ECE 385

Fall 2021

Final Project Report

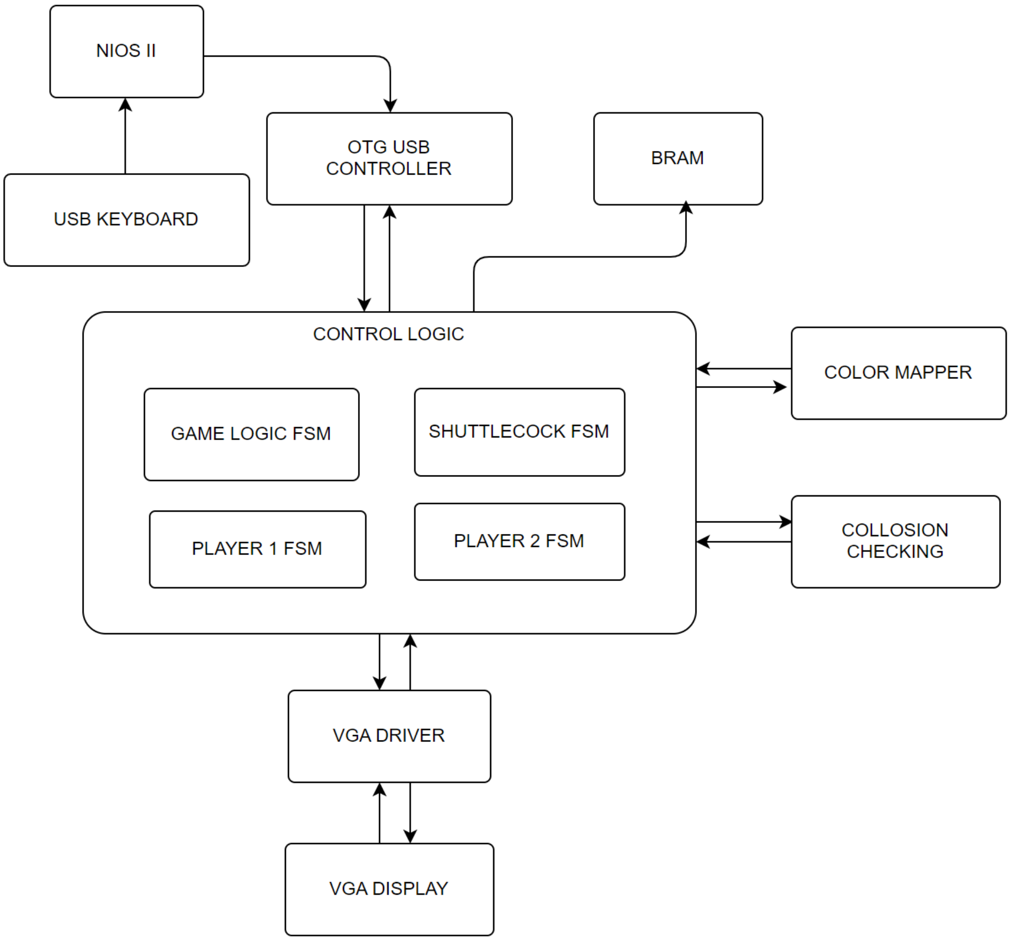
Stick Figure Badminton

Xu Ke / Zhu Xiaohan

LA4/Thursday & 18:00-20:50 Huang Tianhao

1. Introduction

The goal of our final project is to re-design and implement a game called *Stick Figure Badminton* on the FPGA as a System-on-chip. This is a two-player game that two stickmen can only move left and right on their own fields and they need to catch the shuttlecock from each other. If one of them fails to make it, including the shuttlecock falls on his field or hit the net, he will lose the game. Two stickmen will be controlled by one keyboard.



Here is the general flow of our circuit, the idea is basically based on lab 8.

2. Module Description

The most important parts of this circuit are control logic and color mapper, we can describe the hole circuit by describing the input and output of those modules.

<u>Player1FSM: (Same as Player2FSM)</u>

Input: *Reset* - Reset player1 to S1 state to serve the ball

Input: *Clk, frame_clk*

Input: *keycode* - choose state transition

Output: *figure1_state* - control figure1.sv, for player1 state transition

Output: *ball_exist1, ball_shoot1, ball_hit1* - control ball.sv, indicate the motion of the players so that it can give corresponding state of the ball.

Color Mapper:

Input: *Clk*

Input: *DrawX, DrawY:* This signal is generated from the VGA controller and it indicates which current pixel is being drawn. This is important because all object positions are compared to the pixel, both for choosing what color to be drawn and for determining hit detection ("Is" family relies on DrawX, DrawY)

Input: *figure1_data/ figure2_data/ ball_data / background_data* – picture data of players and ball and background: These are the signal generated by each RAM for color mapper to determine the color to print.

Input: *is_figure1/ is_figure2/ is_ball/ is_background* – Basically this family of "Is" logic variables determines whether an object exists at that specific pixel, i.e. if "Is Ship" = 1 then the ship exists at that pixel. The first sort of check that is done when deciding what type of object is present, so it checks all of the "Is" Family.

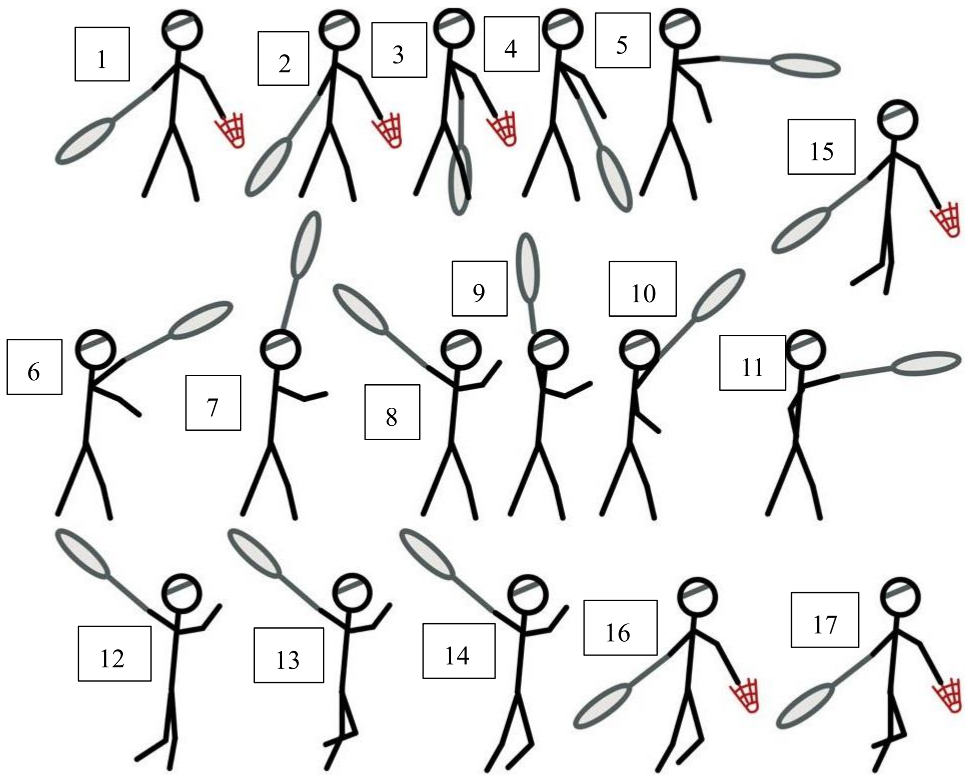
Output: *VGA_R, VGA_G, VGA_B:* These are the only outputs of the color mapper module but very important: These decide the intensity of each color channel for the current pixel being drawn.

3. Design Procedure / State Diagram / Simulation Waveform

Overview of the design procedure:

Our project is only based on lab8 files and used the 385 helper-tool to transform picture to text.

