ECE 385

Spring 2018

Final Project

Final Project

Ziwei Zhu

Nuochen Lyu

Section ABO/ Thursday 3:00-5:50

TA: Zhenhong Liu

Gene Shiue

Introduction

Our final project is a kirby game action adventure 2D game. The game content contains two stages: A normal stage that have three enemies and multiple platforms, and the second stage that contains a boss. The game involves main character actions including jump, attack, running and flying. Enimies would hurt kirby and reduce its health points. If kirby kills enimies they would also gain scores. The moving of kirby applys our gravity system. The image is loaded by both on-chip memory and SRAM with color palette. The image graphics are all designed and drawn by us. In the second stage we designed a AI for our boss which would track and chase our kirby. The control system includes keyboard and mouse which is able to equally manipulate our kirby. The background music starts playing after the game starts. Finally the game includes a speed up function button and a restart button. The whole games includes 12 FSM, three driver interfaces and multiple graphic mapper function.

Game Description

Our goal is to make a game that is like kirby in GBA platform. A adventure action game from Nintendo. We plan to clone most of its features and make a small version using our FPGA board. Below is one of the stage in Kirby.

Our game supports following functions:

--The character kirby in our game is able to walk, jump, fly and attack.

--The enemies would walk and hurt kirby once kirby meets the enemies, which requires collision detection.

--The scene of the game includes the ground and several floating clouds. The boarder of the screen is restricted too.

--The gravity system enable kirby to jump and fly in the air for a period of time. The kirby would fell from the floating clouds.

--There are two stages in the game. First stage is a normal adventure. Second stage is the boss fight.

--Kirby have three health point. If Kirby loss all points it would die.

--Kirby can attack the enimies by inhaling. The enemies would disappeared.

--In the second stage the boss would automatically trace and chase kirby and attack kirby by shooting stars.(A.I.)

--The score would display how many enimies are killed.

--Kirby can be controlled by keyboard. Multiple keys are supported such as left + run. And there is a trick for cheating. Once player press “up, down, left, right”, all the enemies would be killed.
--Kirby can be controlled by mouse. Mouse could perform the same action as keyboard. Also the buttons are clickable by mouse.
--Reset button to restart.
--Speed button to swith kirby into super kirby mode. The kirby would be really fast.
--Background music supported.
--Start menu to start the game.

Features

1. multiple keys (up to and include four)
2. Motion/Action control for kirby (left walk, right walk, jump mid, jump left, jump right, fly left, fly right, stop, attack left, attack right, die)
3. Hp points record (decending)
4. Score points record (acending)
5. Boss AI state machine (trace, range detection, find, move, attack, die)
6. Graphics (color palette, spirts, fix functions, background)
7. On chip memory storage (sprits) and SRAM (two background)
8. Collision detection (for floating clouds, grounds, screen boards and enemies)
9. Mouse control (fly, run, jump)
10. Tuner (speed control)
11. Audio output
12. Two stages
13. Enimies state control
14. Gravity

Design Details

Image processing

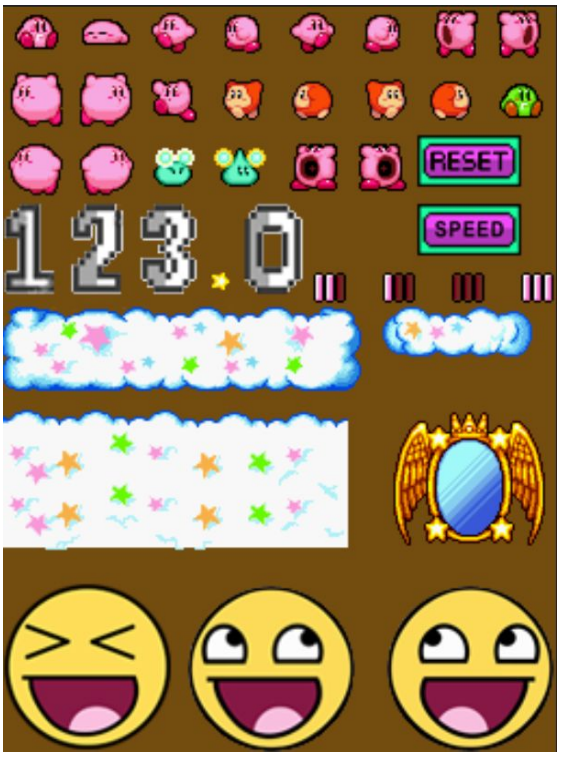
Sprits & fix function
Most of our game image comes from sprits. The kirby, clouds, enimies, boss, buttons, score and HP points. We draw these image and put them into one canvas that have width 256. We select 256 particularly because we could use shift operation to access each spirits during the calculation. For instance, if we need to find a kirby image, we could use the following formula:

address = (DrawY - Xposition) * 256 + DrawY + Yposition

But since width is 256, we could transform into the following formula:

address = (DrawY - Xposition) << 8 + DrawY + Yposition

Shift operation is far more quicker than the normal multiplication. The sprits canves in our project is 256 * 500 which is around 500KB. It would be saves in the on-chip memory.



Background

There are two background for our game because we have two stages. The background picture one is 640*480 and the second picture is 640*300 considering the stage limitation in SRAM. The access and strage is transfromed into different format in SRAM which would be covered later.

On-chip memory

The sprits are saved in the on-chip memory. We designed our on-chip memory into 8*(total pixels in the picture) size. Each pixel is represented by 8 bits. However, we only use 6 bit because the data in the on-chip memory does not represent real RGB color. We use a color palette to store the 48 colors that appears in the game. The data in on-chip memory saves the quiry number of the RGB color. 48 numbers needs at least 6 bit to represent. The declaration of the on-chip memory is listed below:

```
module  frameRAM
(
        input [7:0] data_In,
        input [18:0] write_address, read_address,
        input we, Clk,

        output logic [7:0] data_Out
);

// mem has width of 3 bits and a total of 400 addresses
logic [7:0] mem [0: 90112];

initial
begin
    $readmemh("kirby3.txt", mem);
end


always_ff @ (posedge Clk) begin
    if (we)
        mem[write_address] <= data_In;
    data_Out<= mem[read_address];
end

endmodule
```

The input address of the on-chip memory is controlled by a address dispatcher consist of about 47 separated address wire. Each wire points to a different picture address. For example the kirby address module manage the address of the 12 actions of Kirby. The module would output the needed address for Kirby. The 47 address wire read the data at the same time. But only one wire would be choosed. This is obvious since there could be only one single address input for our on-chip memory. This measures avoid the multiple drive issue.

```
else if(is_diren)
    read_address = read_address_diren;
else if(is_diren2)
    read_address = read_address_diren2;
else if(DrawY >= cloud && DrawY <= cloud + 10'd40 && DrawX >= 10'd475)
    begin
    is_cloud = 1'b1;
    read_address = ((10'd144 + DrawY - cloud) << 8) + DrawX - 10'd475;
    end
else if(DrawY >= cloud_left && DrawY <= cloud_left + 10'd40 && DrawX <= 10'd165)
    begin
    is_cloud = 1'b1;
    read_address = ((10'd144 + DrawY - cloud_left) << 8) + DrawX;
    end
else if(DrawY >= cloud_mid && DrawY <= cloud_mid + 10'd40 && DrawX >= 10'd250 && DrawX <= 10'd415)
    begin
    is_cloud = 1'b1;
    read_address = ((10'd144 + DrawY - cloud_mid) << 8) + DrawX - 10'd250;
    end
else if(DrawY >= cloud_small && DrawY <= cloud_small + 10'd24 && DrawX >= 10'd166 && DrawX <= 10'd235)
    begin
    is_cloud_small = 1'b1;
    read_address = ((10'd144 + DrawY - cloud_small) << 8) + DrawX - 10'd166 + 10'd176;
    end
```

Once the correct address is feeded to the on-chip memory, we could get the data in the color mapper. The data query the color palette so we could get the correct RGB color. Finally the RGB color is outputted to the VGA and displayed on the screen.

