Karan Humpal
Lab 6
06/5/2023
Sign Off: 06/01/23 / 10:31am / 1898
Section A

1 Description

Lab 6 was the toughest lab we've encountered throughout the quarter. In this lab, we explored the use of the VGA port on the BASYS3 board to render images on a monitor. Our task was to develop a game called "Bug Fest". The gameplay involves controlling a slug with BtnU on the BASYS3 board. The goal for the slug is to catch bugs while also landing on or hanging from platforms. The game screen features a blue border, and at the bottom, there's a 'pond'. Contact with this pond results in losing the game. Three platforms move horizontally from left to right. The slug can rest atop a platform when it lands from above or hang from it when hitting the platform from below. However, colliding with the side of a platform concludes the game. The slug experiences continuous downward motion, which we can only counteract when the player presses BtnU The game introduces bugs from either below or above the platforms, adding an extra layer of unpredictability. Furthermore, the platforms are generated in varying sizes. A counter displayed on the hex 7-segment display increments with each bug caught by the slug. The game is started by pressing BtnC on the board.

2 Design

2.1 VGA control

The first module I created was the VGA control module. This is a good starting point because when this module is correctly implemented, the monitor is ready to receive signals from the FPGA, and we can start displaying things on the screen. In order to manipulate the monitor display, two control signals, Hsync and Vsync, along with 12 RGB data signals for each of the 640 x 480 pixels, need to be generated. These signals dictate the color of each pixel, moving systematically in a left-to-right, top-to-bottom fashion, with each pixel requiring one cycle of a provided 25MHz clock. The grid consists of an 'Active Region' (640 x 480 pixels at the top left) which corresponds to the visible screen, while the rest of the grid is off-screen, accounting for the time required for re-positioning between rows and frames. The RGB data signals should be low for pixels outside the Active Region. The Hsync and Vsync signals, low at specific times, ensure synchronization at the start of each row and frame. The horizontal sync (Hsync) is low for the first 96 pixels starting with the 656th pixel in each row, while the vertical sync (Vsync) is

low for the pixels in the 490th and 491st rows only. The VGAcontrol module uses input clock and 10-bit counters (x and y), to iterate over pixels of a 800x525 grid. Pixel addresses are within a 'Active Region' if they fall within specified bounds (640 pixels horizontally and 480 vertically). RGB signals (vgaBlue, vgaRed, vgaGreen) are assigned based on this active region. The synchronization signals, Hsync and Vsync, are controlled by comparing pixel addresses with their corresponding sync regions, specified by hsyncStart, hsyncEnd, vsyncStart, and vsyncEnd. Once the end of a row or frame is reached (tracked by lastpixel_x and lastpixel_y), counters reset, marking the start of the next row or frame.

2.2 Border

The next module I created was the border. The border was an 8 bit wide blue border around the monitor. This was a relatively simple module to create. It works by taking x and y pixel coordinates and a clock signal as inputs, outputting a border signal corresponding to these coordinates. The module sets the border signal high when the pixel coordinates are within specific ranges that outline the screen's edges.

2.2 Pond

The pond module created a blue rectangle on the bottom of the monitor. This was also very simple and done in the exact same way as the border module, by activating the Pond signal when the pixel coordinates (x, y) fall within a certain range.

2.3 Platform

The Platform module generates three moving platforms, with each platform's horizontal position being determined by a counter that increments based on the game's start and frame signals, while not paused. The sizes and spaces (or gaps) between the platforms are defined by the platform_x variable, which is determined by an LFSR (Linear Feedback Shift Register). This results in platforms of varying lengths with dynamic spaces between them. Each platform's position is reset to the right edge of the screen when it reaches the left edge, thereby continuously moving from right to left. The inputs include clk for the system clock, start to initiate the game, pause to pause the game, color_platform to define the platform's color, and x and y for the current screen position. The outputs are platform, which is a boolean representing whether a given pixel is part of a platform, and color_platform, which sets the platform's color.

