LAB 11 DESIGN REPORT

ECE 260 | 07/12/2021

Done by: Michel Shane Keh (A6) | Abrisam Durrani (A9)

Contents

Table of Figures

List of Tables

1. Introduction

In this lab 11, we were tasked to design a digital circuit on the FPGA board, based on the knowledge learnt during ECE260 modules.

In this lab, the project that we have chosen to undertake is the creation of the game, Flappy brick. This project will encompass the following knowledge learnt during the ECE module:

- VGA interface
- Displaying Images with VGA Interface

This report will encompass the Design of our circuit.

2. Introduction to Flappy Brick

Flappy brick is modelled after the popular mobile phone game, Flappy bird, a game developed by Dong Nguyen, a Vietnamese video game artist and programmer [1].

In flappy brick, the user will control a flying red brick, named Reddy and will have to maneuver the brick through an equally sized gaps between a pair of long rectangular blocks placed at random heights. Reddy automatically descends towards the bottom of the screen and will only ascend upon input from the user. Points are scored as the user successfully pass through the pair of rectangular blocks, called obstacle blocks. The game ends when Reddy collides into the rectangular blocks or reaches the bottom of the screen.

3. Flappy Brick Design

Flappy Brick starts with a black background. The display of the game with a static image of Reddy will show when the start switch is flicked up (1). The game is then initiated by pressing the center button on the FPGA board (BTNC).

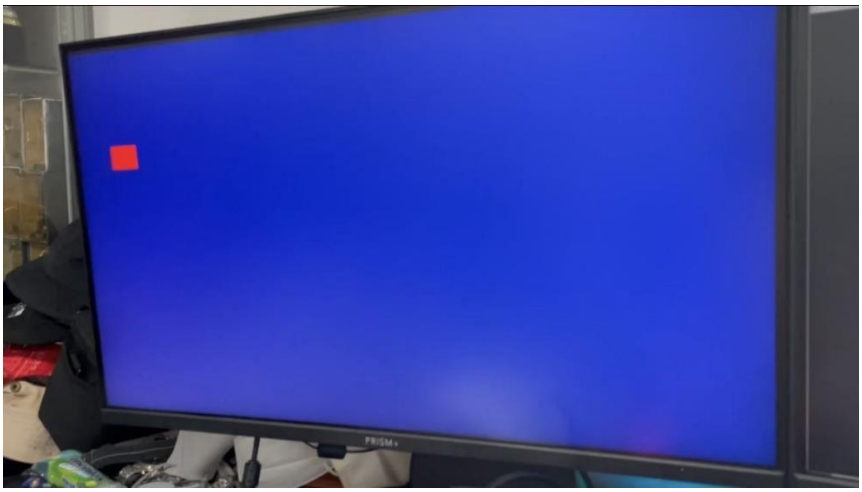

Figure 1: Initial start screen of Flappy Brick

Figure 2: Reddy avoiding the obstacle bricks

The obstacle bricks, whose height are generated randomly, appears one at a time and move towards Reddy. The user will have to control Reddy to pass through the gaps between the obstacle bricks.

The following Signal and button input design for Flappy Brick is as shown in the table below:

Signal Name	Pin Name on Board	Function/Description
Up	BTNC	To control Reddy to move up once button is pressed and to start the game
rstn	BTNU	To restart the game

Start	SW[0]	Intro Screen
audio	SW[15]	Turn on/off the audio
r[2:0]	-	VGA red color signal
g[2:0]	-	VGA green color signal
b[1:0]	-	VGA blue color signal
hs	-	VGA horizontal sync signal
vs	-	VGA vertical sync signal
ubird	-	Upper bound for Reddy
dbird	-	Lower bound for Reddy
ublock	-	Upper bound for obstacle blocks
lblock	-	Lower bound for obstacle blocks
rblock	-	Width bounds for obstacle pipes
usuiji	-	Randomized upper bound coordinate for obstacle block
videoen	-	Signal to enable the display of the game
Pass	-	Flag to increment score when flag is high
Num0[3:0] to Num3[3:0]	-	Counter for the score
Success	-	Flag set high every time Reddy passes through the obstacle. Once this flag is set low, the game automatically ends
Firstup	-	Flag to prevent continuous input if the BTNC is pressed and held

Table 1: Signal description and pin assignment

Table 1 above shows the description of the various signals used for Flappy Brick and the allocated pin assignment if needed.

