기초디지털실험 기말 프로젝트: "Let's Make Antarctic Adventure"!

Dela Rosa Joshua Gabriel | 2019142208

INTRODUCTION AND HOW TO PLAY

In this project we make a simple FPGA based arcade game called "Antarctic Adventure" where a penguin runs for a finite distance while collecting coins and avoiding obstacles. The implementation uses the Programmable Logic (PL) part of the FPGA to handle the graphics and finite state machine of the game while the Processing System (PS) part handles the audio for the background music and interrupts. The project reuses multiple modules from the different experiments conducted throughout the semester as well as new modules and code for additional features. The graphics part heavily reuses Week 11 and 12 source files while the audio part heavily reuses Week 9 source files.

To play the game, simply launch the program on Vitis by first adding the "m" library just like in Week 9 experiment. The game will only start if sw[0] (right switch) is high. The game pauses when it is low. Use BTN3 to move to the left, BTN2 to jump and BTN1 to move to the right. To dodge the obstacle, the jump must occur where the barrier's lower edge is exactly on top of the shadow.

FINITE STATE MACHINE FOR GAME PROGRESSION

In order to make a game, we need a finite state machine that manages the game progression. Since the game consists of a penguin trying to collect coins and avoid obstacles until it runs for a certain distance, we need to able to do 2 things: 1) from a starting finite distance, reduce it as the game goes forward, 2) generate coins and barriers as the game progresses This can be done by intentionally choosing when the coins and barriers come out based on the current remaining distance. The chosen FSM for the game progression is shown in figure 1. We also add shortcuts to FINISH state if the number of remaining lives become 0.

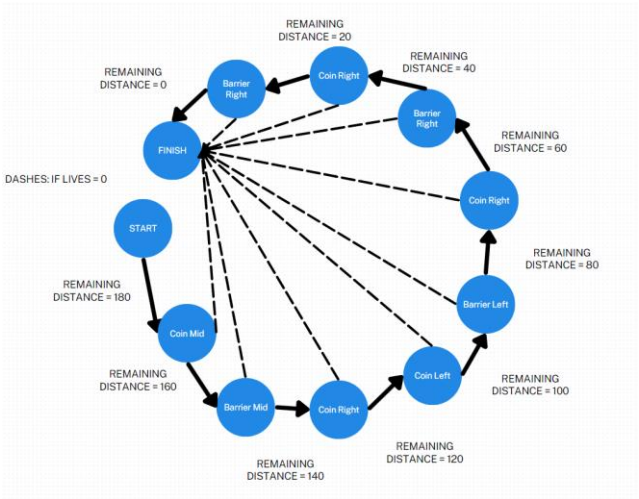

Figure 1: FSM for Game Progression

Figure 2 shows the implementation of the first number of states in Verilog. Basically, the FSM checks the current state and the remaining distance. If the remaining distance becomes a specific value within that state, it moves to the next state and set ups the activation signals for barrier (obstacles) and coins.

```
always@(negedge i_v_sync) begin
    if (ZERO_LIVES == 1) begin
        current_state <= FINISH;
        active_barrier <= RELEASE_NULL;
        active_coin <= RELEASE_NULL;
    end

    // Progression
    if(current_state == START && ZERO_LIVES == 0) begin                    // 0
        if (remaining_distance == 12'd180) begin
            active_barrier <= RELEASE_NULL;
            active_coin <= ON_MID;
            current_state <= COIN1; // move to the next state
        end
    end

    else if (current_state == COIN1 && ZERO_LIVES == 0) begin              // 1
        if(remaining_distance == 12'd160) begin
            active_coin <= RELEASE_NULL;
            active_barrier <= ON_MID;
            current_state <= BARRIER1;
        end
    end

    else if (current_state == BARRIER1 && ZERO_LIVES == 0) begin           //2
        if(remaining_distance == 12'd140) begin
            active_barrier <= RELEASE_NULL;
            active_coin = ON_RIGHT;
            current_state = COIN2;
        end
    end
```

Figure 2: FSM Implementation Snippet (state_machine.sv)

MAKING THE GAME

In this section, we discuss how we make the game step by step as follow.