2.4 Slug

The slug's vertical movement is controlled by a 15 bit counter used in previous labs. The Up input is asserted when the user pushes the button (btnU), as long as the slug is not too low (pond) or too high (top border), not touching the lower part of the platform (denoted by below), and the slug is not dead. The height of the slug is given by a counter, so when Up is asserted, the counter increments, reducing the height and thus making the slug move upwards. The Dw input is asserted when the user is not pushing the button, and it's not touching either the upper or lower part of the platform. When Dw is asserted, the counter decrements, increasing the height and causing the slug to move downwards. This causes the slug to be always falling unless the user provides an external input of btnU. When the slug's height is such that it approaches the top of a platform, the counter's Dw input is deasserted, effectively pausing the slug's downward movement. When the slug approaches the bottom of a platform we pause both Dw and Up of the counter allowing the slug to hang on the bottom of the platform. When the slug is dead we also pause the Up and Dw of the counter.

2.5 Bug

The bug module generates a bug in the game that moves horizontally across the screen from right to left, appearing at different heights. It detects collisions with the slug and flashes after a collision before it disappears. The inputs are clock (the system clock), start (to begin the bug movement), x and y (the current coordinates), and slug (status of the slug). The outputs are bug (the status of the bug), stop (indicates if the bug should stop), and increment (to score points). The frame and frame2 wires represent the end of the screen. If the bug hits the edge, it will reset to the start. The counterUD15L regulates the bug's horizontal movement, with the Up input controlled by the start signal and the frame edges. When start is high and the bug is not flashing (flash is low), the bug moves to the left as the counter increases, thereby decreasing width, which determines the bug's x-coordinate. The bugstate module manages the bug's states, including runtime, flash, increment score, and time-up conditions. It uses the status of the bug and slug as inputs. The EdgeDetector and another counterUD15L handle the scoring system. Whenever a collision is detected (inc is asserted), the score counter increments. The FDRE named toggleHeightFF and the MUX2to1_8bit named height_mux randomly alter the bug's height (y-coordinate). This is by using the last bit of a LFSR to toggle between the high range or low range heights of the bug. The counterUD15L named ate2 controls the bug's flashing mechanism, initiated when the bug collides with the slug. The bug flashes for a period, then disappears. The stop output is asserted during this period to halt the bug.

2.6 Bug State

The bugstate module is the state machine which controls what the bug does. Using D flip-flops, the state transitions are based on conditions such as collisions with a slug, and whether the bug's time is up. These states dictate if the bug is in its normal run state, if it should flash upon a collision, or if it needs to reset after a specific time. These states also decide when the score should increment upon a successful collision with a slug.

State 0 Not Eaten - We wake up in this state and stay in this state so long as there is no collision. We also enter this state as soon as we enter the Reset State.

assign d[0] = (q[0] & ~collision)|q[2];

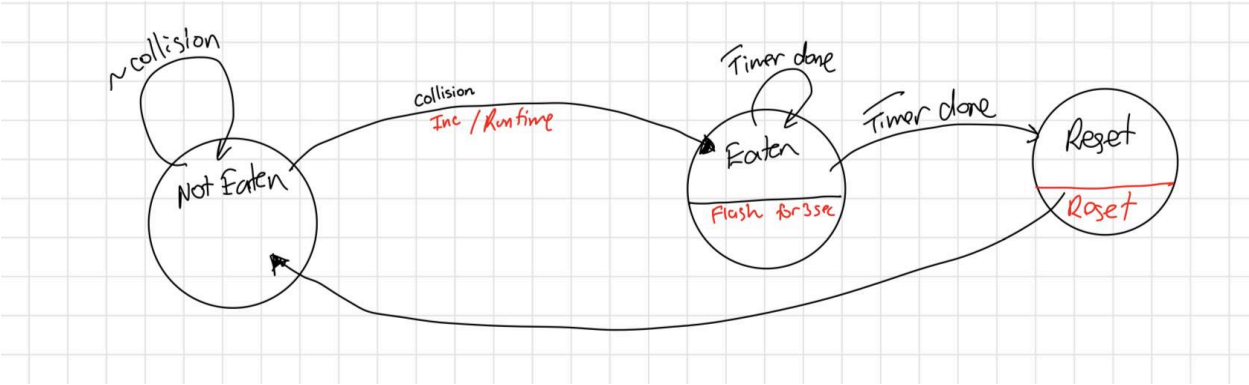
State 1 Eaten - We enter this state when there is a collision, and stay in it until the timer is done.

