].led	frogLight
19	|frogLight:m...Col[13].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:...w[2].midCol[13].led	frogLight
20	|frogLight:m...Col[14].led|	5 (5)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:...w[2].midCol[14].led	frogLight
21	|frogLight:m...Col[11].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:...w[3].midCol[11].led	frogLight
22	|frogLight:m...Col[12].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:...w[3].midCol[12].led	frogLight
23	|frogLight:m...Col[13].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:...w[3].midCol[13].led	frogLight
24	|frogLight:m...Col[14].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:...w[3].midCol[14].led	frogLight
25	|frogLight:rightCol[1].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:rightCol[1].led	frogLight
26	|frogLight:rightCol[2].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:rightCol[2].led	frogLight
27	|frogLight:rightCol[3].led|	6 (6)	1 (1)	0	0	0	0	|DE1_SoC|frogLight:rightCol[3].led	frogLight
28	|inputTug:back|	2 (2)	4 (2)	0	0	0	0	|DE1_SoC|inputTug:back	inputTug
1	|inputDeja...etaPlayer|	0 (0)	2 (2)	0	0	0	0	|DE1_SoC|inputTug...jammer:metaPlayer	inputDejammer
29	|inputTug:forward|	2 (2)	4 (2)	0	0	0	0	|DE1_SoC|inputTug:forward	inputTug
1	|inputDeja...etaPlayer|	0 (0)	2 (2)	0	0	0	0	|DE1_SoC|inputTug...jammer:metaPlayer	inputDejammer
30	|inputTug:left|	2 (2)	4 (2)	0	0	0	0	|DE1_SoC|inputTug:left	inputTug
1	|inputDeja...etaPlayer|	0 (0)	2 (2)	0	0	0	0	|DE1_SoC|inputTug...jammer:metaPlayer	inputDejammer
31	|inputTug:right|	2 (2)	4 (2)	0	0	0	0	|DE1_SoC|inputTug:right	inputTug
1	|inputDeja...etaPlayer|	0 (0)	2 (2)	0	0	0	0	|DE1_SoC|inputTug...jammer:metaPlayer	inputDejammer
32	|loseChecker:lost|	5 (5)	0 (0)	0	0	0	0	|DE1_SoC|loseChecker:lost	loseChecker
33	|normalLight:botDanger|	9 (9)	9 (9)	0	0	0	0	|DE1_SoC|normalLight:botDanger	normalLight
34	|normalLight:topDanger|	8 (8)	8 (8)	0	0	0	0	|DE1_SoC|normalLight:topDanger	normalLight
35	|seg7:vicHex|	7 (7)	0 (0)	0	0	0	0	|DE1_SoC|seg7:vicHex	seg7
36	|victoryLigh...Row[10].led|	5 (3)	3 (2)	0	0	0	0	|DE1_SoC|victoryLight:topRow[10].led	victoryLight
1	|frogLight:victory|	2 (2)	1 (1)	0	0	0	0	|DE1_SoC|victoryLig...d|frogLight:victory	frogLight
37	|victoryLigh...Row[11].led|	5 (3)	3 (2)	0	0	0	0	|DE1_SoC|victoryLight:topRow[11].led	victoryLight
1	|frogLight:victory|	2 (2)	1 (1)	0	0	0	0	|DE1_SoC|victoryLig...d|frogLight:victory	frogLight
38	|victoryLigh...Row[12].led|	5 (3)	3 (2)	0	0	0	0	|DE1_SoC|victoryLight:topRow[12].led	victoryLight
1	|frogLight:victory|	2 (2)	1 (1)	0	0	0	0	|DE1_SoC|victoryLig...d|frogLight:victory	frogLight
39	|victoryLigh...Row[13].led|	5 (3)	3 (2)	0	0	0	0	|DE1_SoC|victoryLight:topRow[13].led	victoryLight
1	|frogLight:victory|	2 (2)	1 (1)	0	0	0	0	|DE1_SoC|victoryLig...d|frogLight:victory	frogLight
40	|victoryLigh...Row[14].led|	5 (3)	3 (2)	0	0	0	0	|DE1_SoC|victoryLight:topRow[14].led	victoryLight
1	|frogLight:victory|	2 (2)	1 (1)	0	0	0	0	|DE1_SoC|victoryLig...d|frogLight:victory	frogLight
41	|victoryLigh...Row[15].led|	5 (3)	3 (2)	0	0	0	0	|DE1_SoC|victoryLight:topRow[15].led	victoryLight
1	|frogLight:victory|	2 (2)	1 (1)	0	0	0	0	|DE1_SoC|victoryLig...d|frogLight:victory	frogLight

Verilog Code

README

Modules:

1. DE1_SoC:

   The DE1_SoC module is the top level-entity in the design. This module overall controls the output LED Array, as well as the HEX0 display on the De1-SoC board. The output is controlled from the input signals of 4 buttons, KEY[3], KEY[2], KEY[1], and KEY[0], and switch SW[9]. The DE1_SoC module controls a travelling green LED starting in the fifth row from the top of the LED Array board that represents a frog. The frog must navigate past orange LED cars in the above rows to reach the top row. The behavior of the circuit originates from the modules instantiated in the DE1_SoC module, which are listed below.

2. inputTug:

   Instantiated four times in DE1_SoC, one for each used keybutton. Receives user input from keybuttons and checks whether it is occurred on a clock edge, or missed the clock edge, and ensures that a single button press only counts as one press, even if held over several clock cycles.

3. inputDejammer:

   Instantiated within inputTug is inputDejammer, which receives the user input from inputTug and connects this input to two flip flops to ensure a stable input for the rest of the system, which also ensures stable outputs.

4. frogLight:

   Instantiated 24 times within the DE1_SoC module to cover 4x6 of the 5x6 playfield section of the LED Array. The frogLight module controls frog movement over the playfield, turning the current light on or off depending on which buttons are pressed (as received by the outputs of inputTug) and whether the adjacent lights are on. This module is used for the starting light that the frog resets to on a new game and is also instantiated once per victory light described below. Upon reset, if the light is specified to be a center light, then it will turn on. Otherwise, it will turn off. If a round is completed, meaning one round was won by a player, it will turn off unless it is a designated center light, then it will turn on.

5. normalLight:

   Instantiated twice in the DE1_SoC module, one for the 1st and 3rd row of the LED Array (with the topmost being the 0th). Controls car movement on the LED Array by controlling a single row and changing the direction of LEDs turning on depending on an input parameter specifying direction. The cars move automatically across the row. Will turn off while reset is active, before returning to “animated” movement across the row after reset.

6. victoryLight:

   Instantiated six times in DE1_SoC to control the top row of the LED Array. Also contains an instantiated frogLight as mentioned above to turn LED on (briefly, since the frog is

promptly sent back to start) when the frog moves to it. Also checks if a win occurs, depending on whether the forward button is pressed and if the light below the victory light is on and is only green (the frog and not a car). If there is a win, the output is later sent to module counter instantiated in DE1_SoC to increment the number of wins for that player, with a maximum of seven wins.

