ECE385 Final Project Report

Frogger in System Verilog

Eric Meyers, Ryan Helsdingen

Section ABG; TAs: Ben Delay, Shuo Liu

May 4th, 2016

emeyer7, helsdin2

I. Introduction

For this project, we recreated the classic game called Frogger. The basic premise of Frogger is to navigate three frogs through obstacles from the bottom to the top of the screen. Frogger must first pass through four lanes of traffic and a river full of lily pads in order to successfully make it to the other side of the map. A frog may die by either colliding with a moving car or falling in the water. There are a total of three frogs that the user must navigate to the other end of the map, and once all three frogs move to their particular ending location, the user wins. If a user dies three times, then the game is over.

This system was developed in System Verilog in Quartus-II on an Altera-DE2-115 FPGA Board, and used software drivers developed in C to communicate with a USB keyboard (to be used as the controller).

II. List of Features

- User controlled frog sprite moves according to grid set on VGA display
  - Up, down, left, or right depending on keyboard input
  - Frog position restricted to on-screen
  - Frog direction held after keyboard direction inputed
  - Frog contains collision algorithms for obstacles
  - A total of three frogs placed into game at three different starting points
    - three endpoints for frogs to get to
    - frog can finish at any endpoint but only one frog can finish per endpoint
- Generation of moving obstacles
  - Four lanes of at most four cars in them
    - Each row moves at specific speed, direction
    - Cars wrap around screen
    - Frogger dies upon impact with car ending the game
  - Four lanes of at most four lilypads in them
    - Each row moves at specific speed, direction
    - Lily pads wrap around screen
    - Frogger lives by stepping onto the lily pad, drowns by falling into the water
    - If the lily pad moves off the screen with Frogger on it, Frogger will fall into the water
- Working game clock at the top of the screen.
  - 60 seconds to complete the level.
  - Clock resets at the start of the game
  - Game ends when the game clock runs out of time
- Color/VGA Monitor Output
  - Detailed sprites give look and feel in graphical user interface
  - Colors allow user to clearly differentiate between obstacle, user controlled frog, and the map background
- Game reset button established that can reset the game at any point.

III. Block Diagram

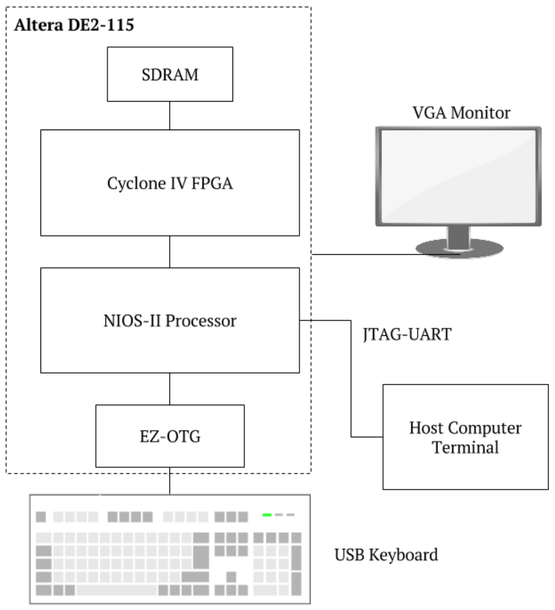
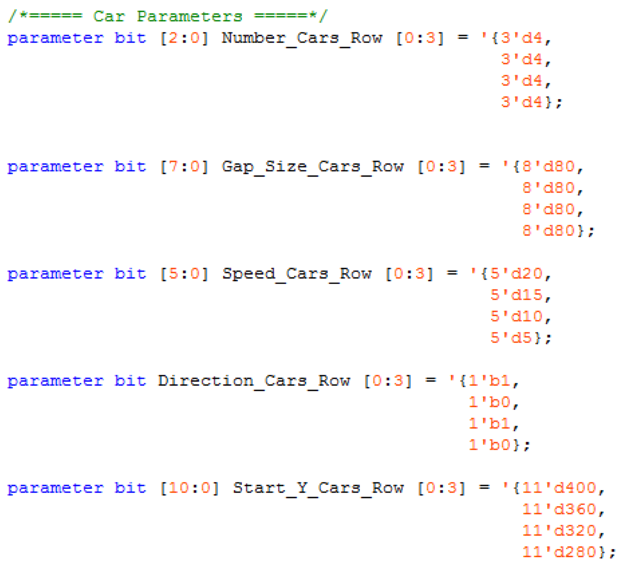


Fig. 1: Top Block Diagram

IV. Purpose of Modules

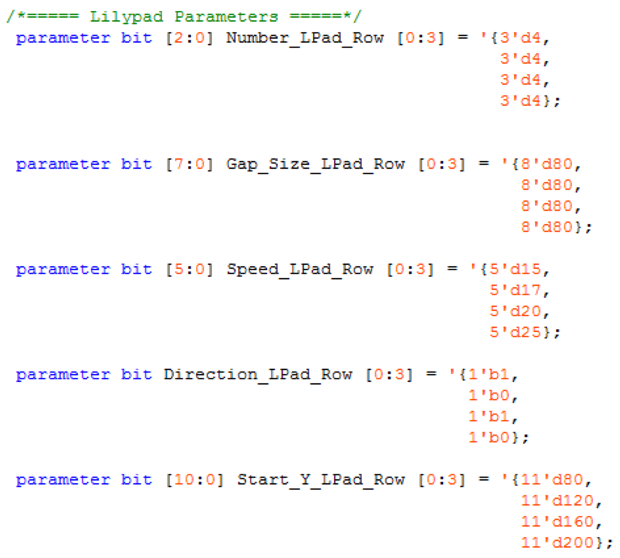
A. *final_frogger_top*

This is the top level module that both initializes all modules to use in the game and initializes their parameters. Three frogs are initialized and depending on the button the user presses (either 1, 2, or 3 on the numpad) either one is active at a single instant. Both cars and lilypads are initialized using parameter constant arrays, as shown in Figure 2 and Figure 3.



```
/*===== Car Parameters =====*/
parameter bit [2:0] Number_Cars_Row [0:3] = '{3'd4,
                                               3'd4,
                                               3'd4,
                                               3'd4};


parameter bit [7:0] Gap_Size_Cars_Row [0:3] = '{8'd80,
                                                 8'd80,
                                                 8'd80,
                                                 8'd80};

parameter bit [5:0] Speed_Cars_Row [0:3] = '{5'd20,
                                              5'd15,
                                              5'd10,
                                              5'd5};

parameter bit Direction_Cars_Row [0:3] = '{1'b1,
                                            1'b0,
                                            1'b1,
                                            1'b0};

parameter bit [10:0] Start_Y_Cars_Row [0:3] = '{11'd400,
                                                 11'd360,
                                                 11'd320,
                                                 11'd280};
```