assign d[1] = (q[0]&collision)| (q[1] & ~Timeup);

State 2 Reset - We enter this state from State 1 when the Timer is finished.

assign d[2] = (q[1] & Timeup);

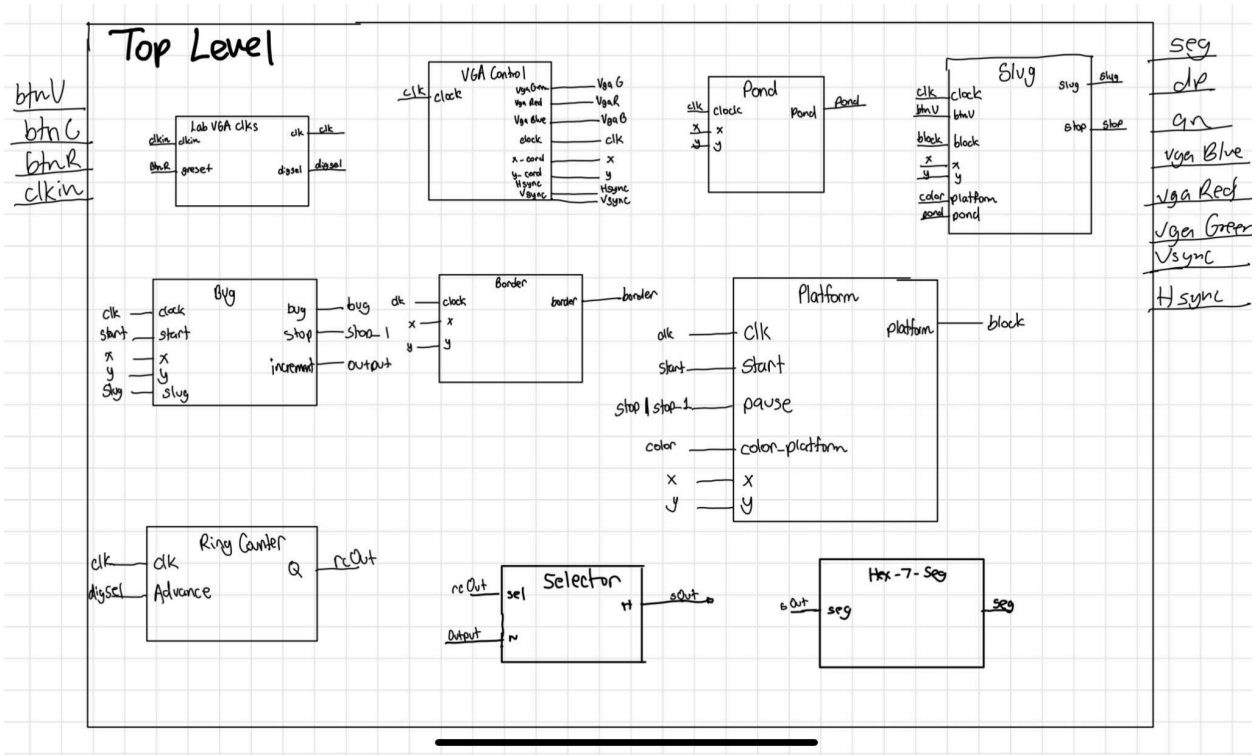
assign Flash = q[1];
assign Runtime = q[1];
assign reset = q[1]&Timeup;
assign Inc = d[1];



2.7 Top

The top module ties together several game elements, with each module contributing to the overall gameplay and visuals. VGAcontrol handles VGA display synchronization and provides the current pixel coordinates used by all other modules to determine their individual

contributions. Pond, Slug, Bug, Border, and Platform modules generate their respective game elements based on these coordinates. The labVGA_clks module generates clock signals necessary for timing control. Slug and Bug modules interact with the Platform module to handle object collisions and movement. The ring_counter and selector modules manage the display of scores on a 7-segment display, while hex7seg converts the score into a format suitable for this display. All the generated elements and scores are then combined and outputted to the VGA display and LEDs.