3.1. Reddy Movement Control

An important design feature in Flappy Brick is the movement of Reddy. By default, Reddy will fall towards the bottom of the screen, unless there is an up movement initiated by the press of the BTNC by the user.

The default movement of Reddy, which is falling, is implemented by increasing the upper and lower bounds of Reddy by a fixed value.

```
begin
   ubird <= ubird-10'd80;
   dbird <= dbird-10'd80;
   firstup<=0;
end
```

Figure 3: Code to implement the default behaviour of Reddy

As shown from figure 3 above, the upper and lower bounds of Reddy are increased by a fixed decimal value of 15, to implement the default falling behaviour of Reddy.

However, whenever BTNC is pressed by the user, Reddy will move up. This is done by decreasing the upper and lower bounds y-coordinate by a fixed value. In addition to this, a flag, firstpress, is set high whenever the button is first pressed. This is to prevent the continuous up movement in case the user presses and hold the button down.

```
if(up==1&&ubird>10'd31)//up button pushed
  begin
    //jump once
    if(firstup==1)
      begin
        ubird <= ubird-10'd80;
        dbird <= dbird-10'd80;
        firstup<=0;
      end
    else if(dbird<10'd510)
      begin
        ubird <= ubird+10'd15;
        dbird <= dbird+10'd15;
      end
  end
```

Figure 4: Code to implement the up movement of Reddy

As seen in figure 4 above, the upper and lower bounds of Reddy is incremented once a button press is detected, and the firstup flag is low.

3.2. Random Obstacle Bricks generator

One of the key design features of Flappy Brick is the generation of the obstacle bricks at random heights. The upper bound of the obstacle brick is assigned a randomly incremented variable, usuiji, as seen in the snippet of the code in Figure 5 below:

```
        begin
            lblock <= 10'd800;
            rblock <= 10'd850;
            ublock <= usuiji;  //Upper bound random
         end
    end
```

Figure 5: Code to generate random heights of the obstacle bricks

Usuiji is updated and incremented at every positive edge of sjclk, a clock that is assigned values randomly from clkdiv[15:0]. Whenever usuiji reaches a value of 400, it will be reset back to 35. This is to ensure that the upper bound of the gap in the waterpipe does not go out of range of the screen.

```
assign sjclk=clkdiv[15];//random number clock
reg [9:0]usuiji=10'd35;
always @(posedge sjclk)
  begin
    if(rst==1)
      usuiji<=10'd150;
    else
      begin
        usuiji <= usuiji+1'b1;
        if(usuiji == 10'd400)
          usuiji <= 10'd35;
      end
  end
```

Figure 6: Code to generate values for variable usuiji

Through this method, we are able to generate varying heights for Reddy to traverse through to vary the difficulty of Flappy Brick.

3.3. Moving the Pipes

Another key design in Flappy Brick is to enable the movement of the pipes towards Reddy, to simulate that the Reddy is flying towards the obstacles. This is done by “shifting” the obstacles by a fixed value.

```
//waterpipe movement
assign pipemoveclk = clkdiv[20]; //
always @ (posedge pipemoveclk or posedge rst)
begin
   if(rst || strtgame==0)
       begin
          lblock <= 10'd800;
          rblock <= 10'd850;
          ublock <= 10'd50;
        end
    else
      begin
        lblock <= lblock - speed;//
        rblock <= rblock - speed;
        if(rblock < 140)//
          begin
              lblock <= 10'd800;
              rblock <= 10'd850;
              ublock <= usuiji;  //Upper bound random
           end
      end
end
```

Figure 7: Code to enable the movement of the obstacle

As seen from figure 7 above, the obstacles are moved by shifting the blocks by a fixed variable, speed.

The difficulty of the game can also be increased by increasing the value of the rate at which the obstacle is shifted from right to left, thereby increasing the speed of which the obstacle is moving towards Reddy. The code is as shown in Figure 8 below:

```
//waterpipe difficulty - speed of the waterpipe
assign difficultclk = clkdiv[29];//
reg [3:0]speed = 10'd8;
always @(posedge difficultclk or posedge rst)
  begin
    if(rst)
        speed <= 8;
     else
      begin
        if(speed < 15)
           speed <= speed + 1'b1;
       end
end
```

Figure 8: Code to increase the difficulty of the game by incrementing the variable speed

3.4. Collision Detection and Scoring system

In Flappy Brick, the aim is to score as many points as possible by traversing through as many obstacles as possible. You will lose the game if you collide into the obstacles or reach the bottom of the screen. As such, the scoring system and collision detection is an important design to the game.