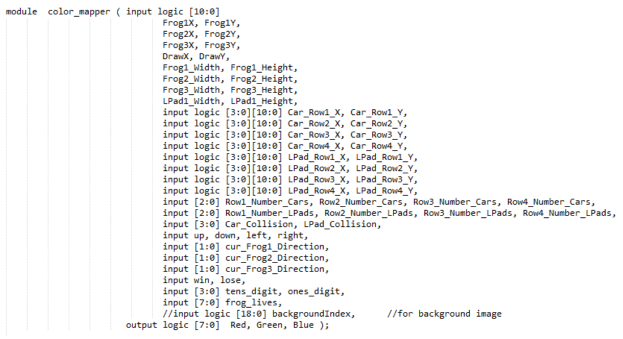
Fig. 2: Car Parameters



```
/*===== Lilypad Parameters =====*/
 parameter bit [2:0] Number_LPad_Row [0:3] = '{3'd4,
                                               3'd4,
                                               3'd4,
                                               3'd4};


 parameter bit [7:0] Gap_Size_LPad_Row [0:3] = '{8'd80,
                                                 8'd80,
                                                 8'd80,
                                                 8'd80};

 parameter bit [5:0] Speed_LPad_Row [0:3] = '{5'd15,
                                              5'd17,
                                              5'd20,
                                              5'd25};

 parameter bit Direction_LPad_Row [0:3] = '{1'b1,
                                            1'b0,
                                            1'b1,
                                            1'b0};

 parameter bit [10:0] Start_Y_LPad_Row [0:3] = '{11'd80,
                                                 11'd120,
                                                 11'd160,
                                                 11'd200};
```

Fig. 3: Lilypad Parameters

B. *Color_Mapper*

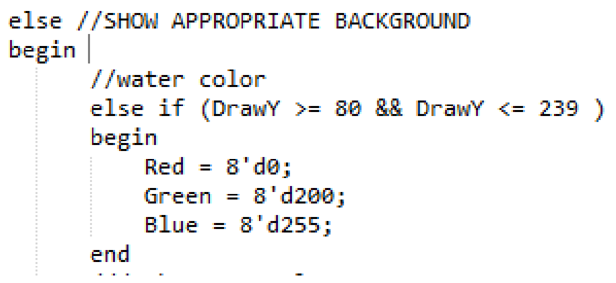


```
module  color_mapper ( input logic [10:0]
                              Frog1X, Frog1Y,
                              Frog2X, Frog2Y,
                              Frog3X, Frog3Y,
                              DrawX, DrawY,
                              Frog1_Width, Frog1_Height,
                              Frog2_Width, Frog2_Height,
                              Frog3_Width, Frog3_Height,
                              LPad1_Width, LPad1_Height,
                              input logic [3:0][10:0] Car_Row1_X, Car_Row1_Y,
                              input logic [3:0][10:0] Car_Row2_X, Car_Row2_Y,
                              input logic [3:0][10:0] Car_Row3_X, Car_Row3_Y,
                              input logic [3:0][10:0] Car_Row4_X, Car_Row4_Y,
                              input logic [3:0][10:0] LPad_Row1_X, LPad_Row1_Y,
                              input logic [3:0][10:0] LPad_Row2_X, LPad_Row2_Y,
                              input logic [3:0][10:0] LPad_Row3_X, LPad_Row3_Y,
                              input logic [3:0][10:0] LPad_Row4_X, LPad_Row4_Y,
                              input [2:0] Row1_Number_Cars, Row2_Number_Cars, Row3_Number_Cars, Row4_Number_Cars,
                              input [2:0] Row1_Number_LPads, Row2_Number_LPads, Row3_Number_LPads, Row4_Number_LPads,
                              input [3:0] Car_Collision, LPad_Collision,
                              input up, down, left, right,
                              input [1:0] cur_Frog1_Direction,
                              input [1:0] cur_Frog2_Direction,
                              input [1:0] cur_Frog3_Direction,
                              input win, lose,
                              input [3:0] tens_digit, ones_digit,
                              input [7:0] frog_lives,
                              //input logic [18:0] backgroundIndex,      //for background image
                         output logic [7:0]  Red, Green, Blue );
```

Fig. 4: Color_mapper.sv input/output.

The Color_Mapper module is the heart of controlling what gets displayed onto the screen. The module was inspired by Lab 8 with the moving ball sprite. This version of Color_Mapper has several jobs including knowing the position of and displaying all 39 sprites for the game at any given instance, establishing the proper orientation of Frogger, displaying the game clock, number of lives, and game title, as well as properly layering the background to the back of the game. Figure 4 is a view of the inputs and outputs of the module showing just how involved this module really is to the Frogger game.

The sprite generation will be discussed in full detail in ”Section XII - Color and Sprite Generation”. Though it is worth noting how the background was implemented with so many sprites on the screen. A large if-else statement was used to control the layers on the screen with front layers appearing earlier in the if-else logic and background layers appearing as the else statement. Figure 5 shows only the first background layer of the else condition, the blue color of the water meant to be displayed behind all lilypad sprites.



```
else //SHOW APPROPRIATE BACKGROUND
begin
    //water color
    else if (DrawY >= 80 && DrawY <= 239 )
    begin
        Red = 8'd0;
        Green = 8'd200;
        Blue = 8'd255;
    end
```

Fig. 5: Generating a background image.

The two variables DrawX and DrawY represent the given pixel that is to be colored in. The two variables run through the if-else logic until they find conditions in which they satisfy. For the example shown in Figure 5, any uncolored pixels left with Y coordinates between 80 and 239 inclusive are filled with the RGB value within the else if statement.

C. *frog*

The frog module contains the logic for the moving component controlled by the user (the “frog”). This entails the logic for what the frog does when detecting it is colliding with a car, a lilypad, water, or the end position. The frog state diagram is explained below in the ”Finite State Machines” section.