1. Display Game Images

We make game images by writing System Verilog code for the background, the penguin runner, coins, obstacles, and the student ID. All of these are to be collected and placed under one module that manages all graphics modules. For the background we write a simple System Verilog code that displays a certain RGB value depending on where that pixel is on the screen. If the pixel is on the upper half, it is painted with sky colors while the bottom half has white color for snow. We also add a road for the penguin by drawing a trapezoid and painting it light gray. This painting part is shown below in figure 3.

```
always_comb begin
    if(i_y <= 120) begin
        sky_r = 8'd63;
        sky_g = 8'd81;
        sky_b = 8'd181;
    end
    else if(i_y <= 240) begin
        sky_r = 8'd33;
        sky_g = 8'd149;
        sky_b = 8'd243;
    end
    else begin
        sky_r = 8'd3;
        sky_g = 8'd168;
        sky_b = 8'd244;
    end
end
```

```
// is this pixel part of the sky or inside the trapezoid?
always_comb begin
    sky = (i_y <= sky_limit);
    if (inside_trapezoid) begin
        paint_r = 8'hE0;  // Green color inside the trapezoid
        paint_g = 8'hE0;
        paint_b = 8'hE0;
    end else begin
        paint_r = sky ? sky_r : 8'hFF; // Sky or background color
        paint_g = sky ? sky_g : 8'hFF;
        paint_b = sky ? sky_b : 8'hFF;
    end
end
```

Figure 3: Background Logic (test_card_simple.sv)

The penguin runner graphic is created by writing a System Verilog module which includes all the frames necessary for the penguin's movement. Figure 4 below shows one frame pixel BITMAP and the color table used to color the bitmap. For the penguin, a total of 4 bitmaps are made.

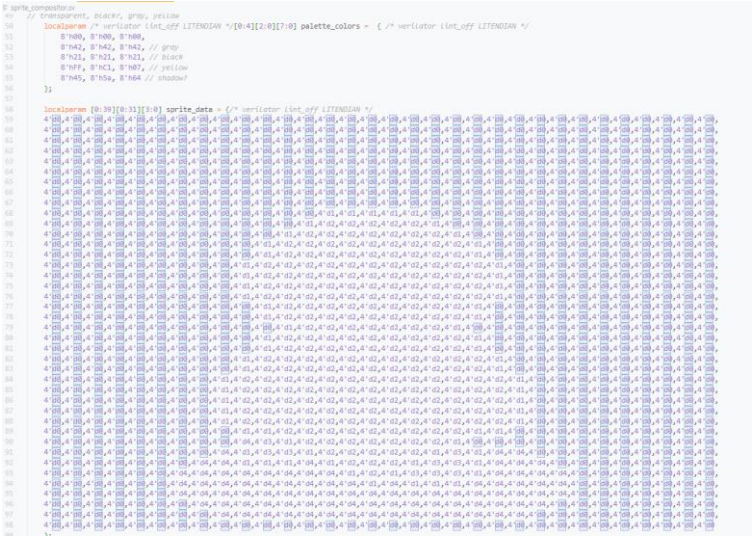

Figure 4: Penguin Runner Making Display (sprite_compositor.sv)

To make a slightly detailed penguin bitmap, we used an online pixel art maker to draw the sprites and use it as a guide. In addition, we used a Verilog Simulator called Verilator to quickly see the results of the System Verilog code without having to wait for Generating Bitstream.

Similarly, we use bitmaps to make display graphics for the coin and the barrier. For the barrier, we just make a simple rectangle colored with red and for the coin we draw a diamond shape and colored it with yellow. We also used the bitmap to make display graphics for the ID.