3 Testing

To make sure my design worked properly, I did a lot of testing during this lab. Since we were using an FPGA, I was able to see errors directly on the monitor. So, instead of doing a lot of simulations, I mostly just made the bitstream and checked the output on the screen.

To help with testing, I used some troubleshooting methods like checking the decimal points (dp) and the LEDs on the FPGA. This was a quick way to see if things were working like they should.

I picked the inputs by guessing what I thought would work best and then testing them out. If something didn't work right, I'd try something else. It was a lot of trial and error. I can't really think of any tricky situations or corner cases that I had to watch out for during this lab. In

general, testing was all about making small changes and checking the results until everything worked as expected.

4 Conclusion

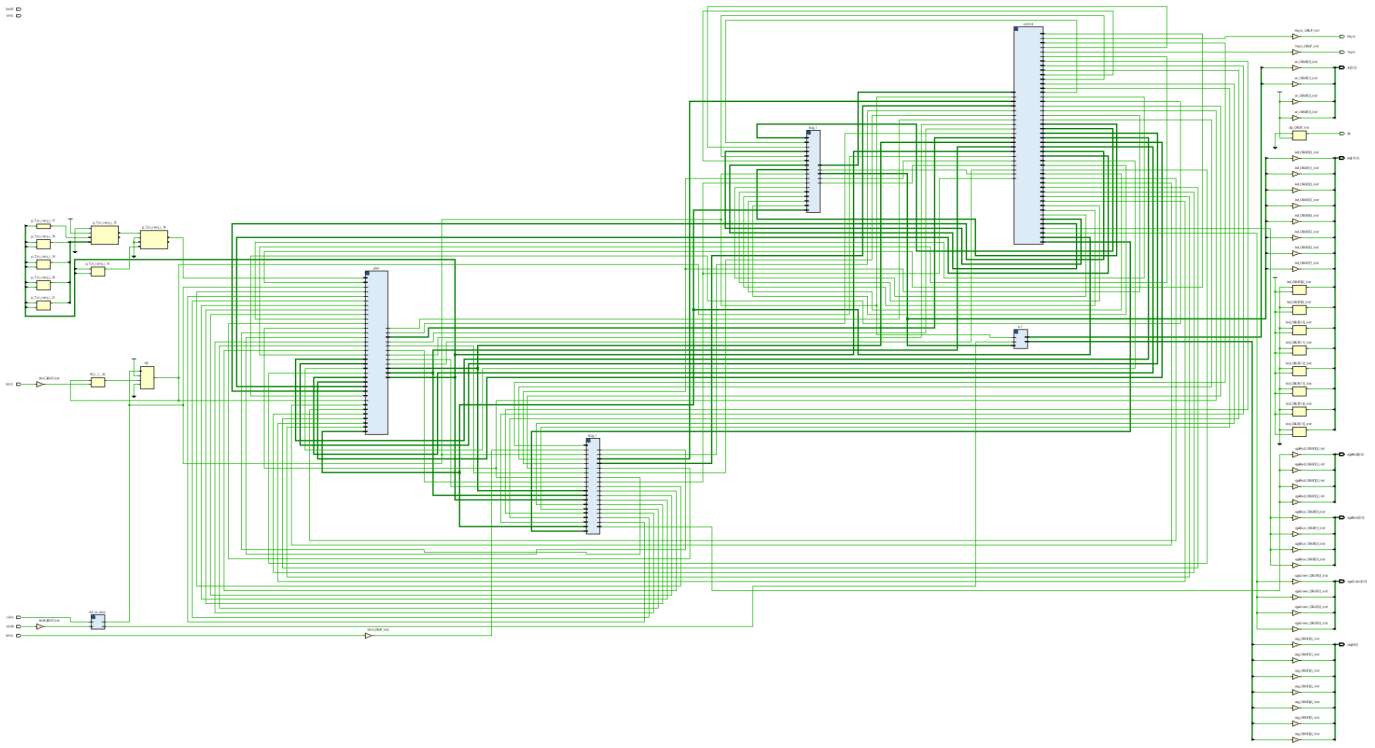
This lab was a big learning experience for me. I put in a lot of hours and learned so much. One of the coolest things I learned was how to program the VGA monitor. I found out how pixels are arranged and how they combine to make the picture we see.