This module takes inputs as the desired start position of the frog, the controller keys (up, down, left, right), lilypad collision signals, car collision signals, and lilypad parameters (so it can sync up with the movements of the lilypad). The module outputs the frog x and y position to display on the screen.

D. *car*

The car module contains the logic for the moving cars in the bottom half of the screen. This module also detects if there is a collision between its location and the location of the frog. It takes input of the desired speed/direction and updates/outputs the position of the car to display on the screen. It also takes inputs of the frog x and y position so it can determine if the car is collided with the frog. If it is, then it outputs a collision signal.

The logic for the collision detection is shown in Figure 12.

E. *car_row*

The car_row module uses a generate block to create four car modules and depending on the number of cars used, it updates the collision signal. It takes inputs as the number of cars, speed/direction of cars, and the gap size in between each car. This is shown in Figure 14

F. *lilypad*

The lilypad module is similar to the functionality of the car module. This module contains the logic for the moving lilypads in the top half of the screen. It takes inputs of the desired direction/speed along with the frog x and y position and outputs the new location of the lilypad to draw to the screen. The frog x and y position are used to determine if a collision is occuring between the frog and the lilypad. This will output a collision signal if so.

Lilypads differ from cars in that they “carry” the frog once the two objects collide. This means that there must be a method to “sync” up the movements of both the lilypad and the frog once they collide. For this reason, the lilypad module also must output its remaining count on its state machine cycle for its speed. This is so that if the frog lands on the lilypad midway through its “move” cycle, then the frog can essentially take over the count of the lilypad and be synced up and move accordingly.

G. *lilypad_row*

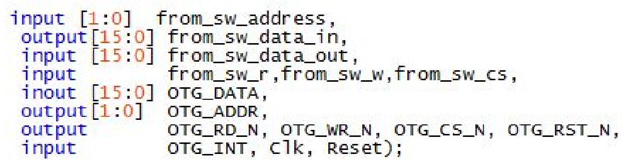
The lilypad_row module is similar to that of the car_row module. Th

H. *game_logic*

Module game_logic.sv controls the overall finite state machine of the Frogger game. This module monitors number of frogs to successfully get to the other side of the screen. It then sends the game into a win or death state depending on the outcome of the game. This module also controls the value of the game clock. If the game clock runs out of time before the three frogs reach the other side, the game ends and the player loses.

I. *hpi_io_intf*

This method takes HPI output values from the NIOS II as input, checks for a call to reset, and if no reset, assigns these values to the proper values to be outputted by the top-level module to the CY7C67200 chip. Inputs and outputs used are shown below.



```
input  [1:0]   from_sw_address,
 output[15:0] from_sw_data_in,
 input  [15:0] from_sw_data_out,
 input         from_sw_r,from_sw_w,from_sw_cs,
 inout  [15:0] OTG_DATA,
 output[1:0]   OTG_ADDR,
 output        OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N,
 input         OTG_INT, Clk, Reset);
```

J. *VGA_controller*

The VGA_controller manages output to the monitor. Inputs include clock and reset. Outputs include several sync signals, a pixel clock specified for 25 MHz and 10 bit horizontal and vertical coordinate signals as shown below.

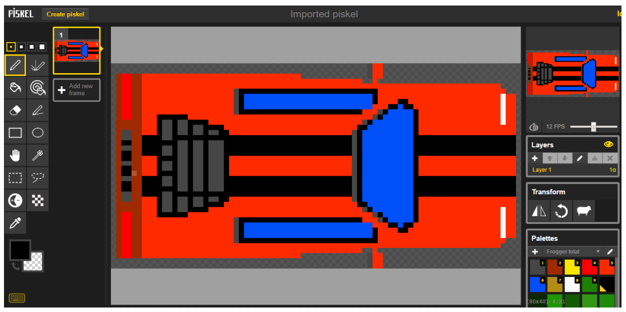


Fig. 6: Generating carright_sprite in Piskel.

V. Circuit Schematics

ERIC SECTION - easy, just when done take screenshots of the circuit, 5 min

VI. Finite State Machines

There were a total of 3 FSMs implemented in Frogger. One for the user-controlled component (the frog), another for the moving components (cars and lilypads), and the last for the game logic.

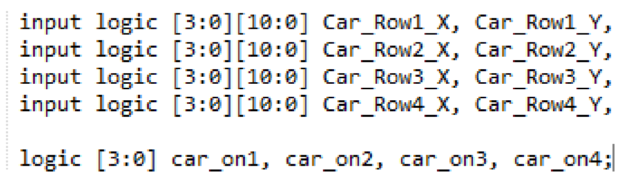
The frog module state diagram controls what happens to frogger when a particular action occurs. For example, if frogger collides with a car or water, it must die. However, if frogger collides with a lilypad, it must move at the same rate as the lilypad. These state machines are shown in Figures 16, 17, and 18.

VII. Color & Sprite Generation

A total of 39 sprites appear on the VGA monitor at any given moment. They consist of three (3) frogs, sixteen (16) lilypads, eight (8) cars facing to the left, eight (8) cars facing to the right, two (2) digits for representing the tens digit and ones digit of the game clock, (1) digit for representing the number of lives left, and (1) sprite for the logo image.

The logo is the least complex displaying a static picture of the game logo, while the frogs are the most complex sprites controlling both location and orientation. The cars and lilypads were complex in their own way since there were multiple sprites on multiple rows that needed position to be tracked. Let's look at the process used to get carright_sprite from the drawing board to the big screen as an example.