SRAM

The SRAM saves two background. The memory format for the SRAM is 16 bit width. However RGB is 24 bits. So we truncated the RGB into 15 bits. Each color channel owns 5 bits. Then a zero is appended to the 15 bit RGB and stored into one SRAM memory. We use Matlab to transform the image into a hex file. And use control panel to upload the image into the SRAM. For a 640*480 image, the hex file is 1400 KB. However the SRAM is only 2MB big so two pictures would exceed the size limitation of the SRAM. Therefore, the second picture is 640*300 considertion the size limitaion.

One stage would select one image so we switch the address for different stage. The input for SRAM has two wires lies in a if statement. The color is compressed from 24 bit RGB into 15 bit RGB.

```
always_comb
begin
    if(stage_one)
        SRAM_ADDR = (DrawY_SRAM * 20'd640) + DrawX_SRAM;
    else
        begin
            if(DrawY <= 10'd315)
            SRAM_ADDR = (DrawY_SRAM * 20'd640) + DrawX_SRAM + 20'd640 * 20'd480;
            else
            SRAM_ADDR = 20'd640 * 20'd480 + 10'd20;
        end
end
```

Color Mapper

The color Mapper is the interface and center management system that controls the graphic output. The working procedure of the color mapper is demonstrated with the following steps. The game logic goes from different modules to color mappers. The color mappers get the signals from the game logic and understand what pattern to be displayed. The the mappers asked the on chip memory and SRAM for the data. The data need to be mapped to the color palette. Finally the RGB is outputted to the VGA and displayed in the screen. The characters spirits are displayed by multiple switch case to prevent high delay and glitches.

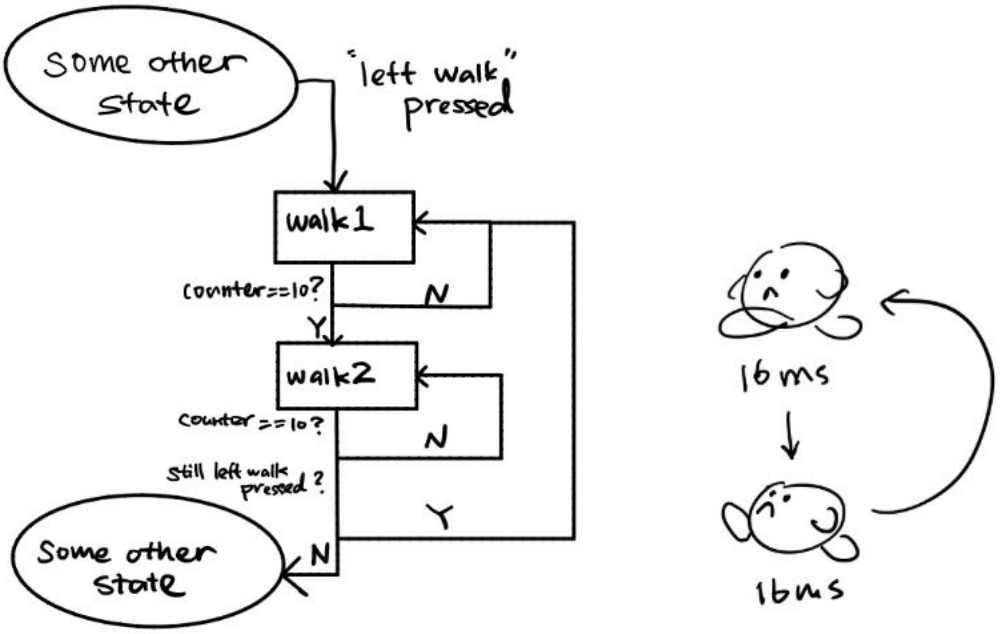
Animation Effect

FSM & Counter

All the animation effect, are literally fast picture switch, as if the object is really moving on the screen. The switch of the pictures/image is done all by finite state machine. And together the timing of the state is controled by a counter. Most importantly, a **frame counter**.
Took the walking of the kirby as a example, the waking moving contains two state.

move the left foot  **move the right foot** 

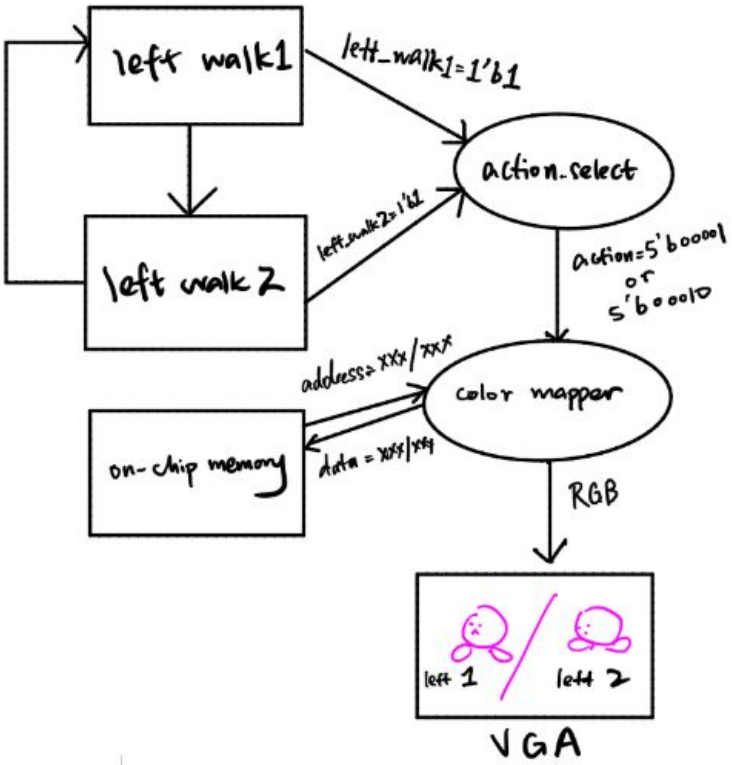
The walking left is basically the switching of these two pictures. So part of the FSM should be developed as following:



The counter is not controlled by the main clk, because it is too fast for human eye. We plan to let the left foot image stay for 10 frame clock. And then switch to the right foot image for another 10 fram clock. The fram clock is feeded by the signal VS which is the vertical signal. The VS signal would clicking at 60 times per second. So in the walk 1 state we would keep looping itself from 10 clock cycle which would create about 16ms still image on the screen. And then once the cycle equals to 10, the FSM drives to the right foot state.

Image Selection

For each state we would give a signal to indicate which picture should be chosen now. The signal could be a singal wire or a coded data. Then the color mapper would process the message and quiry the on-chip memory for the image data. Finally the image is outputted to the VGA scanner. We would able to see a quick running little kirby in the monitor.




```
color_extract color_extract_module( .*);
color_extract_diren color_extract_diren_module(.*);
color_extract_diren color_extract_diren2_module(.*, .diren_a
color_extract_bg color_extract_bg_module(.*);
color_extract_hp color_extract_hp_module(.*);
color_extract_boss color_extract_boss_module(.*);
```