I fully understood how to use the helper tool first, then decomposed the stick figure's motions according to the original game. I redrew each step of motion by hand in my iPad and put them in one picture (As the picture shown below). Then I fit the picture in right size, as well as marked the important coordinates of one state motion: the left upper corner, the center, and the frame size (which was put in figure1.sv and indexed by specific state). So according to our state machine's output, our figure1.sv will choose the right part of the picture to read and show.



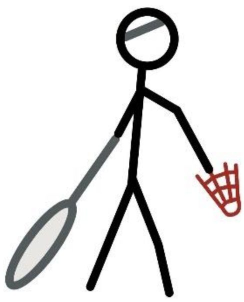
After those above processes, I implemented the FSM and tested whether the state transition works. Then, I implemented the ball's motion. This is also the most difficult part of the project, because I need to consider the collision condition criterion and gravity of the ball, which make the motion of the ball hard to show. Finally, I choose a relatively fuzzy judgment method for the collision condition criterion.

Details of my procedures are listed below:

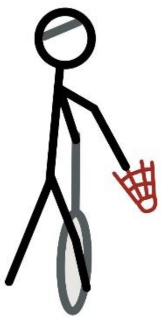
3.1 State transition

In this procedure, all motions of one figure are decomposed to states (which listed in figure1FSM &figure2FSM):

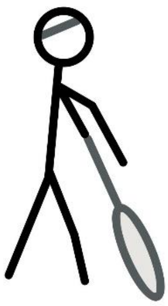
State S1:
The start state of the player who serve the ball.
Output:
ball_exist1 = 1'b0;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 1
(*Condition*) **Next state**:
(keycode *A*) SL1
(ketcode *D*) SR1
(keycode *S*) S2



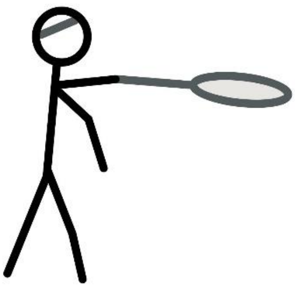
State S2:
One of the transition states of serving the ball.
Output:
ball_exist1 = 1'b0;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 2
(*Condition*) **Next state:**
(*Unconditional*) S3



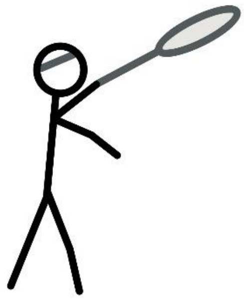
State S3:
One of the transition states of serving the ball.
Output:
ball_exist1 = 1'b0;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 3
(*Condition*) **Next state:**
(*Unconditional*) S4



State S4:
One of the transition states of serving the ball.
The ball now apart from player.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b1;
Corresponding diagram number: 4
(*Condition*) **Next state:**
(*Unconditional*) S5

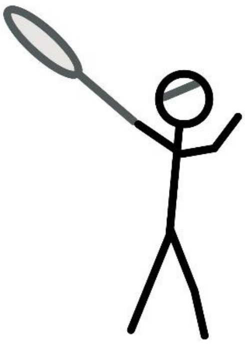


State S5:
One of the transition states of serving the ball.
The ball now apart from player.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 5
(*Condition*) **Next state:**
(*Unconditional*) S6



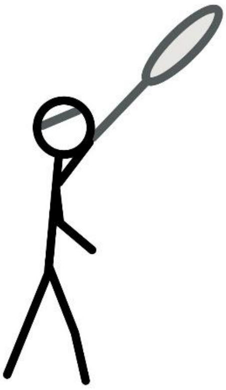
State S6:
One of the transition states of serving the ball.
The ball now apart from player.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 6
(*Condition*) **Next state:**
(*Unconditional*) S7

State S7:
One of the transition states of serving the ball.
The ball now apart from player.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 7
(*Condition*) **Next state:**
(*Unconditional*) W

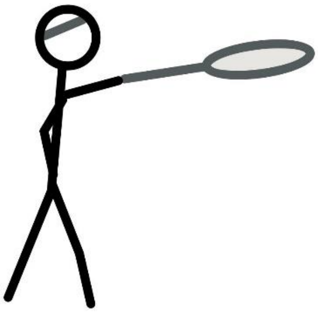


State W:
The waiting state for player to hit the ball.
The ball now apart from player.
Output:
ball_exist1 = 1'b0;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 8
(*Condition*) **Next state**:
(keycode *A*) ML1
(ketcode *D*) MR1
(keycode *S*) H1

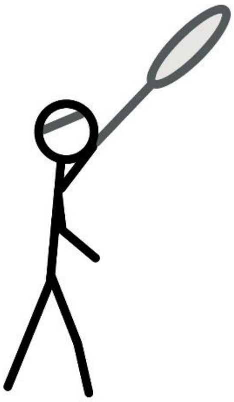
State H1:
One of the transition states of hitting the ball.
The ball now apart from player.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b1;
ball_shoot1 = 1'b0;
Corresponding diagram number: 9
(*Condition*) **Next state:**
(*Unconditional*) H2



State H2:
One of the transition states of hitting the ball.
The ball now apart from player.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b1;
ball_shoot1 = 1'b0;
Corresponding diagram number: 10
(*Condition*) **Next state:**
(*Unconditional*) H3

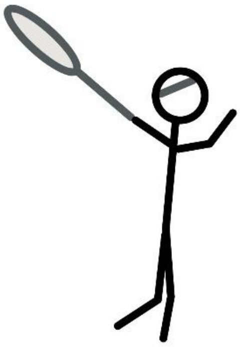


State H3:
One of the transition states of hitting the ball.
The ball now apart from player.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b1;
ball_shoot1 = 1'b0;
Corresponding diagram number: 11
(*Condition*) **Next state:**
(*Unconditional*) H4



State H4:
One of the transition states of hitting the ball.
The ball now apart from player.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 9
(*Condition*) **Next state:**
(*Unconditional*) H5

State H5:
One of the transition states of hitting the ball.
The ball now apart from player.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 8
(*Condition*) **Next state:**
(*Unconditional*) W



State MR1:

One of the transition states of moving when waiting for hitting the ball.

Output:

ball_exist1 = 1'b1;

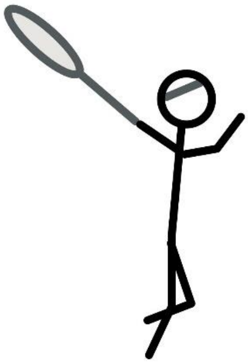
ball_hit1 = 1'b0;

ball_shoot1 = 1'b0;

Corresponding diagram number: 12

(*Condition*) **Next state:**

(*Unconditional*) MR2



State MR2:

One of the transition states of moving when waiting for hitting the ball.

Output:

ball_exist1 = 1'b1;

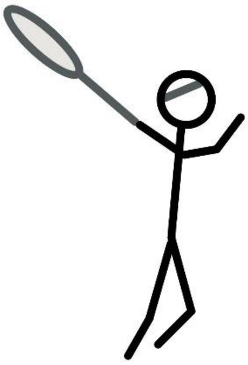
ball_hit1 = 1'b0;

ball_shoot1 = 1'b0;

Corresponding diagram number: 13

(*Condition*) **Next state:**

(*Unconditional*) MR3



State MR3:

One of the transition states of moving when waiting for hitting the ball.

Output:

ball_exist1 = 1'b1;

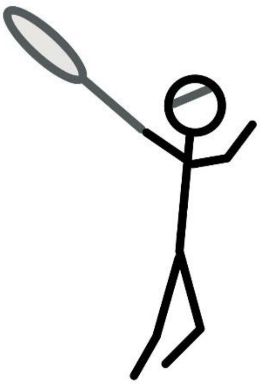
ball_hit1 = 1'b0;

ball_shoot1 = 1'b0;

Corresponding diagram number: 14

(*Condition*) **Next state:**

(*Unconditional*) W



State ML1:

One of the transition states of moving when waiting for hitting the ball.