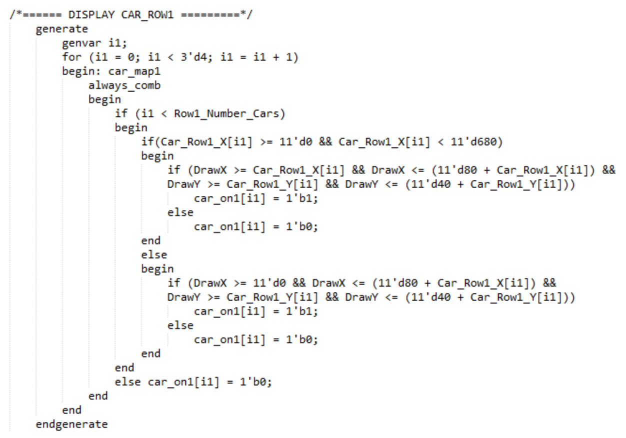
All sprites were born as .png files for clear, simple pixel mappings. It was not as easy as finding .png files from the internet or converting our own images into .png files, however. Even the clearest .png files had thousands of colors due to poor clarity. Developing sprites thus began with Piskel, a free online sprite editor, which allowed us to optimize for the minimal amount of color use. Figure 6 shows the creation of carright_sprite in Piskel. A Java file was used to generate all sprite files for the game mapping each pixel to a color. A palette, or array of colors used amongst all sprites, was also generated based on all sprites used in the project. This process made it easy to map the correct color from the palette.sv to the sprite generation file in the Color_Mapper module.



```
input logic [3:0][10:0] Car_Row1_X, Car_Row1_Y,
input logic [3:0][10:0] Car_Row2_X, Car_Row2_Y,
input logic [3:0][10:0] Car_Row3_X, Car_Row3_Y,
input logic [3:0][10:0] Car_Row4_X, Car_Row4_Y,

logic [3:0] car_on1, car_on2, car_on3, car_on4;
```

Fig. 7: Grabbing inputs for rightcar_sprite.

The Color_Mapper module takes five parts to properly map one of the carright_sprite(s) onto the screen. It first inputs the x and y location for each car_row as well as generates variables such as car_on1 to check if a car is on at a given pixel. Figure 7 shows a screenshot of these lines of code. The carright_sprite was only generated on rows 1 and 3, so those are the only rows cared about for this example.

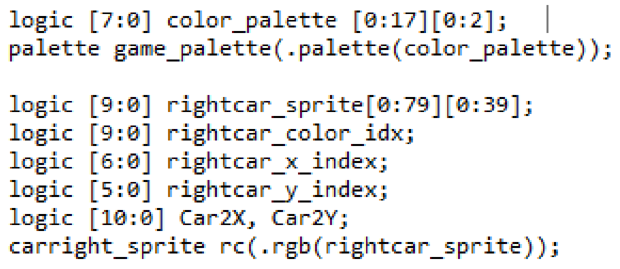


```
/*====== DISPLAY CAR_ROW1 =========*/
    generate
        genvar i1;
        for (i1 = 0; i1 < 3'd4; i1 = i1 + 1)
        begin: car_map1
            always_comb
            begin
                if (i1 < Row1_Number_Cars)
                begin
                    if(Car_Row1_X[i1] >= 11'd0 && Car_Row1_X[i1] < 11'd680)
                    begin
                        if (DrawX >= Car_Row1_X[i1] && DrawX <= (11'd80 + Car_Row1_X[i1]) &&
                        DrawY >= Car_Row1_Y[i1] && DrawY <= (11'd40 + Car_Row1_Y[i1]))
                            car_on1[i1] = 1'b1;
                        else
                            car_on1[i1] = 1'b0;
                    end
                    else
                    begin
                        if (DrawX >= 11'd0 && DrawX <= (11'd80 + Car_Row1_X[i1]) &&
                        DrawY >= Car_Row1_Y[i1] && DrawY <= (11'd40 + Car_Row1_Y[i1]))
                            car_on1[i1] = 1'b1;
                        else
                            car_on1[i1] = 1'b0;
                    end
                end
                else car_on1[i1] = 1'b0;
            end
        end
    endgenerate
```

Fig. 8: Determining location of cars in first car row.

Next four cars are mapped onto their given rows. In Figure 8 we see the first car row get mapped. Pixels specified by DrawX and DrawY values are given a specific car_on1 value, 1 if a car is at that pixel and 0 if there is no car at that pixel.

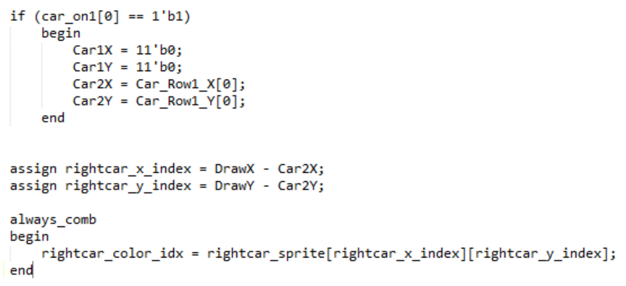
The third part of Color_Mapper initializes the sprite 2D pixel array as well as the palette. The palette only gets initialized once for all sprites. A color index, x and y index are also initialized here. Figure 9 shows this part for the carright_sprite.



```
logic [7:0] color_palette [0:17][0:2];
palette game_palette(.palette(color_palette));

logic [9:0] rightcar_sprite[0:79][0:39];
logic [9:0] rightcar_color_idx;
logic [6:0] rightcar_x_index;
logic [5:0] rightcar_y_index;
logic [10:0] Car2X, Car2Y;
carright_sprite rc(.rgb(rightcar_sprite));
```

Fig. 9: Initializing variables for sprites.

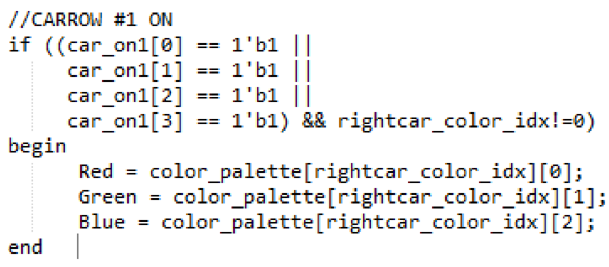
Following this, each car gets its upper left-hand corner (x,y) positioned marked. For this example, the first car in the first row is used. Car2X and Car2Y were designated to point to upper left-hand corner coordinates of this carright_sprite. Variables rightcar_x_index and rightcar_y_index are assigned to be the local x and y index within the sprite. These values are then used to map the proper color value at each pixel location to rightcar_color_idx.



```
if (car_on1[0] == 1'b1)
    begin
        Car1X = 11'b0;
        Car1Y = 11'b0;
        Car2X = Car_Row1_X[0];
        Car2Y = Car_Row1_Y[0];
    end


assign rightcar_x_index = DrawX - Car2X;
assign rightcar_y_index = DrawY - Car2Y;

always_comb
begin
    rightcar_color_idx = rightcar_sprite[rightcar_x_index][rightcar_y_index];
end
```

Fig. 10: Mapping the carright_sprite to its color index.