7. loseChecker:

   Instantiated once within DE1_SoC. Checks the positions of the car and the frog for whether they occupy the same LED following a button press. If they do, a loss is output, which will later be sent to module counter instantiated in DE1_SoC to decrement the number of wins for that player, reducing to no less than 0 wins.

8. counter:

   Instantiated within the DE1_SoC module. Takes output from victoryLight and loseChecker on whether there was a win or loss and updates the total amount of wins a player has, with a maximum of seven and a minimum of 0. Sends output to instantiated seg7 module in the DE1_SoC module, which updates result to HEX0 on the De1-SoC board.

9. seg7:

   Instantiated within the DE1_SoC module is the seg7 module, which assigns case-by-case the HEX display depending on how many times a player has won. A maximum of 7 rounds are won before no more wins are recorded and a minimum of 0 wins are recorded.

10. clock_divider:

    Instantiated within the DE1_SoC module. Is used to modify the clock frequency of the system and the vehicles on the LED Array. The system clock controls most functionality, including the frog movement, button presses, and resets. The clock for the vehicles is much slower, to allow the frog to potentially reach victory.

DE1_SoC and Testbench:

```
// Top-level module that defines the I/Os for the DE-1 SoC board
// Creates and controls a game of Frogger, where a frog starting in
the
      // 5th row from the top of the LED Array will need to reach the
      // uppermost row of the LED Array without running into any of the
      // moving cars. The frog can move freely across the 5*6
playfield,
      // granted that it doesn't run into a car.
// HEX0 keeps track of the number of victories, with a maximum of 7,
      // and upon a victory, the player can play again automatically.
      // If the frog runs into a car, then the victories is subtracted
      // by 1, the lowest being 0. All other HEXs are off.
// KEY[3] controls the frog by moving it one LED left
// KEY[2] controls the frog by moving it one LED right
// KEY[1] controls the frog by moving it one LED forward
// KEY[0] controls the frog by moving it one LED backward
// SW[9] resets the game, meaning that the score drops to 0.
module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, SW, LEDR,
GPIO_1, CLOCK_50);
      output logic [6:0]  HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
      output logic [9:0]  LEDR;
      input  logic [3:0]  KEY;
      input  logic [9:0]  SW;
      output logic [35:0] GPIO_1;
      input logic CLOCK_50;

      // Turn off HEX displays except for HEX0, which displays the
score
      assign HEX1 = '1;
      assign HEX2 = '1;
      assign HEX3 = '1;
      assign HEX4 = '1;
      assign HEX5 = '1;

      /* Set up system base clock to 1526 Hz (50 MHz / 2**(14+1))
      ============================================================*/
      logic [31:0] div_clk;
      parameter sysClock = 14; // 1526 Hz clock signal
      parameter busClock = 23; // 3 Hz clock signal

      // SYSTEM_CLOCK is for all functionality such as button presses,
            // resets, general frog movement
      // BUS_CLOCK controls the speed of the cars and buses traversing
```

```
          // the playfield.
     logic SYSTEM_CLOCK;
     logic BUS_CLOCK;

     clock_divider divider (.clock(CLOCK_50),
.divided_clocks(div_clk));

     // CLOCK SELECT - for simulation or for programming the De1-SoC
     //assign SYSTEM_CLOCK = CLOCK_50;          // for simulation
     //assign BUS_CLOCK = CLOCK_50;          // for simulation
     assign SYSTEM_CLOCK = div_clk[sysClock];
     assign BUS_CLOCK = div_clk[busClock];



     /* Set up LED board driver
     ================================================================
= */
     logic [15:0][15:0]RedPixels; // 16 x 16 array representing red
LEDs
     logic [15:0][15:0]GrnPixels; // 16 x 16 array representing green
LEDs


     logic resetGame;                    // reset - toggle this on
startup
     assign resetGame = SW[9];
     logic resetRound;                                   //
automatically resets the round

     // by repositioning frog to start

     /* Standard LED Driver instantiation - set once and 'forget it'.
     See LEDDriver.sv for more info. Do not modify unless you know
what you are doing! */
     LEDDriver Driver (.CLK(SYSTEM_CLOCK), .RST(resetGame),
.EnableCount(1'b1), .RedPixels(RedPixels),
                           .GrnPixels(GrnPixels), .GPIO_1(GPIO_1));


     /* LED board general control of frogger and cars

     ================================================================
== */
```

```
	// L, R, F, B stores inputs from KEY[3:0] that are sent through instantiated inputTug modules
		// for each button. These modules check for holding down the button (which is only counted once)
		// and also briefly delaying key presses to ensure stability. Output is sent to L, R, F, B.
	logic L, R, F, B;
	inputTug left (.clk(SYSTEM_CLOCK), .reset(resetGame), .key(KEY[3]), .tug(L));
	inputTug right (.clk(SYSTEM_CLOCK), .reset(resetGame), .key(KEY[2]), .tug(R));
	inputTug forward (.clk(SYSTEM_CLOCK), .reset(resetGame), .key(KEY[1]), .tug(F));
	inputTug back (.clk(SYSTEM_CLOCK), .reset(resetGame), .key(KEY[0]), .tug(B));

	// car and frog stores the positions of the cars and the frogs on the LED Array
		// car's 1st and 3rd rows (starting from 0th at the top) is controlled by the
		// output to the RedPixels in the same rows of the LED Array.
		// frog's positions are all controlled later through modules.

	// GrnPixels is controlled by both the positions of the frog and car.
		// Since car also uses RedPixels, cars will appear orange on the LED Array
		// Frog will only appear green
	logic [15:0][15:0] car, frog;
	assign car[1][15:10] = RedPixels[1][15:10];
	assign car[3][15:10] = RedPixels[3][15:10];
	assign GrnPixels = frog + car;

	// Instantiates two normalLight modules to control the cars in the 1st and 3rd rows of
		// the LED Array. The outputs are stored in the corresponding rows for RedPixels.
		// The clock is the slower BUS_CLOCK, which is only used to control the speed
			// of the cars.
		// dir is the direction the cars are traveling.
	normalLight topDanger (.clk(BUS_CLOCK), .reset(resetGame), .dir(1'b1), .lightOn(RedPixels[1][15:10]));
```

```
	normalLight botDanger (.clk(BUS_CLOCK), .reset(resetGame),
.dir(1'b0), .lightOn(RedPixels[3][15:10]));

	// winRound contains whether the frog reached the top row
	// loseRound contains whether the frog ran into a vehicle
	logic winRound;
	logic loseRound;

	// win is controlled by whether the frog reached any LED in the
top row.
	// If the frog reached one of the LEDs, the index of that LED
will go high
	// winRound is set to whether any of the indices for the top row
goes high,
		// meaning the frog reached the top row.
	logic [5:0] win;
	assign winRound = (win >= 1);

	// the frog resets to the starting position if it wins or loses a
round.
	or (resetRound, winRound, loseRound);

	// checks if the frog ran into a car
		// by comparing whether both the positions of the car and
the frog match
		// Outputs result to loseRound
	loseChecker lost (.car(car), .frog(frog), .lose(loseRound));

	// tracks the total wins for the player
	logic [2:0] totalWins;
	// instantiates a 3-bit counter to increment by 1 in the case of
a round win
		// or decrement by 1 in the case of a round loss.
	counter wins (.clk(SYSTEM_CLOCK), .reset(resetGame),
.win(winRound), .lose(loseRound), .out(totalWins));

	// the output of the 3-bit counter is sent to the HEX display