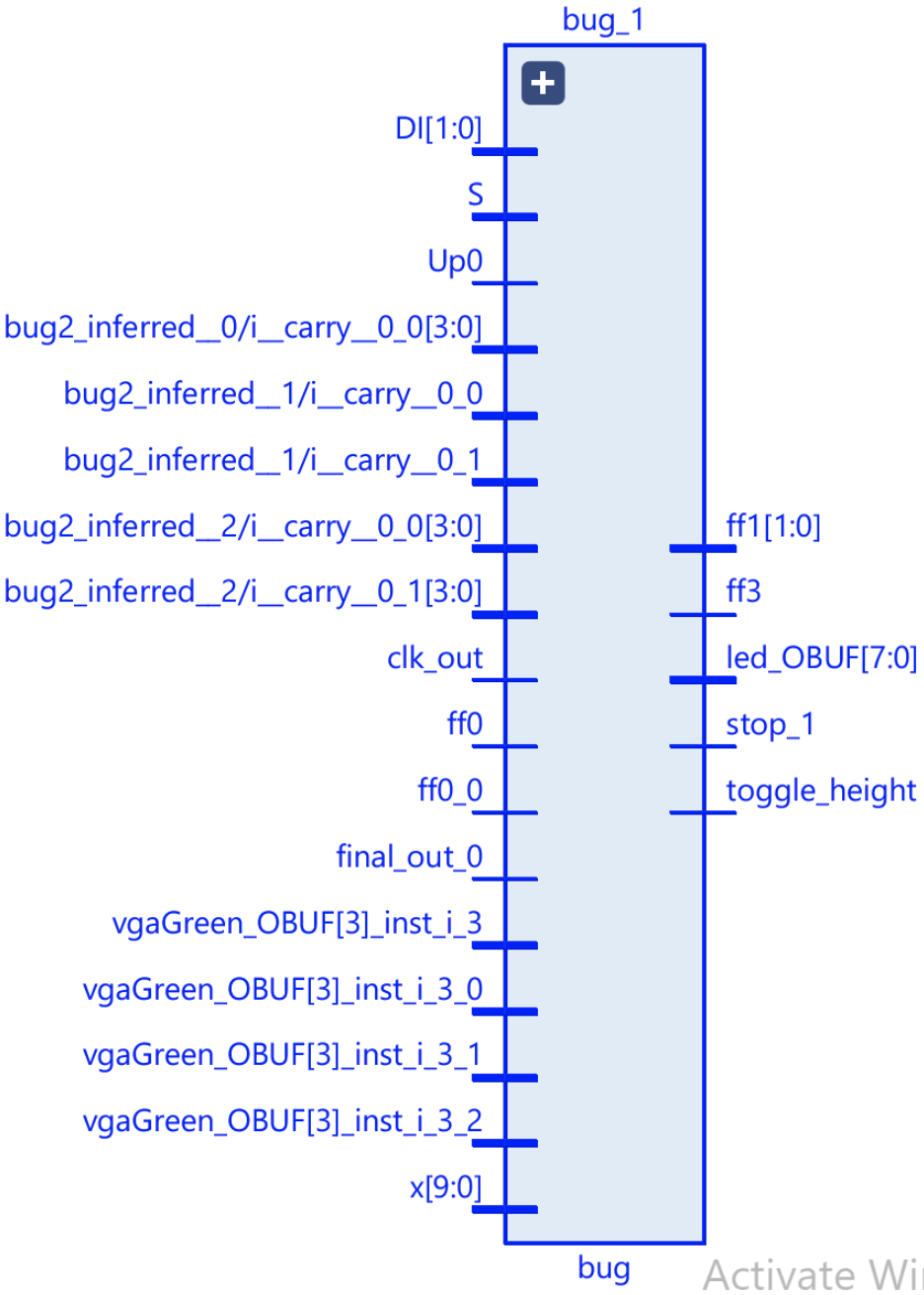
I also got better at programming the FPGA and learned how to use its tools, like the LEDs and dp, to help me when I ran into problems. I had some tough times with getting different parts of my program to work together, but I learned a lot from working through these problems.

