Ben Nguyen
Grant Wheatley
EE 371, Fall 2018
11 December 2018
Final Lab Report

Dodge Snowball

Abstract:

This lab consisted of building a project where we decided on the design of the project. Given the system requirements the project must meet, we constructed a video game called Dodge Snowball.

Dodge Snowball is a two player game. Player one must dodge the snowballs that are spawned by player two. The game is over when player one gets hit by a snowball.

We were able to get the game working however we had a few visual issues on the screen as well as our microphone not working when we are using our camera.

Introduction:

Dodge Snowball is a two player video game with a resolution of 320 x 240. Player one must avoid the snowballs produced by player two. In the event a snowball comes into contact on player one, player two wins. A snowball is considered in contact with player one when a snowball is directly in front of player one, or when a snowball is located one pixel to the right of player one. A representation of this video game is displayed in Figure 1. Figure 1 displays, with respect to the number listed on the Figure 1:

1. Offset from the edge of the left display where locations where player one can toggle. This offset is of size 10 x 240.
2. The location of player one. Player one is size 3x3 and is able to move up or down in steps of three pixels, within a 3 x 240 box. Therefore, player one can toggle between 80 different locations. Figure 2 displays the design of player one.
3. A 270 x 240 snowball field where snowballs can be spawned by player two. Each snowball is 3x3, and will move from right to left, in steps of three pixels. Therefore, there are 90 different locations a snowball can exist, with respect to it's y-location. There are 80 different y-locations. Figure 3 displays an example of three snowballs in a snowball field.
4. Snowballs can be spawned within a 3 x 240 box. A single snowball is displayed, and is spawned relative to player two. There is an offset between player two and the snowball(s) produced of 6 x 240.

5. The location of player two. Player two is size 3x3 and is able to move up or down in steps of three pixels, within a 3 x 240 box. Therefore, player two can toggle between 80 different locations. Figure 2 displays the design of player two.
6. Offset from the edge of the right display where locations where player two can toggle. This offset is of size 10 x 240.

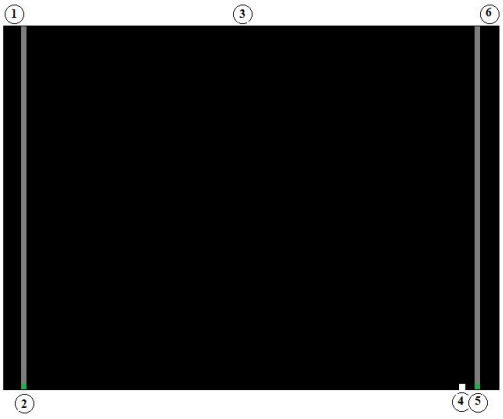

Figure 1: Designed Screen for Dodge Snowball

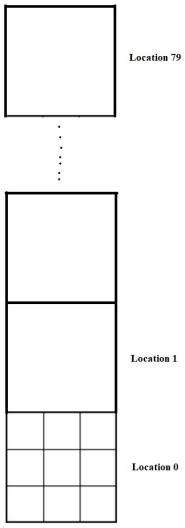

Figure 2: Player One or Player Two Design

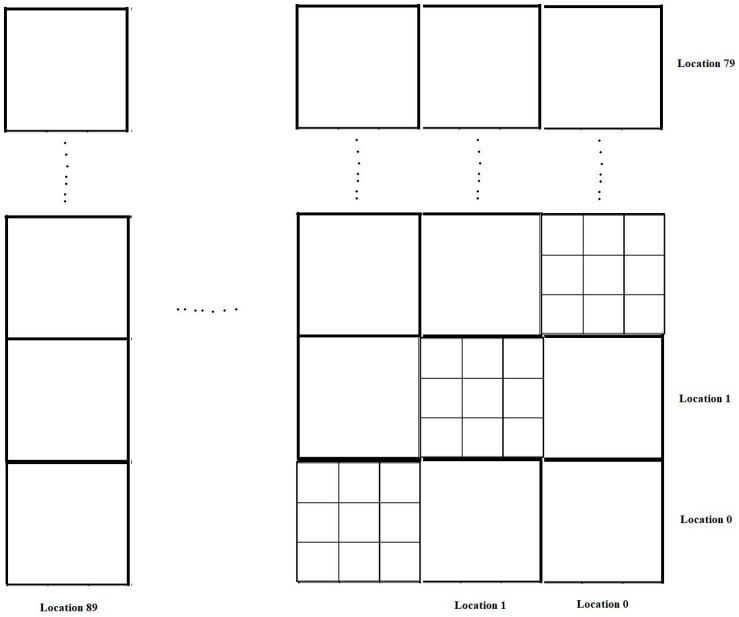

Figure 3: Snowball Field Design

The Dodge Snowball game design meets the project requirements by implementing the following:

- A Camera:
  The camera will be used to paint the player one cursor, player two cursor, and the snowballs. The painted value will be determined by the average red, average blue, and average green values.

- An Input Device that does not use a USB:
  A 2-axis joystick to control the player one cursor. The joystick will be used with the analog to digital converter on the DE1_SoC board.

- VGA:
  The VGA output of the DE1_SoC will be used to display a screen resolution of 320 x 240 on the computer monitor. The Dodge Snowball game will be contained within the 320 x 240 display.

- GPIO functions:
  A general purpose input output (GPIO) pin will turn an LED in the event player two spawns a snowball.

- Microphone:
  The microphone will be used as an input for player two. If the sound entering the microphone reaches above a threshold, then the player two cursor goes up. If the sound entering the microphone is below or equal to the threshold then the player two cursor resets to the bottom of the screen.

Procedure/Results/Analysis:

Steps to Complete the Lab:
To complete this project, we decided to start working on smaller pieces of the project and then bring them all together at the end. We prioritized our work into seven sections.

1. Player 1 controlling an object and can duck on the left side of the VGA screen, using a joystick
2. A microphone that allows player 2 to move the cursor
3. Player 2 controlling a cursor that spawns penguins on the right side of the VGA screen, using a push button to spawn penguins
4. A module that shifts all spawned penguins from right to left
5. A module that uses the camera and paints the background of the VGA screen, using the average values read from the camera
6. A module that lights an LED each time the spawn button was pressed

Description for each Module:

gameBoardTestWithCamera and gameBoardTest:
This is the top level module for the Dodge Snowball game. This module instantiates and prepares the logic needed for the lower level modules. Note that GPIO[18] is assigned to KEY[1] which is the spawn snowball button pressed controlled by player two.

Meanwhile, the gameBoardTest module performs the connections for player one, player two, camera RGB color assignments, and connects the VGA_framebuffer module with the p1_snowball_p2_board module. Figure 4 displays the gameBoardTestWithCamera and gameBoardTest module.

```
module gameBoardTestWithCamera(
    // Joystick
    input logic ADC_DOUT,
    output logic ADC_CONVST, ADC_DIN, ADC_SCLK,

    ///////////// Clocks /////////
    input logic           CLOCK2_50,
    input logic           CLOCK3_50,
    input logic           CLOCK4_50,
    input logic           CLOCK_50,

    ///////////// SDRAM /////////
    output logic [12:0]  DRAM_ADDR,
    output logic [1:0]   DRAM_BA,
    output logic         DRAM_CAS_N,
    output logic         DRAM_CKE,
    output logic         DRAM_CLK,
    output logic         DRAM_CS_N,
    inout        [15:0] DRAM_DQ,
    output logic         DRAM_LDQM,
    output logic         DRAM_RAS_N,
    output logic         DRAM_UDQM,
    output logic         DRAM_WE_N,

    ///////////// SEG7 /////////
    output logic [6:0]   HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,

    ///////////// KEY /////////
    input logic  [3:0]   KEY,

    ///////////// LED /////////
    output logic [9:0]   LEDR,

    ///////////// SW /////////
    input logic  [9:0]   SW,

    ///////////// GPIO_1, GPIO_1 connect to D8M-GPIO /////////
    output logic         CAMERA_I2C_SCL,
    inout                CAMERA_I2C_SDA,
    output logic         CAMERA_PWDN_n,
    output logic         MIPI_CS_n,
    inout                MIPI_I2C_SCL,
    inout                MIPI_I2C_SDA,
    output logic         MIPI_MCLK,
    input  logic         MIPI_PIXEL_CLK,
    input  logic [9:0]   MIPI_PIXEL_D,
    input  logic         MIPI_PIXEL_HS,
    input  logic         MIPI_PIXEL_VS,
    output logic         MIPI_REFCLK,
    output logic         MIPI_RESET_n,
    ///////////// GPIO_0, GPIO_0 connect to GPIO Default /////////
    inout   [35:0] GPIO,


    ///////////// vga frame buffer ////////
    output [7:0] VGA_R,
    output [7:0] VGA_G,
    output [7:0] VGA_B,
    output VGA_BLANK_N,
    output VGA_CLK,
    output VGA_HS,
    output VGA_SYNC_N,
    output VGA_VS
    );

    logic [7:0] nextR;
    logic [7:0] nextG;
    logic [7:0] nextB;

    Cam_2_Color camColor (.*);

    gameBoardTest0 (.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .LEDR, .SW, .CLOCK_50,
    .VGA_R, .VGA_G, .VGA_B, .VGA_BLANK_N, .VGA_CLK, .VGA_HS, .VGA_SYNC_N, .VGA_VS,
    .ADC_DOUT, .ADC_CONVST, .ADC_DIN, .ADC_SCLK, .nextR, .nextG, .nextB, .CLOCK2_50);

    assign GPIO[18] = ~KEY[1];
endmodule
```

```
module gameBoardTest0 (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, CLOCK_50,
    VGA_R, VGA_G, VGA_B, VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, VGA_VS,
    ADC_DOUT, ADC_CONVST, ADC_DIN, ADC_SCLK, nextR, nextG, nextB, CLOCK2_50);

    input logic [7:0] nextR;
    input logic [7:0] nextG;
    input logic [7:0] nextB;
    input logic CLOCK2_50;

    input logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    input logic [9:0] LEDR;

    input logic [3:0] KEY;
    input logic [9:0] SW;
    input logic ADC_DOUT;

    input logic CLOCK_50;
    output logic [7:0] VGA_R;
    output logic [7:0] VGA_G;
    output logic [7:0] VGA_B;
    output logic VGA_BLANK_N;
    output logic VGA_CLK;
    output logic VGA_HS;
    output logic VGA_SYNC_N;
    output logic VGA_VS;
    output logic ADC_CONVST, ADC_DIN, ADC_SCLK;

    logic [10:0] x, y, x_in, y_in;
    logic write_en;
    logic done_signal;
    logic pixel_write;
    logic color_current;
    logic move_signal, black_square;

    logic reset;
    assign reset = ~KEY[2];

    // ********************************** Joy Stick
    logic [11:0] joystick_data;
    joystick_ADC joyADC (.CLOCK_50, .reset, .ADC_DOUT, .ADC_CONVST, .ADC_DIN, .ADC_SCLK, .joystick_data);

    logic [23:0] mic_signal;

    always_ff @(posedge CLOCK_50) begin
        if(reset)
            mic_signal <= 24'd0;
        else
            if(~KEY[0]) mic_signal <= 24'd5001;
            else mic_signal <= 24'd0;
    end
     logic [7:0] Avg_colors_temp [0:2];

     assign Avg_colors_temp[2] = nextB;
     assign Avg_colors_temp[1] = nextG;
     assign Avg_colors_temp[0] = nextR;

    // ********************************** Frame Buffer
    logic [7:0] VGA_colors_out [0:2];
    logic pixel_write_out;
    logic pixel_color;
    VGA_framebuffer fb(.clk50(CLOCK_50), .reset(1'b0), .x, .y,
                .pixel_color, .pixel_write(pixel_write_out),
                .nextB(VGA_colors_out[0]), .nextG(VGA_colors_out[1]), .nextB(VGA_colors_out[2]),
                .VGA_R, .VGA_G, .VGA_B, .VGA_CLK, .VGA_HS, .VGA_VS,
                .VGA_BLANK_n(VGA_BLANK_N), .VGA_SYNC_n(VGA_SYNC_N));

    logic spawn;

    p1_snowball_p2_board gameBoard(.clk(CLOCK_50), .reset, .mic_signal, .spawn_button(~KEY[1]), .joystick_data,
    .Avg_colors_temp, .spawn, .Xout(x), .Yout(y), .VGA_colors_out, .pixel_write_out, .pixel_color);
endmodule
```

Figure 4: gameBoardTestWithCamera and gameBoardTest Module

p1_snowball_p2_board:

The p1_snowball_p2_board module contains modules that allows the VGA screen to be properly painted. This module contains a control module called draw_FSM that decides when P1Input, snowballField, and when P2Input will be drawn. After sequentially deciding which of these three modules to choose from, draw_FSM will enable and send data to draw_TEST. With this given data, draw_TEST will write to the VGA. Each x and y output from the draw_TEST will be used as inputs for the VGA_framebuffer to paint onto the VGA screen. The block diagram of the p1_snowball_p2_board is displayed in Figure 5. Figure 6 displays the p1_snowball_p2_board module and it's testbench is displayed in Figure 7.