through instantiated seg7, which
		// takes in the total wins and the HEX to display the value
to.
	seg7 vicHex (.count(totalWins), .leds(HEX0));


	// generate variables i and j cycle through the rows or columns
of the LED Array
```

```
          // to instantiate a module to control each necessary LED.
     genvar i, j;
     generate
          // instantiates victoryLight modules for the six leftmost
LEDs on the top row of the LED Array.
               // the victoryLights are off on reset and are only
affected by
               // the forward button and whether the light below them
is on.
          // If the frog is below a victory light and the forward
button is pressed,
               // the frog is victorious and returns to start.
          for(i=15; i>9; i--) begin :topRow
               victoryLight led (.clk(SYSTEM_CLOCK),
.reset(resetRound),
                                        .F(F), .NB(frog[1][i]),
                                        .lightOn(frog[0][i]),
.win(win[-i + 15]));
          end

          // Instantiates frogLight modules for the leftmost column
for the first 5 rows of the
               // LED Array, without the edge cases.
          // However, if the frog presses left on the leftmost
column, the frog will "cycle" to the
               // rightmost LED on the playfield, which is the 6th
LED from the left.
          // The frog's movement is therefore continuous throughout
the field.
          for(i=1; i<4; i++) begin :leftCol
               frogLight led (.clk(SYSTEM_CLOCK), .reset(resetGame),
.roundWin(resetRound),
                                        .L(L), .R(R), .F(F), .B(B),
                                        .NL(frog[i][10]),
.NR(frog[i][14]), .NF(frog[i-1][15]), .NB(frog[i+1][15]),
                                        .center(1'b0),
.lightOn(frog[i][15]));
          end

          // Instantiates frogLight modules for the middle rows and
columns of the playfield
               // for the first 5 rows and 6 columns of the LED
Array.
          for (j=1; j<4; j++) begin :midRow
               for(i=14; i>10; i--) begin :midCol
```

```
                    frogLight led (.clk(SYSTEM_CLOCK),
.reset(resetGame), .roundWin(resetRound),
                                                .L(L), .R(R), .F(F),
.B(B),
                                                .NL(frog[j][i+1]),
.NR(frog[j][i-1]), .NF(frog[j-1][i]), .NB(frog[j+1][i]),
                                                .center(1'b0),
.lightOn(frog[j][i]));
               end
          end


          // Instantiates frogLight modules for the rightmost column
of the playfield
               // for the first 5 rows of the LED Array.
          // If the frog presses right on this column, the frog will
"cycle" to the
               // leftmost LED on the playfield.
          // The frog's movement is therefore continuous throughout
the field.
          for(i=1; i<4; i++) begin :rightCol
               frogLight led (.clk(SYSTEM_CLOCK), .reset(resetGame),
.roundWin(resetRound),
                                        .L(L), .R(R), .F(F), .B(B),
                                        .NL(frog[i][11]),
.NR(frog[i][15]), .NF(frog[i-1][10]), .NB(frog[i+1][10]),
                                        .center(1'b0),
.lightOn(frog[i][10]));
          end


          // Instantiates frogLight modules for the bottommost row of
the playfield
               // (5th row of LED Array), not including edge cases.
          // The third LED from the left is the starting point for
the frog,
               // which will be on after any reset.
          for(i=14; i>10; i--) begin :botRow
               if (i==13) begin
                    frogLight led (.clk(SYSTEM_CLOCK),
.reset(resetGame), .roundWin(resetRound),
                                        .L(L), .R(R), .F(F), .B(B),
                                        .NL(frog[4][i+1]),
.NR(frog[4][i-1]), .NF(frog[3][i]), .NB(1'b0),
```

```
                                        .center(1'b1),
.lightOn(frog[4][i]));
                end else begin
                     frogLight led (.clk(SYSTEM_CLOCK),
.reset(resetGame), .roundWin(resetRound),
                                        .L(L), .R(R), .F(F), .B(B),
                                        .NL(frog[4][i+1]),
.NR(frog[4][i-1]), .NF(frog[3][i]), .NB(1'b0),
                                        .center(1'b0),
.lightOn(frog[4][i]));
                end
           end

     endgenerate

     // instantiates frogLight modules for the bottom left and bottom
right corners of the
           // playfield (the edge cases). As with the columns, if the
frog presses left
           // on the leftmost LED, it will skip to the rightmost LED,
appearing like
           // the playfield is continuous. The same happens for the
right side but
           // skipping to the left.
     frogLight botLeft (.clk(SYSTEM_CLOCK), .reset(resetGame),
.roundWin(resetRound),
                                        .L(L), .R(R), .F(F), .B(B),
                                        .NL(frog[4][10]),
.NR(frog[4][14]), .NF(frog[3][15]), .NB(1'b0),
                                        .center(1'b0),
.lightOn(frog[4][15]));
     frogLight botRight (.clk(SYSTEM_CLOCK), .reset(resetGame),
.roundWin(resetRound),
                                        .L(L), .R(R), .F(F), .B(B),
                                        .NL(frog[4][11]),
.NR(frog[4][15]), .NF(frog[3][10]), .NB(1'b0),
                                        .center(1'b0),
.lightOn(frog[4][10]));



endmodule

     // Test and simulate the DE1_SoC module by testing switch inputs
for reset
```

```
	// as well as testing key input to verify design.
module DE1_SoC_testbench();
	logic [6:0]  HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
	logic [9:0]  LEDR;
	logic [3:0]  KEY;
	logic [9:0]  SW;
	logic [35:0] GPIO_1;
	logic CLOCK_50;

	// sets up a DE1_SoC named as dut for testbench.
	DE1_SoC dut (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, SW, LEDR,
GPIO_1, CLOCK_50);

	// Set up a simulated clock.
	parameter CLOCK_PERIOD=100;
	initial begin
	CLOCK_50 <= 0;
	// Forever toggle the clock
	forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50;
	end

	// Test the design.
	initial begin
		// test moving forward twice.
		// Expected result is that single button presses will shift
the center light up once
			// and if the frog occupies the same LED as a car,
then it will reset back to
			// starting position.

		// Game reset, resets all lights and hex displays
		SW[9] <= 1;
			 @(posedge CLOCK_50);
			// testing loss from start.
			// Expected response is that the frog stays (since it
was reset to start)
			KEY[3:0] <= 4'b0010;
		 @(posedge CLOCK_50);
			KEY[3:0] <= 4'b0;
		    @(posedge CLOCK_50);

			// testing movement with buttons as well as returning
to start on loss
		SW[9] <= 1;
			 @(posedge CLOCK_50);
```

```
            SW[9] <= 0;
                        @(posedge CLOCK_50);
                KEY[3:0] <= 4'b0010;
             @(posedge CLOCK_50);
                KEY[3:0] <= 4'b0;
               @(posedge CLOCK_50);
                KEY[3:0] <= 4'b0010;
             @(posedge CLOCK_50);
                KEY[3:0] <= 4'b0;
               @(posedge CLOCK_50);
                KEY[3:0] <= 4'b0100;
             @(posedge CLOCK_50);
                KEY[3:0] <= 4'b0;
               @(posedge CLOCK_50);
                KEY[3:0] <= 4'b0010;
             @(posedge CLOCK_50);
                KEY[3:0] <= 4'b0;
      repeat(5) @(posedge CLOCK_50);
      $stop; // End the simulation.
      end
endmodule
```