All of the bitmaps used are small to be directly displayed on the screen. Hence, the bitmaps are stretched by some amount. For example, the coin bitmaps are sized 16x16 and are enlarged by first calculating the distance of the current pixel to the upper left corner of the sprite which gives the current pixel on the bitmap. This pixel is repeated multiple times which gives the enlarging effect. This is implemented as shown below in figure 5.

```
49      assign sprite_hit_x = (i_x >= sprite_x) && (i_x < sprite_x + stretch);
50      assign sprite_hit_y = (i_y >= sprite_y) && (i_y < sprite_y + stretch);
51      assign sprite_render_x = (i_x - sprite_x)>>stretch_factor;
52      assign sprite_render_y = (i_y - sprite_y)>>stretch_factor;
53
54      reg [1:0] selected_palette;
55      assign selected_palette = sprite_data[sprite_render_y][sprite_render_x];
56      assign o_red    = (sprite_hit_x && sprite_hit_y) ? palette_colors[selected_palette][2] : 8'hXX;
57      assign o_green  = (sprite_hit_x && sprite_hit_y) ? palette_colors[selected_palette][1] : 8'hXX;
58      assign o_blue   = (sprite_hit_x && sprite_hit_y) ? palette_colors[selected_palette][0] : 8'hXX;
59      assign o_sprite_hit = active ? (sprite_hit_y & sprite_hit_x) && (selected_palette != 2'd0): 1'b0;
```

Figure 5: Enlarging the small bitmap image

2. Making the Background Effect

We make the background effect by simply writing module for two sprites where one moves to the bottom left and the other to the bottom write. The graphic for the terrain is also made by writing bitmaps and overly stretching the x-direction and stretching the image on y-direction with a small amount. The background effect is achieved by letting both of the sprites start at a designated pixel location then moving down and either left or right one pixel at every positive edge of the vertical sync signal. Once it the upper left corner hits the edge of the screen, it moves back to its original position. Figure 6 below shows the System Verilog code for the right moving sprite. The right sprite instead has (-1) on line 72.

```
70      always@(posedge i_v_sync) begin
71          if(ACTIVE && STATE_CHECK == 0) begin
72              sprite_x <= sprite_x + 1;
73              sprite_y <= sprite_y + 1;
74
75              if(sprite_x >= 1280 || sprite_y >= 720) begin
76                  sprite_x <= 880;
77                  sprite_y <= 360;
78              end
79          end
80      end
```

Figure 6: Background Effect Code (terrain_right.sv)

3. Making the Runner Move

We designate three buttons to make the runner move. BTN3 is used to make it move to the right (of its current location), BTN2 to jump, and BTN1 to move to the left. The penguin does not teleport to the new lane on the press of the button but instead slowly moves to the new location. Jumping is implemented by using two extra frames which moves the penguin upper while the shadow remains on the ground.

Two always blocks are used in implementing these mechanisms. First block activates at positive edge of vertical sync. It checks if a button is pressed and raises a signal that triggers the second always block. On the second always block, it receives the signal on what button is pressed and changes the lane of the penguin (state only, not current visual). Going back to the first block, if the penguin's current lane is X and it is currently not in the base position for the lane X, it slowly moves the sprite to the base position for that lane. For the jump, the first block activates jump only if the penguin is on the ground. If so, it changes the penguin's state which corresponds to the different airborne positions. Different airborne states correspond to different bitmaps to be used by the sprite to show jumping animation. Figure 7 below shows the penguin jump animation logic.

```
239     assign selected_palette = (CURRENT_JUMP_STATE == GROUND) ? (FEET_STATE ? sprite_data[sprite_render_y][sprite_render_x] : sprite_data1[sprite_render_y][sprite_render_x])
240                             :((CURRENT_JUMP_STATE == MID_UP || CURRENT_JUMP_STATE == MID_DOWN) ? sprite_data_jump0[sprite_render_y][sprite_render_x]
241                             : sprite_data_jump1[sprite_render_y][sprite_render_x]);
```

Figure 7: Penguin Jump Animation Logic (sprite_compositor.sv)