3.4.1. Scoring System

The scoring system is design such that every time Reddy passes through the obstacles, a flag, pass, will be set high, and while the flag is high, the score and incremented. The total score is designed to be displayed on the 7-LED Segment on the FPGA Board.

```
reg success = 1;//
reg first ;//prevent double scoring
reg [7:0]score=0;
initial first <= 1;
initial pass <= 0; //when pass is high, score increases
always @ (posedge flyclk or posedge rst)
begin
   if(rst)
       begin
         success <= 1;
        // score <= 0;
         pass<=0;
         first <= 1;
       end
    else if(lblock < 231 && rblock > 200)//bird meet pipe
       begin
          if((ubird < ublock || dbird > (ublock + 150))&&!start)//collision
            begin
             success <= 0;
             pass <= 0;
            end
             else if( first == 1&&success==1)
             begin
               pass <= 1;//pass a waterpipe
               first <= 0;
             end
        end
     else if(dbird>=510)
       begin
         success<=0;
         pass<=0;
       end
     else
       begin
        first <= 1;
        pass <= 0;
       end
end
```

Figure 9: Code for scoring system and collision detection

Figure 9 above shows the overall code for the scoring system. As previously described, the pass flag will be set high if Reddy's position on the screen does not match with the upper and lower bounds of the obstacles. Once the pass flag is high, the score will then be incremented.

```
always@(posedge pass or posedge rst)
begin
    if(rst)
    begin
        num0 <= 0;
        num1 <= 0;
        num2 <= 0;
        num3 <=0;
    end
    else if(num0 == 9)
    begin
        num0 <= 0;
        if(num1 == 9)
        begin
            num1 <= 0;
            if(num2 == 9)
            begin
                num2 <= 0;
                if(num3 == 9)
                    num3 <= 0;
                else
                    num3 <= num3 + 1;
            end
            else
                num2 <= num2 + 1;
        end
        else
            num1 <= num1 + 1;
    end
    else
        num0 <= num0 + 1;

end
```

Figure 10: Code for incrementing the score

Figure 10 above shows the code for incrementing the score. As defined in Table 1, num0[3:0] to num3[3:0] are variables to keep the score. Each num variable is to represent a single numeric decimal digit, with num0 representing ones to num3 representing thousands respectively. As such, whenever the value of each num reaches 9, the variable is reset, and the following numeric decimal place value is incremented.

3.4.2. Collision Detection

```
else if(lblock < 231 && rblock > 200)//bird meet pipe
   begin
      if((ubird < ublock || dbird > (ublock + 150))&&!start)//collision
        begin
         success <= 0;
         pass <= 0;
        end
         else if( first == 1&&success==1)
         begin
           pass <= 1;//pass a waterpipe
           first <= 0;
         end
```

Figure 11: Code for collision detection

As seen from figure 11, Reddy is considered colliding with the obstacle when the position of Reddy is within the upper or lower block of the obstacles. When collision happens, the pass flag is set low, and the success flag is also set low.

Once the success flag is set low, the game automatically ends, and a red colored background is printed. This is as shown in the code in figure 11 below:

```
if(success == 0 && start==0)begin
    //gameover
    if(vc < 511 && vc >=31 && hc < 784 && hc >=144)
      begin
          r <= 3'b111;
          g <= 3'b000;
          b <= 2'b00;
      end
    else
      begin
          r <= 3'b000;
          g <= 3'b000;
          b <= 2'b00;
      end
end
```

Figure 12: Code to indicate the game is over. Once the game is lost, a red screen is displayed