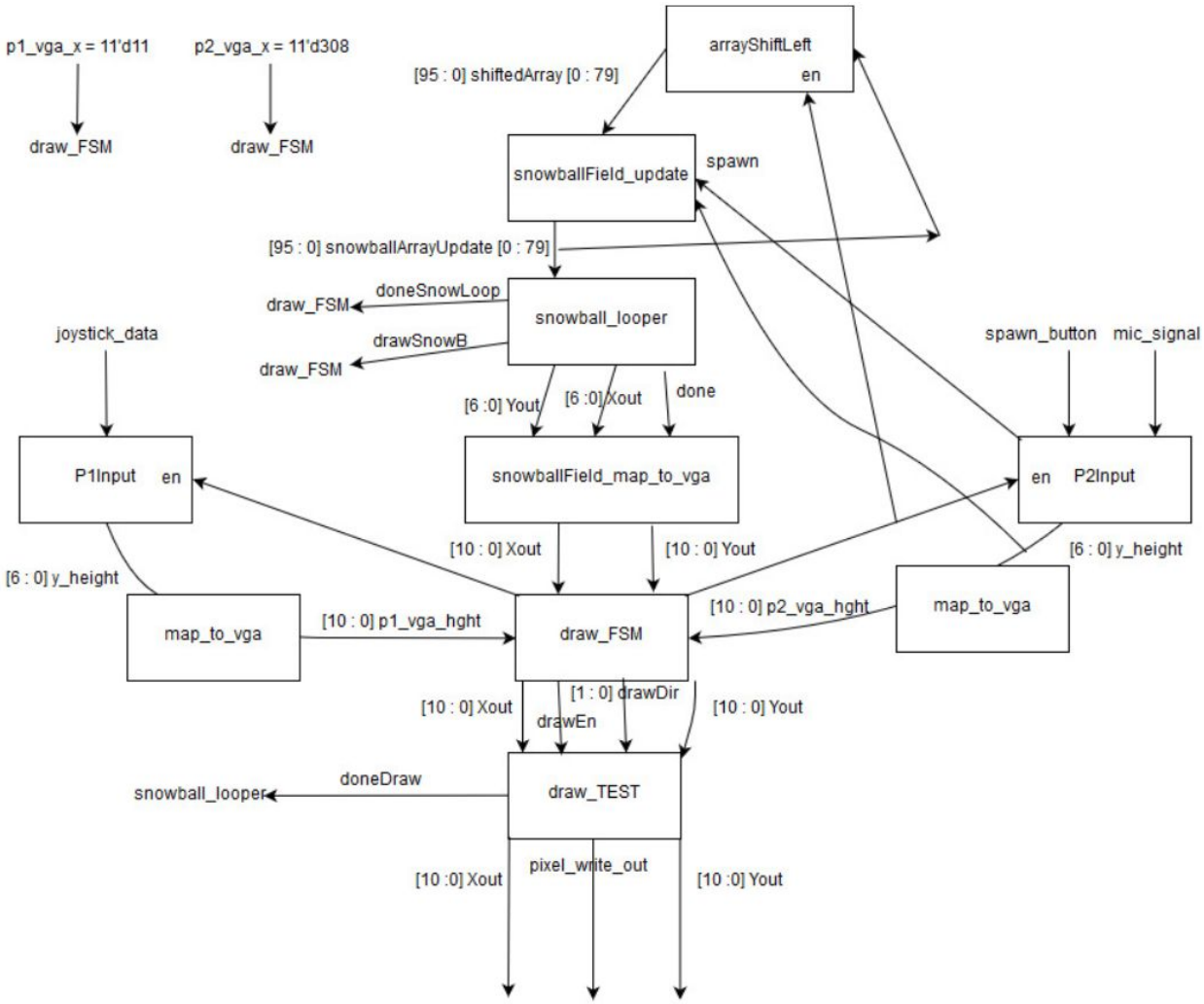

Figure 5: p1_snowball_p2_board Block Diagram

```
module p1_snowball_p2_board (clk, reset, mic_signal, spawn_button, joystick_data,
   Avg_colors_temp, spawn, Xout, Yout, VGA_colors_out, pixel_write_out, pixel_color);
   input logic clk, reset, spawn_button;
   input logic [11:0] joystick_data;
// logic [95:0] snowballArrayIn [0:79];
   input logic [7:0] Avg_colors_temp [0:2];
   input logic [23:0] mic_signal;
   output logic spawn;
   output logic [10:0] Xout, Yout;
   output logic [7:0] VGA_colors_out [0:2];
   output logic pixel_write_out, pixel_color;

   // ------------------------logic-----------------------------------------
   logic [6:0] p1_hght, p2_hght, snowball_height;

   // done signals
   logic doneSnowLoop;
   logic doneDraw;

   // draw signals
   logic drawSnowB;
   logic drawEn;
   logic [1:0] drawDir;

   // location of draw inputs with respect to the vga screen "y-coordinates"
   logic [10:0] p1_vga_hght, p2_vga_hght, snowB_vga_hght;

   // "player locations respective to vga screen x-coordinates"
   logic [10:0] p1_vga_x, p2_vga_x, snowB_vga_x;
   assign p1_vga_x = 11'd11;
   assign p2_vga_x = 11'd308;

   // snowball location on the snowball field map
   logic [6:0] snowball_map_x, snowball_map_hght;

   // pixel constants
   parameter vgaBottommostPixel = 239;
   parameter pixel = 1;

   // draw location with respect to the vga screen
   logic [10:0] draw_vga_x, draw_vga_hght;

   logic [95:0] snowballArrayUpdate [0:79];
   logic [95:0] shiftedArray [0:79];

   logic playerEn;
   logic spawnSnowB;

   // ***************************** Player 1 ****************************
   P1Input p1 (.clk(clk), .reset(reset), .enable(playerEn), .joystick_data(joystick_data), .y_height(p1_hght))

   // ***************************** Snowball Field ****************************
   // snowball array shift
   arrayShiftLeft #(96, 80) leftShifter (.clk(clk), .reset, .enable(playerEn),
      .oldArray(snowballArrayUpdate), .shiftedArray(shiftedArray));

   // shifted array goes into the snowball updater (spawning from player 2)
   snowballField_update snowB_field_update(.clk, .yVal(p2_hght), .spawn, .snowballArrayIn(shiftedArray),
      .snowballArrayOut(snowballArrayUpdate));

   // Updated snowball array looper for the mapper, to map each x and y
   snowball_looper sl (.clk(clk), .en(doneDraw), .reset(reset), .snowballArrayIn(snowballArrayUpdate), // good
      .done(doneSnowLoop), .Xout(snowball_map_x), .Yout(snowball_map_hght), .draw(drawSnowB));

   // Maps the x, y value of the snowball field to the vga screen
   snowballField_map_to_vga map(.Xin(snowball_map_x), .Yin(snowball_map_hght), // good
      .Xout(snowB_vga_x), .Yout(snowB_vga_hght));

   // ***************************** Player 2 ****************************
   P2Input p2 (.clk(clk), .reset(reset), .mic_signal(mic_signal), .spawn_button(spawn_button),
      .enable(playerEn), .y_height(p2_hght), .spawn(spawn));

   // ***************************** Control for drawing the Snowball Game ****************************
   draw_FSM boardFSM (.clk(clk), .reset(reset), .doneDraw(doneDraw), .doneSnowLoop(doneSnowLoop),
      .drawSnowB(drawSnowB), .Xin0(p1_vga_x), .Yin0(p1_vga_hght), .Xin1(snowB_vga_x), .Yin1(snowB_vga_hght),
      .Xin2(p2_vga_x), .Yin2(p2_vga_hght), .Xout(draw_vga_x), .Yout(draw_vga_hght), .en(drawEn),
      .dir(drawDir), .playerEn);

   // ***************************** Output of the Control Logic ****************************
   draw_TEST drawTest(.clk(clk), .reset(reset), .direction(drawDir), .enable(drawEn), .x_in(draw_vga_x),
      .y_in(draw_vga_hght), .Avg_colors(Avg_colors_temp), .x(Xout), .y(Yout), .VGA_colors(VGA_colors_out),
      .pixel_write(pixel_write_out), .done_signal(doneDraw), .pixel_color);


   always_comb begin
      // convert snowball field y-coordinate to vga screen y-coordinate
      p1_vga_hght = vgaBottommostPixel - (p1_hght * 3) - pixel;
      p2_vga_hght = vgaBottommostPixel - (p2_hght * 3) - pixel;
   end

endmodule
```

Figure 6: p1_snowball_p2_board Module

```
module p1_snowball_p2_board_testbench();
   // inputs
   logic clk, reset, mic_signal, spawn_button;
   logic [11:0] joystick_data;

   // outputs
   logic spawn;
   logic [10:0] Xout, Yout;
   logic [7:0] VGA_colors_out [0:2];
   logic pixel_write_out;

   p1_snowball_p2_board dut (.clk, .reset, .mic_signal, .spawn_button, .joystick_data,
      .spawn, .Xout, .Yout, .VGA_colors_out, .pixel_write_out);

   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   integer a;
   integer b;

   initial begin
      reset = 1; @(posedge clk);

      reset = 0;
      mic_signal = 1;
      spawn_button = 1;
      joystick_data = 12'd3840;
       input snowball field is all zeroes
      for (a = 0; a < 80; a++) begin
         for (b = 0; b < 96; b++) begin
            snowballArrayIn[a][b] = 1'b0;
         end
      end

      // input snowball fields has 3 1's ("snowball")
      snowballArrayIn[0][0] = 1'b1; @(posedge clk); // 1
      snowballArrayIn[40][47] = 1'b1; @(posedge clk); // 2
      snowballArrayIn[79][95] = 1'b1; @(posedge clk); // 3

      // in delay state
      repeat (249997) begin
         @(posedge clk);
      end
      @(posedge clk);

      // in p1 state
      repeat (27) begin
         @(posedge clk);
      end
      @(posedge clk);

      //snowball mapper (80 * 95 = 7600) + (1 * 18) + 1
      repeat (7619) begin
         @(posedge clk);
      end

      // in p2 state
      repeat (18) begin
         @(posedge clk);
      end
      @(posedge clk);@(posedge clk);

      $stop;
   end
endmodule
```

Figure 7: p1_snowball_p2_board Testbench Module

<u>Player 1 Controls:</u>

Player 1 controls their cursor on the left side of the screen by moving a joystick up and down in the y-direction, as seen in Figure 8 below. Since we are only using the y-direction of the joystick, we only need connect S-Y to use one channel of the ADC and we choose to use Channel 1. In Figure 8 above the joystick you can see the connection pins on the DE1-SoC for the ADC which Figure 9 below shows which pins on the ADC Port we are using.

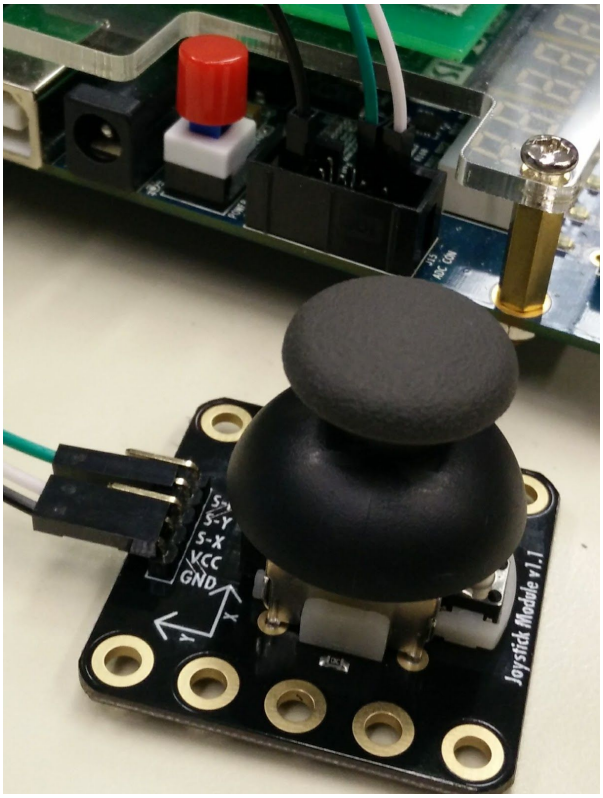

Figure 8: Joystick

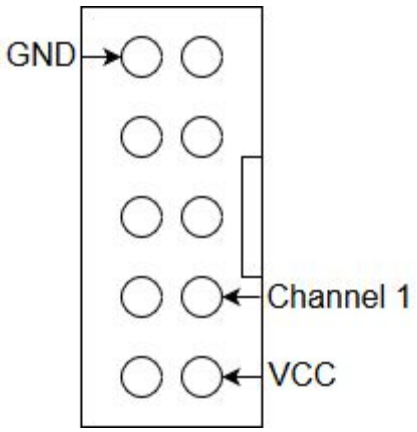

Figure 9: ADC Pins

The controls for player 1 are broken up into two modules: joystick_ADC.sv (for getting data about the joystick position) and P1Input.sv (for controlling the position of the player 1 cursor). Figure 10 below shows a block diagram for these modules.

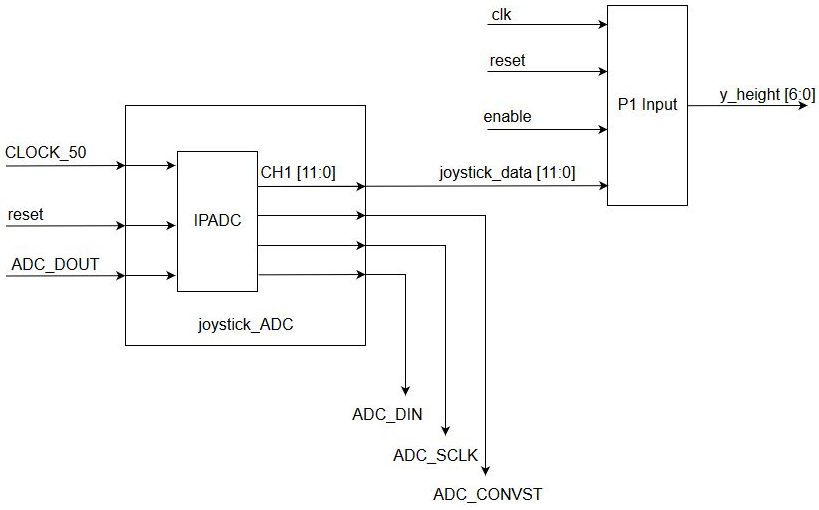

Figure 10: Player 1 Controls Block Diagram

The joystick_ADC module handles getting the joystick through the IPADC module which is a ADC module that was made using the IP Catalog which will be explained why in the results section below. The ADC_DOUT, ADC_DIN, ADC_SCLK, and ADC_CONVST signals are used as top level input and output signals similar to using the onboard HEX displays or switches. The 12 bit data on Channel 1 is set out of the joystick_ADC to the P1Input module. Since there is 12 bit for data, the ADC will output values from 0 to 4095. For our joystick a value 0 would mean pushing to -y on the joystick. A value of 4095 would mean pushing to +y on the joystick.