Similarly, we give the penguin a running animation by using two pixel maps that alternate the movement of its feet which is swapped every few frames which is triggered by a signal coming from the FSM called [SPRITE_REFRESHER] as output and [REFRESHER] inside the state_machine.sv file.

```
39      // Timer.. do something only if not wait or finish.
40      always@(posedge i_v_sync) begin
41          if(current_state != FINISH && GAME_SWITCH != 0) begin
42              counter <= counter + 1'b1;
43              if(counter == 6'b11_1111) begin
44                  remaining_distance = remaining_distance - 1; // slowly reduce the distance
45              end
46              if(counter[3:0] == 4'b1111) begin
47                  REFRESHER = ~REFRESHER;
48              end
49          end
50      end
```

Figure 8: Sprite animation refresher (state_machine.sv)

4. Making moving obstacles and coins

To make the obstacles (barriers) and coins move, we simply add logic that moves the sprite location towards the bottom of the screen. Since there are three lanes, we write three modules for both barrier and coin on each lane for a total of 6 modules. Barriers and coins on the middle only change their y-coordinates, sprites on the left and right changes their x-coordinate as well.

To make these sprites go larger as they approach the penguin, we add a logic that dynamically stretches the pixel map based on the y-coordinates of the sprite. Figure 9 below shows the stretching logic for the coin on the left. The coin receives an activation signal from the coin_generator.sv which manages generation of coins (obstacles have barrier_generation.sv). Once active, the coin moves depending on which lane is it assigned. At certain y-coordinate value, it is resized by using the [stretch] and [stretch_factor] variables which is used in selecting the pixel to be shown on the screen.

```
66      always@(posedge i_v_sync) begin
67          if (active == 1) begin
68              sprite_y <= sprite_y + 10;
69              sprite_x <= sprite_x - 10;
70              if(sprite_y > 720) begin
71                  sprite_y <= 720; // hide
72                  sprite_x <= 16'd640 - 16'd64;
73  //                IN_PLACE <= 0;
74              end
75
76              if(sprite_y >= 360-50 && sprite_y < 440-50) begin
77                  stretch <= 32;
78                  stretch_factor <= 1;
79              end
80
81              else if(sprite_y >= 480-50 && sprite_y < 600-50) begin
82                  stretch <= 64;
83                  stretch_factor <= 2;
84              end
85
86              else if (sprite_y >= 500) begin
87                  stretch <= 128;
88                  stretch_factor <= 3;
89  //                IN_PLACE <= 1;   // this coin is now ready to be hit
90              end
91
92          end
93          else if (active == 0) begin
94              sprite_y <= 360-50;
95              sprite_x <= 16'd640 - 16'd64;
96  //            IN_PLACE <= 0;
97          end
```

Figure 9: Making sprites bigger as they come closer (coin_left.sv)

5. Making effects of coin

The score display is managed by two modules named score_header_compositor.sv which shows the word "SCORE" and the module digit0.sv which shows a 1-bit digit on the screen. digit0.sv file takes in a value to be shown on the screen. Based on that value, it picks an appropriate pixel map of a bit

and displays it on the screen. Figure 10 below shows a snippet of this logic.

```
135     logic [1:0] selected_palette;
136     always_comb begin
137         if (value == 12'd0) begin
138             selected_palette = sprite_data0[sprite_render_y][sprite_render_x];
139         end
140 
141         else if (value == 12'd1) begin
142             selected_palette = sprite_data1[sprite_render_y][sprite_render_x];
143         end
144 
145         else if (value == 12'd2) begin
146             selected_palette = sprite_data2[sprite_render_y][sprite_render_x];
147         end
```

Figure 10: Displaying score (digit0.sv)

The value taken by digit0.sv file is given by coin_generator.sv which tracks if a collision with a coin is detected. Figure 11 below shows the logic which detect collision. Each coin instantiated sends a signal to the coin_generator.sv module which tells it that it is in place and is ready to be hit. The penguin from sprite_compositor.sv also sends signal to the generator on where the current lane is. If there is a match, it corresponds to a hit, score is increased by 1 and a signal is sent back to the FSM at state_machine.sv to deactivate the spawn signal. If there is no collision and the coin moves past the lower edge of the screen, the coin keep itself at the bottom of the screen, lowers its [IN_POSITION] signal and waits until FSM lowers the activation signal. Since the deactivation signal from FSM takes 2 clock cycles, we read the score value from second LSB for correctness.

```
113     always@(posedge i_v_sync) begin
114         if(COIN_L_IN_POSITION && current_lane == 2'b01) begin // left
115             score <= score + 1'b1;
116             temp_hit <= 1;
117         end
118 
119         else if(COIN_M_IN_POSITION && current_lane == 2'b10) begin // mid
120                 score <= score + 1'b1;
121             temp_hit <= 1;
122         end
123 
124         else if(COIN_R_IN_POSITION && current_lane == 2'b11) begin
125                 score <= score + 1'b1;
126             temp_hit <= 1;
127         end
128         else if (active == 2'b00) begin // set to low only when no barrier is active
129             temp_hit <= 0;
130         end
131     end
```