Output:

ball_exist1 = 1'b1;

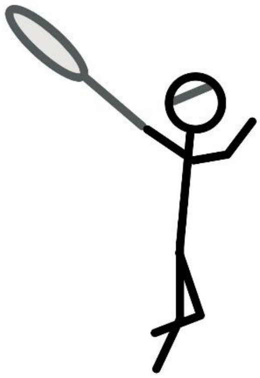
ball_hit1 = 1'b0;

ball_shoot1 = 1'b0;

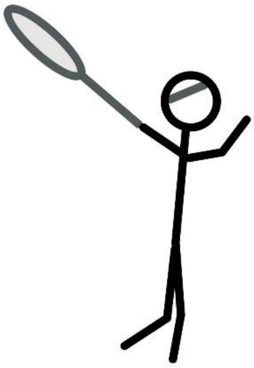
Corresponding diagram number: 14

(*Condition*) **Next state:**

(*Unconditional*) ML2



State ML2:
One of the transition states of moving when waiting for hitting the ball.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 13
(*Condition*) **Next state:**
(*Unconditional*) ML3



State ML3:
One of the transition states of moving when waiting for hitting the ball.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 12
(*Condition*) **Next state:**
(*Unconditional*) W

State SR1:
One of the transition states of moving when serving the ball.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 15
(*Condition*) **Next state:**
(*Unconditional*) SR2

State SR2:
One of the transition states of moving when serving the ball.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 17
(*Condition*) **Next state:**
(*Unconditional*) SR3

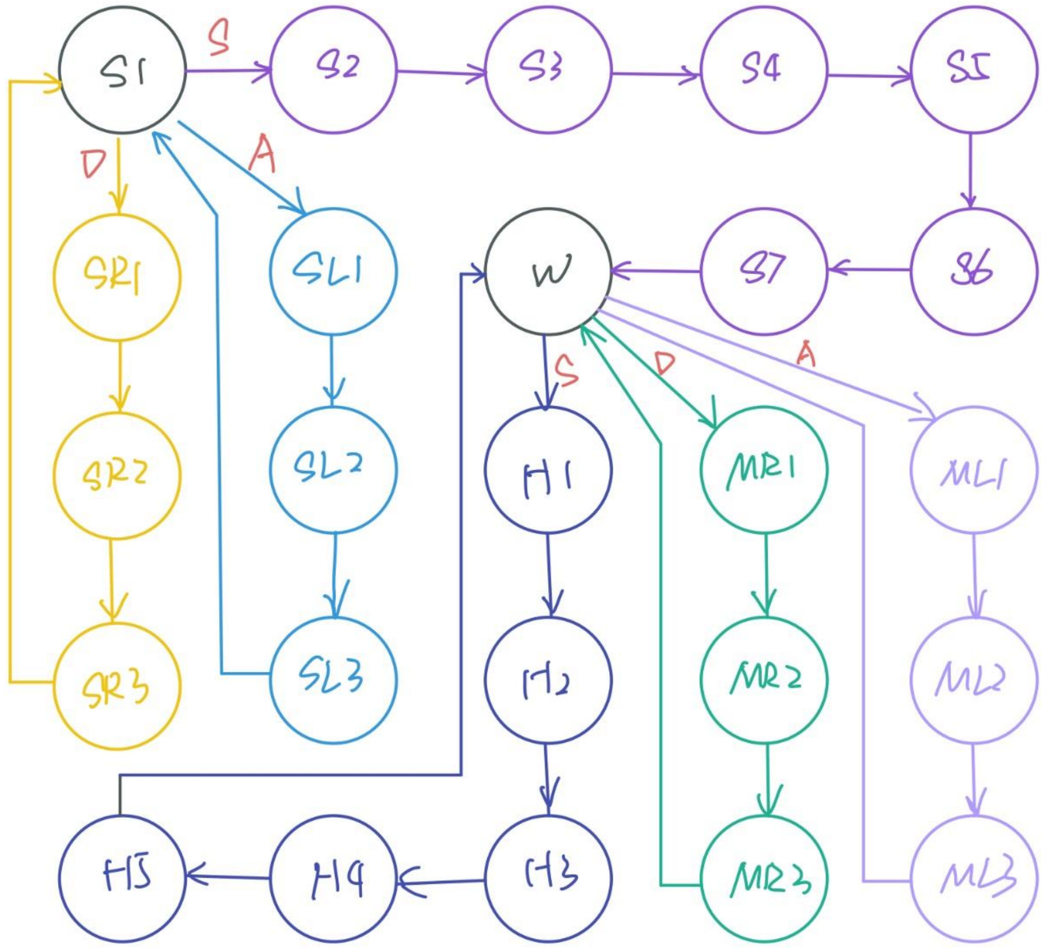
State SR3:
One of the transition states of moving when serving the ball.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 16
(*Condition*) **Next state:**
(*Unconditional*) S1

State SL1:
One of the transition states of moving when serving the ball.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 16
(*Condition*) **Next state:**
(*Unconditional*) SL2

State SL2:
One of the transition states of moving when serving the ball.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 17
(*Condition*) **Next state:**
(*Unconditional*) SL3

State SL3:
One of the transition states of moving when serving the ball.
Output:
ball_exist1 = 1'b1;
ball_hit1 = 1'b0;
ball_shoot1 = 1'b0;
Corresponding diagram number: 15
(*Condition*) **Next state:**
(*Unconditional*) S1

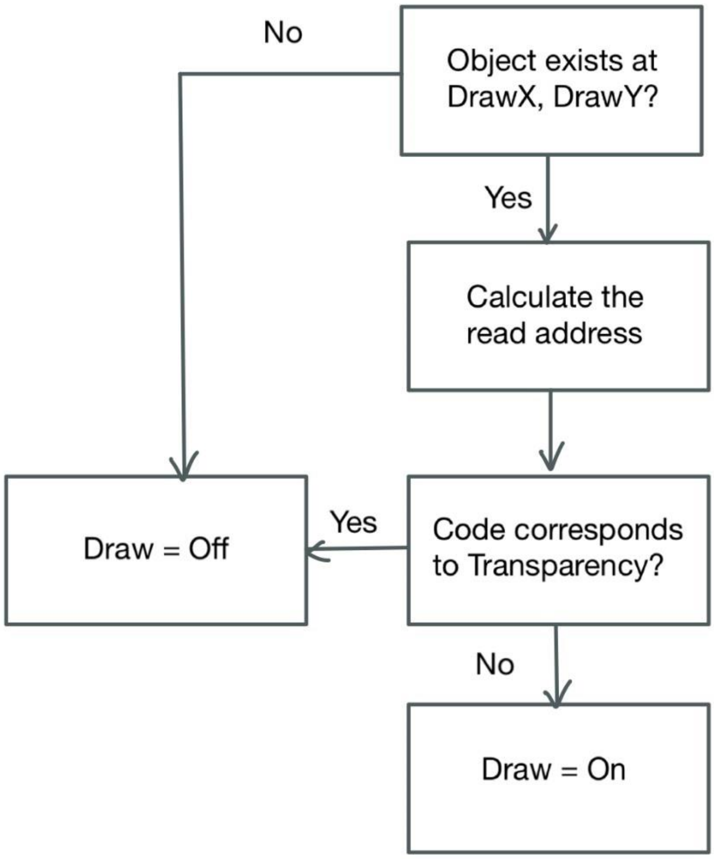
State machine:



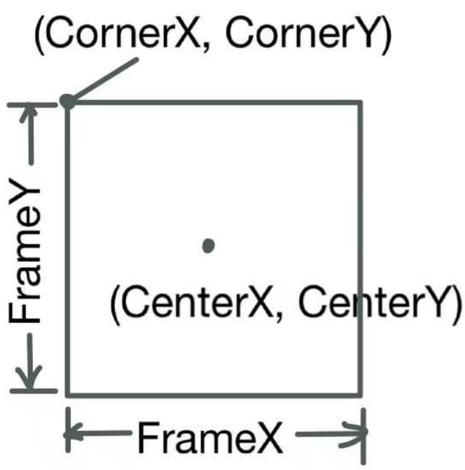
Although there are 25 states here, we actually have 17 states pictures since some states share one picture. Move left and move right can use the same three states' picture but with opposite directions. I first calculated the states than exported the corresponding part of 17 states picture, this can help me save much more memory than if I use 25 different pictures to read. The following algorithm will give the relative read address for our choice of state.