3.5. Music

We implemented a background music in the game. When the game is booted up, the music will start to play. By using switch[15], when it is off, the music will stop and when it is on, it will start playing the audio through the audio port.

```
always @ ( posedge  clk_4Hz )begin   //An eternal east wind for everyone here
   if ( len == 246 )begin
      len = 0 ;
   end
   else begin
      len = len + 1 ;
   end
  case ( len )
   0 : j = 12 ; 1 : j = 12 ; 2 : j = 12 ; 3 : j = 12 ; 4 : j = 12 ;
   5 : j = 12 ; 6 : j = 13 ; 7 : j = 13 ; 8 : j = 12 ; 9 : j = 12 ;
   10 : j = 11 ; 11 : j = 10 ; 12 : j = 10 ; 13 : j = 9 ; 14 : j = 9 ;
   15 : j = 1 ; 16 : j = 1 ; 17 : j = 1 ; 18 : j = 1 ; 19 : j = 9 ;
   20 : j = 9 ; 21 : j = 5 ; 22 : j = 5 ; 23 : j = 5 ; 24 : j = 5 ;
   25 : j = 5 ; 26 : j = 5 ; 27 : j = 6 ; 28 : j = 6 ; 29 : j = 6 ;
   30 : j = 6 ; 31 : j = 5 ; 32 : j = 5 ; 33 : j = 5 ; 34 : j = 5 ;
   35 : j = 8 ;
   36 : j = 8 ;
   37 : j = 10 ;
   38 : j = 12 ;
   39 : j = 12 ;
   40 : j = 10 ;
   41 : j = 10 ;
```

Figure 13: Switch case for the song

As shown in figure 13 above, we can see that the variable len increments by 1 till it reaches 246 which is the duration of the song. In each case, we can see that the variable j is assigned a number. This number will then be used in another switch case as shown in figure 14 below.

```
always @ ( posedge  clk_4Hz )begin
     case ( j )
    'd0 : origin = 'b0 ;
    'd1 : origin = 'd4916 ;   //low
    'd2 : origin = 'd6168 ;
    'd3 : origin = 'd7281 ;
    'd4 : origin = 'd7791 ;
    'd5 : origin = 'd8730 ;
    'd6 : origin = 'd9565 ;
    'd7 : origin = 'd10310 ;
    'd8 : origin = 'd010647 ;   //middle
    'd9 : origin = 'd011272 ;
    'd10 : origin = 'd011831 ;
    'd11 : origin = 'd012087 ;
    'd12 : origin = 'd012556 ;
    'd13 : origin = 'd012974 ;
    'd14 : origin = 'd013346 ;
    'd15 : origin = 'd13516 ;   //high
    'd16 : origin = 'd13829 ;
    'd17 : origin = 'd14108 ;
    'd18 : origin = 'd11535 ;
    'd19 : origin = 'd14470 ;
    'd20 : origin = 'd14678 ;
    'd21 : origin = 'd14864 ;
    default : origin = 'd011111 ;
   endcase
end
```

Figure 14: Switch case for audio inputs

Depending on the input j, it will assign the variable, origin, to a number, which will then be transferred to count. Then, count is incremented till it reaches 16363, before it takes in the new value of the next sound. All these is to determine which frequency the music is played at, being the lows, mids and highs. As shown in figure 15 below.

```
always @ ( posedge  clk_6MHz )begin
    if ( count == 16383 )begin
        count = origin ;
        audiof =~ audiof ;
    end
    else begin
        count = count + 1 ;
    end
end
```

Figure 15: Audio frequency output

3.6. Schematic

The figure 16 below shows the schematic of our designed circuit for Flappy Brick:

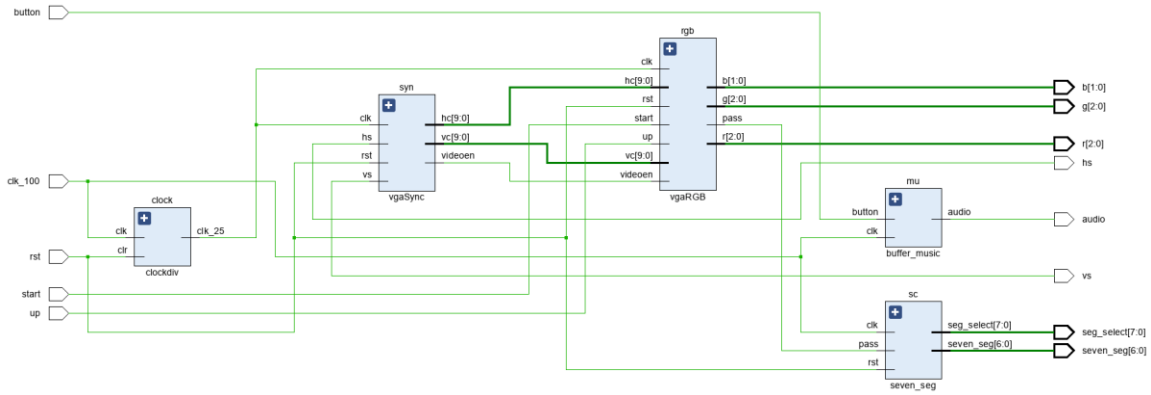

Figure 16: Schematic for Flappy Brick

Figure 17 below shows the synthesized circuit for our Flappy Brick Design:

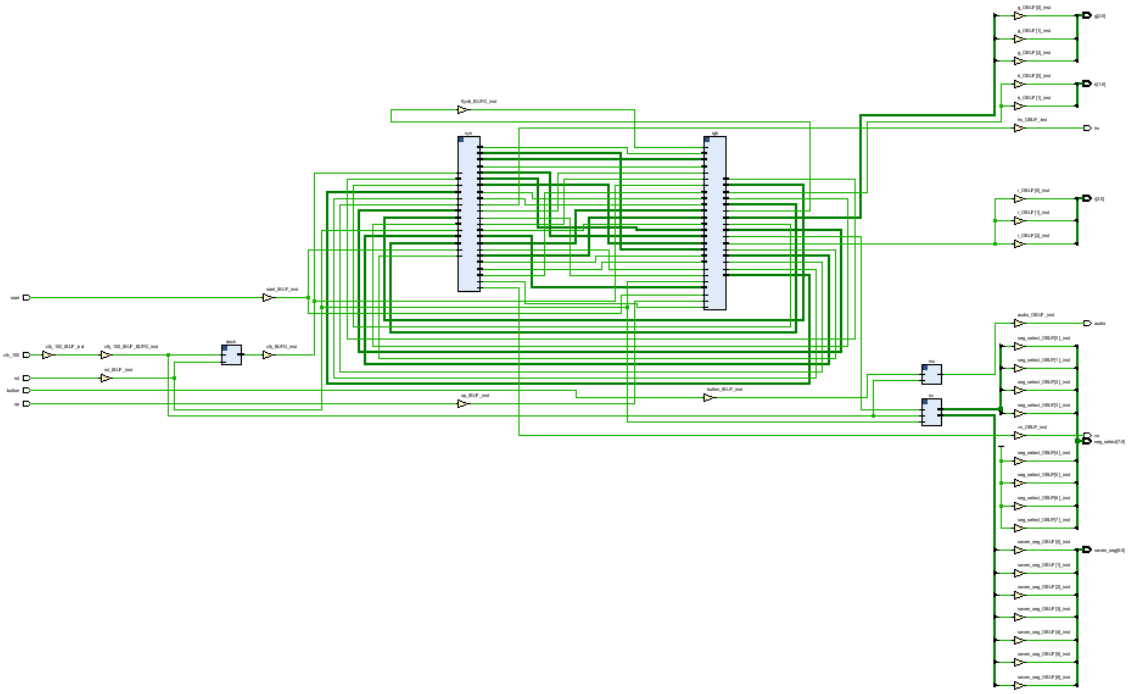

Figure 17: Synthesized Schematic for Flappy Brick

4. Conclusion

The objective of this lab was to create a design circuit that would enable us to apply the knowledge thought in ECE260. The choice of designing Flappy Brick has enabled us to achieve this objective.

Flappy brick required the displaying on bricks onto a screen by using the FPGA VGA interface. This required us to use the knowledge in Lab 5 and Lab 6 of ECE260. It required us to understand and generate the required timing control signals, and the generation of the VGA HSYNC and VSYNC Signals. In addition to this, the Flappy Brick game required us to be able to control the bricks on the screen, in which we used the knowledge from Lab 6 to implement.

Furthermore, this project has also enabled us to experiment and try controlling and interfacing with the FPGA audio interface. Doing so required us to put several knowledge and lab practices learned during ECE260.

In conclusion, this Flappy Brick design enabled us to practice the skills and knowledge though in ECE260 and has given as a good understanding of the practical use of the lessons taught in this module.

5. References

[1] R. Williams, "Whati s Flappy Bird The game taking the App Store by storm," The Daily Telegraph, January 2014. [Online]. Available: https://www.telegraph.co.uk/technology/news/10604366/What-is-Flappy-Bird-The-game-taking-the-App-Store-by-storm.html. [Accessed December 2021].