The P1Input module outputs the current y_height of the player 1 cursor from 0 to 79 since we divided the vertical space of the play field by 80 rows. If the enable signal is true then if the joystick_data is above 3839 then the y_height will increase by one. If the joystick_data is below 256 then the y_height will decrease by one. We have an enable signal to synchronize the movement of both player cursors and the snowball to make sure they don't move at the speed of our CLOCK_50.

Figure 11 below shows the SystemVerilog code for the joystick_ADC module and Figure 12 below shows the P1Input module.

```
module joystick_ADC (
    input logic CLOCK_50,
    input logic reset,
    input logic ADC_DOUT,
    output logic ADC_CONVST, ADC_DIN, ADC_SCLK,
    output logic [11:0] joystick_data
    );

    logic [7:0][11:0] data;

    IPADC (.CLOCK(CLOCK_50),
           .ADC_SCLK,
           .ADC_CS_N(ADC_CONVST),
           .ADC_DIN,
           .ADC_DOUT,
           .RESET(reset),
           .CH0(data[0]),
           .CH1(data[1]),
           .CH2(data[2]),
           .CH3(data[3]),
           .CH4(data[4]),
           .CH5(data[5]),
           .CH6(data[6]),
           .CH7(data[7]));

    assign joystick_data = data[1];

endmodule
```

Figure 11: joystick_ADC Module

```
module P1Input(
   input logic clk, reset, enable,
   input logic [11:0] joystick_data,
   output logic [6:0] y_height);

   logic [6:0] y;

   parameter THRESHOLD_HIGH = 3839;
   parameter THRESHOLD_LOW = 256;

   always_ff @(posedge clk) begin
      if(reset)
         y <= 0;
      else if(enable) begin

         if(y_height < 7'd79 && joystick_data > THRESHOLD_HIGH) //increase y
            y <= y + 1;
         else if(y_height > 7'd0 && joystick_data < THRESHOLD_LOW) //decrease y
            y <= y - 1;

      end
   end

   assign y_height = y;

endmodule

module P1Input_testbench();
   logic clk, reset, enable;
   logic [11:0] joystick_data;
   logic [6:0] y_height;
   P1Input dut (.*);
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end
   initial begin
   integer i;
                                                        @(posedge clk);
   reset <=1; enable <=0; joystick_data <=12'h000; @(posedge clk);
   reset <=0;                                      @(posedge clk);
   $stop;
   end
endmodule
```

Figure 12: P1Input Module

Player 2 Controls:

Player 2 was originally going to controls their cursor on the right side of the screen by using their voice. We ran into issues when using a microphone and a camera which will be explained more in the results section below. This section will first explain our original plan and code. Then we will explain our workaround.

Player 2 was going to be able to move their cursor upward if the value of the left channel of the microphone was greater than a threshold value. If they ever went below the threshold value their cursor would immediately go back down to the bottom of the screen. Player 2 is also able to spawn snowballs which if they press KEY 1 then a snowball is spawned at the current y_height value that the player 2 cursor is at. Figure 13 below shows a block diagram of how the mic_input and P2Input modules are hooked up.

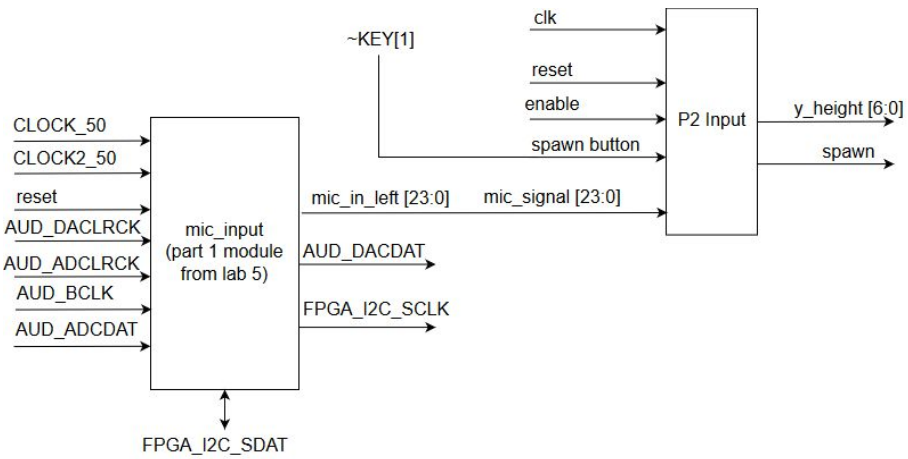

Figure 13: Player 2 Control Block Diagram

The mic_input module is similar to the part 1 module as in our Lab 5 however instead of sending the mic_in_left and right to the speaker, we redirected mic_in_left out of the module so that we can use it in our P2Input module.

The P2Input module outputs the current y_height of the player 2 cursor from 0 to 79 since we divided the vertical space of the play field by 80 rows. If the enable signal is true then if the mic_signal is above 5000 then the y_height will increase by one. If the mic_signal is equal or below 5000 then the y_height resets to 0. We chose 5000 as the threshold as a placeholder value that we would change once we got the mic working with our game. We have an enable signal to synchronize the movement of both player cursors and the snowball to make sure they don't move at the speed of our CLOCK_50.

The results section below will go into more detail of why our microphone was not working along with the camera, we created a workaround to at least be able to have player 2 able to move. In the level above our P2Input module, we had a always_ff block that sets a mic_signal to 5001 if KEY 0 is pressed, otherwise set the mic_signal to 0. Thus the key press would simulate player 2 speaking louder than the threshold and letting go of the button would be speaker quieter than the threshold.

Figure 14 below shows the SystemVerilog code for the mic_input module and Figure 15 below shows the P2Input module.

```
module mic_input (CLOCK_50, CLOCK2_50, reset, mic_in_left, FPGA_I2C_SCLK, FPGA_I2C_SDAT, AUD_XCK,
                  AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK, AUD_ADCDAT, AUD_DACDAT);

   input CLOCK_50, CLOCK2_50;
   input reset;
   // I2C Audio/Video config interface
   output FPGA_I2C_SCLK;
   inout FPGA_I2C_SDAT;
   // Audio CODEC
   output AUD_XCK;
   input AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK;
   input AUD_ADCDAT;
   output AUD_DACDAT;

   output [23:0] mic_in_left;

   // Local wires.
   wire read_ready, write_ready, read, write;
   wire [23:0] readdata_left, readdata_right;
   wire [23:0] writedata_left, writedata_right;

   ////////////////////////////////
   //     Our code goes here     //
   ////////////////////////////////


   assign mic_in_left = readdata_left;


   assign read = read_ready;

/////////////////////////////////////////////////////////////////////////////////
// Audio CODEC interface.
//
// The interface consists of the following wires:
// read_ready, write_ready - CODEC ready for read/write operation
// readdata_left, readdata_right - left and right channel data from the CODEC
// read - send data from the CODEC (both channels)
// writedata_left, writedata_right - left and right channel data to the CODEC
// write - send data to the CODEC (both channels)
// AUD_* - should connect to top-level entity I/O of the same name.
//         These signals go directly to the Audio CODEC
// I2C_* - should connect to top-level entity I/O of the same name.
//         These signals go directly to the Audio/Video Config module
/////////////////////////////////////////////////////////////////////////////////
   clock_generator my_clock_gen(
      // inputs
      CLOCK2_50,
      reset,

      // outputs
      AUD_XCK
   );

   audio_and_video_config cfg(
      // Inputs
      CLOCK_50,
      reset,

      // Bidirectionals
      FPGA_I2C_SDAT,
      FPGA_I2C_SCLK
   );

   audio_codec codec(
      // Inputs
      CLOCK_50,
      reset,

      read, write,
      writedata_left, writedata_right,

      AUD_ADCDAT,

      // Bidirectionals
      AUD_BCLK,
      AUD_ADCLRCK,
      AUD_DACLRCK,

      // Outputs
      read_ready, write_ready,
      readdata_left, readdata_right,
      AUD_DACDAT
   );

endmodule
```

Figure 14: mic_input Module

```
module P2Input #(parameter MIC_THRESHOLD = 5000) (
   input logic clk, reset,
   input logic [23:0] mic_signal,
   input logic spawn_button,
   input logic enable,
   output logic [6:0] y_height,
   output logic spawn);

   logic [6:0] y;

   always_ff @(posedge clk) begin
      if(reset)
         y <= 7'b0;
      else if (enable) begin

         if(y_height < 7'd79 & mic_signal > MIC_THRESHOLD)
            //increase y
            y <= y + 1;

         else if(y_height <= 7'd79 & mic_signal <= MIC_THRESHOLD)
            //set y to 0
            y <= 0;
      end
   end

   assign spawn = spawn_button;
   assign y_height = y;
endmodule

module P2Input_testbench();
   logic clk, reset, spawn_button, enable;
   logic [23:0] mic_signal;
   logic [6:0] y_height;
   logic spawn;
   P2Input dut (.*);
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end
   initial begin
   integer i;
                                                  @(posedge clk);
   reset<=1; mic_signal<=24'd0; spawn_button<=0;@(posedge clk);
   reset<=0; enable<= 1;                          @(posedge clk);
   mic_signal<=24'd5001;                          @(posedge clk);
   for(i=0; i<82; i++)
                                                  @(posedge clk);
   mic_signal<=24'd0;                             @(posedge clk);
                                                  @(posedge clk);
   mic_signal<=24'd5001;                          @(posedge clk);
                                                  @(posedge clk);
                                                  @(posedge clk);
   spawn_button<=1;                               @(posedge clk);
   spawn_button<=0;                               @(posedge clk);
                                                  @(posedge clk);
   spawn_button<=1;                               @(posedge clk);
                                                  @(posedge clk);
   spawn_button<=0;                               @(posedge clk);
                                                  @(posedge clk);
   $stop;
   end
endmodule
```

Figure 15: P2Input Module

Camera:

For Dodge Snowball we used a camera to get the average color of the surroundings and set the player cursor and the snowballs to that average color. The driver for the provided camera directly sends what it sees to the screen. To get the average color we had change the camera driver to poll the Red, Green, and Blue values at the last stage inside the camera driver before they would get set to be drawn on the screen. This new driver is called Cam_2_Color as seen in Figure 16 through # below. From our knowledge, the camera driver is able to draw a new pixel every cycle. So with each clock cycle of the clock set to the camera driver we will get the next pixel color. We also know that the camera outputs a picture with a resolution of 640 by 480 pixels which would mean we would have to average 307,200 red, green, and blue samples. To not output to the VGA screen, I change all outputs for the VGA screen in the camera driver to be internal logic.

```
1
2   //=======================================================
3   //  Camera to Color Module
4   //=======================================================
5
6   // Controls:
7   // KEY[2]: reset the system.
8   // KEY[3]: Run the autofocus system.
9   // SW[9]: Choose between full and central auto-focus, plus yellow rectangle.
10
11  module Cam_2_Color(
12
13     ////////////// CLOCK //////////
14     input                       CLOCK2_50,
15     input                       CLOCK3_50,
16     input                       CLOCK4_50,
17     input                       CLOCK_50,
18
19     ////////////// SDRAM //////////
20     output          [12:0]      DRAM_ADDR,
21     output          [1:0]       DRAM_BA,
22     output                      DRAM_CAS_N,
23     output                      DRAM_CKE,
24     output                      DRAM_CLK,
25     output                      DRAM_CS_N,
26     inout           [15:0]      DRAM_DQ,
27     output                      DRAM_LDQM,
28     output                      DRAM_RAS_N,
29     output                      DRAM_UDQM,
30     output                      DRAM_WE_N,
31
32     ////////////// SEG7 //////////
33     output           [6:0]      HEX0,
34     output           [6:0]      HEX1,
35     output           [6:0]      HEX2,
36     output           [6:0]      HEX3,
37     output           [6:0]      HEX4,
38     output           [6:0]      HEX5,
39
40     ////////////// KEY //////////
41     input            [3:0]      KEY,
42
43     ////////////// SW //////////
44     input            [9:0]      SW,
45
46     ////////////// GPIO_1, GPIO_1 connect to D8M-GPIO //////////
47     output                      CAMERA_I2C_SCL,
48     inout                       CAMERA_I2C_SDA,
49     output                      CAMERA_PWDN_n,
50     output                      MIPI_CS_n,
51     inout                       MIPI_I2C_SCL,
52     inout                       MIPI_I2C_SDA,
53     output                      MIPI_MCLK,
54     input                       MIPI_PIXEL_CLK,
55     input            [9:0]      MIPI_PIXEL_D,
56     input                       MIPI_PIXEL_HS,
57     input                       MIPI_PIXEL_VS,
58     output                      MIPI_REFCLK,
59     output                      MIPI_RESET_n,
60     ////////////// GPIO_0, GPIO_0 connect to GPIO Default //////////
61     inout           [35:0]      GPIO,
62
63     output [7:0] nextR,
64     output [7:0] nextG,
65     output [7:0] nextB
66  );
67
68  /*    VGA logic/wire */
69     wire                        VGA_BLANK_N;
70     wire            [7:0]       VGA_B;
71     wire                        VGA_CLK;
72     wire            [7:0]       VGA_G;
73     wire                        VGA_HS;
```