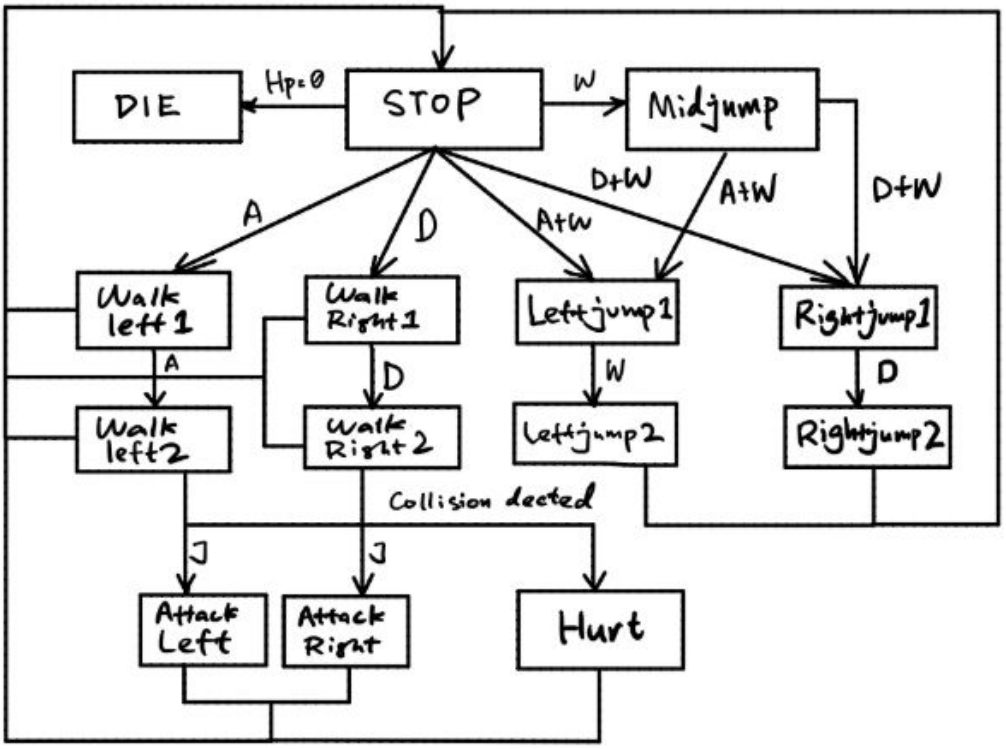
Kirby: the main charater

Movement & State

The kirby is our main character which involves complicated actions. The state of the kirby is the following:

“stop, sleep1, hurt, dead, leftwalk1, leftwalk2, rightwalk1, rightwalk2, midjump, leftjump, rightjump, leftjump2, rightjump2, stop2, leftsuck, rightsuck”

A FSM is applied to control the state of the kirby. Each state outputted proper signals in order to control the image and interaction of the Kirby. The finite state machine is displayed below.



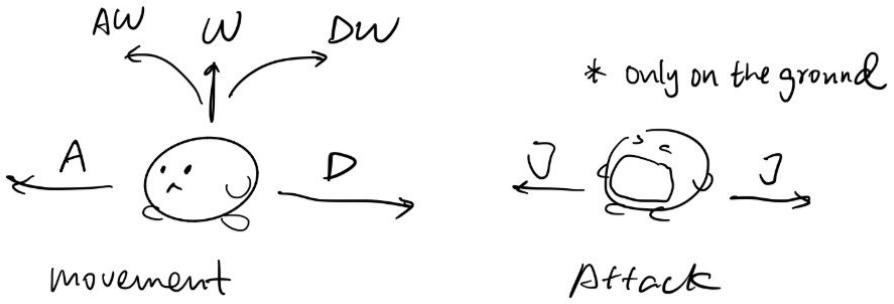
Controls

As the input of the kirby. The movement and action of the kirby is controlled by keyboard and mouse. The keyboard and mouse driver feed the data to kirby module. Our kirby module would decoded the data. The data is either a 32 bit keycode or position of the mouse.

For the keycode, a switch case handles all the keyboard input combination by a huge switch case:

```
// Update position and motion only at rising edge of frame clock
if (frame_clk_rising_edge && start_game && ~freeze && ~leftButton && ~rightButton)
begin
    unique case(keycode[15:0])
        16'h001A:  // w (up)
            begin
                jump = 1'b1;
                Ball_X_Motion_in = 1'b0;
            end
        16'h0004:   //A(left)
            begin
                Ball_X_Motion_in = (~(Ball_X_Step) + 1'b1);
                walk_left = 1'b1;
                pre_direction = 1'b0;
            end
        16'h0007: //D(right)
            begin
                Ball_X_Motion_in = Ball_X_Step;
                walk_right = 1'b1;
                pre_direction = 1'b1;
            end
        16'h0016://S(down)
            begin
                Ball_X_Motion_in = 1'b0;
            end
        16'h041a:  // w (up) + A (left) left jump
```

Each case gives two kind of signals. One for FSM to tell FSM what is the next state. And the other one is usually the step/motion of the kirby. For example in the above image in case 0004, we set the Ball_X_Motion_in = -1.
Mouse is similar to the keyboard input. But mouse would gives two additional data, x position and y position. So we could let our kirby follows the mouse by comparing the relative position between the mouse and the kirby.

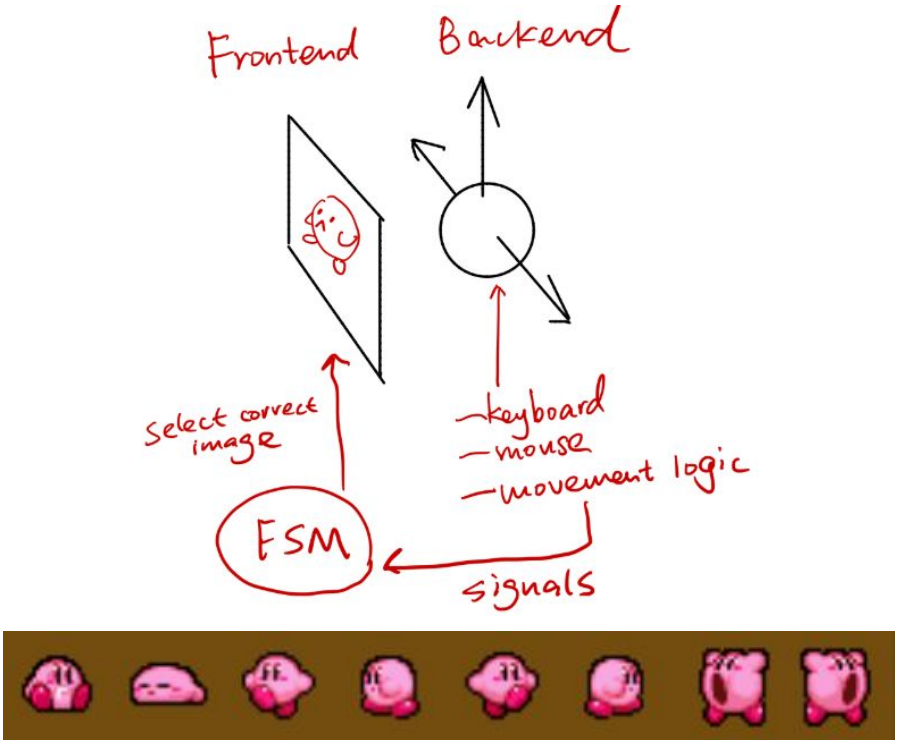


Movement

```
// Update registers
always_ff @ (posedge Clk)
begin
    if (Reset)
    begin
        Ball_X_Pos <= Ball_X_Center;
        Ball_Y_Pos <= Ball_Y_Center;
        Ball_X_Motion <= 10'd0;
        Ball_Y_Motion <= Ball_Y_Step;
            pre_direction_reg <= 1'b1;
    end
    else
    begin
        Ball_X_Pos <= Ball_X_Pos_in;
        Ball_Y_Pos <= Ball_Y_Pos_in;
        Ball_X_Motion <= Ball_X_Motion_in;
        Ball_Y_Motion <= Ball_Y_Motion_in;
            pre_direction_reg <= pre_direction;
    end
```

The movement of the kirby is built on the game logic for ball.sv, which is very similar to the game in lab 8. The kirby would update the X velocity when it receive a left or right command. The kirby would also update the Y velocity when it receive a jump signal. But notice that since kirby is different from a flat ball game. Here the Y position is determined by another module called gravity which would be introduced later. The movement would also update the position of the kirby in the screen for the interactions.