3.2 Sprite algorithm

Flow diagram:



Center, Corner and Frame are defined as:



Those data are given by the FSM output to index the coordinates:

```
assign CenterX1 = figureStateCenterX[state1];   // Center position
assign CenterY1 = figureStateCenterY[state1];
assign CornerX1 = figureStateCornerX[state1];   // left up corner
assign CornerY1 = figureStateCornerY[state1];
assign FrameX1 = figureStateFrameX[state1];  // the frame size
assign FrameY1 = figureStateFrameY[state1];

assign CenterX2 = figureStateCenterX[state2];   // Center position
assign CenterY2 = figureStateCenterY[state2];
assign CornerX2 = figureStateCornerX[state2];   // left up corner
assign CornerY2 = figureStateCornerY[state2];
assign FrameX2 = figureStateFrameX[state2];  // the frame size
assign FrameY2 = figureStateFrameY[state2];
```

And those coordinates are listed below, all were recorded by pixel tool:

```
assign figureStateCenterX = '{10'd69,10'd155 ,10'd219 ,10'd278 ,10'd343 ,
                              10'd21  ,10'd123 ,10'd222 ,10'd270 ,10'd320 ,
                              10'd412 ,10'd69  ,10'd162 ,10'd259 ,10'd463 ,
                              10'd352 ,10'd465};//TODO

assign figureStateCenterY = '{10'd22,10'd22  ,10'd21  ,10'd21  ,10'd21  ,
                              10'd198 ,10'd198 ,10'd198 ,10'd198 ,10'd198 ,
                              10'd198 ,10'd332 ,10'd332 ,10'd333 ,10'd72  ,
                              10'd334 ,10'd334};//TODO

assign figureStateCornerX = '{10'd0 ,10'd106 ,10'd194 ,10'd257 ,10'd321 ,
                              10'd0   ,10'd103 ,10'd151 ,10'd248 ,10'd299 ,
                              10'd387 ,10'd0   ,10'd92  ,10'd188 ,10'd394 ,
                              10'd286 ,10'd396};//TODO

assign figureStateCornerY = '{10'd0 ,10'd0   ,10'd0   ,10'd0   ,10'd0   ,
                              10'd160 ,10'd107 ,10'd145 ,10'd122 ,10'd142 ,
                              10'd172 ,10'd283 ,10'd283 ,10'd283 ,10'd49  ,
                              10'd306 ,10'd311};//TODO

assign figureStateFrameX = '{10'd106,10'd88  ,10'd63  ,10'd64  ,10'd112 ,
                             10'd103  ,10'd48  ,10'd97  ,10'd51  ,10'd85  ,
                             10'd115  ,10'd92  ,10'd96  ,10'd98  ,10'd108 ,
                             10'd101  ,10'd106};//TODO

assign figureStateFrameY = '{10'd108,10'd107 ,10'd114 ,10'd114 ,10'd106 ,
                             10'd123  ,10'd176 ,10'd138 ,10'd161 ,10'd141 ,
                             10'd111  ,10'd137 ,10'd137 ,10'd138 ,10'd107 ,
                             10'd116  ,10'd110};//TODO
```

Equation to calculate whether object exist at DrawX, DrawY:

$$DrawX >= figure1\_x - (CenterX1-CornerX1)$$

$$DrawX <= figure1\_x + (CornerX1+FrameX1-CenterX1)$$

$$DrawY >= figure1\_y - (CenterY1-CornerY1)$$

$$DrawY <= figure1\_y + (CornerY1+FrameY1-CenterY1)$$

Equation to calculate the read address:

$$read\_address1=CenterX1-(figure1\_x-DrawX)+(CenterY1-(figure1\_y-DrawY))*total\_length$$

For figure2, since its only the inverse of figure1, we still read the same picture, but we need to change a little about the equation:

$$DrawX >= figure2\_x - (CornerX2+FrameX2-CenterX2)$$

$$DrawX <= figure2\_x + (CenterX2-CornerX2)$$

Equation to calculate the read address of figure2 also need some modification:

$$read\_address2=CenterX2+(figure2\_x-DrawX)+(CenterY2-(figure2\_y-DrawY))* total\_length$$

3.3 Ball motion

For ball's motion, I considered the collision condition in a very simple way.

When the control signal of FSM gives that the ball is either in player1 hand or player 2 hand, that is, when ball_exist1==1 &ball_exist2==1, the ball will appear at the position where the player holds the ball. We calculate the relative position of ball from figure1:

Ball_X_Pos <= figure1_x + 10'd33;

Ball_Y_Pos <= figure1_y + 10'd51;

Then update the ball's position and motion.

Now, we divide the collision condition in these ways:

a) When ball is flying: keep motion in x direction, have gravity in y direction

Condition: ball_exist1==1 &ball_exist2==1

$v_x' = v_x$

$v_y' = v_y + gt$

b) When ball collides the wall: opposite direction of original x direction, have gravity in y direction

Condition: X_Pos reach X_Min or X_Max

$v_x' = -v_x$

$v_y' = v_y + gt$

c) When ball hits the ground: game over, no velocity

Condition: Y_Pos reach Y_Min

$v_x' = 0$

$v_y' = 0$

d) When ball hits the bat: opposite direction of original x direction, a initial velocity in y direction

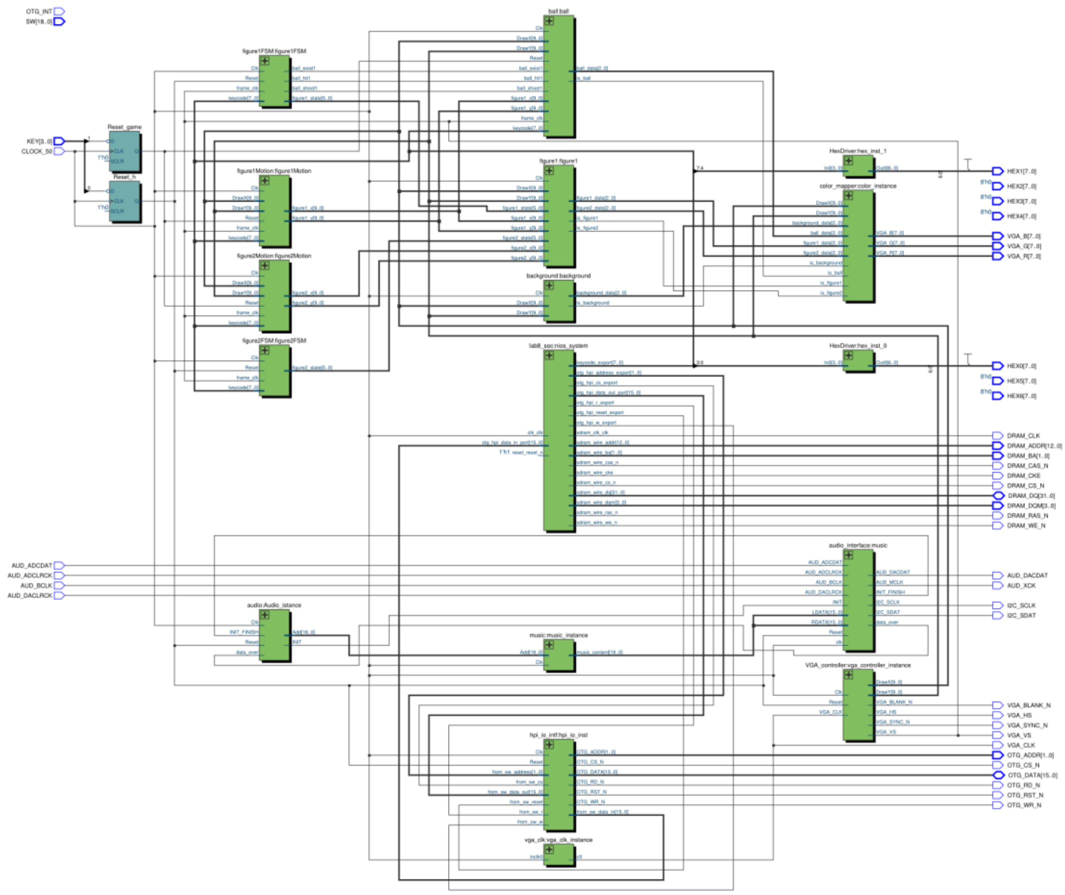
Condition: (X_Pos, Y_Pos) in range of the bat swing area