The fifth stage within Color_Mapper is to finally draw the sprite to the screen. A large if statement checks for DrawX and DrawY values to contain car_on values. This means the car exists at that pixel and it can be mapped to its proper color in the sprite. Figure 11 shows this being done for the carright_sprite. The RGB values are properly mapped to the correct values for the pixel and outputted from Color_Mapper to the VGA_controller to be displayed onto the VGA monitor.



```
//CARROW #1 ON
if ((car_on1[0] == 1'b1 ||
    car_on1[1] == 1'b1 ||
    car_on1[2] == 1'b1 ||
    car_on1[3] == 1'b1) && rightcar_color_idx!=0)
begin
     Red = color_palette[rightcar_color_idx][0];
     Green = color_palette[rightcar_color_idx][1];
     Blue = color_palette[rightcar_color_idx][2];
end
```

Fig. 11: Establishing RGB for each pixel in carright_sprite.

Every sprite that moved went through these five stages in the Color_Mapper module in order to be displayed. The game clock, lives counter, and game logo could bypass the first two stages since their position on the screen remained static. The game clock and lives counter required inputs to the Color_Mapper module to declare which number to display on the screen.

One last added feature not mentioned was the ability to control the direction of the frogs. A direction variable was used to record the last direction key used on the frog and orient the frog based on that direction. Matrix transformations were then used to rotate the 2D pixel array of the frog to the proper direction.

VIII. Difficulty

Producing the Frogger game in less than five weeks was no simple task. The team came across a number of issues on a range of the features that were implemented into the game. Most of the problems encountered came about from dealing with the System Verilog software. What would be a simple function in a high-level programming language became rather tedious in System Verilog and required some thought in order to successfully implement.

One of the most difficult tasks in making the game was collision control for the Frogger sprite with obstacles, particularly with the lily pads. The algorithm for collision control turned into a long AND/OR statement for the cars. Collision for the frog with the cars was easy once that long statement was put into place and modified for precision. It was easy because a simple contact between the frog sprite and car sprite killed the frog and ended the game. Creating collision control with the lily pads was far more difficult to implement. The frog should land on the lily pad and live, floating across the screen holding its position fixed to the lily pad until another direction key is pressed. Clock issues came into play when trying to match up the speeds of the two colliding sprites, especially when the two sprites have different frame clocks.

Another issue was generating the high level graphics for the sprites and background. The on-chip memory alone was not enough to hold the amount of colors and pixels of all graphics desired for the project. SDRAM had to be implemented to store the memory for all 640 x 480 pixels on the screen.

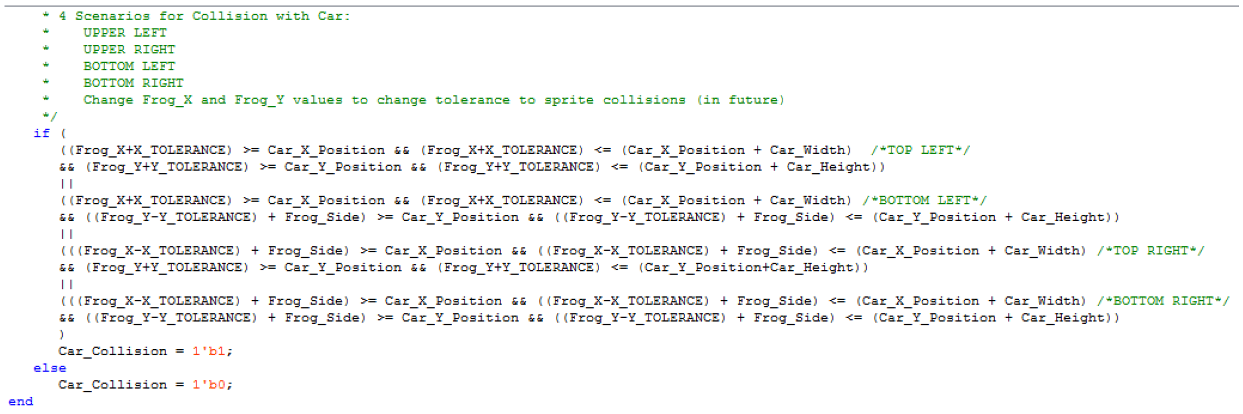
A LOT MORE COULD BE ADDED HERE

IX. Conclusion

Resource	Value
LUT	
DSP	
Memory (BRAM)	
Flip-Flop	
Frequency	
Static Power	
Dynamic Power	
Total Power	