Figure 11: Coin collision detection (coin_generator.sv)

The graphic effect is managed by sprite_compositor.sv where it "snoops" the signal coming from the coin_generator.sv to state_machine.sv. It sees the collision signal and acts appropriately by changing the output RGB colors to make all sprite pixels' yellow as shown below in figure 12.

```
243     assign o_red   = (sprite_hit_x && sprite_hit_y) ? (barrier_hit ? 8'hFF : ((coin_hit) ? 8'hFF :  palette_colors[selected_palette][2])) : 8'hXX;
244     assign o_green = (sprite_hit_x && sprite_hit_y) ? (barrier_hit ? 8'h00 : ((coin_hit) ? 8'hFF :  palette_colors[selected_palette][1])) : 8'hXX;
245     assign o_blue  = (sprite_hit_x && sprite_hit_y) ? (barrier_hit ? 8'h00 : ((coin_hit) ? 8'h00 :  palette_colors[selected_palette][0])) : 8'hXX;
246     assign o_sprite_hit = (sprite_hit_y & sprite_hit_x) && (selected_palette != 2'd0);
```

Figure 12: Graphic effect logic for coin and obstacle (sprite_compositor.sv)

The audio effect is achieved by using an interrupt. The interrupt signal is sent by the HDMI_TOP.v module which manages all visual logic to the ZYNQ7 processing system which generates audio based on the interrupt signal. Details on the interrupt and audio will be discussed in the final step as in the making of this project, we first finished the graphics part then implemented the audio part.

6. Making effect of obstacles

Similar with step 5, the obstacles follow the same logic for collision with the sprite. It checks if the barrier is in place and if the penguin is on the correct lane and generates a barrier collision signal. Instead of a score, a life point system is used and is managed by life_compositor.sv which also displays the current remaining lives on the upper left of the screen. At every positive edge of the barrier collision signal, it reduces the number of lives and updates the current bitmap to be used.

The graphic effect is similar with the coin effect wherein the sprite_compositor.sv module snoops of the barrier collision signal and changes all output pixels to show red. The audio effect is also done using interrupts sent to the ZYNQ Processing System and is discussed at the last step of this report.

7. Making end point

The end point is achieved after the penguin runs for a finite amount of distance. As explained in the FSM section of this report, the game progresses forward as we reduce the remaining distance down to zero and end the game. Figure 8 above shows the continuous reduction of the current distance which is used by the FSM to generate coins and barriers as the game progresses.

The amount of remaining distance is sent by the FSM to the display module named three_digit.sv which shows the remaining distance in hundreds. The logic is similar to how the score is displayed on the screen only that we have three bits instead of 1. To properly show the right number for each digit, we decompose the number by recursively. For example, to find the hundredths digit, we check if it is bigger than a certain hundred number and subtract the original value by that number.

As soon as the remaining distance hits zero, FSM changes the current state to FINISH as shown on the FSM in figure 1. This signal is snooped by final_score.sv module which displays the text “FINISH” on the middle of the screen and originally included feature that shows the final score hence the name. The text “FINISH” is displayed using bitmap and has similar logic with the sprite compositor for the student ID.

Since the FINISH state is also triggered when we ran out of lives, the FSM constantly checks if the life_compositor.sv module raised ZERO_LIVES signal which signals that there are no more lives left hence the FSM raises the signal to tell every moving sprite to stop and tells final_score.sv module to start displaying labels on the screen.