Kirby Frontend: Skin



The above game logic for Kirby build the base or backend of the character. The character is moving correctly. However, how do we see the correct image/skin of the kirby in different state. If we do not have a way to select the proper spirt for Kirby. Kirby would be only a ball. Here we use the FSM described above to manage all the possible state of the Kirby. The backend pass the control signal to the FSM. And the FSM further output the need image to the color mapper. The message of the Kirby action is encoded as following:

```
//kirby's action
always_comb begin
    if(midjump_1)
        kirby_action = 5'b00111;
    else if(walk_left_1)
        kirby_action = 5'b00001;
    else if(walk_left_2)
        kirby_action = 5'b00010;
    else if(walk_right_1)
        kirby_action = 5'b00011;
    else if(walk_right_2)
        kirby_action = 5'b00100;
    else if(leftjump_1)
        kirby_action = 5'b00101;
    else if(leftjump_2)
        kirby_action = 5'b01101;
    else if(rightjump_1)
        kirby_action = 5'b00110;
    else if(rightjump_2)
        kirby_action = 5'b01110;
    else if(sleep_1)
        kirby_action = 5'b01000;
    else if(leftsuck_1)
        kirby_action = 5'b01001;
    else if(rightsuck_1)
        kirby_action = 5'b01010;
    else
        kirby_action = 5'b00000;
end
```

The kirby action is passed to the color mapper, the color mapper took in charge of finding real image sprite in the on-chip memory. And them map the color of the memory to the correct RGB value by the palelet.

```
unique case(kirby_action)
    5'b00001:  // left walk
        begin
            read_address_kirby = ((19'd15 - (ball_y_pos - draw_y_pos)) << 8)  + (19'd79 - (ball_x_pos - draw_x_pos));
        end
    5'b00010:  // left walk 2
        begin
            read_address_kirby = ((19'd15 - (ball_y_pos - draw_y_pos)) << 8)  + (19'd111 - (ball_x_pos - draw_x_pos));
        end
    5'b00011:  // right walk
        begin
            read_address_kirby = ((19'd15 - (ball_y_pos - draw_y_pos)) << 8)  + (19'd143 - (ball_x_pos - draw_x_pos));
        end
    5'b00100:  // right walk 2
        begin
            read_address_kirby = ((19'd15 - (ball_y_pos - draw_y_pos)) << 8)  + (19'd175 - (ball_x_pos - draw_x_pos));
        end
    5'b00101:  // left jump
        begin
            read_address_kirby = ((19'd47 - (ball_y_pos - draw_y_pos)) << 8)  + (19'd15 - (ball_x_pos - draw_x_pos));
        end
    5'b00110:  // right jump
        begin
            read_address_kirby = ((19'd47 - (ball_y_pos - draw_y_pos)) << 8)  + (19'd45 - (ball_x_pos - draw_x_pos));
        end
    5'b00111:  // mid jump
        begin
            read_address_kirby = ((19'd47 - (ball_y_pos - draw_y_pos)) << 8)  + (19'd79 - (ball_x_pos - draw_x_pos));
        end
    5'b01000:  // sleep
        begin
            read_address_kirby = ((19'd15 - (ball_y_pos - draw_y_pos)) << 8)  + (19'd45 - (ball_x_pos - draw_x_pos));
        end
    5'b01001:  // suck left
        begin
            read_address_kirby = ((19'd15 - (ball_y_pos - draw_y_pos)) << 8)  + (19'd207 - (ball_x_pos - draw_x_pos));
        end
    5'b01010:  // suck right
        begin
            read_address_kirby = ((19'd15 - (ball_y_pos - draw_y_pos)) << 8)  + (19'd239 - (ball_x_pos - draw_x_pos));
        end
    5'b01101:  // left jump 1
        begin
            read_address_kirby = ((19'd79 - (ball_y_pos - draw_y_pos)) << 8)  + (19'd16 - (ball_x_pos - draw_x_pos));
        end
    5'b01110:  // right jump 2
        begin
            read_address_kirby = ((19'd79 - (ball_y_pos - draw_y_pos)) << 8)  + (19'd47 - (ball_x_pos - draw_x_pos));
        end
    default: //stop
        begin
            read_address_kirby = ((19'd15 - (ball_y_pos - draw_y_pos)) << 8)  + (19'd15 - (ball_x_pos - draw_x_pos));
        end
endcase
```

The address of the sprite is feeded to the address wire for kirby. The main address would be activated once scanner is in the kirby figure. The kriby figure is determined by a circle which owns the size of kirby and a center.

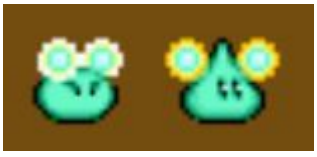
```
int DistX, DistY, Size;
assign DistX = DrawX - Ball_X_Pos;
assign DistY = DrawY - Ball_Y_Pos;
assign Size = kirby_size;

always_comb begin
    if ( ( DistX*DistX + DistY*DistY) <= (Size*Size) )
        is_ball = 1'b1;
    else
        is_ball = 1'b0;
end
```

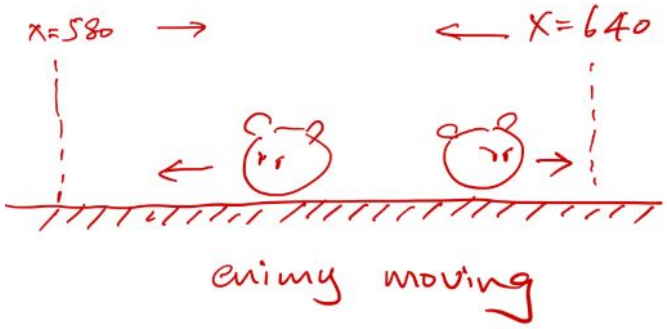
Enemies & Collision Detection

There are two kind of enemies. The first kind of enemies is the red kirby below.

The second kind of enemies is the thunder monster below.



The red kirby is moving left and right in a cycle. The red kirby is also controlled by a FSM consisting four states: “leftstate1, leftstate2, rightstate1, rightstate2”. The red kirby would have a size and position. The position and size would be compared with the position of Kirby for collision detection. The thounder kirby owns two states “shining, ~shining”. The enemies module is basically a simplified version of kirby. The enimies would move in a range specified by us.



AI: Boss

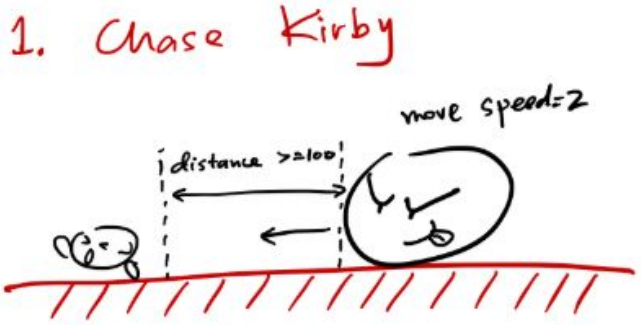
The boss appearing in the second stage. We make a smart A.I that is able to chase, stop, attack and escape. It involve the communication between the FSM of boss and FSM of the kirby. A complicated FSM is designed for the boss. In each state, the position from Kirby and position from Boss is compared.

```
always_comb
    begin
    attack_left = 1'b0;
    attack_right = 1'b0;
    next_state = state;
        case(state)
        leftwalk1:
            if(killboss)
            next_state = die;
            else if(counter == 5'd10)
            next_state = leftwalk2;
            else if(left)
            next_state = rightwalk1;
            else if(Ball_X > Ball_X_Pos && (Ball_X - Ball_X_Pos) > range) //chase
            next_state = rightwalk1;
            else if(Ball_X < Ball_X_Pos && (Ball_X_Pos - Ball_X) < range) //attack
            next_state = attackleft;
            else
            next_state = leftwalk1;
        leftwalk2:
            if(killboss)
            next_state = die;
            else if(counter == 5'd20)
            next_state = leftwalk1;
            else if(left)
            next_state = rightwalk1;
            else if(Ball_X > Ball_X_Pos && (Ball_X - Ball_X_Pos) > range) //chase
            next_state = rightwalk1;
            else if(Ball_X < Ball_X_Pos && (Ball_X_Pos - Ball_X) < range) //attack
            next_state = attackleft;
            else
            next_state = leftwalk2;
        rightwalk1:
            if(killboss)
            next_state = die;
            else if(counter == 5'd10)
```

The four scene are explained below.