Ball_hit1=1 or Ball_hit2 = 1

$v_x' = -v_x$

$v_y' = v_y + v_i$

4. Block Diagram



5. SV Code

```
module lab8( input              CLOCK_50,
             input        [3:0]  KEY,          //bit 0 is set up as Reset
             input        [18:0] SW,           // only for test
             output logic [7:0]  HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6,
             //output logic [7:0]  LEDG,
             // VGA Interface
             output logic [7:0]  VGA_R,        //VGA Red
                                 VGA_G,        //VGA Green
                                 VGA_B,        //VGA Blue
             output logic        VGA_CLK,      //VGA Clock
                                 VGA_SYNC_N,   //VGA Sync signal
                                 VGA_BLANK_N,  //VGA Blank signal
                                 VGA_VS,       //VGA virtical sync signal
                                 VGA_HS,       //VGA horizontal sync signal
             // CY7C67200 Interface
             inout  wire  [15:0] OTG_DATA,     //CY7C67200 Data bus 16 Bits
             output logic [1:0]  OTG_ADDR,     //CY7C67200 Address 2 Bits
             output logic        OTG_CS_N,     //CY7C67200 Chip Select
                                 OTG_RD_N,     //CY7C67200 Write
                                 OTG_WR_N,     //CY7C67200 Read
                                 OTG_RST_N,    //CY7C67200 Reset
             input               OTG_INT,      //CY7C67200 Interrupt
             // SDRAM Interface for Nios II Software
             output logic [12:0] DRAM_ADDR,    //SDRAM Address 13 Bits
             inout  wire  [31:0] DRAM_DQ,      //SDRAM Data 32 Bits
             output logic [1:0]  DRAM_BA,      //SDRAM Bank Address 2 Bits
             output logic [3:0]  DRAM_DQM,     //SDRAM Data Mast 4 Bits
             output logic        DRAM_RAS_N,   //SDRAM Row Address Strobe
                                 DRAM_CAS_N,   //SDRAM Column Address Strobe
                                 DRAM_CKE,     //SDRAM Clock Enable
                                 DRAM_WE_N,    //SDRAM Write Enable
                                 DRAM_CS_N,    //SDRAM Chip Select
                                 DRAM_CLK,      //SDRAM Clock
            input AUD_ADCDAT,
            input AUD_DACLRCK,
            input AUD_ADCLRCK,
            input AUD_BCLK,
            output logic AUD_DACDAT,
            output logic AUD_XCK,
            output logic I2C_SCLK,
            output logic I2C_SDAT
                    );
```

Module: lab8.sv

Input & Output: Shown in diagram

Description: This module is the toplevel of our final project. It assigns all the inputs and outputs to the right place.

Purpose: This module is used to make FPGA and our code in Eclipse interact with each other.

```
/*
 * ECE385-HelperTools/PNG-To-Txt
 * Author: Rishi Thakkar
 *
 */
module background (
   input Clk,
   //input logic background_exist,
   input logic [9:0] DrawX, DrawY,
   output logic [2:0] background_data,
   output logic is_background
);
   // screen size
   parameter [9:0] SCREEN_WIDTH =  10'd480;
   parameter [9:0] SCREEN_LENGTH = 10'd640;
   parameter [9:0] RESHAPE_LENGTH = 10'd320;
   //--------------------load memory-----------------//
   logic [18:0] read_address;
   assign read_address = DrawX/2 + DrawY/2*RESHAPE_LENGTH;
   background_RAM background_RAM(.*);
   always_comb begin
      is_background = 1'b1;
   end
endmodule

module  background_RAM
(
      input [18:0] read_address,
      input Clk,

      output logic [2:0] background_data
);

// mem has width of 3 bits and a total of 307200(640x480) addresses
//logic [2:0] mem [0:307199]; // 640x480 = 307200
logic [2:0] mem [0:76799];// 320x240 = 76800
initial
begin
    $readmemh("background.txt", mem);// read into mem
end
always_ff @ (posedge Clk) begin
   background_data<= mem[read_address];

end

endmodule
```

Module: background.sv

Input & Output: Shown in diagram

Description: This module is used to store the background picture data to on-chip memory then read those data to background_data for Color_Mapper to assign color data.