8. Making background music

The background music is added to the game by re-using the files from Week 9. Since the music audio is completely independent of the game logic, we run them in parallel: the game progresses and the music plays at the same time. Starting from the Week 9 files, we add the HDMI_TOP.v module from our project and integrate it on the block design, and update the XDC file and add the necessary ports.

This is now the part where we add the interrupts to make the audio effects for coin and barrier (obstacle) collisions. To enable interrupts, we simply add a IRQ_F2P interrupt port on the ZYNQ7 processing system. Since there are two interrupt signals, we use a [Concat] IP core to concatenate the coin interrupt and barrier interrupts into one 2-bit signal. The addition of the interrupt feature is shown in figure 13 below.

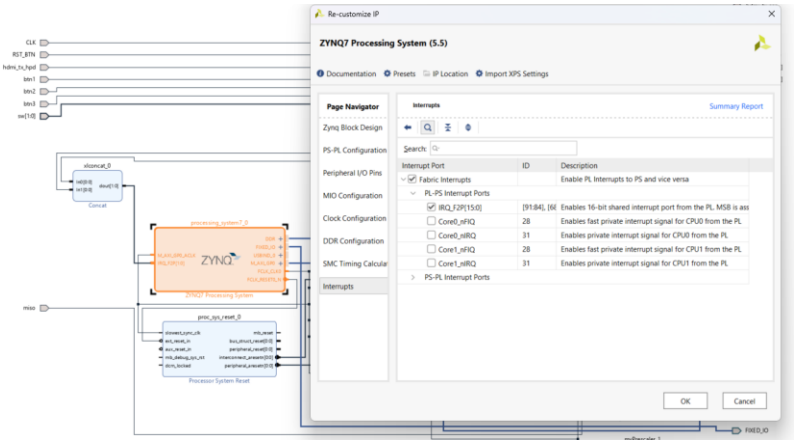

Figure 13: Integration of Interrupt Feature

The full block diagram design in shown below in figure 14. We can clearly see that the game part and the audio part are completely in parallel where the PS manages the audio while the PL manages the game visuals and logic.

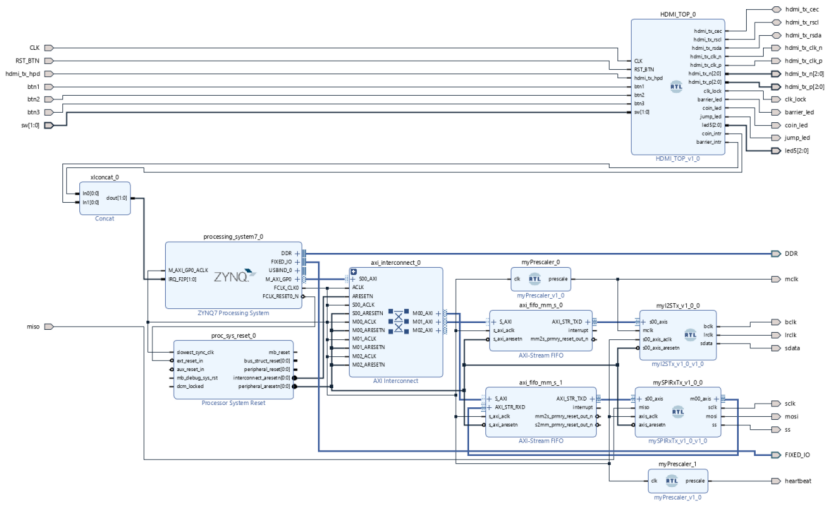

Figure 14: Block Diagram of the Project

We now handle the interrupt in Vitis. The interrupt handler code for coin collision and obstacle (barrier) collision is shown below in figure 15. To generate the sound, we simply pick a single note and ramp it up to a much higher octave to be clearly heard as the majority of the music notes is somewhat low pitched. We make a simple sound pattern for the barrier interrupt and reverse the pattern for the coin interrupt.

```
192 void Barrier_Handler(void *InstancePtr){
193     xil_printf("BARRIER HIT\r\n");
194     note(1, 4*440);
195     note(1, 8*440);
196     note(1, 4*440);
197     note(1, 8*440);
198 }
199
200 void Coin_Handler(void *InstancePtr)
201 {
202     xil_printf("COIN HIT\r\n");
203     note(1, 8*440);
204     note(1, 4*440);
205     note(1, 8*440);
206     note(1, 4*440);
207 }
```