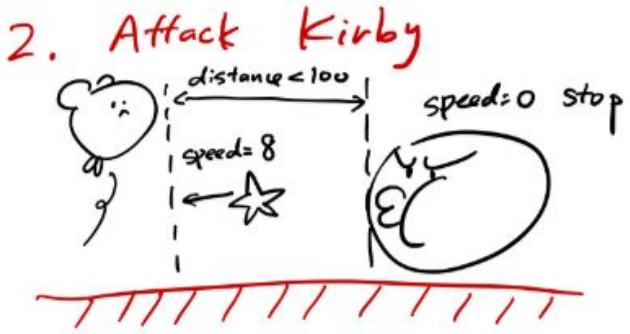
Chase and trace

If the distance between the boss and kirby is bigger than 100 pxiel. The boss would activate the chase state. The boss would actively follow the kirby left or right by comparing the X position. The boss would approach the kirby until the distance is near enough. Then the FSM would jump out of trance state and decided what is the next state.



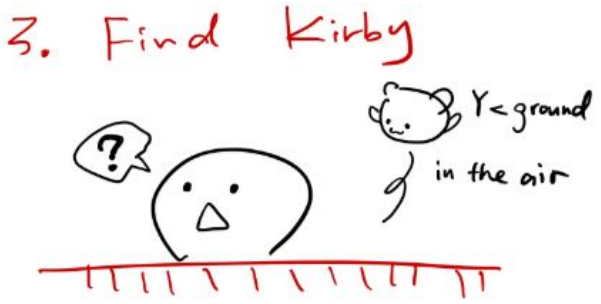
Attack

Once the distance between boss and kirby is smaller than 100 pxiel. If Kirby is in the ground, FSM would activate the attack state. Spark would be emitted from the mouth of the boss. The boss would target the position of the kirby and keep shooting until kirby is either too far or goes into the opposite position.



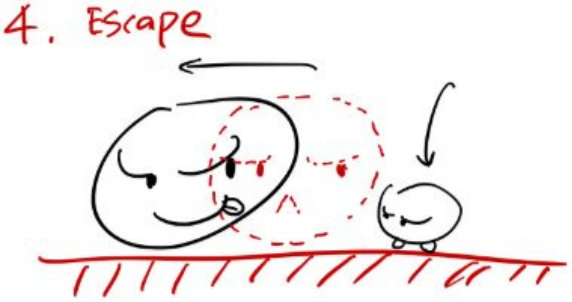
Find

This is the intermediate state in the FSM machine. If our Kirby is in the air, the boss would “think” and try to move to the correct next state. Also all the state would fall into this find state to make the boss react more sensitive and faster. The find state is also the pre state for the escape state.



Escape

The escape state is when the Kirby is in the air, if Kirby is trying to fall on the boss’s head. The boss would escape and move in the opposite direaction in order to avoid the Kirby. Since if Kirby falls on boss’s head. Boss die.



Front-end

We use a “Awesome Face” meme since our boss must be awesome. The movement of the boss contains only three states. In each state, the demanded face is needed. The boss is also a modified Kirby but with bigger size.

Game scenes

Besides the background loaded from SRAM before the game, we also needs the detailed scenes to be put in the front, such the clouds. As other sprites, the clouds are paletterized, and the color indices are stored in on-chip memory. We first hard code the clouds coordinates in the module, color-mapper to depict them. To let Kirby jump and stand above the cloud, we take down the boundary conditions in gravity.sv.

Health Point & Score

HP

Our Kirby owns three point of HP. If all the health point is lost, the Kirby die. With the FSM machine we have full HP, two HP, one HP and zero HP. This acted as a flip flop that saves the HP status of the Kirby. The FSM is controlled by the hurt signal which telles the FSM that Kirby is hitted by a enemies. The state need to be changed. In each state the FSM would output the correct HP bar image to the color mapper. And a die signal to the Kirby module in order to freeze the Kirby and end the game.

```
always_comb
    begin
    next_state = state;
        case(state)
        full:
            if(hurt)
            next_state = twol;
            else
            next_state = full;
        twol:
            if(hurt)
            next_state = twol;
            else
            next_state = two;
        two:
            if(hurt)
            next_state = onel;
            else
            next_state = two;
        onel:
            if(hurt)
            next_state = onel;
            else
            next_state = one;
        one:
            if(hurt)
            next_state = dead;
            else
            next_state = one;
        dead:
            next_state = dead;
        default : ;
        endcase
    end
```

The hurt signal is done by the position compare.

```
always_comb
begin
   diren1_alive2 = 1'b0;
   diren2_alive2 = 1'b0;
   diren3_alive2 = 1'b0;
   if(Ball_X_Pos >= (diren_X_Pos - 10'd50) && Ball_X_Pos <= diren_X_Pos && Ball_Y_Pos == diren_Y_Pos && rightsuck_1)
       diren1_alive2 = 1'b1;
   if(Ball_X_Pos <= (diren_X_Pos + 10'd50) && Ball_X_Pos >= diren_X_Pos && Ball_Y_Pos == diren_Y_Pos && leftsuck_1)
       diren1_alive2 = 1'b1;
   if(Ball_X_Pos >= (diren_X_Pos2 - 10'd50) && Ball_X_Pos <= diren_X_Pos2 && Ball_Y_Pos == diren_Y_Pos2 && rightsuck_1)
       diren2_alive2 = 1'b1;
   if(Ball_X_Pos <= (diren_X_Pos2 + 10'd50) && Ball_X_Pos >= diren_X_Pos2 && Ball_Y_Pos == diren_Y_Pos2 && leftsuck_1)
       diren2_alive2 = 1'b1;
   if(Ball_X_Pos >= (diren_X_Pos3 - 10'd50) && Ball_X_Pos <= diren_X_Pos3 && Ball_Y_Pos == diren_Y_Pos3 && rightsuck_1)
       diren3_alive2 = 1'b1;
   if(Ball_X_Pos <= (diren_X_Pos3 + 10'd50) && Ball_X_Pos >= diren_X_Pos3 && Ball_Y_Pos == diren_Y_Pos3 && leftsuck_1)
       diren3_alive2 = 1'b1;
end
```

Score

```
assign diren_counter = {diren1_alive, diren2_alive, diren3_alive};
```

```
unique case(diren_counter
    3'b000:
        begin
            read_address
        end
    3'b001:
        begin
            read_address
        end
    3'b010:
        begin
            read_address
        end
    3'b100:
        begin
            read_address
```

The score is acted the same way as HP. Basically it is a acending counter that saves status too. If the kirby has kill a enimes. The score would receieve a kill signal and increment the score counter by one. There are totally three enemies in the stage I so max score is three. The score checking is done by concoding the three enemies alive wire into a three bit number and see the situation for all combination.