inputTug and Testbench:

```
// inputTug takes in the input from a player and checks whether their
button press
      // was on a clock edge and ensure that
      // holding the button down only counts as one button press

// clk is the clock used for controlling input and output timing
// reset makes any button presses count as misses during the time
reset is active
// key is the respective key for each player
// tug is the output of whether the button press is counted or not.
module inputTug (clk, reset, key, tug);
      input logic clk;
      input logic reset;
      input logic key;
      output logic tug;

      // internal logic out is whether or not the button counts as a
press or not,
            // which is later sent to inputDejammer, which will ensure
overall input stability.
      logic out;
      // specifies various states of the button press. A miss is when
the button
            // was not pressed on a clockedge. A hit is when the button
was pressed on a
            // clockedge. And a cheat is when the button is held for
too long over multiple
            // clock edges (the button was held down).
      enum logic [1:0] {miss=2'b00, hit=2'b01, cheat=2'b10} ps, ns;
      // This logic describes all the possible state transitions from
ps to ns
      always_comb begin
            // out is determined by the LSB of the possible states,
                  // therefore, if it is a miss, then out is LOW
                  // if it is a hit, then out is HIGH
                  // if it is a cheat, then out is LOW
            out = ps[0];
            case (ps)
                  miss: if (!key)                                    ns =
miss;
                              else
      ns = hit;

                  hit: if (!key)
      ns = miss;
```

```
                              else
      ns = cheat;

               cheat: if (!key)                                        ns =
miss;
                              else
      ns = cheat;
            endcase
      end


      // D Flip Flop implementation (DFFs)
      always_ff @(posedge clk) begin
            if (reset)
                  ps <= miss; // any button presses during reset count
as misses
            else
                  ps <= ns; // otherwise, advances to next state in
state diagram
      end

      // instantiates an inputDejammer, which sends previously defined
"out" to two additional
            // flip flops to ensure stability. The final result is sent
to tug as output.
      inputDejammer metaPlayer (.clk(clk), .playerIn(out), .tug(tug));


endmodule


//Test/Simulate the State Machine
module inputTug_testbench();
      // creates corresponding variables to model inputDejammer module
      logic clk, reset;
      logic key;
      // when tug is true, the playfield light should shift
      logic tug;

      // initializes inputDejammer module for testing with name dut
      inputTug dut (clk, reset, key, tug);

      // Set up a simulated clock to toggle (from low to high or high
to low)
      // every 50 time steps
      parameter CLOCK_PERIOD=100;
      initial begin
            clk <= 0;
```

```
            forever #(CLOCK_PERIOD/2) clk <= ~clk;//toggle the clock
indefinitely
      end

// Set up the inputs to the design.  Each line represents a clock
cycle
// Simulation sends the state machine into possible states of button
presses,
      // including single button presses, misses, or holding down the
key.
      // Expected results are that whenever a button is pressed, even
when holding down,
      // it only ever counts as one button press.
      initial begin
                  reset <= 1;
             @(posedge clk);
                  reset <= 0;
                  key <= 0;                                     repeat(2)
@(posedge clk);
                  key <= 1;                                     repeat(2)
@(posedge clk);
                  key <= 0;                                     repeat(2)
@(posedge clk);
                  key <= 1;                                     repeat(10)
@(posedge clk);
            $stop; // End the simulation
      end
endmodule
```

inputDejammer and Testbench:

```
// inputDejammer ensures that user input from either player
      // is output only after two flip flops to ensure stability.
      // This ensures stable outputs without having completely random
input combinations.

// clk is the clock used for controlling input and output timing
// playerIn is whether a player has pressed the button on the clock
edge
// tug is the output the button press to be counted.
module inputDejammer (clk, playerIn, tug);
      input logic clk;
      input logic playerIn;
      output logic tug;

      // internal logic d1 is used to connect the output of one flip
flop to the next
      logic d1;

      // D Flip Flop implementation (DFFs) The first flip flop.
      always_ff @(posedge clk) begin
            d1 <= playerIn;
      end

      // D Flip Flop implementation (DFFs) The second flip flop sends
to output tug
      always_ff @(posedge clk) begin
            tug <= d1;
      end


endmodule


//Test/Simulate the State Machine
module inputDejammer_testbench();
      // creates corresponding variables to model inputDejammer module
      logic clk;
      logic playerIn;
      logic tug;

      // initializes inputDejammer module for testing with name dut
      inputDejammer dut (clk, playerIn, tug);
```

```
      // Set up a simulated clock to toggle (from low to high or high
to low)
      // every 50 time steps
      parameter CLOCK_PERIOD=100;
      initial begin
            clk <= 0;
            forever #(CLOCK_PERIOD/2) clk <= ~clk;//toggle the clock
indefinitely
      end

// Set up the inputs to the design.  Each line represents a clock
cycle
// Simulation shows results for button presses as inputs, expected
results
      // are that the inputs are the same as outputs, with a two-clock
cycle delay.
      integer i;
      initial begin
                  playerIn <= 0;                                  repeat(2)
@(posedge clk);
                  playerIn <= 1;                                  repeat(2)
@(posedge clk);
                  playerIn <= 0;                                  repeat(2)
@(posedge clk);
                  playerIn <= 1;                                 repeat(10)
@(posedge clk);
            $stop; // End the simulation
      end
endmodule
```