Figure 15: Interrupt Handlers (helloworld.c)

To activate the interrupt, we simply initialize the interrupts by explicitly referring to the registers for the interrupts. Apparently, IRQ_F2P needs to be tied to a pin that is labeled interrupt. However, we do not have such pin and that the interrupt signal is coming from a Verilog Module. Hence when building the project, xparamaters.h would not include such parameters and variables to refer to the interrupt register as it sees no interrupt pins connected to the IRQ_F2P pin. Hence, we explicitly connect the interrupt handlers to the 61st and 62nd bits of the interrupt registers. Figure X below shows the connection for handling the coin collision interrupt where XPAR_FABRIC_EXT_IRQ_INTR is tied to 61U. For barrier collision interrupt we just tie it to 62U.

```
229 int interrupt_init()
230 {
231     int Status;
232
233     GicConfig = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
234     if (NULL == GicConfig) {
235         return XST_FAILURE;
236     }
237     Status = XScuGic_CfgInitialize(&InterruptController, GicConfig, GicConfig->CpuBaseAddress);
238     if (Status != XST_SUCCESS) {
239         return XST_FAILURE;
240     }
241
242     Status = SetUpInterruptSystem(&InterruptController);
243     if (Status != XST_SUCCESS) {
244         return XST_FAILURE;
245     }
246
247     Status = XScuGic_Connect(&InterruptController, XPAR_FABRIC_EXT_IRQ_INTR, (Xil_ExceptionHandler)Coin_Handler, (void *)NULL);
248     if (Status != XST_SUCCESS) {
249         return XST_FAILURE;
250     }
251
252     XScuGic_Enable(&InterruptController, XPAR_FABRIC_EXT_IRQ_INTR);
253
254     return XST_SUCCESS;
255 }
```

Figure 16: Interrupt Initialization {Coin Interrupt) (helloworld.c)

For the barrier collision interrupt initialization, we omit the first 3 checks and proceed with line 247 as the coin handler is initialized first which also set ups the necessary interrupt initializations.

DISCUSSION AND POTENTIAL ISSUES

In this project, we have successfully implemented a working FPGA based arcade game. The project mainly reused multiple different source files given throughout the semester with focus on source files used in Week 9 and Week 11-12. The implementation used both the PL and PS parts of the FPGA which works together in making the complete game.

The project involved multiple revisions and debugging sessions throughout its completion. Most of the debugging came from properly fixing and connecting different modules and generators. From these debugging sessions, we have some modules that does not seem to work accordingly with the code and was fixed by applying proper measures based on the observations. For example, life_compositor.sv would not raise ZERO_LIVES signal to end the game when LIVES is not equal to zero. Based on the observation, the game ends when the penguin hits 4 barriers instead of 3. Hence, the fix was to end the game when LIVES is equal to 1 instead. Graphically, when run on the FPGA, the game ends when there are no more lives left as seen on the screen as required.

Another issue faced was the random game speed. Sometimes the game would run fast and sometimes slow. The current version runs the full game at around 1 minute and 30 seconds. The problem may come from slow logic blocks. We have not checked if this is a physical board problem however it may also be the case.

In the design of the sprites and graphics, we heavily used third party applications such as PIXILART to draw the bitmaps for the different sprites. We also used a Verilog simulator called Verilator to quickly run check the Verilog implementations of the bitmaps without having to directly Generate Bitstream. Details on how to use Verilator (on Linux and MacOs only) can be found in this site.

Finally, due to page limitations, we could not fully explain the minute details of the implementation of each block. In this report, we only included the most significant parts for each step and omitted the specific details on why it works and only wrote the general overview on how we implemented and approached the problem for each step.