Keyboard & Multiple Keys

The keyboard is build on the base of lab8. We modified the C code and Qsys so that we could read 4 keys at the same time. Firstly the code provided in lab 8 can read two keycode in address 0x5c. We created a variable that is a 32 bit unsigned integer and get another two keycode from address 0x5f. We concatenated the two 16 bit number into this 32 bit unsigned integer. And then we modified the output from PIC to 32 bit width so that we could get the 4 keycode from the hardware side. Then the keycode could be prossed normally just like lab 8. But one thing to notice, the keycode from the FPGA keyboard is sorted! However, other keyboard is probably not. So we need to take care of the sequence of the keycode.

```
input [31:0]      keycode,          //newly added keypress code
```

Mouse

We use a PS2 mouse in this game. The module of PS2 mouse driver is provided on KT tech 385 website. We integrate the driver into the top level and maker sure that the pin assignments of PS2 ports are already set. The driver detects the mouse's movement and send the clicks and coordinates signals to the top level. This device let us control the character's action by mouse and click the buttons on the top of the screen. The output position and left & right button are used to support motion and menu button clicking.

```
module  mouse ( input         Clk,                // 50 MHz clock
                              Reset,              // Active-high reset signal
                              frame_clk,
                                     leftButton, middleButton, rightButton, /,
                input [9:0]   DrawX, DrawY, cursorX, cursorY,   // Current pixel
                output logic  is_mouse
              );
```

Restart & Speed button

As mention in the mouse section. This feature is implemented based on the PS2 mouse. We hard-code the boundary of the two buttons. And when the mouse click is detected in the range, the restart or speed-up signal will be active. The restart works similarly as RESET button on the board. And the speed button let the user to tune the Kirby's action 10 times faster than the normal mode. We set the speed of the Kirby by let the motion_in to be a big number.

```
else if(cursorX > 10'd48 && cursorX < 10'd96 && cursorY <= 10'd24 && rightButton && start_game && Ball_X_Step == 10'd2) //
begin
    Ball_X_Step <= 10'd10;
    Ball_Y_Step <= 10'd10;
end
```

Audio output

We used the audio driver from the official website and write a interface to control the audio output. In our game we played background music and keep looping in the game. The audio notes "do re mi fa so la si" are tunned by us. We arranged the songe by the seven notes and stored them as wires.

```
    parameter [9:0] d1 = 7'd45;
     parameter [9:0] d2 = 7'd50;
     parameter [9:0] d3 = 7'd55;
     parameter [9:0] d4 = 7'd59;
     parameter [9:0] d5 = 7'd66;
     parameter [9:0] d6 = 7'd73;
     parameter [9:0] d7 = 7'd83;
     parameter [9:0] dx = 7'd00;

int duration, counter;
enum logic [1:0] {Normal} state,next_state;

logic [31:0][6:0] BGM;
assign BGM[0][6:0] = d3;//d3;
assign BGM[1][6:0] = d3;//d3;
assign BGM[2][6:0] = d4;//d4;
assign BGM[3][6:0] = d5;//d5;
assign BGM[4][6:0] = d5;//d5;
assign BGM[5][6:0] = d4;//d4;
assign BGM[6][6:0] = d3;//d3;
assign BGM[7][6:0] = d2;
assign BGM[8][6:0] = d1;
assign BGM[9][6:0] = d1;
assign BGM[10][6:0] = d2;
assign BGM[11][6:0] = d3;
assign BGM[12][6:0] = d3;
assign BGM[13][6:0] = d2;
assign BGM[14][6:0] = d2;
assign BGM[15][6:0] = d3;
assign BGM[16][6:0] = d3;
assign BGM[17][6:0] = d4;
assign BGM[18][6:0] = d5;
assign BGM[19][6:0] = d5;
assign BGM[20][6:0] = d4;
assign BGM[21][6:0] = d3;
assign BGM[22][6:0] = d2;
assign BGM[23][6:0] = d1;
assign BGM[24][6:0] = d1;
assign BGM[25][6:0] = d2;
assign BGM[26][6:0] = d3;
assign BGM[27][6:0] = d2;
assign BGM[28][6:0] = d1;
assign BGM[29][6:0] = d1;
assign BGM[30][6:0] = dx;
assign BGM[31][6:0] = dx;
```

We firstly initialize the driver. And start outputting the song to the left channel port once we receive the initial finish singal.

```
state <= next_state;

    if(Init_Finish)
        begin
            duration <= duration + 1;
            if(data_over)
            begin
                RDATA <= RDATA + BGM[counter][6:0];
            end
            else//DO
            begin
                Init <= 1'b0;
                LDATA <= LDATA;
                if(counter < 32)
                begin
                    counter <= counter + duration/100000;
                    duration <= duration%100000;
                end
                else
                begin
                    counter <= 0;
                end
            end
        end
```

Modules Description

Module: lab8 (lab8.sv)
Inputs: CLOCK_50, OTG_INT, [3:0] KEY, AUD_BCLK,AUD_ADCDAT, AUD_DACLRCK, AUD_ADCLRCK
Outputs: VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK, [1:0] OTG_ADDR, [1:0] DRAM_BA, [3:0] DRAM_DQM, [6:0] HEX0, [6:0] HEX1, [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, [12:0] DRAM_ADDR, [19:0] SRAM_ADDR, SRAM_UB_N, SRAM_LD_N, SRAM_CE_N, SRAM_OE_N, AUD_DACDAT, I2C_SDAT,I2C_SCLK,AUD_XCK
InOut: [15:0] OTG_DATA, [31:0] DRAM_DQ, [15:0] SRAM_DQ
Description: This is the top level of the project, which is developed according to the top level of lab8. (That's why we forgot to rename this file.) This module integrates the game logic modules, the color mapper module, VGA controller, audio_interface, hpi_to_intf interface, NiosII system, mouse driver and Hex driver. This module wraps all the modules built on FPGA together and output and receive signals through the assigned pins in order to communicate with other hardwares on DE2-115.
Purpose: This module is the top entity to wrap all the System Verilog modules. It is responsible for wiring up all the components, coordinating different functionalities and communicating with other hardwares.

Module: hpi_io_intf (hpi_io_intf.sv)
Inputs: Clk, Reset, from_sw_r,from_sw_w,from_sw_cs, [1:0] from_sw_address, [15:0] from_sw_data_out
Outputs: OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N, [1:0] OTG_ADDR, [15:0] from_sw_data_in
InOut: [15:0] OTG_DATA

Description: This module is the interface between NIOS II and EZ-OTG chip. EZ-ORG chip is responsible for receiving and storing the signals and data from USB port. Signals transmitted by USB port are primarily processed by the EZ-OTG chip and is waiting to be polled by SOC on NIOS II. This module creates a tri-state buffer to temporarily store the data and address for the chip to ensure the two-way data transmission.
Purpose: Interface that contains a tri-state buffer to guarantee the data transmission between NIOS II system and EZ-OTG chip.

Module: HexDriver (HexDriver.sv)
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This is a hexadecimal display driver. This module is directly borrowed from lab8 and only used for debugging propuse in this lab.
Purpose: This module change 3 bit input into 7 bit hexadecimal LED display signal.

Module: color_mapper (Color_Mapper.sv)
Inputs: Clk, is_ball, is_diren, is_diren2, is_boss, is_mouse, is_spark, attack_left, attack_right, Reset, frame_clk, rightsuck_1, leftsuck_1, stage_one,[3:0] hp_action,[4:0] kirby_action, diren_action, diren_action2, boss_action, [7:0] data_Out,[9:0] cursorX, cursorY, DrawX, DrawY, Ball_X_Pos, Ball_Y_Pos, boss_X_Pos, boss_Y_Pos, diren_X_Pos, diren_Y_Pos, diren_X_Pos2, diren_Y_Pos2, spark_X_Pos, spark_Y_Pos, [31:0] keycode,
Outputs: [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, [19:0] SRAM_ADDR, [18:0] read_address, hurt
Inout: [15:0] SRAM_DQ,
Description: The two most important input signals for this module are drawX and drawY. They are provided from VGA controller. After receiving every coordinates on the screen, color-mapper then searches and calculated the RGB value for that specific pixel and send it back to VGA controller. This module can receive data from both on-chip memory and SRAM. The color palette indices of the sprites are stored in on-chip memory while the background colors are stored in SRAM. Each figures and widgets in the game have their own position information sent to this module so that color mapper can easily tell which figure should enlight which pixels on screen.
Purpose: This module translates all the game figures, statues and positions into graphic information, and provides the RGB information for every pixel on VGA.

Module: color_extract.sv modules (incluing color_extract.sv, color_extract_diren.sv, color_extract_boss.sv, color_extract_hp.sv, color_extract_bg,)
Inputs: [9:0] DrawX, DrawY, Ball_X_Pos, Ball_Y_Pos; [4:0] kirby_action
Outputs: [18:0] read_address_kirby
Description: All the modules listed above serve the same functionality but targeting on different objects. They contains the always_comb blocks that carry out pure logic calculations to map the figures' VGA coordinates to the real addresses on the on-chip memory.
Purpose: The corresponding on-chip memory addresses are calculated according to the given VGA coordinates and the sprite's position on the memory.