Purpose: This module is used to place our background at the right place of the screen.

```
module figure1(
   input Clk,                 // 50 MHz clock
   input logic [9:0] DrawX, DrawY,
   input logic [9:0] figure1_x, figure1_y,    // from figure1Motion output
   input logic [9:0] figure2_x, figure2_y,    // from figure2Motion output
   input logic [5:0] figure1_state, figure2_state, // from figure1FSM and figure2FSM
   output logic [2:0] figure1_data, figure2_data,
   output logic  is_figure1, is_figure2       // whether current pixel belongs to figure1/2 or background
);

   // screen size
   parameter [9:0] SCREEN_WIDTH =  10'd480;  // Y
   parameter [9:0] SCREEN_LENGTH = 10'd640;  // X
   parameter [9:0] FIGURE1_WIDTH =  10'd424; // Y
   parameter [9:0] FIGURE1_LENGTH =  10'd502;// X

   //--------------------load memory-----------------//
   logic [18:0] read_address1, read_address2;
   logic [9:0] CenterX1, CenterX2;     // figure1 center in the collection graph
   logic [9:0] CenterY1, CenterY2;
   logic [9:0] CornerX1, CornerX2;     // the frame left up corner
   logic [9:0] CornerY1, CornerY2;
   logic [9:0] FrameX1, FrameX2;    // the frame size
   logic [9:0] FrameY1, FrameY2;
   logic [5:0] state1, state2;
   figure1_RAM figure1_RAM(.*);
// figure2_RAM figure2_RAM(.*);

     // use state as index to find the center of saber
   logic [9:0] figureStateCenterX[0:16];
   logic [9:0] figureStateCenterY[0:16];
   logic [9:0] figureStateCornerX[0:16];
   logic [9:0] figureStateCornerY[0:16];
   logic [9:0] figureStateFrameX[0:16];
   logic [9:0] figureStateFrameY[0:16];
```

```
assign figureStateCenterX = '{10'd69,10'd155 ,10'd219 ,10'd278 ,10'd343 ,
                              10'd21  ,10'd123 ,10'd222 ,10'd270 ,10'd320 ,
                              10'd412 ,10'd69  ,10'd162 ,10'd259 ,10'd463 ,
                              10'd352 ,10'd465};//TODO

assign figureStateCenterY = '{10'd22,10'd22  ,10'd21  ,10'd21  ,10'd21  ,
                              10'd198 ,10'd198 ,10'd198 ,10'd198 ,10'd198 ,
                              10'd198 ,10'd332 ,10'd332 ,10'd333 ,10'd72  ,
                              10'd334 ,10'd334};//TODO

assign figureStateCornerX = '{10'd0  ,10'd106 ,10'd194 ,10'd257 ,10'd321 ,
                              10'd0   ,10'd103 ,10'd151 ,10'd248 ,10'd299 ,
                              10'd387 ,10'd0   ,10'd92  ,10'd188 ,10'd394 ,
                              10'd286 ,10'd396};//TODO

assign figureStateCornerY = '{10'd0  ,10'd0   ,10'd0   ,10'd0   ,10'd0   ,
                              10'd160 ,10'd107 ,10'd145 ,10'd122 ,10'd142 ,
                              10'd172 ,10'd283 ,10'd283 ,10'd283 ,10'd49  ,
                              10'd306 ,10'd311};//TODO

assign figureStateFrameX = '{10'd106,10'd88  ,10'd63  ,10'd64  ,10'd112 ,
                             10'd103 ,10'd48  ,10'd97  ,10'd51  ,10'd85  ,
                             10'd115 ,10'd92  ,10'd96  ,10'd98  ,10'd108 ,
                             10'd101 ,10'd106};//TODO

assign figureStateFrameY = '{10'd108,10'd107 ,10'd114 ,10'd114 ,10'd106 ,
                             10'd123 ,10'd176 ,10'd138 ,10'd161 ,10'd141 ,
                             10'd111 ,10'd137 ,10'd137 ,10'd138 ,10'd107 ,
                             10'd116 ,10'd110};//TODO

assign CenterX1 = figureStateCenterX[state1];    // Center position
assign CenterY1 = figureStateCenterY[state1];
assign CornerX1 = figureStateCornerX[state1];    // left up corner
assign CornerY1 = figureStateCornerY[state1];
assign FrameX1 = figureStateFrameX[state1];   // the frame size
assign FrameY1 = figureStateFrameY[state1];

assign CenterX2 = figureStateCenterX[state2];    // Center position
assign CenterY2 = figureStateCenterY[state2];
assign CornerX2 = figureStateCornerX[state2];    // left up corner
assign CornerY2 = figureStateCornerY[state2];
assign FrameX2 = figureStateFrameX[state2];   // the frame size
assign FrameY2 = figureStateFrameY[state2];
```

```
   // Compute whether the pixel corresponds to figure1/2 or background
    /* Since the multiplicants are required to be signed, we have to first cast them
       from logic to int (signed by default) before they are multiplied. */
   always_comb begin
      state1 = figure1_state;
      state2 = figure2_state;
      read_address1 = 19'b0;
      is_figure1 = 1'b0;
      read_address2 = 19'b0;
      is_figure2 = 1'b0;
      if ((DrawX >= figure1_x - (CenterX1-CornerX1) || figure1_x < (CenterX1-CornerX1)) && DrawX <= figure1_x + (CornerX1+FrameX1-
          (DrawY >= figure1_y - (CenterY1-CornerY1) || figure1_y < (CenterY1-CornerY1)) && DrawY <= figure1_y + (CornerY1+FrameY1-
            is_figure1 = 1'b1;
            read_address1 = CenterX1-(figure1_x - DrawX) + (CenterY1-(figure1_y - DrawY))*FIGURE1_LENGTH;
            //              x position in figure1           y position in figure1
       end
      if ((DrawX >= figure2_x - (CornerX2+FrameX2-CenterX2) || figure2_x < (CenterX2-CornerX2)) && DrawX <= figure2_x + (CenterX2-
          (DrawY >= figure2_y - (CenterY2-CornerY2) || figure2_y < (CenterY2-CornerY2)) && DrawY <= figure2_y + (CornerY2+FrameY2-
            is_figure2 = 1'b1;
            read_address2 = CenterX2+(figure2_x - DrawX) + (CenterY2-(figure2_y - DrawY))*FIGURE1_LENGTH;
            //              x position in figure2           y position in figure2
       end
     end
endmodule


module figure1_RAM(  |
   input [18:0] read_address1, read_address2,//write_address,
   input Clk,

   output logic [2:0] figure1_data, figure2_data
);

// mem has width of n bits and a total of xxx addresses
logic [2:0] mem [0:212847]; // 424x502 = 212848  212847
initial
begin
    $readmemh("figure1.txt", mem);// read into mem
end


always_ff @ (posedge Clk) begin
   figure1_data<= mem[read_address1];// get data accroding to read_address computed above
   figure2_data<= mem[read_address2];
   end

endmodule
```

Module: figure1.sv

Input & Output: Shown in diagram

Description: This module is used to store the figure1 picture data to on-chip memory then by the specific read address according to the data from FSM to read those data to figure1_data and figure2_data for Color_Mapper to assign color data.

Purpose: This module is used to place certain state figure1 and figure2 at the right place of the screen.

```
// color_mapper: Decide which color to be output to VGA for each pixel.
module  color_mapper (  input logic Clk,
                        input logic [2:0]background_data,
                        input logic [2:0]figure1_data, figure2_data, ball_data, //basket_data,
                        input        is_figure1, is_figure2, //is_basket,    // whether current pixel belongs to figure
                        input        is_background,
                        input        is_ball,
                        input         [9:0] DrawX, DrawY,       // Current pixel coordinates
                        output logic [7:0] VGA_R, VGA_G, VGA_B // VGA RGB output
                     );

    logic [7:0] Red, Green, Blue;
    logic [23:0] background_color,figure1_color, figure2_color, ball_color;//basket_color,
    logic [23:0] color;

    //----------------color palette----------------//
    logic [23:0] background_palette [0:7];
    logic [23:0] figure1_palette[0:7];

    assign background_palette = '{24'hffffff, 24'h78837b, 24'h474d4b, 24'h454b47,
                                  24'h986120, 24'he6aa54, 24'h297ba2, 24'h00537e};
                                  // '0xffffff', '0x78837b', '0x474d4b', '0x454b47', '0x986120', '0xe6aa54', '0x297ba2',


    assign figure1_palette = '{24'h000000, 24'hffffff, 24'hdcddd8, 24'h545e5f,
                               24'h5b6161, 24'heae9e5, 24'h973b2e, 24'ha2272c};
                                //  black       white       grey                                              red
                                // '0x000000', '0xffffff', '0xdcddd8', '0x545e5f', '0x5b6161', '0xeae9e5' , '0x973b2e',

    assign background_color = background_palette[background_data];
    assign figure1_color = figure1_palette[figure1_data];
    assign figure2_color = figure1_palette[figure2_data];
    assign ball_color = figure1_palette[ball_data];

    // Output colors to VGA
    assign VGA_R = color[23:16];
    assign VGA_G = color[15:8];
    assign VGA_B = color[7:0];

    // Assign color based on is_ball signal
    always_comb
    begin
        if (is_figure1 == 1'b1 && figure1_color != 24'hFFFFFF )
        begin
            color = figure1_color;
        end
        else if (is_figure2 == 1'b1 && figure2_color != 24'hFFFFFF )
        begin
            color = figure2_color;
        end
        else if ( is_ball == 1'b1 && ball_color != 24'hFFFFFF)
        begin
            color = ball_color;
        end
        else if ( is_background == 1'b1)
        begin
            color = background_color;
        end
        else
        begin
            color = 24'h00FF00;
        end
    end
endmodule
```

Module: color_mapper.sv

Input & Output: Shown in diagram

Description: This module decides which color to be output to VGA for each pixel and whether the pixel belongs to figure1 or figure2 or ball or background and uses RGB color selection.