Figure 16: Cam_2_Color PAGE 1

```
74      wire          [7:0]     VGA_R;
75      wire                    VGA_SYNC_N;
76      wire                    VGA_VS;
77
78    //=============================================================================
79    // Added code to insert Filter.sv into the output path
80    //=============================================================================
81      // The signals from the system to the filter.
82      wire                    pre_VGA_BLANK_N;
83      wire          [7:0]     pre_VGA_B;
84      wire          [7:0]     pre_VGA_G;
85      wire                    pre_VGA_HS;
86      wire          [7:0]     pre_VGA_R;
87      wire                    pre_VGA_SYNC_N;
88      wire                    pre_VGA_VS;
89      // The signals from the filter to the VGA
90      wire                    post_VGA_BLANK_N;
91      wire          [7:0]     post_VGA_B;
92      wire          [7:0]     post_VGA_G;
93      wire                    post_VGA_HS;
94      wire          [7:0]     post_VGA_R;
95      wire                    post_VGA_SYNC_N;
96      wire                    post_VGA_VS;
97
98      Filter #(.WIDTH(640), .HEIGHT(480))
99         filter (.VGA_CLK(VGA_CLK),
100                .iVGA_B(pre_VGA_B), .iVGA_G(pre_VGA_G), .iVGA_R(pre_VGA_R),
101                .iVGA_HS(pre_VGA_HS), .iVGA_VS(pre_VGA_VS),
102                .iVGA_SYNC_N(pre_VGA_SYNC_N), .iVGA_BLANK_N(pre_VGA_BLANK_N),
103                .oVGA_B(post_VGA_B), .oVGA_G(post_VGA_G), .oVGA_R(post_VGA_R),
104                .oVGA_HS(post_VGA_HS), .oVGA_VS(post_VGA_VS),
105                .oVGA_SYNC_N(post_VGA_SYNC_N), .oVGA_BLANK_N(post_VGA_BLANK_N),
106                .KEY(KEY[1:0]), .SW(SW[8:0]));
107
108     assign VGA_BLANK_N = post_VGA_BLANK_N;
109     assign VGA_B = post_VGA_B;
110     assign VGA_G = post_VGA_G;
111     assign VGA_HS = post_VGA_HS;
112     assign VGA_R = post_VGA_R;
113     assign VGA_SYNC_N = post_VGA_SYNC_N;
114     assign VGA_VS = post_VGA_VS;
115
116   //=============================================================================
117   // REG/WIRE declarations
118   //=============================================================================
119
120
121   wire  [15:0]SDRAM_RD_DATA;
122   wire        DLY_RST_0;
123   wire        DLY_RST_1;
124   wire        DLY_RST_2;
125
126   wire        SDRAM_CTRL_CLK;
127   wire        D8M_CK_HZ ;
128   wire        D8M_CK_HZ2 ;
129   wire        D8M_CK_HZ3 ;
130
131   wire [7:0] RED   ;
132   wire [7:0] GREEN  ;
133   wire [7:0] BLUE       ;
134   wire [12:0] VGA_H_CNT;
135   wire [12:0] VGA_V_CNT;
136
137   wire        READ_Request ;
138   wire  [7:0] B_AUTO;
139   wire  [7:0] G_AUTO;
140   wire  [7:0] R_AUTO;
141   wire        RESET_N  ;
142
143   wire        I2C_RELEASE ;
144   wire        AUTO_FOC ;
145   wire        CAMERA_I2C_SCL_MIPI ;
146   wire        CAMERA_I2C_SCL_AF;
```

Figure 16: Cam_2_Color PAGE 2

```
wire        CAMERA_MIPI_RELAESE ;
wire        MIPI_BRIDGE_RELEASE ;

wire        LUT_MIPI_PIXEL_HS;
wire        LUT_MIPI_PIXEL_VS;
wire [9:0]  LUT_MIPI_PIXEL_D  ;
wire        MIPI_PIXEL_CLK_;
wire [9:0]  PCK;
//=============================================================
// Structural coding
//=============================================================
//--INPU MIPI-PIXEL-CLOCK DELAY
CLOCK_DELAY  del1(  .iCLK (MIPI_PIXEL_CLK),  .oCLK (MIPI_PIXEL_CLK_ ) );


assign LUT_MIPI_PIXEL_HS=MIPI_PIXEL_HS;
assign LUT_MIPI_PIXEL_VS=MIPI_PIXEL_VS;
assign LUT_MIPI_PIXEL_D =MIPI_PIXEL_D ;

//------UART OFF --
assign UART_RTS =0;
assign UART_TXD =0;

//------ MIPI BRIGE & CAMERA RESET  --
assign CAMERA_PWDN_n  = 1;
assign MIPI_CS_n      = 0;
assign MIPI_RESET_n   = RESET_N ;

//------ CAMERA MODULE I2C SWITCH  --
assign I2C_RELEASE    = CAMERA_MIPI_RELAESE & MIPI_BRIDGE_RELEASE;
assign CAMERA_I2C_SCL =( I2C_RELEASE  )?  CAMERA_I2C_SCL_AF  : CAMERA_I2C_SCL_MIPI ;

//----- RESET RELAY  --
RESET_DELAY       u2 (
                     .iRST  ( KEY[2] ),
                     .iCLK  ( CLOCK2_50 ),
                     .oRST_0( DLY_RST_0 ),
                     .oRST_1( DLY_RST_1 ),
                     .oRST_2( DLY_RST_2 ),
                     .oREADY( RESET_N)

                  );

//------ MIPI BRIGE & CAMERA SETTING  --
MIPI_BRIDGE_CAMERA_Config    cfin(
                      .RESET_N           ( RESET_N ),
                      .CLK_50            ( CLOCK2_50 ),
                      .MIPI_I2C_SCL      ( MIPI_I2C_SCL ),
                      .MIPI_I2C_SDA      ( MIPI_I2C_SDA ),
                      .MIPI_I2C_RELEASE  ( MIPI_BRIDGE_RELEASE ),
                      .CAMERA_I2C_SCL    ( CAMERA_I2C_SCL_MIPI ),
                      .CAMERA_I2C_SDA    ( CAMERA_I2C_SDA ),
                      .CAMERA_I2C_RELAESE( CAMERA_MIPI_RELAESE )
             );

//------MIPI / VGA REF CLOCK  --
pll_test pll_ref(
                      .inclk0 ( CLOCK3_50 ),
                      .areset ( ~KEY[2]   ),
                      .c0( MIPI_REFCLK    ) //20Mhz

    );

//------MIPI / VGA REF CLOCK  -
VIDEO_PLL pll_ref1(
                      .inclk0 ( CLOCK2_50 ),
                      .areset ( ~KEY[2] ),
                      .c0( VGA_CLK )        //25 Mhz
    );
//------SDRAM CLOCK GENNERATER  --
sdram_pll u6(
                     .areset( 0 ) ,
                     .inclk0( CLOCK_50 ),
```

Figure 16: Cam_2_Color PAGE 3

```
220                     .c1    ( DRAM_CLK ),       //100MHZ   -90 degree
221                     .c0    ( SDRAM_CTRL_CLK )  //100MHZ     0 degree
222
223                 );
224
225 //------SDRAM CONTROLLER --
226 Sdram_Control     u7 (  // HOST Side
227                     .RESET_N     ( KEY[2] ),
228                     .CLK         ( SDRAM_CTRL_CLK ) ,
229                     // FIFO Write Side 1
230                     .WR1_DATA    ( LUT_MIPI_PIXEL_D[9:0] ),
231                     .WR1         ( LUT_MIPI_PIXEL_HS & LUT_MIPI_PIXEL_VS ) ,
232
233                     .WR1_ADDR    ( 0 ),
234                     .WR1_MAX_ADDR( 640*480 ),
235                     .WR1_LENGTH  ( 256 ) ,
236                     .WR1_LOAD    ( !DLY_RST_0 ),
237                     .WR1_CLK     ( MIPI_PIXEL_CLK_),
238
239                     // FIFO Read Side 1
240                     .RD1_DATA    ( SDRAM_RD_DATA[9:0] ),
241                     .RD1         ( READ_Request ),
242                     .RD1_ADDR    (0 ),
243                     .RD1_MAX_ADDR( 640*480 ),
244                     .RD1_LENGTH  ( 256  ),
245                     .RD1_LOAD    ( !DLY_RST_1 ),
246                     .RD1_CLK     ( VGA_CLK ),
247
248                     // SDRAM Side
249                     .SA          ( DRAM_ADDR ),
250                     .BA          ( DRAM_BA ),
251                     .CS_N        ( DRAM_CS_N ),
252                     .CKE         ( DRAM_CKE ),
253                     .RAS_N       ( DRAM_RAS_N ),
254                     .CAS_N       ( DRAM_CAS_N ),
255                     .WE_N        ( DRAM_WE_N ),
256                     .DQ          ( DRAM_DQ ),
257                     .DQM         ( DRAM_DQM  )
258                     );
259
260 //------ CMOS CCD_DATA TO RGB_DATA --
261
262 RAW2RGB_J           u4 (
263                     .RST         ( pre_VGA_VS ),
264                     .iDATA       ( SDRAM_RD_DATA[9:0] ),
265
266                     //-----------------------------------
267                     .VGA_CLK     ( VGA_CLK ),
268                     .READ_Request ( READ_Request ),
269                     .VGA_VS      ( pre_VGA_VS ),
270                     .VGA_HS      ( pre_VGA_HS ),
271
272                     .oRed        ( RED  ),
273                     .oGreen      ( GREEN),
274                     .oBlue       ( BLUE )
275
276
277                     );
278 //------AOTO FOCUS ENABLE  --
279 AUTO_FOCUS_ON  vd(
280                      .CLK_50      ( CLOCK2_50 ),
281                      .I2C_RELEASE ( I2C_RELEASE ),
282                      .AUTO_FOC    ( AUTO_FOC )
283               ) ;
284
285
286 //------AOTO FOCUS ADJ  --
287 FOCUS_ADJ adl(
288                      .CLK_50        ( CLOCK2_50 ) ,
289                      .RESET_N       ( I2C_RELEASE ),
290                      .RESET_SUB_N   ( I2C_RELEASE ),
291                      .AUTO_FOC      ( KEY[3] & AUTO_FOC ),
292                      .SW_Y          ( 0 ),
```

Figure 16: Cam_2_Color PAGE 4

```
293                         .SW_H_FREQ     ( 0 ),
294                         .SW_FUC_LINE   ( SW[9] ),
295                         .SW_FUC_ALL_CEN( SW[9] ),
296                         .VIDEO_HS      ( pre_VGA_HS ),
297                         .VIDEO_VS      ( pre_VGA_VS ),
298                         .VIDEO_CLK     ( VGA_CLK ),
299                         .VIDEO_DE      (READ_Request) ,
300                         .iR            ( R_AUTO ),
301                         .iG            ( G_AUTO ),
302                         .iB            ( B_AUTO ),
303                         .oR            ( pre_VGA_R ) ,
304                         .oG            ( pre_VGA_G ) ,
305                         .oB            ( pre_VGA_B ) ,
306
307                         .READY         ( READY ),
308                         .SCL           ( CAMERA_I2C_SCL_AF ),
309                         .SDA           ( CAMERA_I2C_SDA )
310   );
311
312   //------VGA Controller  --
313
314   VGA_Controller    u1 (  // Host Side
315                         .oRequest( READ_Request ),
316                         .iRed    ( RED    ),
317                         .iGreen  ( GREEN  ),
318                         .iBlue   ( BLUE   ),
319
320                         //   VGA Side
321                         .oVGA_R  ( R_AUTO[7:0] ),
322                         .oVGA_G  ( G_AUTO[7:0] ),
323                         .oVGA_B  ( B_AUTO[7:0] ),
324                         .oVGA_H_SYNC( pre_VGA_HS ),
325                         .oVGA_V_SYNC( pre_VGA_VS ),
326                         .oVGA_SYNC  ( pre_VGA_SYNC_N ),
327                         .oVGA_BLANK ( pre_VGA_BLANK_N ),
328                         //   Control Signal
329                         .iCLK       ( VGA_CLK ),
330                         .iRST_N     ( DLY_RST_2 ),
331                         .H_Cont     ( VGA_H_CNT ),
332                         .V_Cont     ( VGA_V_CNT )
333         );
334
335      color_Buffer colorBuffer (.clk(VGA_CLK),
336                  .reset(~KEY[2]),
337                  .VGA_R(post_VGA_R),
338                  .VGA_G(post_VGA_G),
339                  .VGA_B(post_VGA_B),
340                  .nextR(nextR),
341                  .nextG(nextG),
342                  .nextB(nextB),
343                  .HEX0(HEX0),
344                  .HEX1(HEX1),
345                  .HEX2(HEX2),
346                  .HEX3(HEX3),
347                  .HEX4(HEX4),
348                  .HEX5(HEX5));
349
350
351   //--LED DISPLAY--
352   CLOCKMEM  ck1 ( .CLK(VGA_CLK )   ,.CLK_FREQ  (25000000  ) , . CK_1HZ (D8M_CK_HZ   )  )
             ;//25MHZ
353   CLOCKMEM  ck2 ( .CLK(MIPI_REFCLK   )   ,.CLK_FREQ  (20000000   ) , . CK_1HZ (D8M_CK_HZ2  )
   ) ;//20MHZ
354   CLOCKMEM  ck3 ( .CLK(MIPI_PIXEL_CLK_)   ,.CLK_FREQ  (25000000  ) , . CK_1HZ (D8M_CK_HZ3  )
   )  ;//25MHZ
355
356   endmodule
357
```