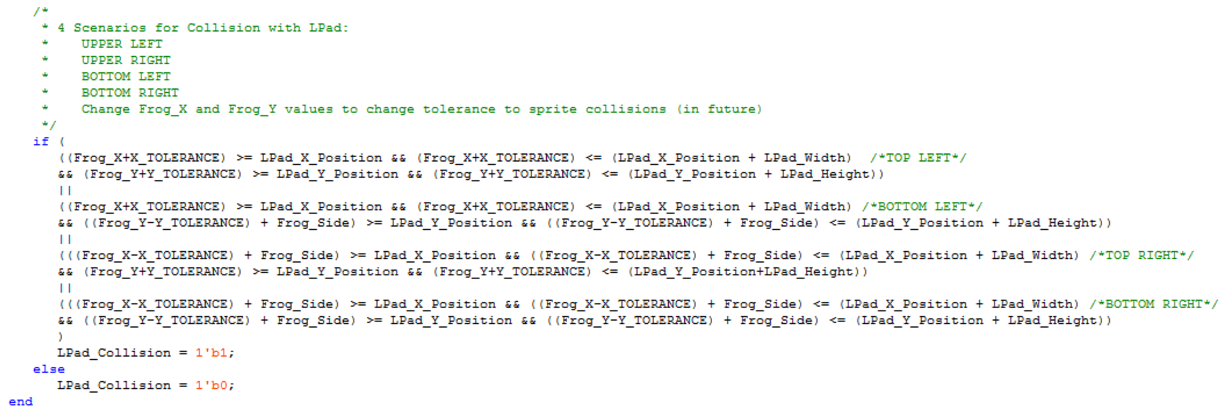
TABLE I: Design Statistics

X. Figures



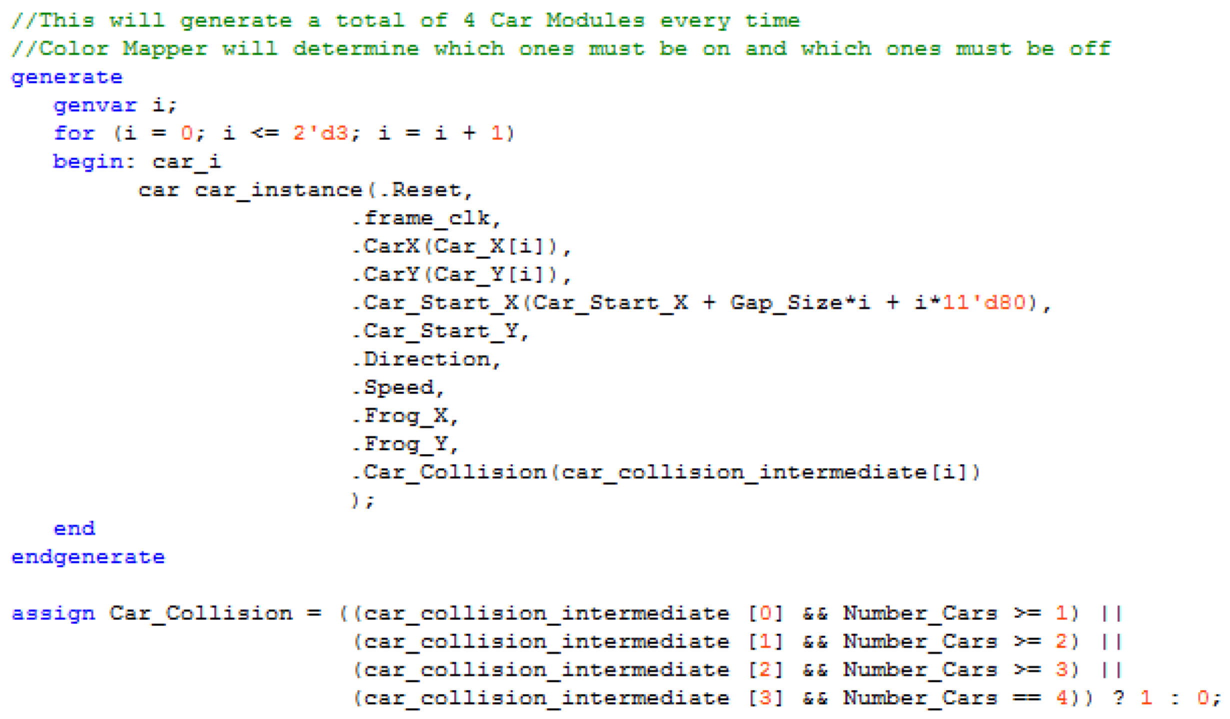
```
     * 4 Scenarios for Collision with Car:
     *    UPPER LEFT
     *    UPPER RIGHT
     *    BOTTOM LEFT
     *    BOTTOM RIGHT
     *    Change Frog_X and Frog_Y values to change tolerance to sprite collisions (in future)
     */
    if (
       ((Frog_X+X_TOLERANCE) >= Car_X_Position && (Frog_X+X_TOLERANCE) <= (Car_X_Position + Car_Width)  /*TOP LEFT*/
       && (Frog_Y+Y_TOLERANCE) >= Car_Y_Position && (Frog_Y+Y_TOLERANCE) <= (Car_Y_Position + Car_Height))
       ||
       ((Frog_X+X_TOLERANCE) >= Car_X_Position && (Frog_X+X_TOLERANCE) <= (Car_X_Position + Car_Width) /*BOTTOM LEFT*/
       && ((Frog_Y-Y_TOLERANCE) + Frog_Side) >= Car_Y_Position && ((Frog_Y-Y_TOLERANCE) + Frog_Side) <= (Car_Y_Position + Car_Height))
       ||
       (((Frog_X-X_TOLERANCE) + Frog_Side) >= Car_X_Position && ((Frog_X-X_TOLERANCE) + Frog_Side) <= (Car_X_Position + Car_Width) /*TOP RIGHT*/
       && (Frog_Y+Y_TOLERANCE) >= Car_Y_Position && (Frog_Y+Y_TOLERANCE) <= (Car_Y_Position+Car_Height))
       ||
       (((Frog_X-X_TOLERANCE) + Frog_Side) >= Car_X_Position && ((Frog_X-X_TOLERANCE) + Frog_Side) <= (Car_X_Position + Car_Width) /*BOTTOM RIGHT*/
       && ((Frog_Y-Y_TOLERANCE) + Frog_Side) >= Car_Y_Position && ((Frog_Y-Y_TOLERANCE) + Frog_Side) <= (Car_Y_Position + Car_Height))
       )
       Car_Collision = 1'b1;
    else
       Car_Collision = 1'b0;
end
```