Purpose: This module is used to draw the figure1, figure2, ball, background, and implement RGB colors on screen.

```
module  figure1Motion (
                input           Clk,                // 50 MHz clock
                                Reset,              // Active-high reset signal
                                frame_clk,          // The clock indicating a new frame (~60Hz)
                input [9:0]     DrawX, DrawY,       // Current pixel coordinates
                input [7:0]     keycode,            // keyboard press
                output logic [9:0] figure1_x,
                output logic [9:0] figure1_y
               );


    parameter [9:0] figure1_X_Center = 10'd160;  // Start X position
    parameter [9:0] figure1_Y_Center = 10'd360;  // Start Y position
    // motion range
    parameter [9:0] figure1_X_Min = 10'd40;       // Leftmost point on the X axis
    parameter [9:0] figure1_X_Max = 10'd300;     // Rightmost point on the X axis
    parameter [9:0] figure1_Y_Min = 10'd0;       // Topmost point on the Y axis
    parameter [9:0] figure1_Y_Max = 10'd440;     // Bottommost point on the Y axis
    // motion step
    parameter [9:0] figure1_X_Step = 10'd1;      // Step size on the X axis
    parameter [9:0] figure1_Y_Step = 10'd1;      // Step size on the Y axis

     logic [9:0] figure1_X_Pos, figure1_X_Motion, figure1_Y_Pos, figure1_Y_Motion;
     logic [9:0] figure1_X_Pos_in, figure1_Y_Pos_in;


//////// Do not modify the always_ff blocks. ////////
     // Detect rising edge of frame_clk
     logic frame_clk_delayed, frame_clk_rising_edge;
     always_ff @ (posedge Clk) begin
         frame_clk_delayed <= frame_clk;
         frame_clk_rising_edge <= (frame_clk == 1'b1) && (frame_clk_delayed == 1'b0);
     end
     // Update registers
     always_ff @ (posedge Clk)
     begin
         if (Reset) // back to original place and don't move
         begin
             figure1_X_Pos <= figure1_X_Center;
             figure1_Y_Pos <= figure1_Y_Center;
         end
         else
         begin
             figure1_X_Pos <= figure1_X_Pos_in;
             figure1_Y_Pos <= figure1_Y_Pos_in;
         end
     end
     always_comb
     begin
         // By default, keep motion and position unchanged
         figure1_X_Pos_in = figure1_X_Pos;
         figure1_Y_Pos_in = figure1_Y_Pos;
         figure1_x = figure1_X_Pos;
         figure1_y = figure1_Y_Pos;
         figure1_X_Motion = 10'd0;
         figure1_Y_Motion = 10'd0;

         // Update position and motion only at rising edge of frame clockv
         if (frame_clk_rising_edge)
         begin
           case(keycode)
             8'h04: // A: Go left
                begin
                   figure1_X_Motion = (~(figure1_X_Step) + 1'b1);
                   figure1_Y_Motion = 10'h000;
                end
             8'h07: // D: Go right
                begin
                   figure1_X_Motion = figure1_X_Step;
                   figure1_Y_Motion = 10'h000;
                end
             8'h1a: // W: Jump not use now
                begin
                   figure1_Y_Motion = 10'h000;//(~(figure1_Y_Step) + 1'b1);
                   figure1_X_Motion = 10'h000;
                end
             8'h16: // S: Bat not use now
                begin
                   figure1_Y_Motion = 10'h000;//figure1_Y_Step;
                   figure1_X_Motion = 10'h000;
                end
             default:
                begin
                end
             endcase

             // Update the figure1's position with its motion
             figure1_X_Pos_in = figure1_X_Pos + figure1_X_Motion;
             figure1_Y_Pos_in = figure1_Y_Pos + figure1_Y_Motion;
          end
     end
endmodule
```

Module: figure1Motion.sv

Input & Output: Shown in diagram

Description: This module updates the position and motion of figure1 only at the rising edge of frame clock and unlike what we did in lab 8, if no keys are pressed it will not change the motion.

Purpose: This module is used to calculate the positions and reacts to keypresses which are from the user via the keyboard.

Module: figure2Motion.sv (almost same as figure1Motion.sv)

Input & Output: Shown in diagram

Description: This module updates the position and motion of figure2 only at the rising edge of frame clock and unlike what we did in lab 8, if no keys are pressed it will not change the motion.

Purpose: This module is used to calculate the positions and reacts to keypresses which are from the user via the keyboard.

```
module figure1FSM(input       Clk,                // 50 MHz clock
                               Reset,              // Active-high reset signal
                               frame_clk,          // The clock indicating a new frame (~60Hz)
                input  [7:0]   keycode,
                output  [5:0] figure1_state,
                output logic ball_exist1,
                output logic ball_shoot1,
                output logic ball_hit1);

    //logic frame_clk_delayed, frame_clk_rising_edge;
    logic [5:0] counter,inner_counter;
    logic set_zero;

    enum logic [4:0] { S1, S2, S3, S4, S5, S6, S7,
                       W, H1, H2, H3, H4, H5,
                       SR1, SR2, SR3, SL1, SL2, SL3,
                       MR1, MR2, MR3, ML1, ML2, ML3} State, Next_state;   // Internal state logic

   always_ff @ (posedge frame_clk)
   begin
      counter<=inner_counter;
      if(set_zero)
      counter<=6'b0;
   end

   always_comb
   begin
      inner_counter=counter+1;
   end

   always_ff @ (posedge Clk)
   begin
      if (Reset)
         State <= S1;
      else
         State <= Next_state;
   end
```

```
always_ff @ (posedge frame_clk)
begin
   set_zero=1'b0;
   // Default next state is staying at current state
   Next_state = State;
   unique case (State)
      S1 :
         case(keycode)
            8'h04: // A: Go left
               Next_state = SL1;
            8'h07: // D: Go right
               Next_state = SR1;
            8'h16: // S: Hit
               Next_state = S2;
            default :
               Next_state = S1;
         endcase
      S2 :
         Next_state = S3;
      S3 :
         Next_state = S4;
      S4 :
         Next_state = S5;
      S5 :
         Next_state = S6;
      S6 :
         Next_state = S7;
      S7 :
         Next_state = W;
      W :
         case(keycode)
            8'h04: // A: Go left
               Next_state = ML1;
            8'h07: // D: Go right
               Next_state = MR1;
            8'h16: // S: Hit
               Next_state = H1;
            default :
               Next_state = W;
         endcase
```

```
// Assign control signals based on current state
case (State)
   S1 :
      begin
      ball_exist1 = 1'b0;
      ball_hit1 = 1'b0;
      ball_shoot1 = 1'b0;
      figure1_state = 10'd0;
      end
   S2 :
      begin
      ball_exist1 = 1'b0;
      ball_hit1 = 1'b0;
      ball_shoot1 = 1'b0;
      figure1_state = 10'd1;
      end
   S3 :
      begin
      ball_exist1 = 1'b0;
      ball_hit1 = 1'b0;
      ball_shoot1 = 1'b0;
      figure1_state = 10'd2;
      end
   S4 :
      begin
      ball_exist1 = 1'b1;
      ball_hit1 = 1'b0;
      ball_shoot1 = 1'b1;
      figure1_state = 10'd3;
      end
   S5 :
      begin
      ball_exist1 = 1'b1;
      ball_hit1 = 1'b0;
      ball_shoot1 = 1'b0;
      figure1_state = 10'd4;
      end
```

Module: figure1FSM.sv

Input & Output: Shown in diagram

Description: This module defines our state machine of figure1, which determines the next state and some output variable for the current state in order to control figure1 motion.

Purpose: This module regulates the states of our figure1 so that it can continuously show its movement when swing and run. It also assigns proper values to some control signals to make the system function properly.

Module: figure2FSM.sv (almost same as figure1FSM.sv)

Input & Output: Shown in diagram

Description: This module defines our state machine of figure2, which determines the next state and some output variable for the current state in order to control figure2 motion.