Figure 16: Cam_2_Color PAGE 5

To do this averaging, we created a module called colorAverager (as seen in Figure 17 below) which adds the 307,200 red, green, and blue samples respectively, and divides by 307,200 to get the average color from one full picture from the camera. These average red, green, and blue values are sent back to a module called color_Buffer (as seen in Figure 18 below) which every ¼ of a second takes the average color that the colorAverager produces and sets it as nextR, nextG, and nextB to be used by our edited VGA_framebuffer module. The color_Buffer module also displays the average red, green, blue values on the HEX displays in hexadecimal format. This was originally meant for debugging but left it in to let users see the actual average color value too.

```
module colorAverager (
   input logic clk, reset, en,
   input logic [7:0] VGA_R, VGA_G, VGA_B,
   output logic [7:0] Avg_R, Avg_G, Avg_B,
   output logic done
   );

   logic [26:0] tempR, tempG, tempB;
   logic [18:0] count;

   always_ff @ (posedge clk) begin
      if(reset) begin
         tempR <= 0;
         tempG <= 0;
         tempB <= 0;
         Avg_R <= 0;
         Avg_G <= 0;
         Avg_B <= 0;
         count <= 19'd0;
         done <= 1;
      end

      else if (count == 19'd0 & en) begin //enable when want to start averaging new frame
         tempR <= tempR + VGA_R;
         tempG <= tempG + VGA_G;
         tempB <= tempB + VGA_B;
         count <= count + 1;
         done <= 0;
      end

      else if (count == 19'd307200) begin //when after adding 640*480 pixels
         count <= 19'd0;
         Avg_R <= tempR/(640*480);
         Avg_G <= tempG/(640*480);
         Avg_B <= tempB/(640*480);
         tempR <= 0;
         tempG <= 0;
         tempB <= 0;
         done <= 1;
      end

      else if(done == 0) begin //add the current RGB colors into the temp
      tempR <= tempR + VGA_R;
      tempG <= tempG + VGA_G;
      tempB <= tempB + VGA_B;
      count <= count + 1;
      end

   end

endmodule

module colorAverager_testbench();
   logic clk, reset, en;
   logic [7:0] VGA_R, VGA_G, VGA_B;
   logic [7:0] Avg_R, Avg_G, Avg_B;
   logic done;

   colorAverager dut (.*);
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end
   initial begin
   integer i;
                                                                  @(posedge clk);
   reset <= 1; en <= 0; VGA_R <= 8'h00; VGA_G <= 8'h00; VGA_B <= 8'h00; @(posedge clk);
   reset <= 0; en <= 1; VGA_R <= 8'hFF; VGA_G <= 8'h10; VGA_B <= 8'h00; @(posedge clk);
               en <= 0;                 VGA_G <= 8'h00;                 @(posedge clk);
   repeat (153610) begin
      VGA_G <= 8'h10;
      @(posedge clk);
      VGA_G <= 8'h00;
      @(posedge clk);
   end
               en <= 1; VGA_R <= 8'h00;                                 @(posedge clk);
               en <= 0; VGA_R <= 8'hFF;                                 @(posedge clk);
   repeat (153610) begin
      VGA_R <= 8'h00;
      @(posedge clk);
      VGA_R <= 8'hFF;
      @(posedge clk);
   end

   $stop;
   end
endmodule
```

Figure 17: colorAverager Module

```
module color_Buffer(
   input logic clk, reset,
   input logic [7:0] VGA_R, VGA_G, VGA_B,
   output logic [7:0] nextR, nextG, nextB,
   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5);


   logic en;
   logic [7:0] Avg_R, Avg_G, Avg_B;
   logic done;

   colorAverager colorAvg (.clk,
                           .reset,
                           .en,
                           .VGA_R, .VGA_G, .VGA_B,
                           .Avg_R, .Avg_G, .Avg_B,
                           .done);

   logic [22:0] count;

   always_ff @(posedge clk) begin
      if(reset) begin
         nextR <= 8'd0;
         nextG <= 8'd0;
         nextB <= 8'd0;
         en <= 1;
         count <= 22'd0;
      end
      else if (count == 23'd0) begin
         count <= count + 1;
         en <= 1;
      end
      else if(count == 23'd6250000) begin //After 1/4 of a second
         nextR <= Avg_R;
         nextG <= Avg_G;
         nextB <= Avg_B;
         count <= 23'd0;
      end

      else
         count <= count +1;
   end

   /* set hex displays to display average r,g,b values in hex */

   seg7 seg7_hex0(.bcd(nextB[3:0]), .leds(HEX0));
   seg7 seg7_hex1(.bcd(nextB[7:4]), .leds(HEX1));

   seg7 seg7_hex2(.bcd(nextG[3:0]), .leds(HEX2));
   seg7 seg7_hex3(.bcd(nextG[7:4]), .leds(HEX3));

   seg7 seg7_hex4(.bcd(nextR[3:0]), .leds(HEX4));
   seg7 seg7_hex5(.bcd(nextR[7:4]), .leds(HEX5));

endmodule
```

Figure 18: color_Buffer Module

We had to edit the VGA_framebuffer (as seen in Figure 19 below) to allow for colored pixels instead of black and white. The simplest way we could come up with was to replace the logic to draw a white pixel if pixel_color was true with draw a pixel based on nextR, nextG, and nextB if pixel_color was true.

```
/*
 * Color VGA Framebuffer
 *
 * Edited from orginal which was made by:
 * Stephen A. Edwards, Columbia University
 */

module VGA_framebuffer(
 input logic        clk50, reset,
 input logic [10:0] x, y, // Pixel coordinates
 input logic        pixel_color, pixel_write,
 input logic [7:0] nextR, nextG, nextB,

 output logic [7:0] VGA_R, VGA_G, VGA_B,
 output logic       VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 *HCOUNT 1599 0              1279       1599 0
 *                _____________          _____
 * _____________|    Video      |________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *                _____________________  _____________
 * |____|       VGA_HS          |____|
 */

  parameter HACTIVE      = 11'd 1280,
            HFRONT_PORCH = 11'd 32,
            HSYNC        = 11'd 192,
            HBACK_PORCH  = 11'd 96,
            HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH; //1600

  parameter VACTIVE      = 10'd 480,
            VFRONT_PORCH = 10'd 10,
            VSYNC        = 10'd 2,
            VBACK_PORCH  = 10'd 33,
            VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH; //525

  logic [10:0]             hcount; // Horizontal counter
  logic                    endOfLine;

  always_ff @(posedge clk50 or posedge reset)
    if (reset)               hcount <= 0;
    else if (endOfLine) hcount <= 0;
    else                hcount <= hcount + 11'd 1;

  assign endOfLine = hcount == HTOTAL - 1;

  // Vertical counter
  logic [9:0]              vcount;
  logic                    endOfField;

  always_ff @(posedge clk50 or posedge reset)
    if (reset)               vcount <= 0;
    else if (endOfLine)
      if (endOfField)    vcount <= 0;
      else               vcount <= vcount + 10'd 1;

  assign endOfField = vcount == VTOTAL - 1;

  // Horizontal sync: from 0x520 to 0x57F
  // 101 0010 0000 to 101 0111 1111
  assign VGA_HS = !( (hcount[10:7] == 4'b1010) & (hcount[6] | hcount[5]));
  assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

  assign VGA_SYNC_n = 1; // For adding sync to video signals; not used for VGA

  // Horizontal active: 0 to 1279     Vertical active: 0 to 479
  // 101 0000 0000  1280         01 1110 0000  480
  // 110 0011 1111  1599         10 0000 1100  524
  logic                    blank;
  assign blank = ( hcount[10] & (hcount[9] | hcount[8]) ) |
       ( vcount[9] | (vcount[8:5] == 4'b1111) );

  // Framebuffer memory: 640 x 480 = 307200 bits

  logic                    framebuffer [307199:0];
  logic [18:0]                read_address, write_address;

  assign write_address = x + (y << 9) + (y << 7) ; // x + y * 640
  assign read_address = (hcount >> 1) + (vcount << 9) + (vcount << 7);

  logic                    pixel_read;

  always_ff @(posedge clk50) begin
     if (pixel_write) framebuffer[write_address] <= pixel_color;
     if (hcount[0]) begin
       pixel_read <= framebuffer[read_address];
       VGA_BLANK_n <= ~blank; // Keep blank in sync with pixel data
     end
  end

  assign VGA_CLK = hcount[0]; // 25 MHz clock: pixel latched on rising edge

  assign {VGA_R, VGA_G, VGA_B} = pixel_read ? {nextR, nextG, nextB} : 24'b0;

endmodule
```

Figure 19: VGA_framebuffer Module

arrayShiftLeft:

This module shifts the given 96 x 80 array each time it is enabled on a clock cycle. This module is enabled when the draw_FSM sends a playerEn which is contingent on the FSM transitioning from delay to p1. This FSM is displayed in Figure 25. This will allow the snowball shifted while on player 1, prior to performing the snowball looper. The arrayShiftLeft module and it's testbench is displayed in Figure 20.

```
module arrayShiftLeft #(parameter n=96, m=80) (
   input logic clk, reset, enable,
   input logic [n-1:0] oldArray [0:m-1],
   output logic [n-1:0] shiftedArray [0:m-1]);

   logic [n-1:0] tempArray [0:m-1];

   always_ff @(posedge clk) begin
      if(reset) begin
         for (int i = 0; i<m; i++)
            tempArray[i] <= 0;
         end
      else if (enable) begin
         for (int i = 0; i < m; i++)
            tempArray[i] <= oldArray[i] << 1;
      end
   end

   assign shiftedArray = tempArray;

endmodule

module arrayShiftLeft_testbench();
   logic clk, reset, enable;
   logic [95:0] oldArray [0:79];
   logic [95:0] shiftedArray [0:79];

   arrayShiftLeft #(4,3) dut (.*);
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end
   initial begin
   integer i;
                                          @(posedge clk);
   reset <=1; enable <=0;
   for(i=0; i<4; i++)
      oldArray[i] <= 0;
                                          @(posedge clk);
                                          @(posedge clk);
   reset <=0;
   for(i=0; i<4; i++)
      oldArray[i] <= i+3;
                                          @(posedge clk);          @(posedge clk);

   enable <=1;                            @(posedge clk);
   enable <=0;                            @(posedge clk);
                                          @(posedge clk);
   for(i=0; i<4; i++)
      oldArray[i] <= i+4;
                                          @(posedge clk);          @(posedge clk);

   enable <=1;                            @(posedge clk);
   enable <=0;                            @(posedge clk);
                                          @(posedge clk);
   $stop;
   end
endmodule
```

Figure 20: arrayShiftLeft and Testbench Module

snowballField_update:

This module receives the shifted array, given from the arrayShiftLeft module, and spawns a snowball on the rightmost column at a specified y-value. This spawn signal and y-value is received from player two, because player two has a y-height and can decide to spawn a snowball. The snowballField_update module and it's testbench is displayed in Figure 21.

```
// spawns a snowball on the far right side of the snowball field, when given the y-value and spawn signal
module snowballField_update(clk, yVal, spawn, snowballArrayIn, snowballArrayOut);
   input logic clk, spawn;
   input logic [6:0] yVal;  //0 to 80
   input logic [95:0] snowballArrayIn [0:79];  //width to height
   output logic [95:0] snowballArrayOut [0:79];

   logic [95:0] snowballArray [0:79];

   always_ff @(posedge clk) begin
      if (spawn) begin
//       // get prev snowballArray
         snowballArray = snowballArrayIn;
//
         // add snowball
         snowballArray[yVal][0] = 1'b1;

//       snowballArrayOut <= snowballArray;
      end
      else snowballArray = snowballArrayIn;
   end

   assign snowballArrayOut = snowballArray;
endmodule

module snowballField_update_testbench();
   logic clk, spawn;
   logic [6:0] yVal;  //0 to 80
   logic [95:0] snowballArrayIn [0:79];  //width to height
   logic [95:0] snowballArrayOut [0:79];

   snowballField_update dut (.clk, .yVal, .spawn, .snowballArrayIn, .snowballArrayOut);

   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   integer i = 0;

   initial begin
      spawn = 0;
      yVal = 0;

      // input snowball field is all zeroes
      for (i = 0; i < 80; i++) begin
         snowballArrayIn[i] = 96'd0;
      end @(posedge clk);

      spawn = 1; @(posedge clk);  // spawn a snowball at yVal = 0

      snowballArrayIn[0][0] = 96'd1;
      yVal = 79; @(posedge clk);  // spawn a snowball at yVal = 79

      @(posedge clk);

      $stop;
   end
endmodule
```

Figure 21: snowballField_update and Testbench Module

snowball_looper:

This module receives the updated snowball array from the snowballField_update, and iterates through each index within the snowball array. Since the array is 96 x 80, the snowball_looper will perform 7680 iterations. In the event a snowball is located at an index in the array, the snowball_looper will map the 96 x 80 snowball array to the VGA screen which is 320 x 240. The snowball_looper will also inform the draw_FSM to enable the draw signal. When the snowball_looper is finished iterating through the snowball array and when the snowball is drawn, the draw_FSM moves from the snowball to player two state, as seen in Figure 25. The snowball_looper module and it's testbench is displayed in Figure 22 and 23, respectively.

```
module snowball_looper (clk, en, reset, snowballArrayIn, done, Xout, Yout, draw);
   input logic clk, en, reset;
   input logic [95:0] snowballArrayIn [0:79];
   output logic done, draw;
   output logic [6:0] Xout, Yout;

   parameter rowLength = 96;
   parameter heightLength = 80;

   integer i;
   integer j;

   always_ff @(posedge clk) begin
      if (reset) begin
         i <= 0;
         j <= 0;

         done <= 1'b1;
         draw <= 1'b0;
         Xout <= j;
         Yout <= i;
      end
      else if (en) begin
            if (done) begin
               i <= 0;
               j <= 0;
               draw <= 1'b0;
            end
            else if (i == 79 && j == 95) begin
               // no longer increase i and j
               done <= 1'b1;  // done

               if (snowballArrayIn[i][j] == 1'b1 && !done) begin
                  draw <= 1'b1;
               end
               else begin
                  draw <= 1'b0;
               end
            end

            else if (j == 95) begin
               // increase i, and reset j
               i <= i + 1;
               j <= 0;

               done <= 1'b0;  // not done
               draw <= 1'b0;
            end
            else begin  // j != 95
               // increase j
               j <= j + 1;

               done <= 1'b0;

               if (j != 95) begin
                  if (snowballArrayIn[i][j] == 1'b1) begin
                     draw <= 1'b1;  // draw
                  end
                  else begin
                     draw <= 1'b0;
                  end
               end
               else begin
                  draw <= 1'b0;
               end
            end

            Xout <= j;
            Yout <= i;
      end
      else begin  // !en ("wait") because draw3x3 is drawing
         i <= i;
         j <= j;
         done <= 1'b0;
         draw <= 1'b0;
         Xout <= j;
         Yout <= i;
      end
   end
endmodule
```