frogLight and Testbench:

```
// module frogLight defines basic movement for the frog.
// also has additionally functionality for center light

// clk is the clock used for controlling input and output timing
// reset will turn the lights off when active, except the specified
center
      // light, which will be on.
// roundWin is whether or not the round has been won, meaning
      // that the frog should reset back to the start (the designated
center light).
// L is true when left key (KEY[3]) is pressed
// R is true when the right key (KEY[2]) is pressed
// F is true when the right key (KEY[1]) is pressed
// B is true when the right key (KEY[0]) is pressed
// NL is true when the light on the left is on
// NR is true when the light on the right is on
// NF is true when the light forward is on
// NB is true when the light backward is on
// center is an input for whether this light is the center light, 0 if
not.
// lightOn is whether the light should be on or not, 1 if on.
module frogLight (clk, reset, roundWin, L, R, F, B, NL, NR, NF, NB,
center, lightOn);
      input logic clk, reset, roundWin;
      input logic L, R, F, B, NL, NR, NF, NB, center;
      // when lightOn is true, the normal light should be on.
      output logic lightOn;

      // two states for a light, either off or on
      enum logic {off=1'b0, on=1'b1} ps, ns;
      // This logic describes all the possible state transitions from
ps to ns
      always_comb begin
            // by default, if not satisfying any other designated
inputs, stay in the present state
            ns = ps;
            // otherwise, depending on whether the Left, Right,
Forward, and Backward buttons are pressed,
                  // as well as whether the Next Left, Next Right, Next
Forward,
                  // and Next Backward lights are on, turn the light on
or off.
            case (ps)
                  off: if ({L, R, F, B, NL, NR, NF, NB} == 8'b10000100)
                        ns = on;
```

```
                    else if ({L, R, F, B, NL, NR, NF, NB} ==
8'b01001000)                    ns = on;
                    else if ({L, R, F, B, NL, NR, NF, NB} ==
8'b00100001)                    ns = on;
                    else if ({L, R, F, B, NL, NR, NF, NB} ==
8'b00010010)                    ns = on;

               on: if ({L, R, F, B, NL, NR, NF, NB} == 8'b10000000)
                         ns = off;
                    else if ({L, R, F, B, NL, NR, NF, NB} ==
8'b01000000)                    ns = off;
                    else if ({L, R, F, B, NL, NR, NF, NB} ==
8'b00100000)                    ns = off;
                    else if ({L, R, F, B, NL, NR, NF, NB} ==
8'b00010000)                    ns = off;
          endcase
     end

     // Output logic
     // Output to lightOn matches the present state, which is encoded
          // to represent whether it is HIGH or LOW
     assign lightOn = ps;

     // D Flip Flop implementation (DFFs)
     always_ff @(posedge clk) begin
          // if on reset or a roundWin
               // turn the light on if it is the designated center
light,
               // otherwise turn off.
          if (reset | roundWin) begin
               if (center)
                    ps <= on;
               else
                    ps <= off;
          // if not on reset or roundWin, advance to the next state
in the state
               // diagram
          end else
               ps <= ns;
     end

endmodule


//Test/Simulate the State Machine
module frogLight_testbench();
     // creates corresponding variables to model frogLight module
     logic clk, reset, roundWin;
```

```
      logic L, R, F, B, NL, NR, NF, NB, center;
      // when lightOn is true, the normal light should be on.
      logic lightOn;

      // initializes frogLight module for testing with name dut
      frogLight dut (clk, reset, roundWin, L, R, F, B, NL, NR, NF, NB,
center, lightOn);

      // Set up a simulated clock to toggle (from low to high or high
to low)
      // every 50 time steps
      parameter CLOCK_PERIOD=100;
      initial begin
            clk <= 0;
            forever #(CLOCK_PERIOD/2) clk <= ~clk;//toggle the clock
indefinitely
      end

// Simulation with different situations
      initial begin
            // Tests general functionality after reset with multiple
combinations of button presses,
                  // including neither pressed, both pressed, one
pressed with neither neighboring lights on,
                  // and one pressed with a neighboring light on.

            // Expected results include that the light turns on only if
an adjacent light is on and the button
                  // pressed is opposite of the direction that adjacent
light is.
                  // Meaning, for the light to turn on when the next
left light is on, right button must be pressed.

            // Additionally, button presses should not affect current
light when no adjacent lights are on
            // Current light should not turn on if there is no button
press even when adjacent lights are on.
            reset <= 1;
                                                      @(posedge clk); //
Always reset FSMs at start
            reset <= 0; center <= 0; roundWin <= 0;
                                    {L, R, F, B, NL, NR, NF, NB} <= 4'b0;
      repeat(2) @(posedge clk);
                                    {L, R, NL, NR} <= 4'b1100;
            repeat(2) @(posedge clk);
                                    {L, R, NL, NR} <= 4'b1000;
            repeat(2) @(posedge clk);
```

```
                                {L, R, NL, NR} <= 4'b1001;
            repeat(2) @(posedge clk);
                                {L, R, NL, NR} <= 4'b0;
                                {F, B, NF, NB} <= 4'b1000;
            repeat(2) @(posedge clk);
                                {F, B, NF, NB} <= 4'b1001;
            repeat(5) @(posedge clk);
            // tests response on roundWin.Expected results is that
light is off
                        // when it is not a center light
            reset <= 0; center <= 0; roundWin <= 1;
                         @(posedge clk);
                                {L, R, NL, NR} <= 4'b0000;
            repeat(2) @(posedge clk);
                                {L, R, NL, NR} <= 4'b0110;
            repeat(2) @(posedge clk);
                                {L, R, NL, NR} <= 4'b1001;
            repeat(5) @(posedge clk);
            // test that light is on for reset since it is the
designated center light
            reset <= 0; center <= 1;
                                 @(posedge clk);
                                {L, R, NL, NR} <= 4'b0000;
            repeat(2) @(posedge clk);
                                {L, R, NL, NR} <= 4'b0110;
            repeat(2) @(posedge clk);
                                {L, R, NL, NR} <= 4'b1001;
            repeat(5) @(posedge clk);
            $stop; // End the simulation
      end
endmodule
```