Purpose: This module regulates the states of our figure2 so that it can continuously show its movement when swing and run. It also assigns proper values to some control signals to make the system function properly.

```
module hpi_io_intf( input        Clk, Reset,
                    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, from_sw_reset, // Active low
                    inout [15:0] OTG_DATA,
                    output[1:0]  OTG_ADDR,
                    output       OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N // Active low
                   );

// Buffer (register) for from_sw_data_out because inout bus should be driven
//   by a register, not combinational logic.
logic [15:0] from_sw_data_out_buffer;

// TODO: Fill in the blanks below.
always_ff @ (posedge Clk)
begin
    if(Reset)
    begin
        from_sw_data_out_buffer <= 16'h0000;
        OTG_ADDR                <= 2'b00;
        OTG_RD_N                <= 1'b1;
        OTG_WR_N                <= 1'b1;
        OTG_CS_N                <= 1'b0;
        OTG_RST_N               <= 1'b0;
        from_sw_data_in         <= 16'h0000;
    end
    else
    begin
        from_sw_data_out_buffer <= from_sw_data_out;
        OTG_ADDR                <= from_sw_address;
        OTG_RD_N                <= from_sw_r;
        OTG_WR_N                <= from_sw_w;
        OTG_CS_N                <= from_sw_cs;
        OTG_RST_N               <= 1'b1;
        from_sw_data_in         <= OTG_DATA;
    end
end

// OTG_DATA should be high Z (tristated) when NIOS is not writing to OTG_DATA inout bus.
// Look at tristate.sv in lab 6 for an example.
assign OTG_DATA = ~from_sw_w ? from_sw_data_out_buffer : {16'bZ};

endmodule
```

Module: hpi_io_intf

Input & Output: Shown in diagram

Description: This module is the interface between NIOS II and EZ-OTG chip, a hardware tri-state buffer using buffer (register) for from_sw_data_out.

Purpose: This module is used to send read, write, cs, reset, data and address signals to the EZ-OTG chip, and OTG_DATA should be high Z (tristated) when NIOS is not writing to OTG_DATA inout bus.

```
module  VGA_controller (input              Clk,         // 50 MHz clock
                                           Reset,       // Active-high reset signal
                        output logic       VGA_HS,      // Horizontal sync pulse.  Active low
                                           VGA_VS,      // Vertical sync pulse.  Active low
                        input              VGA_CLK,     // 25 MHz VGA clock input
                        output logic       VGA_BLANK_N, // Blanking interval indicator.  Active low.
                                           VGA_SYNC_N,  // Composite Sync signal.  Active low.  We don't use it in this lab,
                                                        // but the video DAC on the DE2 board requires an input for it.
                        output logic [9:0] DrawX,       // horizontal coordinate
                                           DrawY        // vertical coordinate
                        );

    // 800 pixels per line (including front/back porch)
    // 525 lines per frame (including front/back porch)
    parameter [9:0] H_TOTAL = 10'd800;
    parameter [9:0] V_TOTAL = 10'd525;

    logic VGA_HS_in, VGA_VS_in, VGA_BLANK_N_in;
    logic [9:0] h_counter, v_counter;
    logic [9:0] h_counter_in, v_counter_in;

    assign VGA_SYNC_N = 1'b0;
    assign DrawX = h_counter;
    assign DrawY = v_counter;

    // VGA control signals.
    // VGA_CLK is generated by PLL, so you will have to manually generate it in simulation.
    always_ff @ (posedge VGA_CLK)
    begin
        if (Reset)
        begin
            VGA_HS <= 1'b0;
            VGA_VS <= 1'b0;
            VGA_BLANK_N <= 1'b0;
            h_counter <= 10'd0;
            v_counter <= 10'd0;
        end
        else
        begin
            VGA_HS <= VGA_HS_in;
            VGA_VS <= VGA_VS_in;
            VGA_BLANK_N <= VGA_BLANK_N_in;
            h_counter <= h_counter_in;
            v_counter <= v_counter_in;
```

Module: VGA_controller

Input & Output: Shown in diagram

Description: This module handles the synchronization of signals where VS implies vertical sync and HS implies horizontal sync of the VGA signal we are outputting in addition to "drawing" pixels

Purpose: This module is used to display the ball bouncing on the screen, as an output from the FPGA

Platform Designer Modules

⊟ clk_0	Clock Source
clk_in	Clock Input
clk_in_reset	Reset Input
clk	Clock Output
clk_reset	Reset Output

This is the clock module which simply the 50Mhz generated by the FPGA. The clk goes from here to all the other clocks inputs

⊟ onchip_memory2_0	On-Chip Memory (RAM or ROM) I...
clk1	Clock Input
s1	Avalon Memory Mapped Slave
reset1	Reset Input

This is our on-chip memory, which is often smaller than SRAM in size but faster and actually on the chip. The data width is 32 bits and the total memory size is 16 bytes

⊟ **sdram**	SDRAM Controller Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
wire	Conduit

This is our SDRAM that we use to store the software program due to the limited on-chip memory. We have to use an SDRAM controller to interface with the bus since we have row/column addressing and constantly needs to refresh in order to retain data.

⊟ **sdram_pll**	ALTPLL Intel FPGA IP
inclk_interface	Clock Input
inclk_interface_...	Reset Input
pll_slave	Avalon Memory Mapped Slave
c0	Clock Output
c1	Clock Output

This module generates the clock that goes into the SDRAM. The PLL allows us to account for delays, specifically 3ns in order to have the SDRAM wait for the outputs to stabilize.

⊟ **sysid_qsys_0**	System ID Peripheral Intel FP...
clk	Clock Input
reset	Reset Input
control_slave	Avalon Memory Mapped Slave

This is an ID checker which ensure the compatibility between hardware and software.

⊟ **nios2_gen2_0**	Nios II Processor
clk	Clock Input
reset	Reset Input
data_master	Avalon Memory Mapped Master
instruction_master	Avalon Memory Mapped Master
irq	Interrupt Receiver
debug_reset_request	Reset Output
debug_mem_slave	Avalon Memory Mapped Slave
custom_instructi...	Custom Instruction Master

This is an IP based 32-bit CPU which can programmed using a high-level language.

⊟ **keycode**	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a simple 8 bit-wide PIO block, which outputs the keycode from the IO_READ (keyboard).

⊟ **otg_hpi_address**	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a simple PIO block, which outputs the 2-bit value corresponding to the specific HPI register.

⊟ **otg_hpi_data**	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a simple 32 bit-wide PIO block, which is inout because data is both read from and written to here.

⊟ **otg_hpi_r**	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a simple PIO block, which is a 1bit output corresponding to a “read” enable signal

⊟ **otg_hpi_w**	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a simple PIO block, which is a 1bit output corresponding to a “write” enable signal

⊟ **otg_hpi_cs**	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a simple PIO block, which is a 1bit output corresponding to a “chip enable” signal

⊟ **otg_hpi_reset**	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a simple PIO block, which is a 1bit output corresponding to a “reset” signal

6. Design statistics and Discussions

LUT	**2756**
DSP	**0**
Memory (BRAM)	**1087488**
Flip-Flop	**2184**
Frequency	**127.81Mhz**
Static Power	**105.20mW**
Dynamic Power	**0.75mW**
Total Power	**180.57mW**

7. Conclusion

I encountered many flaws when debugging, except those basic syntax errors that raised by Quartus, something like forgetting to declare the new variable in scope, wrong assignment of FSM states……Those errors are fixed by compare my output with the correct output to see where is the error, I also use the RTL viewer to see the port connection to debug.

In demo, we failed to show our ball in screen, that might because the collision condition is not right, so that the ball just flashed at one second. We reviewed our code again and made some changes. Also, since the key is interrupted, we cannot move two players at the same time, which might cause the inequality. This problem can be solved but need a lot of modification.

In summary, we almost completed a game *Stickman Badminton.* Though it's not as our expected before, but the motion is really smooth. I learned a lot from this project, especially how to use FSM to give control signals that make every part work properly as a whole entity, also how to give correct inputs and outputs between different modules. Also, beside consolidating the knowledge I learned in the course, I learned how to use sprite and compress the picture.