Figure 22: snowball_looper Module

```
module snowball_looper_testbench();
   logic clk, en, reset;
   logic [95:0] snowballArrayIn [0:79];
   logic done, draw;
   logic [6:0] Xout, Yout;

   snowball_looper dut (.clk, .en, .reset, .snowballArrayIn, .done, .Xout, .Yout, .draw);

   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   integer a;
   integer b;

   initial begin
      reset = 1'b1; @(posedge clk);
      reset = 1'b0; @(posedge clk);

      en = 1'b0; @(posedge clk);

      // input snowball field is all zeroes
      for (a = 0; a < 80; a++) begin
         for (b = 0; b < 96; b++) begin
            snowballArrayIn[a][b] = 1'b0; @(posedge clk);
         end
      end
      @(posedge clk);

      // input snowball fields has 3 1's ("snowball")
      snowballArrayIn[0][0] = 1'b1; @(posedge clk); // 1
      snowballArrayIn[40][47] = 1'b1; @(posedge clk); // 2
      snowballArrayIn[79][95] = 1'b1; @(posedge clk); // 3

      a = 0;
      b = 0; @(posedge clk);

      // go through the loop with enable on
      for (a = 0; a < 80; a++) begin
         for (b = 0; b < 96; b++) begin
            if (a == 0) begin
               en = 1'b1; @(posedge clk);
               en = 1'b0;
               b = b + 1; @(posedge clk); //due to enable
               repeat (18) begin
                  en = 1'b0; @(posedge clk);
               end
            end
            else if (a == 40) begin
               en = 1'b1; @(posedge clk);
               en = 1'b0;
               b = b + 1; @(posedge clk); //due to enable
               repeat (18) begin
                  en = 1'b0; @(posedge clk);
               end

            end
            else if (a == 79) begin
               en = 1'b1; @(posedge clk);
               en = 1'b0;
               b = b + 1; @(posedge clk); //due to enable
               repeat (18) begin
                  en = 1'b0; @(posedge clk);
               end
            end

            en = 1'b1; @(posedge clk);
         end
      end
      repeat (5) begin
      @(posedge clk);
      end

      reset = 1'b1; @(posedge clk);
      $stop;
   end
endmodule
```

Figure 23: snowball_looper Testbench Module

snowballField_map_to_vga:

This module maps the 96 x 80 snowball array to the VGA screen which is 320 x 240. The 96 x 80 snowball array corresponds to 288 x 240 pixels on the VGA screen, as the remaining pixels are used as either offsets, player one, or player two, as mentioned in the introduction. Recall that each snowball in the 96 x 80 snowball array is size 3x3 on the VGA screen which is why 96 x 80 corresponds to 288 x 240.

The formula to map the 96 x 80 snowball array can be shown in Figure 24.

```
// Maps the 2D array snowball field to the center value of it's respective placement on the VGA screen
module snowballField_map_to_vga(Xin, Yin, Xout, Yout);
   input logic [6:0] Xin, Yin;
   output logic [10:0] Xout;
   output logic [10:0] Yout;

   logic [4:0] offsetRight;
   logic [8:0] vgaRightmostPixel;
   logic [7:0] vgaBottommostPixel;
   logic pixel;

   assign offsetRight = 5'd19;
   assign vgaRightmostPixel = 9'd319;
   assign vgaBottommostPixel = 8'd239;
   assign pixel = 1'b1;

   always_comb begin
      Xout = vgaRightmostPixel - (Xin * 3) - offsetRight - pixel;
      Yout = vgaBottommostPixel - (Yin * 3) - pixel;
   end
endmodule

module snowballField_map_to_vga_testbench();
   logic [6:0] Xin, Yin;
   logic [10:0] Xout;
   logic [10:0] Yout;

   snowballField_map_to_vga dut (.Xin, .Yin, .Xout, .Yout);

   initial begin
      // @ origin(bottomRight) - output should be (299, 238)
      Xin = 11'b0000000;
      Yin = 11'b0000000; #10;

      // @ bottomLeft - output should be (14, 238)
      Xin = 11'd95;
      Yin = 11'd0; #10;

      // @ topRight - output should be (299, 1)
      Xin = 11'd0;
      Yin = 11'd79; #10;

      // @ topLeft - output should be (14, 1)
      Xin = 11'd95;
      Yin = 11'd79; #10;

      // A point somewhere around the middle
      Xin = 11'd48;
      Yin = 11'd40; #10;
      $stop;
   end
endmodule
```

Figure 24: snowballField_map_to_vga and Testbench Module

draw_FSM:

This module is a finite state machine that determines which entity to draw on the VGA screen. This finite state machine is displayed in Figure 25. The three entities that are drawn on the screen are player one, snowball array, and player two. draw_FSM consists of four states. These states are delay, p1, snowball, and p2.

The delay state does not enable the module that writes to the VGA screen and consists of a counter that can reach up to 6250000. After reaching a count of 6250000 which is equivalent to 20 ns, using a 50Mhz clock, the state changes from delay to p1. This transition gives an output to inform draw_TEST to begin drawing player 1.

The p1 state does not move to the snowball state until player 1 has been drawn onto the VGA screen. After receiving a doneDraw signal from the draw_TEST module, the state changes from player 1 to snowball.

The snowball does not move to the p2 state until each index in the snowball array has been iterated and until a doneDraw signal is received from the draw_TEST module. In addition, during iteration of the snowball array, if a snowball is detected, the draw_TEST module will be enabled to draw the snowball that was found. During the transition from snowball to p2, a signal is sent to draw_TEST to begin drawing player two.

The p2 state does not move to the delay state until player two has been drawn onto the VGA screen. After receiving a doneDraw signal from the draw_TEST module, the state changes from

player 2 to delay. During this transition, the counter is reset to 0, to allow the delay to count towards 6250000. The draw_FSM module and it's testbench is displayed in Figure 26 and 27, respectively.

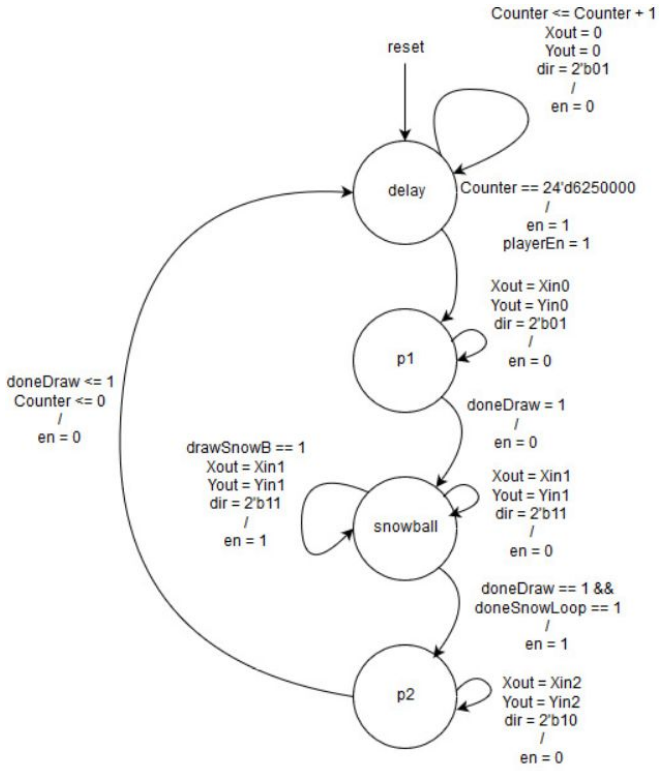

Figure 25: Drawing Control Finite State Machine

```
module draw_FSM (clk, reset, doneDraw, doneSnowLoop, drawSnowB, Xin0, Yin0, Xin1, Yin1, Xin2, Yin2, Xout, Yout,
   en, dir, playerEn);
   input logic clk, reset, doneDraw, doneSnowLoop, drawSnowB;
   input logic [10:0] Xin0, Yin0, Xin1, Yin1, Xin2, Yin2;  // 0 == P1, 1 == Snowball, 2 == P2
   output logic en, playerEn;
   output logic [1:0] dir;
   output logic [10:0] Xout, Yout;

   // State Variables
   enum {delay, p1, snowball, p2} ps, ns;

   logic [23:0] counter;

   always_comb begin
      case (ps)
         delay: begin
            Xout = 0;
            Yout = 0;
            dir = 2'b01;  // snowball dir, but does not matter since enable == 0

            // was 12500000, using 1000
            if (counter == 24'd6250000) begin
               ns = p1;
            end
            else begin
               ns = delay;
            end
         end
         p1: begin
            Xout = Xin0;
            Yout = Yin0;
            dir = 2'b01;

            if (doneDraw) begin
               ns = snowball;
            end
            else begin
               ns = p1;
            end
         end
         snowball: begin
            Xout = Xin1;
            Yout = Yin1;
            dir = 2'b11;

            if (doneSnowLoop && doneDraw) begin
               ns = p2;
            end
            else begin
               ns = snowball;
            end
         end
         p2: begin
            Xout = Xin2;
            Yout = Yin2;
            dir = 2'b10;

            if (doneDraw) begin
               ns = delay;
            end
            else begin
               ns = p2;
            end
         end
      endcase
   end

   always_ff @(posedge clk)
      if (reset) begin
         ps <= delay;
         counter <= 24'd0;
      end
      else begin
         if (ps == delay) begin
            counter <= counter + 24'b1;
         end
         else if (ns == delay) begin
            counter <= 0;
         end

         ps <= ns;
      end

   assign en = (ps == delay && ns == p1) | // transition from delay to p1
               (ps == snowball && drawSnowB == 1) | // in snowball and drawSnowB
               (ps == snowball && ns == p2); // transition from snowball to p2


   assign playerEn = (ps == delay && ns == p1);  // transition delay to p1
endmodule
```

Figure 26: draw_FSM Module

```
module draw_FSM_testbench();
   logic clk, reset, doneDraw, doneSnowLoop, drawSnowB;
   logic [10:0] Xin0, Yin0, Xin1, Yin1, Xin2, Yin2;  // 0 == P1, 1 == Snowball, 2 == P2
   logic en, playerEn;
   logic [1:0] dir;
   logic [10:0] Xout, Yout;

   draw_FSM dut (.clk, .reset, .doneDraw, .doneSnowLoop, .drawSnowB, .Xin0, .Yin0, .Xin1, .Yin1, .Xin2,
      .Yin2, .Xout, .Yout, .en, .dir, .playerEn);

   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   initial begin
      // at delay state
      reset = 1;
      doneDraw = 0;
      doneSnowLoop = 0;
      Xin0 = 7'd0;
      Yin0 = 7'd0;
      Xin1 = 7'd155;
      Yin1 = 7'd118;
      Xin2 = 7'd299;
      Yin2 = 7'd238;
      drawSnowB = 0; @(posedge clk);
      reset = 0; @(posedge clk);

      // delay state for 250000 clock cycles
      repeat (1000) begin
       @(posedge clk);
      end

      // at p1
      repeat (27) begin
      @(posedge clk);  // stay at p1 for 27 clock cycles
      end

      // go to snowball state
      doneDraw = 1; @(posedge clk);

      // stay at snowball state
      repeat (10) begin
      drawSnowB = 1;
      @(posedge clk);  // stay at snowball for 10 clock cycles
      end

      drawSnowB = 0;
      // now at p2
      doneSnowLoop = 1; @(posedge clk);

      // stay at p2
      doneDraw = 0; @(posedge clk);
      repeat (10) begin
      @(posedge clk);  // stay at snowball for 10 clock cycles
      end

      // move back to delay
      doneDraw = 1; @(posedge clk);
      // stay at delay *********** change delay value to a high value
      repeat (10) begin
      @(posedge clk);
      end
      // ****************************************2nd rotation
      doneDraw = 0;
      doneSnowLoop = 0;
      // delay state for 250000 clock cycles
      repeat (1000) begin
       @(posedge clk);
      end

      // at p1
      repeat (27) begin
      @(posedge clk);  // stay at p1 for 27 clock cycles
      end

      // go to snowball state
      doneDraw = 1; @(posedge clk);

      // stay at snowball state
      repeat (10) begin
      drawSnowB = 1;
      @(posedge clk);  // stay at snowball for 10 clock cycles
      end

      drawSnowB = 0;
      // now at p2
      doneSnowLoop = 1; @(posedge clk);

      // stay at p2
      doneDraw = 0; @(posedge clk);
      repeat (10) begin
      @(posedge clk);  // stay at snowball for 10 clock cycles
      end

      // move back to delay
      doneDraw = 1; @(posedge clk);
      // stay at delay *********** change delay value to a high value
      repeat (10) begin
      @(posedge clk);
      end

      $stop;
   end
```

Figure 27: draw_FSM Testbench Module

Drawing and Erasing Controls:

After we created a way to order the drawing of the players cursor and the snowballs, we needed a way to actual erase the old location of each on screen and draw the updated location. To do this, we created a finite state machine that interacts with a 3 pixels by 3 pixels box drawer to erase and draw. Figure 28 below shows the ASM chart for the finite state machine.