Fig. 12: Car Collision Logic



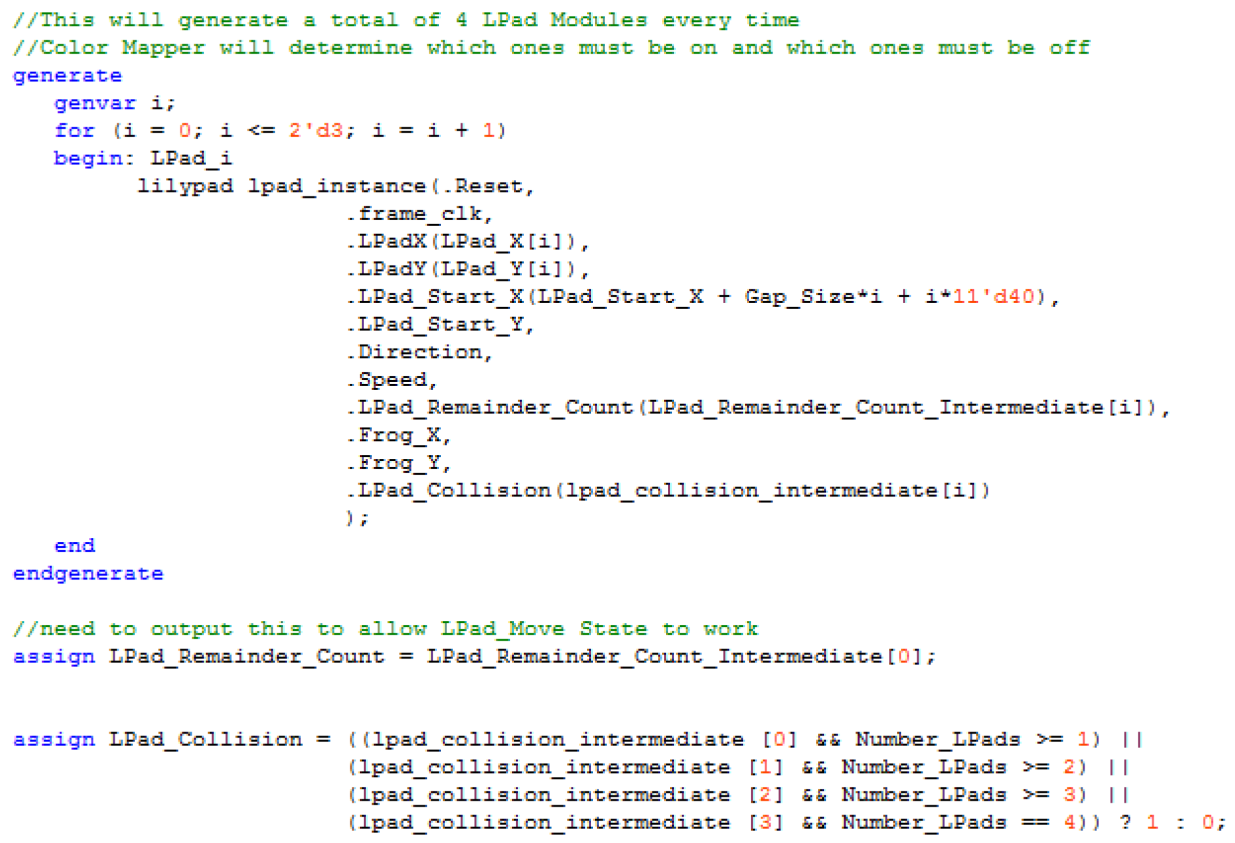
```
    /*
     * 4 Scenarios for Collision with LPad:
     *    UPPER LEFT
     *    UPPER RIGHT
     *    BOTTOM LEFT
     *    BOTTOM RIGHT
     *    Change Frog_X and Frog_Y values to change tolerance to sprite collisions (in future)
     */
    if (
       ((Frog_X+X_TOLERANCE) >= LPad_X_Position && (Frog_X+X_TOLERANCE) <= (LPad_X_Position + LPad_Width)  /*TOP LEFT*/
       && (Frog_Y+Y_TOLERANCE) >= LPad_Y_Position && (Frog_Y+Y_TOLERANCE) <= (LPad_Y_Position + LPad_Height))
       ||
       ((Frog_X+X_TOLERANCE) >= LPad_X_Position && (Frog_X+X_TOLERANCE) <= (LPad_X_Position + LPad_Width) /*BOTTOM LEFT*/
       && ((Frog_Y-Y_TOLERANCE) + Frog_Side) >= LPad_Y_Position && ((Frog_Y-Y_TOLERANCE) + Frog_Side) <= (LPad_Y_Position + LPad_Height))
       ||
       (((Frog_X-X_TOLERANCE) + Frog_Side) >= LPad_X_Position && ((Frog_X-X_TOLERANCE) + Frog_Side) <= (LPad_X_Position + LPad_Width) /*TOP RIGHT*/
       && (Frog_Y+Y_TOLERANCE) >= LPad_Y_Position && (Frog_Y+Y_TOLERANCE) <= (LPad_Y_Position+LPad_Height))
       ||
       (((Frog_X-X_TOLERANCE) + Frog_Side) >= LPad_X_Position && ((Frog_X-X_TOLERANCE) + Frog_Side) <= (LPad_X_Position + LPad_Width) /*BOTTOM RIGHT*/
       && ((Frog_Y-Y_TOLERANCE) + Frog_Side) >= LPad_Y_Position && ((Frog_Y-Y_TOLERANCE) + Frog_Side) <= (LPad_Y_Position + LPad_Height))
       )
       LPad_Collision = 1'b1;
    else
       LPad_Collision = 1'b0;
end
```

Fig. 13: Lilypad Collision Logic



```
//This will generate a total of 4 Car Modules every time
//Color Mapper will determine which ones must be on and which ones must be off
generate
   genvar i;
   for (i = 0; i <= 2'd3; i = i + 1)
   begin: car_i
         car car_instance(.Reset,
                          .frame_clk,
                          .CarX(Car_X[i]),
                          .CarY(Car_Y[i]),
                          .Car_Start_X(Car_Start_X + Gap_Size*i + i*11'd80),
                          .Car_Start_Y,
                          .Direction,
                          .Speed,
                          .Frog_X,
                          .Frog_Y,
                          .Car_Collision(car_collision_intermediate[i])
                          );
   end
endgenerate

assign Car_Collision = ((car_collision_intermediate [0] && Number_Cars >= 1) ||
                        (car_collision_intermediate [1] && Number_Cars >= 2) ||
                        (car_collision_intermediate [2] && Number_Cars >= 3) ||
                        (car_collision_intermediate [3] && Number_Cars == 4)) ? 1 : 0;
```

Fig. 14: Car Row Generation



```
//This will generate a total of 4 LPad Modules every time
//Color Mapper will determine which ones must be on and which ones must be off
generate
   genvar i;
   for (i = 0; i <= 2'd3; i = i + 1)
   begin: LPad_i
         lilypad lpad_instance(.Reset,
                               .frame_clk,
                               .LPadX(LPad_X[i]),
                               .LPadY(LPad_Y[i]),
                               .LPad_Start_X(LPad_Start_X + Gap_Size*i + i*11'd40),
                               .LPad_Start_Y,
                               .Direction,
                               .Speed,
                               .LPad_Remainder_Count(LPad_Remainder_Count_Intermediate[i]),
                               .Frog_X,
                               .Frog_Y,
                               .LPad_Collision(lpad_collision_intermediate[i])
                               );
   end
endgenerate

//need to output this to allow LPad_Move State to work
assign LPad_Remainder_Count = LPad_Remainder_Count_Intermediate[0];


assign LPad_Collision = ((lpad_collision_intermediate [0] && Number_LPads >= 1) ||
                         (lpad_collision_intermediate [1] && Number_LPads >= 2) ||
                         (lpad_collision_intermediate [2] && Number_LPads >= 3) ||
                         (lpad_collision_intermediate [3] && Number_LPads == 4)) ? 1 : 0;
```