normalLight and Testbench:

```
// module normalLight defines basic functionality for passing
vehicles.

// clk is the clock used for controlling input and output timing
// reset will turn certain LEDs off when active
// dir is which direction the cars should head, 0 for right, 1 for
left.
// lightOn specified which LEDs in a row of 6 LEDs is on or off.
module normalLight (clk, reset, dir, lightOn);
      input logic clk, dir, reset;
      output logic [5:0] lightOn;

      // ps and ns of the lights will allow for transitioning between
states
      logic [5:0] ps, ns;
      // count specified which state to transition to. count is
incremented or
            // decremented depending on dir.
      logic [2:0] count;
      // This logic describes all the possible states for a row of
vehicle
            // lights.
      always_comb begin
            case (count)
                  default: ns = 6'b000000;
                  3'b000: ns = 6'b001100;
                  3'b001: ns = 6'b100110;
                  3'b010: ns = 6'b010011;
                  3'b011: ns = 6'b001001;
                  3'b100: ns = 6'b000100;
                  3'b101: ns = 6'b100010;
                  3'b110: ns = 6'b110001;
                  3'b111: ns = 6'b011000;
            endcase
      end

      // D-FF implementation
      always_ff @(posedge clk) begin
                  // on reset, turn LEDs off
                  if (reset)
                        ps <= 6'b0;
                  else begin
                  // otherwise head to next state,
                  // which is determined by count
                        // if heading right (dir is 0), then
```

```
                              // increment count until a 7. Once
approaching
                              // 7, set count back to 0.
                        // if heading right (dir is 1), then do the
                              // reverse of above.
                        ps <= ns;
                              if(dir == 0) begin
                                    if(count < 7)
                                          count <= count + 1;
                                    else
                                          count <= 3'b0;
                              end else begin
                                    if(count > 0)
                                          count <= count - 1;
                                    else
                                          count <= 3'b111;
                              end
                  end
      end

      // Output logic
      // Output to lightOn matches the present state, which is encoded
            // to represent whether each LED in the row is HIGH or LOW
      assign lightOn = ps;
endmodule


//Test/Simulate the State Machine
module normalLight_testbench();
      // creates corresponding variables to model normalLight module
      logic clk, reset, dir;
      // when lightOn is true, the corresponding LEDs should be on.
      logic [5:0] lightOn;

      // initializes normalLight module for testing with name dut
      normalLight dut (clk, reset, dir, lightOn);

      // Set up a simulated clock to toggle (from low to high or high
to low)
      // every 50 time steps
      parameter CLOCK_PERIOD=100;
      initial begin
            clk <= 0;
            forever #(CLOCK_PERIOD/2) clk <= ~clk;//toggle the clock
indefinitely
      end
```

```
// Simulation sends the state machine from reset to the various states
automatically
     initial begin
          reset <= 1;
                              @(posedge clk); // Always reset FSMs
at start
          // check movement for dir = 0
          // Expected result is that the LEDs are "animated"
rightward
          reset <= 0; dir <= 0;                              repeat
(10) @(posedge clk);
          reset <= 1;
                              @(posedge clk);
          // check movement for dir = 1
          // Expected result is that the LEDs are "animated" leftward
          reset <= 0; dir <= 1;                              repeat
(10) @(posedge clk);
          $stop; // End the simulation
     end
endmodule
```

victoryLight and Testbench:

```
// module victoryLight defines additional functionality for a victory
(top) light.

// clk is the clock used for controlling input and output timing
// reset turns the lights off on a new round
// F is true when forward key is pressed
// NB is true when the light backward is on
// lightOn is whether the light should turn on. Though, this only
      // happens very briefly since the frog is reset
      // to start once it reaches a victory light.
// win is whether the frog reached a victoryLight, 1 if it did.
module victoryLight (clk, reset, F, NB, lightOn, win);
      input logic clk, reset;
      input logic F, NB;
      // when lightOn is true, the light should be on.
      output logic lightOn;
      output logic win;


      // instantiates a frogLight for basic functionality of a frog
light, turning on when it is reached,
            // specifying that this is not a center light
      frogLight victory (.clk(clk), .reset(reset), .roundWin(win),
.L(1'b0), .R(1'b0), .F(F), .B(1'b0),
                                          .NL(1'b0), .NR(1'b0),
.NF(1'b0), .NB(NB), .center(1'b0), .lightOn(lightOn));


      // player wins if the frog reaches a victory light by pressing
forward when the next
            // bottom light is on.
      enum logic [1:0] {roundCont=2'b00, roundWin=2'b01,
roundStop=2'b10} ps, ns;
      // This logic describes all the possible state transitions from
ps to ns
      always_comb begin
            win <= ps[0];
            case (ps)
                  roundStop:
            ns = roundStop;
                  roundWin:
      ns = roundStop;
                  roundCont: if (NB && F)
      ns = roundWin;
                                    else
            ns = roundCont;
```

```
            endcase
      end

      // D-FF implementation
      always_ff @(posedge clk) begin
            // on reset, set state to one allowing play
            if (reset) begin
                  ps <= roundCont;
            // otherwise, head to next state in state diagram
            end else
                  ps <= ns;
      end
endmodule

//Test/Simulate the State Machine
module victoryLight_testbench();
      // creates corresponding variables to model victoryLight module
      logic clk, reset;
      logic F, NB;
      // when lightOn is true, the normal light should be on.
      logic lightOn;
      logic win;

      // initializes victoryLight module for testing with name dut
      victoryLight dut (clk, reset, F, NB, lightOn, win);

      // Set up a simulated clock to toggle (from low to high or high
to low)
            // every 50 time steps
      parameter CLOCK_PERIOD=100;
      initial begin
            clk <= 0;
            forever #(CLOCK_PERIOD/2) clk <= ~clk;//toggle the clock
indefinitely
      end

// Simulation sends the state machine into various scenarios.
      initial begin
            reset <= 1;
@(posedge clk); // Always reset FSMs at start
            reset <= 0;     // test player functionality with no button
press, a button press but no adjacent light on,
                                    // and button press with the
adjacent light on.
                                    // Expected results are that
there is no response when only the button
                                          // is HIGH or no buttons
are HIGH.
```

```
                                        // Expect a win when the forward
button is pressed and the next bottom LED is on
                                                // and only one win occurs
                                    {F, NB} <= 2'b00;
                                    repeat(2) @(posedge clk);
                                    {F, NB} <= 2'b10;
                                    repeat(2) @(posedge clk);
            reset <= 1;
                                                                    @(posedge
clk);
            reset <= 0;
                                    {F, NB} <= 2'b00;
                                    repeat(2) @(posedge clk);
                                    {F, NB} <= 2'b01;
                                    repeat(2) @(posedge clk);
            reset <= 1;
                                                                    @(posedge
clk);
            reset <= 0;
                                    {F, NB} <= 2'b00;
                                    repeat(2) @(posedge clk);
                                    {F, NB} <= 2'b11;
                                    repeat(5) @(posedge clk);
                                    {F, NB} <= 2'b00;
                                    repeat(2) @(posedge clk);
                                    {F, NB} <= 2'b11;
                                    repeat(5) @(posedge clk);
            $stop; // End the simulation
      end
endmodule
```

loseChecker and Testbench:

```
// loseChecker determines whether the frog ran into a car or not
// car is the position of the cars on the 16x16 LED Array
// frog is the position of the frog on the 16x16 LED Array
// lose is whether the frog ran into a car or not, 1 if it did.
module loseChecker (car, frog, lose);
      input logic [15:0][15:0] car; // array of red LEDs
      input logic [15:0][15:0] frog; // array of green LEDs
      output logic lose;

      // lose is determined if the position of the frog matches
            // the position of a car. This is only relevant for the
            // 1st and 3rd rows of the LED Array (starting with 0th
            // row at the top), where the cars are.
      assign lose = ((car[1][15] && frog[1][15]) |
            (car[1][14] && frog[1][14]) |
            (car[1][13] && frog[1][13]) |
            (car[1][12] && frog[1][12]) |
            (car[1][11] && frog[1][11]) |
            (car[1][10] && frog[1][10]) |
            (car[3][10] && frog[3][10]) |
            (car[3][11] && frog[3][11]) |
            (car[3][12] && frog[3][12]) |
            (car[3][13] && frog[3][13]) |
            (car[3][14] && frog[3][14]) |
            (car[3][15] && frog[3][15]));
endmodule

// test situations for the frog and car positions
module loseChecker_testbench();
      // creates corresponding variables to model loseChecker module
      logic [15:0][15:0] car; // 16x16 array of red LEDs
      logic [15:0][15:0] frog; // 16x16 array of green LEDs
      logic lose;

      // initializes loseChecker module for testing with name dut
      loseChecker dut (.car, .frog, .lose);

      // checks whether a loss is output when the car and frog occupy
the same position.
            // Expected results is that lose goes HIGH.
      // also checks whether a loss is recorded when they are in
different positions.
            // Expected result is that lose stays LOW.
```

```
	initial begin
		car = '0; frog = '0;
		{car[1][15], frog[1][15]} = 2'b00;
			#10;
		{car[1][15], frog[1][15]} = 2'b10;
			#10;
		{car[1][15], frog[1][15]} = 2'b11;
			#10;
		{car[1][15], frog[1][15]} = 2'b00;
			#10;
		{car[1][15], frog[3][15]} = 2'b11;
			#10;
		$stop; // End the simulation
	end
endmodule
```