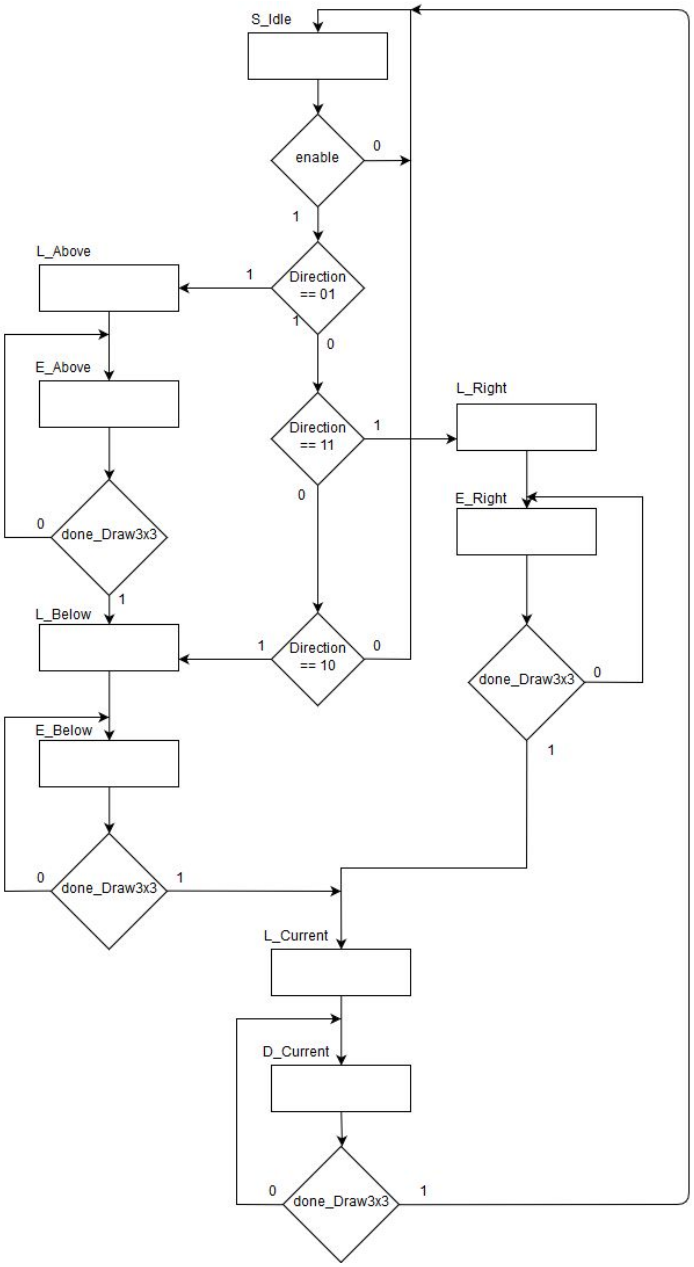

Figure 28: ASM Chart for draw_erase_fsm module

Looking at the ASM chart above, the draw_erase_fsm module works by if there is a enable signal then based on the direction code provided it follows through a path to erase and draw. For

player 1, we need to erase above and below since we do not know if the previous location was above or below us. For player 2 since it is most of the time moving upwards, we only need to erase from below. For snowballs since they are only moving to the left we only need to erase from the right of the current snowball position. For the state names: “L_” means load values, “E_” means erase, and “D_” means draw. We need a load state before erasing/drawing since the 3x3 pixel box drawer we made needed to have the pixels loaded one cycle before the drawer is enabled. Figures 29 and 30 below show the SystemVerilog for the draw_erase_fsm and the draw_erase_Datapath modules.

```
module draw_erase_Datapath (
   input logic clk, reset,
   input logic load_Above, load_Below, load_Right, load_Current,
   input logic [10:0] x_in, y_in,
   input logic [7:0] Avg_colors [0:2],
   output logic [10:0] x_toDraw, y_toDraw,
   output logic [7:0] VGA_colors [0:2],
   output logic pixel_color
   );

   always_ff @(posedge clk) begin

      if(load_Above)begin
         x_toDraw <= x_in;
         y_toDraw <= y_in - 3;
         VGA_colors[0] <= 8'h00;
         VGA_colors[1] <= 8'h00;
         VGA_colors[2] <= 8'h00;
         pixel_color <= 0;
      end

      if(load_Below)begin
         x_toDraw <= x_in;
         y_toDraw <= y_in + 3;
         VGA_colors[0] <= 8'h00;
         VGA_colors[1] <= 8'h00;
         VGA_colors[2] <= 8'h00;
         pixel_color <= 0;
      end

      if(load_Right) begin
         x_toDraw <= x_in + 3;
         y_toDraw <= y_in;
         VGA_colors[0] <= 8'h00;
         VGA_colors[1] <= 8'h00;
         VGA_colors[2] <= 8'h00;
         pixel_color <= 0;
      end

      if(load_Current) begin
         x_toDraw <= x_in;
         y_toDraw <= y_in;
         VGA_colors <= Avg_colors;
         pixel_color <= 1;
      end
   end


endmodule
```

Figure 29: draw_erase_Datapath module

```
module draw_erase_FSM (
   input logic clk, reset,
   input logic [1:0] direction, //01 = P1, 10 = P2, 11 = Snowballs
   input logic enable,
   input logic done_Draw3x3,
   output logic draw_Draw3x3,
   output logic done_signal, load_Above, load_Below, load_Right, load_Current
   );

   enum {S_Idle, L_Above, E_Above, L_Below, E_Below, L_Right, E_Right, L_Current, D_Current}
 ps, ns;
   /* State Lables:
    * S_Idle  = Idle State
    *
    * L_Above = Load Above values and delay for one clock cycle
    * E_Above = Erase section above given coordinates
    *
    * L_Below = Load Below values and delay for one clock cycle
    * E_Below = Erase section below given coordinates
    *
    * L_Right = Load Right values and delay for one clock cycle
    * E_Right = Erase section right of given coordinates
    *
    * L_Current = Load Current values and delay for one clock cycle
    * D_Current = Draw section of given coordinates
    */

   always_comb begin
      case (ps)
         S_Idle:  if(enable)
                     if      (direction == 2'b01) ns = L_Above;
                     else if (direction == 2'b10) ns = L_Below;
                     else if (direction == 2'b11) ns = L_Right;
                     else ns = S_Idle;
                  else                            ns = S_Idle;

         L_Above:                    ns = E_Above;

         E_Above: if (!done_Draw3x3) ns = E_Above;
                  else               ns = L_Below;

         L_Below:                    ns = E_Below;

         E_Below: if (!done_Draw3x3) ns = E_Below;
                  else               ns = L_Current;

         L_Right:                    ns = E_Right;

         E_Right: if (!done_Draw3x3) ns = E_Right;
                  else               ns = L_Current;

         L_Current:                  ns = D_Current;

         D_Current: if (!done_Draw3x3) ns = D_Current;
                    else               ns = S_Idle;
      endcase
   end

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

   assign draw_Draw3x3 = (ps == L_Above & ns == E_Above) |
                         (ps == L_Below & ns == E_Below) |
                         (ps == L_Right & ns == E_Right) |
                         (ps == L_Current & ns == D_Current);

   assign load_Above = (ps != L_Above & ns == L_Above);
   assign load_Below = (ps != L_Below & ns == L_Below);
   assign load_Right = (ps != E_Right & ns == E_Right);
   assign load_Current = (ps != L_Current & ns == L_Current);
   assign done_signal = (ps == S_Idle);
endmodule

module draw_erase_FSM_testbench();
   logic clk, reset;
   logic enable, done_Draw3x3, draw_Draw3x3;
   logic [1:0] direction;
   logic load_Above, load_Below, load_Right, load_Current;
   logic done_signal;

   draw_erase_FSM dut (.*);
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end
   initial begin
   integer i;
                                                  @(posedge clk);
   reset <= 1; enable <= 0; done_Draw3x3 <= 1; direction <= 2'b01;   @(posedge clk);
   reset <= 0;                                                       @(posedge clk);
               enable <= 1;                                          @(posedge clk);
               enable <= 0; done_Draw3x3 <= 0;                       @(posedge clk);
   repeat (18) begin
      @(posedge clk);
   end
                            done_Draw3x3 <= 1;                       @(posedge clk);
                             done_Draw3x3 <= 0;                      @(posedge clk);
   repeat (18) begin
      @(posedge clk);
   end
                            done_Draw3x3 <= 1;                       @(posedge clk);
                             done_Draw3x3 <= 0;                      @(posedge clk);
   repeat (18) begin
      @(posedge clk);
   end
                            done_Draw3x3 <= 1;                       @(posedge clk);
                                                                     @(posedge clk);
                                                                     @(posedge clk);
                                                                     @(posedge clk);
                                                                     @(posedge clk);

   $stop;
   end
endmodule
```

Figure 31: draw_erase_FSM module

To be able to draw the 3 pixels by 3 pixel squares used by the player cursor and snowballs we created a module called draw3x3 which takes in the center coordinate of a square and draws a 3 pixel by 3 pixel square around it when enabled. Figure 32 and 33 below show the code of the draw3x3 module.

```
module draw3x3 (
   input logic clk, reset,
   input logic [10:0] x_in, y_in,
   input logic write_en,
   output logic [10:0] x, y,
   output logic pixel_write, done_signal);


   logic [10:0] x_temp, y_temp;
   logic [3:0] count;

   always_ff @(posedge clk) begin
      if(reset) begin
         count <= 4'd8;
         x_temp <= x_in;
         y_temp <= y_in;
         done_signal <= 0;
         pixel_write <= 0;
      end

      else if(count == 4'd8 & write_en) begin
         x_temp <= x_in - 1;
         y_temp <= y_in - 1;
         pixel_write <= 1;
         count <= 4'd0;
         done_signal <= 0;
      end

      else if (count == 4'd8 & !write_en) begin
         x_temp <= x_in;
         y_temp <= y_in;
         pixel_write <= 0;
         done_signal <= 1;
      end

      else if (count == 4'd7) begin
         x_temp <= x_in + 1;
         y_temp <= y_in + 1;
         count <= count + 1;
      end

      else if (count == 4'd6) begin
         x_temp <= x_in;
         y_temp <= y_in + 1;
         count <= count + 1;
      end

      else if (count == 4'd5) begin
         x_temp <= x_in - 1;
         y_temp <= y_in + 1;
         count <= count + 1;
      end

      else if (count == 4'd4) begin
         x_temp <= x_in + 1;
         y_temp <= y_in;
         count <= count + 1;
      end

      else if (count == 4'd3) begin
         x_temp <= x_in;
         y_temp <= y_in;
         count <= count + 1;
      end

      else if (count == 4'd2) begin
         x_temp <= x_in - 1;
         y_temp <= y_in;
         count <= count + 1;
      end

      else if (count == 4'd1) begin
         x_temp <=  x_in + 1;
         y_temp <= y_in - 1;
         count <= count + 1;
      end

      else if (count == 4'd0) begin
         x_temp <= x_in;
         y_temp <= y_in - 1;
         count <= count + 1;
         pixel_write <= 1;
      end

   end

   assign x = x_temp;
   assign y = y_temp;

endmodule

module draw3x3_testbench();
   logic clk, reset;
   logic [10:0] x_in, y_in;
   logic write_en;
   logic [10:0] x, y;
   logic pixel_write, done_signal;

   draw3x3 dut (.*);
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end
   initial begin
   integer i;
                                                                  @(posedge clk);
   reset <=1; x_in<=5; y_in<=5;  write_en <=0;  @(posedge clk);
   reset <=0;                                                     @(posedge clk);
                                                                  @(posedge clk);
               x_in<=3; y_in<=3;              write_en <=1; @(posedge clk);
                                              write_en <=0; @(posedge clk);
   for(i=0; i<11; i++)
                                                                  @(posedge clk);
                                                                  @(posedge clk);

   $stop;
   end
endmodule
```

Figure 32: draw3x3 module PAGE 1

Figure 33: draw3x3 module PAGE 2

We then created a module called draw_TEST to act as a top level module to connect the draw_erase_fsm to the draw_erase_Datapath as well as the draw3x3 module. Figure 34 below show the code for the draw_TEST module.

```
module draw_TEST(
   input logic clk, reset,
   input logic [1:0] direction, //01 = P1, 10 = P2, 11 = Snowballs
   input logic enable,
   input logic [10:0] x_in, y_in,
   input logic [7:0] Avg_colors [0:2],
   output logic [10:0] x, y,
   output logic [7:0] VGA_colors [0:2],
   output logic pixel_write,
   output logic done_signal,
   output logic pixel_color
   );

   logic done_Draw3x3;
   logic draw_Draw3x3;
   logic load_Above, load_Below, load_Right, load_Current;
   draw_erase_FSM FSM (.clk,
                       .reset,
                       .direction,
                       .enable,
                       .done_Draw3x3,
                       .draw_Draw3x3,
                       .done_signal,
                       .load_Above,
                       .load_Below,
                       .load_Right,
                       .load_Current);

   logic [10:0] x_toDraw, y_toDraw;
   draw_erase_Datapath Datapath (.clk,
                                 .reset,
                                 .load_Above,
                                 .load_Below,
                                 .load_Right,
                                 .load_Current,
                                 .x_in,
                                 .y_in,
                                 .Avg_colors,
                                 .x_toDraw,
                                 .y_toDraw,
                                 .pixel_color,
                                 .VGA_colors);

   draw3x3 boxdraw (.clk,
                    .reset,
                    .x_in(x_toDraw),
                    .y_in(y_toDraw),
                    .write_en(draw_Draw3x3),
                    .x,
                    .y,
                    .pixel_write,
                    .done_signal(done_Draw3x3));

endmodule

module draw_TEST_testbench();
   initial begin
   integer i;

   @(posedge clk);
   reset<=1; direction<=2'b10; enable<=0; x_in<=0; y_in<=0; Avg_colors<={8'h10, 8'h00, 8'h00
};  @(posedge clk);
   reset<=0;
                                 enable<=1; x_in<=5; y_in<=5;
                                       @(posedge clk);
                                 enable<=0;
                                                      @(posedge clk);
   repeat(25) begin
      @(posedge clk);
   end

   $stop;
   end
endmodule
```

Figure 34: draw_TEST module

Simulation for each Module:

P1_snowball_p2_board Simulation:

P1Input Simulation:

Figure 35 below displays the joystick_data being greater than the upper threshold of 3839 thus increasing the y_height output, and then decreasing since the joystick_data is lower than the lower threshold of 256.