Module: shining.sv

Inputs: Clk, Reset, frame_clk,
Outputs: shine_out
Description: This module contains a finite state machine that counts the number of clock cycles and apply two different sprites for that game figure to create a ‘shining effect’.
Purpose: This is a small FSM that controls the shining effects of a small figure in the first stage.

Module: ball (ball.sv)
Inputs: Clk, Reset, frame_clk, hurt, leftButton, rightButton, [9:0] DrawX, [9:0] DrawY, [9:0] cursorX, [9:0] cursorY, [9:0] boss_X_Pos, [9:0]boss_Y_Pos, [31:0] keycode
Outputs: is_ball, , jump_out, leftsuck_1, rightsuck_1, start_game, stage_one, killboss, [3:0] hp_action, [4:0] kirby_action, [4:0] counter_out, [9:0] Ball_X_Pos, [9:0] Ball_Y_Pos
Description: The module is referred to the ball module in lab8. But we make it way more complicated in this project. It contains various conditions to check the character's current status, whether it suppose to walk, jump or attack. It also coordinated the keycodes from the keyboard and the clicks from the mouse to support the two facilities together. This module uses a finite state machine to calculate Kirby’s coordinates on the screen and the actions that Kirby is going to take in the next clock cycle. This module then send the action signals to walk.sv to pick up the correct animation effect for the character.
Purpose: This module covers all the game logic of the character, Kirby. It control’s Kirby’s next action and movement.

Module: boss.sv
Inputs: Clk, Reset, frame_clk, [9:0] DrawX, [9:0] DrawY, [9:0]Ball_X, [9:0]Ball_Y,
Outputs: is_boss, attack_left, attack_right, [4:0] boss_action, [9:0] boss_X_Pos, [9:0]boss_Y_Pos
Description: This module is the ‘wrapper’ module that outputs the boss’s action and attacks. It also decides where to draw the boss on the screen according to the inputs DrawX and DrawY.
Purpose: This module checks the boundary conditions of the boss in the stage two to let it bounce back and forth on the ground.

Module: walk_boss.sv
Inputs: frame_clk, Reset, Clk, left, right, killboss,[9:0] Ball_X_Pos, [9:0] Ball_Y_Pos, [9:0] Ball_X, [9:0] Ball_Y,
Outputs:go_left, go_right, attack_left, attack_right, [4:0] boss_action
Description: As walk.sv used for Kirby’s action determinator, this module is used to determine the boss’s action by an finite state machine. It takes the Kirby’s position and makes the enemy always walk and attack towards Kirby by comparing it’s coordinator with Kirby’s.
Purpose: Decides the boss’s action according to the FSM and sends out the necessary actions to the color mapper to depict the boss on the screen.

Module:diren.sv (including diren2.sv, diren3.sv)
Inputs: Clk, Reset, frame_clk, [9:0] DrawX, [9:0] DrawY
Outputs: is_diren, [4:0] diren_action, [9:0] diren_X_Pos, [9:0] diren_Y_Pos

Description: Theses three modules controls the three enemy's actions seperately. But they all follow the same control logic. By comparing with DrawX, DrawY and diren_X_Pos and diren_Y_Pos, the module is able to decide which pixels on VGA should be drawn as the enemies. It also allows the enemies to walk back and forth along the pre-defined path.
Purpose: It controls the three enemies' actions and movements and sent out the corresponding signals to color-mapper.

Module: spark.sv
Inputs: Clk, Reset, frame_clk, attack_left, attack_right, killboss, [9:0] boss_X_Pos, [9:0]boss_Y_Pos,[9:0] DrawX, [9:0] DrawY
Outputs: is_spark, [9:0] spark_X_Pos ,[9:0] spark_Y_Pos
Description: This module controls the star's movement that is emitted by the boss. Boss module send the signal to attack to the right or the left to this module to let the star flying towards Kirby. It will fly to the boundary of the screen if it miss Kirby.
Purpose: This module represents the little star emit by the boss. The little star starts from the enemy's position and flying horizontally towards Kirby.

Module: gravity.sv
Inputs: Clk, Reset, frame_clk, stage_one_in, [9:0] DrawX, [9:0] DrawY, jump, [9:0] boss_X_Pos, [9:0]boss_Y_Pos
Outputs: stop, killboss, [9:0] Ball_Y_Motion_in
Description: It is included in the module ball.sv that mainly controls kirby's action. It not only simulates Kirby's jumping motion but also send the signal to its upper level to tell when Kirby reach the ground. We also hardcode in the position of the floating clouds so that Kirby can jump up to the clouds and fall from it when reach the ends.
Purpose: This is the gravity system that controls kirby's vertical movement. It deals with two major conditions: jumping and falling.

Module:walk.sv
Inputs:frame_clk, walk_left, walk_right, jump, Reset, Clk, ground, jump_left, jump_right,suck_left, suck_right, hurting, die,
Outputs:walk_left_1, walk_left_2, walk_right_1, walk_right_2, midjump_1, leftjump_1, leftjump_2, rightjump_1, rightjump_2, sleep_1, leftsuck_1, rightsuck_1, [4:0] counter_out, freeze
Description: This module takes the action signal generated by ball. sv that feed the signals into different cases and conditions in the finite state machine. For example, both walk and jump needs two pictures to be switched in between. We set a counter to coordinate with the FSM to create the animation effects. The current action may also be interrupted by higher-priority cases such as being attacked or die. This module sends out the desired action codes to ball.sv again to let color-mapper loads the sprites.
Purpose: This module renders Kirby's movement, which is responsible for the animation effect.

Module: hp.sv
Inputs: Clk, Reset, frame_clk, [9:0] DrawX, [9:0] DrawY
Outputs: [3:0] hp_action, hurting, die

Description: This is the simple finite state machine that loads the picture representing Kirby's health bar. The finite state machine can also keep track of Kirby's rest lives to tell when Kirby is hurt or dead and send out the signals to the upper modules.
Purpose: This module creates the health bar for Kirby. Kirby has three lives in total to bear three attacks from the enemies. When Kirby is attacked third time, the health bar will be empty and the game is over.

Module: VGA_controller.sv
Inputs: Clk, Reset, VGA_CLK,
Outputs: VGA_BLANK_N,VGA_SYNC_N, [9:0] DrawX, [9:0] DrawY, VGA_HS, VGA_VS
Description: This is the provided module directly come from lab 8. It is a steady finite state machine that follows the clock and VGA clock signals to scan every pixel on the VGA monitor one by one. DrawX and DrawY are the cordinates of the current lightened pixel on the screen.
Purpose: It sends out stead VGA pulse to the VGA monitor to tell when and which pixel is supposed to be lighten.

Module: frameRAM.sv
Inputs:[7:0] data_In, [18:0] write_address,[18:0] read_address, we, Clk,
Outputs:[7:0] data_Out
Description: This module first load the palettelized sprites into on-chip memory. And let the color-mapper directly get the data related to the RGB color indices of the palette according to the memory address.
Purpose: This module loads a file to on-chip memory and let other modules directly access the memory contents based on the memory address.

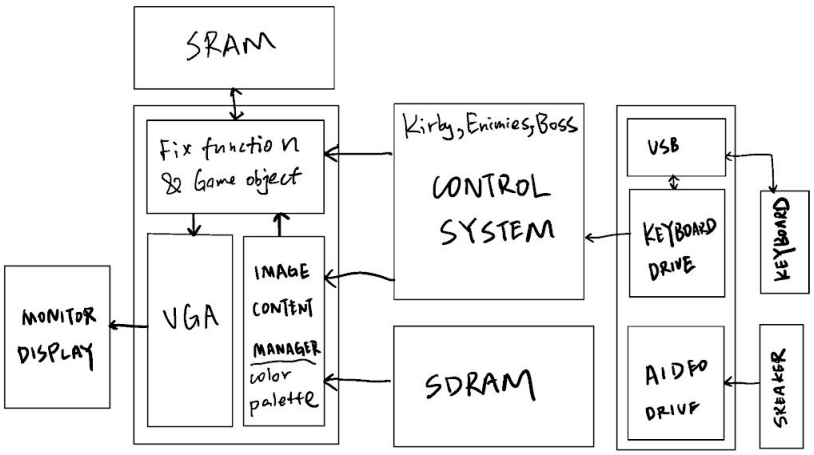
Module:mouse.sv
Inputs: Clk, Reset, frame_clk, leftButton, middleButton, rightButton, [9:0] DrawX, [9:0] DrawY, cursorX, cursorY,
Outputs: is_ball
Description: This module takes the coordinates of the mouse and depict a dot on the screen.
Purpose: A GUI to depict the mouse on screen.