Fig. 15: Lilypad Row Generation

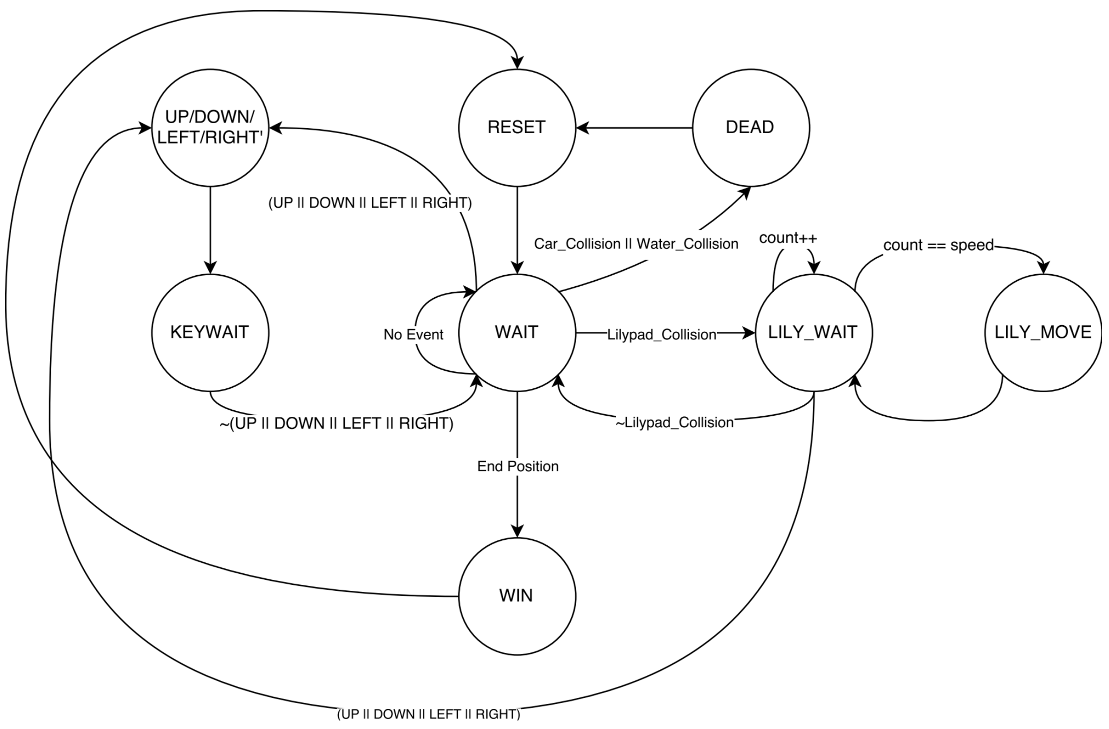


Fig. 16: Frogger State Diagram

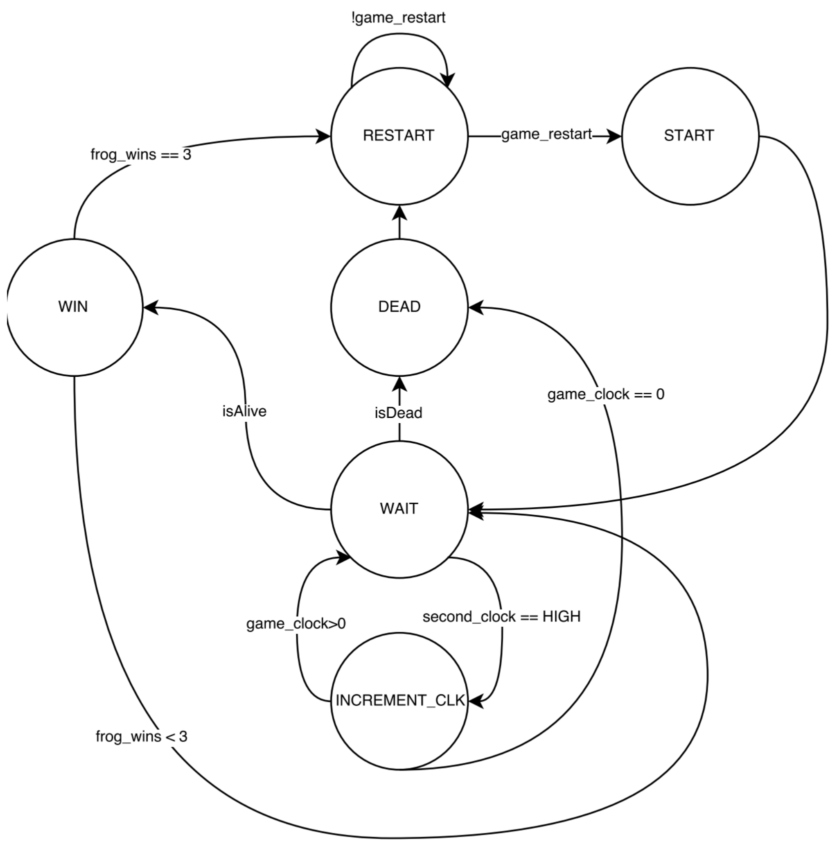


Fig. 17: Game Logic State Diagram

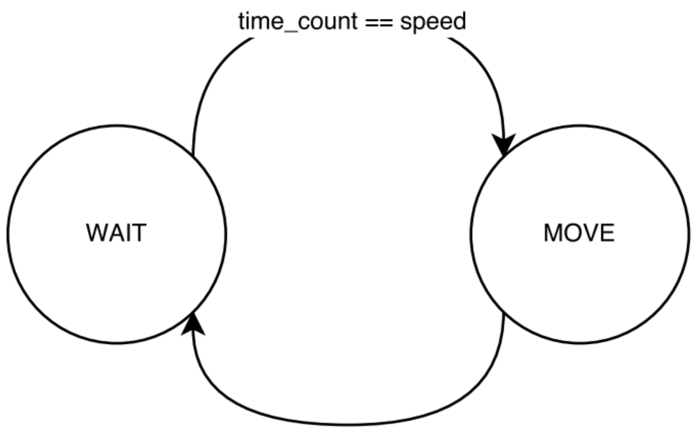


Fig. 18: Moving Obstacles State Diagram

Appendix