counter and Testbench:

```
// module counter defines a 3-bit counter for counting 7 total won
rounds
      // for one player

// clk is the clock used for controlling input and output timing
// reset makes the count reset to 0
// win is whether a round was won
// out is the total times the player has won in the current game
module counter (clk, reset, win, lose, out);
      input logic clk, reset;
      input logic win, lose;
      output logic [2:0] out;


      // D Flip Flop implementation (DFFs)
      always_ff @(posedge clk) begin
            // on reset, the counter resets to 0
            if (reset)
                  out <= 3'b0;
            else
                  // if a round was won and the total number of times a
player has won
                        // is less than 7, then increase the total wins
by 1.
                  if (win && out < 7)
                        out <= out + 1;
                  // if a round was LOST and the total number of times a
player has won
                        // is more than 0, then decrease the total wins
by 1.
                  else if (lose && out > 0)
                        out <= out - 1;
                  else
                        // otherwise, maintain the same win count.
                        out <= out;
      end

endmodule


//Test/Simulate the State Machine
module counter_testbench();
      // creates corresponding variables to model inputDejammer module
      logic clk, reset;
      logic win, lose;
      logic [2:0] out;
```

```
      // initializes inputDejammer module for testing with name dut
      counter dut (clk, reset, win, lose, out);

      // Set up a simulated clock to toggle (from low to high or high
to low)
      // every 50 time steps
      parameter CLOCK_PERIOD=100;
      initial begin
            clk <= 0;
            forever #(CLOCK_PERIOD/2) clk <= ~clk;//toggle the clock
indefinitely
      end

      // Set up the inputs to the design.  Each line represents a clock
cycle
      // Tests responses to won rounds, or rounds that have not been
won yet.
      // Expected results are that when a win is recorded, the count
increases by 1 with a max of 7,
      // If a loss is recorded, the count decreases by 1, with a
minimum of 0.
      initial begin
                  reset <= 1;
        @(posedge clk); // Always reset FSMs at start
                  reset <= 0;      win <= 0; lose <= 0;
@(posedge clk);
                  win <= 1;
@(posedge clk);
                  win <= 0;                                   repeat(2)
@(posedge clk);
                  lose <= 1;
@(posedge clk);
                  lose <= 0;                                  repeat(2)
@(posedge clk);
                  lose <= 1;
@(posedge clk);
                  lose <= 0;                                  repeat(2)
@(posedge clk);
                  win <= 1;
@(posedge clk);
                  win <= 0;                                   repeat(2)
@(posedge clk);
                  win <= 1;
@(posedge clk);
                  win <= 0;                                   repeat(2)
@(posedge clk);
```

```
                win <= 1;
@(posedge clk);
                win <= 0;                                   repeat(2)
@(posedge clk);
                win <= 1;
@(posedge clk);
                win <= 0;                                   repeat(2)
@(posedge clk);
                win <= 1;
@(posedge clk);
                win <= 0;                                   repeat(2)
@(posedge clk);
                win <= 1;
@(posedge clk);
                win <= 0;                                   repeat(2)
@(posedge clk);
                win <= 1;
@(posedge clk);
                win <= 0;                                   repeat(2)
@(posedge clk);
                win <= 1;
@(posedge clk);
                win <= 0;                                   repeat(2)
@(posedge clk);

                $stop; // End the simulation
           end
endmodule
```

seg7:

```
// module seg7 controls output to a hex display depending on count.
      // output leds is a number from 0 through 7 depending on how many
rounds were
      // won by respective player.
// This module is based off the template given by Lab3.
module seg7 (count, leds);
      input  logic  [2:0] count;
      output logic  [6:0] leds;

      // count's value corresponds how many rounds a player has won
            // minus however many times they lost. The hex displays
            // the corresponding value by specifying exactly which
segments are on or off.
      always_comb begin
            case (count)
                  //          Light: 6543210
                  default: leds = 7'b1111111; // none active
                  3'b000: leds = ~7'b0111111; // 0
                  3'b001: leds = ~7'b0000110; // 1
                  3'b010: leds = ~7'b1011011; // 2
                  3'b011: leds = ~7'b1001111; // 3
                  3'b100: leds = ~7'b1100110; // 4
                  3'b101: leds = ~7'b1101101; // 5
                  3'b110: leds = ~7'b1111101; // 6
                  3'b111: leds = ~7'b0000111; // 7
            endcase
      end
endmodule
```

clock_divider:

```
// This module divides the on-board FPGA clock at 50Mhz to
// divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz,
// [24] = 1.5Hz, [25] = 0.75Hz, ...and so on.
module clock_divider (clock, divided_clocks);
      // creates input variables reset and clock with type logic
      input logic clock;
      // creates output variables divided_clocks[31:0] with type logic.
      output logic  [31:0]  divided_clocks = 0;

      // generates the appropriate size for the divided clock
      always_ff @(posedge clock) begin
            divided_clocks <= divided_clocks + 1;
      end
endmodule
```