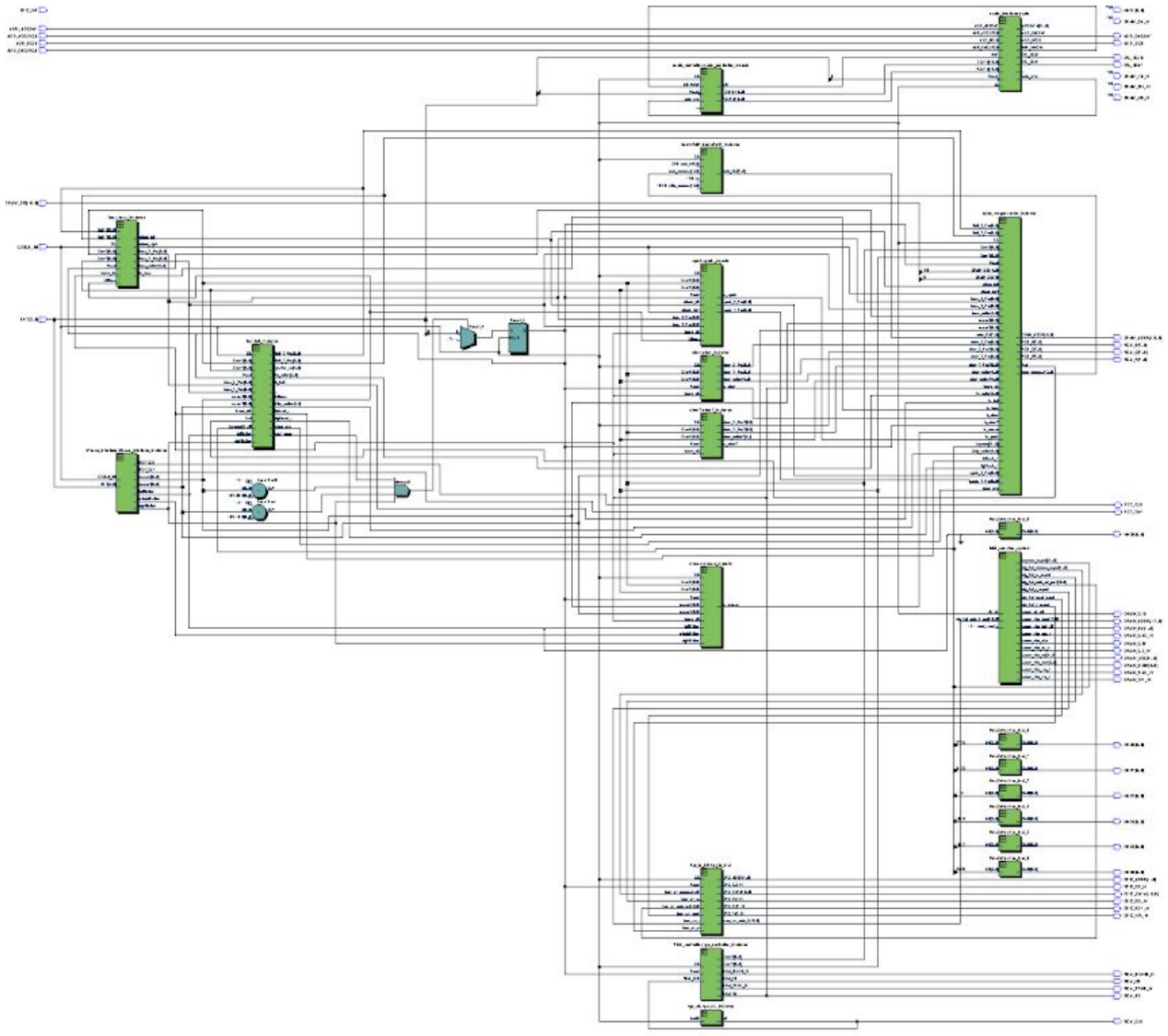
Module: Mouse_interface.sv
Inputs: CLOCK_50, [3:0] KEY,
Outputs: leftButton, middleButton, rightButton, [9:0] cursorX, [9:0] cursorY
Inout: PS2_CLK, PS2_DAT,
Description: The Inout ports PS2_CLK, PS2_DAT are already connected to the physical PS2 ports by the assigned pins. this module can detects the motion and click of the mouse and sends out the signals whenever needed.
Purpose: This is the driver for the PS2 mouse and output the mouse actions to its upper level.

Module: audio_controller.sv
Inputs: Init_Finish, Reset, Clk, data_over, frame_clk
Outputs:[15:0] LDATA,[15:0] RDATA, Init

Description: We hard-code each pitch this module, and put all the pitches in sequence to create an audio clip by ourselves. So when it is loaded into FPGA, the audio clip will be played in loop.
Purpose: This module contains a simple audio clip that is manually coded into. It outputs the sounds through the audio port on FPGA in collaborate with audio_interface.vhd(, which is provided on the course website.)

Block Diagram





This is the block diagram of our circuit generated by Quartus.

Design Procedure

We first did a brainstrom and make a proposal. The kirby game needs tons of graphic and controls. So we decided to let game logic to receiving all the game logic. And them use FSMs to communicate between the game back end to the image front end. After we finished the backend. The game has motion and stage. We started to make pictures. We firstly decided to use on-chip memory for the sprites since it requires the fastest response. And put the background into the SRAM because the background need to be outputted always which requires a different port. Once we have correctly saves the data into the memory. We build address dipather from each character and object. The color mapper has a mange system that select each wire and get the correct data. We also decided to use a color palette because it saves a large amount of memory. With a 48 bit color palette, we are able to

squeeze the size of one pixel from 24 bit to 6 bit but the quality is still very good. Once the framework is done. We keeped adding object to the game stage.

Then additional features such as score, hp, music, mouse and AI are added once the main game is finished. These modules can be separately build so that is why we implement them in the second half of the project. Finally we improve the graphic and add our own personal feature such as humor to improve the user experience of the game. Also corner case are considered during the debug in the end. We tested our game to make sure it is very stable and there would be no problem is user entered illegal command.

Statistics

LUT	8282
DSP	60
Memory	672768
Flip-Flop	2603
Frequency	13.66MHz
Static Power	106.27mW
Dynamic Power	148.18mW
Total Power	359.57mW

For the compile report, one most significant noticement is that the frequency decrease from to 50 MHz to 13.66MHz in this project, which is due to the massive case switches and condition check. But it's still enough for this game to be implemented smoothly. We also notice that the 256*382 sprite picture stored in on-chip memory takes the 17% resources, which in turn shows the limited memory storage of FPGA. As for LUT ,Flip-Flops and power, they are also increase dramatically comparing with our normal labs because of the complexity of this project.

Conclusion

Our final project integrates both peripheral devices and SV logic as an entity. As for hardware devices, we utilize PS2 mouse, audio output through headphones, keyboards and VGA monitors. For game logic which are mainly implemented by SystemVerilog, we realize most of our design based on finite state machine. Signals in each state is transmitted between upper and lower modules to push the game going smoothly. Even this whole project is implemented in hardware description language, we can still sense the convention of dividing the project into front end and back end, to ensure the sanity and efficiency of the work. The game is eventually performed as we expected in the demo.

We are here to thanks for the assistance and instructions provided from out instructor, teaching assistants and other students.