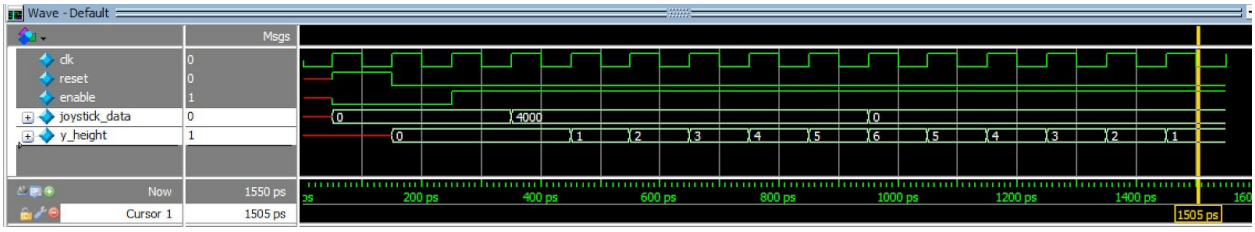

Figure 35: P1Input Simulation

P2Input Simulation:

Figure 36 below displays the mic_signal being greater than the threshold of 5000 and thus increasing the y_height output and then jumping back down to zero since the mic_signal went below the threshold of 5000.

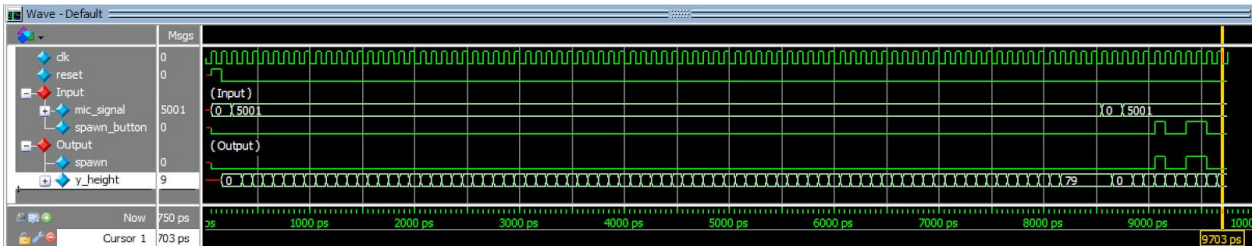

Figure 36: P2Input Simulation

snowballField_update Simulation:

Figure 37 displays a snowballArrayOut adding a snowball from the snowballArrayIn at the specified yVal location, when the spawn signal is on.

Figure 37: snowballField_update Simulation

Snowball_looper Simulation:

Figure 38 displays the snowballArrayIn first being initialized to zeroes, and then containing three snowballs. These snowballs are located at snowballArrayIn[0][0], snowballArrayIn[40][47], snowballArrayIn[79][95]. Performing the snowball_looper reveals that there are three snowballs located in their respective locations.

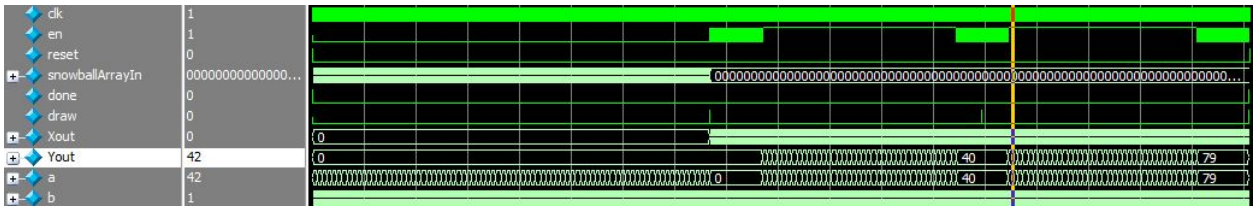

Figure 38: snowball_looper Simulation

snowballField_map_to_vga Simulation:

Figure 39 displays the Xin and Yin values mapping to the proper Xout and Yout values, corresponding to the VGA screen. The four points were chosen for each corner of the snowball field.

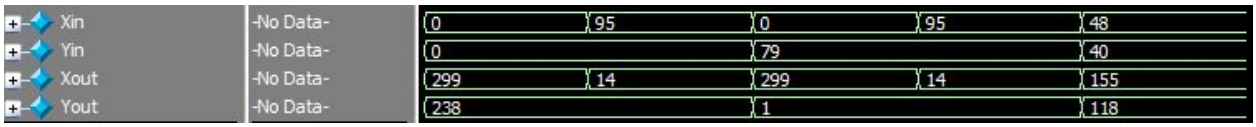

Figure 39: snowballField_map_to_vga Simulation

draw_TEST Simulation:

Figure 40 displays the draw_TEST module drawing a the next location for player 1 at test coordinates of (11, 5). We have highlighted an area with a white rectangle to focus on.

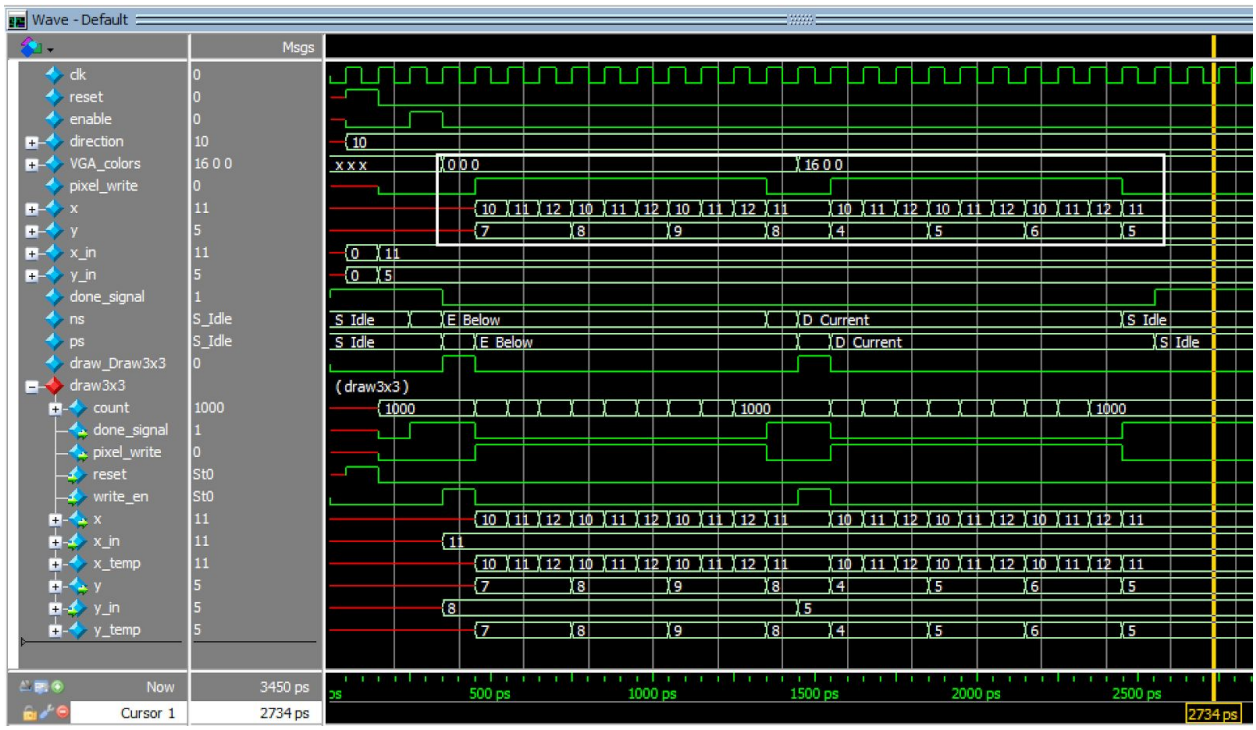

Figure 40: draw_TEST Simulation

In the white rectangle in Figure 40 above, we can see that the square below (11,5) which is (11,8) (due to the y-values on the screen increasing as you go down) begin drawn but with

VGA_colors as 0,0,0 which is black. Then the current location is drawn at (11,5) with VGA_colors of 10,0,0.

Overall Description of the System:

Despite the some of the bugs presented in Dodge Snowball, Dodge Snowball is effectively functional and implements all of the specifications except for the microphone, due to conflicts implementing both the camera and microphone. Instead of the microphone, KEY[1:0] was used to simulate the microphone. In the end, two players can still play Dodge Snowball. Figure 41 displays player one, player two, and the snowballs spawned by player two.

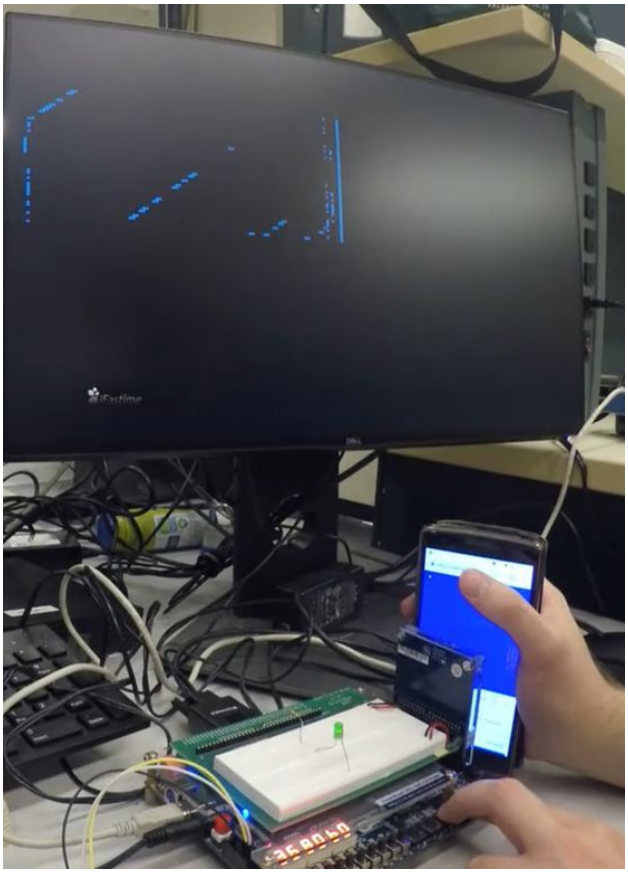

Figure 41: Overall System in Action

Analysis of Results / Error:

Player 1 Control:

While creating the ADC for the joystick, we ran into a problem that the ADC driver provided did not work for us. With help from Tony Faubert on the Canvas Discussion Board, we were able to follow his tutorial in using the IP Catalog to create a ADC module which worked for our needs.

Player 2 Control:

While trying to add the microphone into our project, when we uploaded our code onto the DE1-SoC our camera decided to stop sending average color information. We think this might be due to the microphone drivers provided interacting with camera driver in a unseen away. We decided that showing our camera working for the demo was more important than showing our microphone working since we had already shown the microphone working in a previous lab report. Thus we made KEY 0 act as if the player 2 voice is above the threshold if pressed, or below the threshold if not pressed.

Camera:

Although the averaging of the colors from the camera does work, there are a few slight issues. We decided to only set every ¼ of a second average color as the color drawn on screen since we didn't want any quick random changes. However this resulted in slight flickering every ¼ of a second since the average color was different. We should have averaged the average colors every ¼ of a second to get a smoother average color so that there would be less of a flicker.

Also our camera seemed to have a blue bias where the average color would always be slightly more blue than it should be. Thus red turned into a light red/orange, green turned into greenish blue, and blue turned into a lighter blue. We could have subtracted an amount from the average blue to do some color correction however we wanted to keep the average color values the same as they were coming from the camera.

Snowball Field:

Occasional snowballs remain at the spawn location. Additionally, we did not get a chance to remove the snowballs at the leftmost column. To do this correctly, we would have to add more states into our draw_FSM to make sure we always draw black to where the last column of snowballs are before our 2d array shifts them off and we lose track of them.

Player Cursors Not Erasing Properly:

We had some issues with the player cursors not erasing properly while simulating the draw_TEST module for different cases worked. We decided to try to only erase the pixels we needed to based off a suggestion from our TA Shuowei. Next time we could try to just erase the whole section of where the player cursors can be and then draw the new location.

Conclusion:

Aside from the mentioned logic bugs, lack of implementation of the microphone which was problematic when implemented with the camera, lack of time to implement a module to detect if player one is in contact with a snowball, and lack of time to implement a module that removes the snowballs at the leftmost side of the snowball field, Dodge Snowball meets all of the other specifications and implements all of the vital modules.

In the end, Dodge Snowball is playable between two players.

Appendices:

- https://canvas.uw.edu/courses/1219718/discussion_topics/4558139

Miscellaneous:

Figure 42 displays the flow summary.

Flow Status	Successful - Tue Dec 11 20:54:20 2018
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SOC_D8M_RTL
Top-level Entity Name	gameBoardTestWithCamera
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	7,911 / 32,070 (25 %)
Total registers	17963
Total pins	200 / 457 (44 %)
Total virtual pins	0
Total block memory bits	395,264 / 4,065,280 (10 %)
Total DSP Blocks	2 / 87 (2 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	3 / 6 (50 %)
Total DLLs	0 / 4 (0 %)

Figure 42: Flow Summary

The total number of hours to complete the lab is 100 hours altogether.