LEDDriver:

```
// A driver for the 16x16x2 LED display expansion board.
// Read below for an overview of the ports.
// IMPORTANT: You do not need to necessarily modify this file. But if
you do, be sure you know what you are doing.

// FREQDIV: (Parameter) Sets the scanning speed (how often the display
cycles through rows)
//          The CLK input divided by 2^(FREQDIV) is the interval at
which the driver switches rows.
// GPIO_1: (Output) The 36-pin GPIO1 header, as on the DE1-SoC board.
// RedPixels: (Input) A 16x16 array of logic items corresponding to
the red pixels you'd like to have lit on the display.
// GrnPixels: (Input) A 16x16 array of logic items corresponding to
the green pixels you'd like to have lit on the display.
// EnableCount: (Input) Whether to continue moving through the rows.
// CLK: (Input) The system clock.
// RST: (Input) Resets the display driver. Required during startup
before use.
module LEDDriver #(parameter FREQDIV = 0) (GPIO_1, RedPixels,
GrnPixels, EnableCount, CLK, RST);
    output logic [35:0] GPIO_1;
    input logic [15:0][15:0] RedPixels ;
    input logic [15:0][15:0] GrnPixels ;
    input logic EnableCount, CLK, RST;

    reg [(FREQDIV + 3):0] Counter;
    logic [3:0] RowSelect;
    assign RowSelect = Counter[(FREQDIV + 3):FREQDIV];

    always_ff @(posedge CLK)
    begin
        if(RST) Counter <= 'b0;
        if(EnableCount) Counter <= Counter + 1'b1;
    end

    assign GPIO_1[35:32] = RowSelect;
    assign GPIO_1[31:16] = { GrnPixels[RowSelect][0],
GrnPixels[RowSelect][1], GrnPixels[RowSelect][2],
GrnPixels[RowSelect][3], GrnPixels[RowSelect][4],
GrnPixels[RowSelect][5], GrnPixels[RowSelect][6],
GrnPixels[RowSelect][7], GrnPixels[RowSelect][8],
GrnPixels[RowSelect][9], GrnPixels[RowSelect][10],
GrnPixels[RowSelect][11], GrnPixels[RowSelect][12],
```

```
GrnPixels[RowSelect][13], GrnPixels[RowSelect][14],
GrnPixels[RowSelect][15] };
    assign GPIO_1[15:0] = { RedPixels[RowSelect][0],
RedPixels[RowSelect][1], RedPixels[RowSelect][2],
RedPixels[RowSelect][3], RedPixels[RowSelect][4],
RedPixels[RowSelect][5], RedPixels[RowSelect][6],
RedPixels[RowSelect][7], RedPixels[RowSelect][8],
RedPixels[RowSelect][9], RedPixels[RowSelect][10],
RedPixels[RowSelect][11], RedPixels[RowSelect][12],
RedPixels[RowSelect][13], RedPixels[RowSelect][14],
RedPixels[RowSelect][15] };
endmodule

module LEDDriver_Test();
    logic CLK, RST, EnableCount;
    logic [15:0][15:0]RedPixels;
    logic [15:0][15:0]GrnPixels;
    logic [35:0] GPIO_1;

    LEDDriver #(.FREQDIV(2)) Driver(.GPIO_1, .RedPixels, .GrnPixels,
.EnableCount, .CLK, .RST);

    initial
    begin
        CLK <= 1'b0;
        forever #50 CLK <= ~CLK;
    end

    initial
    begin
        EnableCount <= 1'b0;
        RedPixels <= '{default:0};
        GrnPixels <= '{default:0};
        @(posedge CLK);

        RST <= 1; @(posedge CLK);
        RST <= 0; @(posedge CLK);
        @(posedge CLK); @(posedge CLK); @(posedge CLK);

        GrnPixels[1][1] <= 1'b1; @(posedge CLK);
        EnableCount <= 1'b1; @(posedge CLK); #1000;
        RedPixels[2][2] <= 1'b1;
        RedPixels[2][3] <= 1'b1;
        GrnPixels[2][3] <= 1'b1; @(posedge CLK); #1000;
        EnableCount <= 1'b0; @(posedge CLK); #1000;
```

```
        GrnPixels[1][1] <= 1'b0; @(posedge CLK);
        $stop;

    end
endmodule

module LEDDriver_TestPhysical(CLOCK_50, RST, Speed, GPIO_1);
    input logic CLOCK_50, RST;
    input logic [9:0] Speed;
    output logic [35:0] GPIO_1;
    logic [15:0][15:0]RedPixels;
    logic [15:0][15:0]GrnPixels;
    logic [31:0] Counter;
    logic EnableCount;

    LEDDriver #(.FREQDIV(15)) Driver (.CLK(CLOCK_50), .RST,
.EnableCount, .RedPixels, .GrnPixels, .GPIO_1);

    //                      F E D C B A 9 8 7 6 5 4 3 2 1 0
    assign RedPixels[00] = '{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
    assign RedPixels[01] = '{1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1};
    assign RedPixels[02] = '{1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1};
    assign RedPixels[03] = '{1,0,1,1,0,0,0,0,0,0,0,0,1,1,0,1};
    assign RedPixels[04] = '{1,0,1,0,1,1,1,1,1,1,1,1,0,1,0,1};
    assign RedPixels[05] = '{1,0,1,0,1,1,0,0,0,0,1,1,0,1,0,1};
    assign RedPixels[06] = '{1,0,1,0,1,0,1,1,1,1,0,1,0,1,0,1};
    assign RedPixels[07] = '{1,0,1,0,1,0,1,0,1,1,0,1,0,1,0,1};
    assign RedPixels[08] = '{1,0,1,0,1,0,1,1,0,1,0,1,0,1,0,1};
    assign RedPixels[09] = '{1,0,1,0,1,0,1,1,1,1,0,1,0,1,0,1};
    assign RedPixels[10] = '{1,0,1,0,1,1,0,0,0,0,1,1,0,1,0,1};
    assign RedPixels[11] = '{1,0,1,0,1,1,1,1,1,1,1,1,0,1,0,1};
    assign RedPixels[12] = '{1,0,1,1,0,0,0,0,0,0,0,0,1,1,0,1};
    assign RedPixels[13] = '{1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1};
    assign RedPixels[14] = '{1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1};
    assign RedPixels[15] = '{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};

    assign GrnPixels[00] = '{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
    assign GrnPixels[01] = '{0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0};
    assign GrnPixels[02] = '{0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0};
    assign GrnPixels[03] = '{0,1,0,1,1,1,1,1,1,1,1,1,1,0,1,0};
    assign GrnPixels[04] = '{0,1,0,1,1,0,0,0,0,0,0,1,1,0,1,0};
    assign GrnPixels[05] = '{0,1,0,1,0,1,1,1,1,1,1,0,1,0,1,0};
    assign GrnPixels[06] = '{0,1,0,1,0,1,1,0,0,1,1,0,1,0,1,0};
    assign GrnPixels[07] = '{0,1,0,1,0,1,0,1,0,0,1,0,1,0,1,0};
    assign GrnPixels[08] = '{0,1,0,1,0,1,0,0,1,0,1,0,1,0,1,0};
```

```
    assign GrnPixels[09] = '{0,1,0,1,0,1,1,0,0,1,1,0,1,0,1,0};
    assign GrnPixels[10] = '{0,1,0,1,0,1,1,1,1,1,1,0,1,0,1,0};
    assign GrnPixels[11] = '{0,1,0,1,1,0,0,0,0,0,0,1,1,0,1,0};
    assign GrnPixels[12] = '{0,1,0,1,1,1,1,1,1,1,1,1,1,0,1,0};
    assign GrnPixels[13] = '{0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0};
    assign GrnPixels[14] = '{0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0};
    assign GrnPixels[15] = '{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1};

    always_ff @(posedge CLOCK_50)
    begin
        if(RST) Counter <= 'b0;
        else
        begin
            Counter <= Counter + 1'b1;
            if(Counter >= Speed)
            begin
                EnableCount <= 1'b1;
                Counter <= 'b0;
            end
            else EnableCount <= 1'b0;
        end
    end
endmodule
```