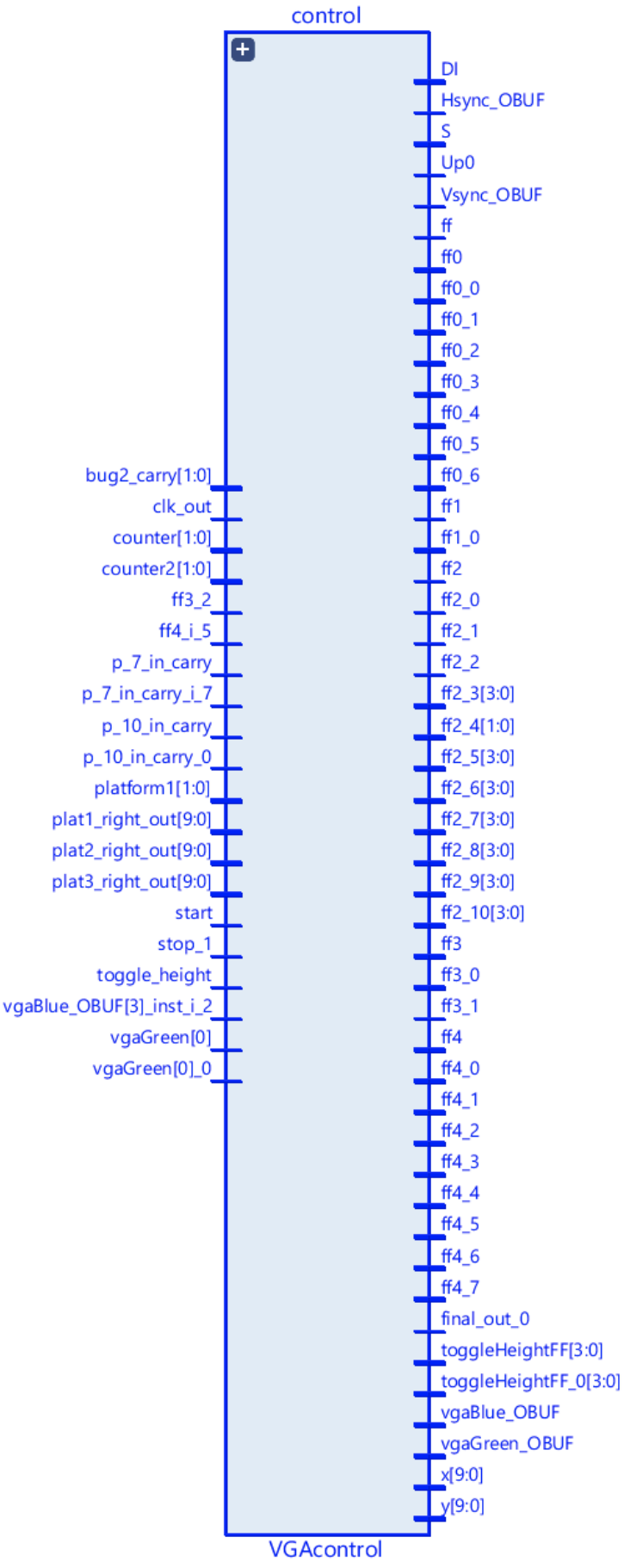
If I had to do this lab again, I would plan more before I started coding. I would also try to write better code that uses less of the FPGA's resources. This lab taught me a lot and has made me a better programmer.

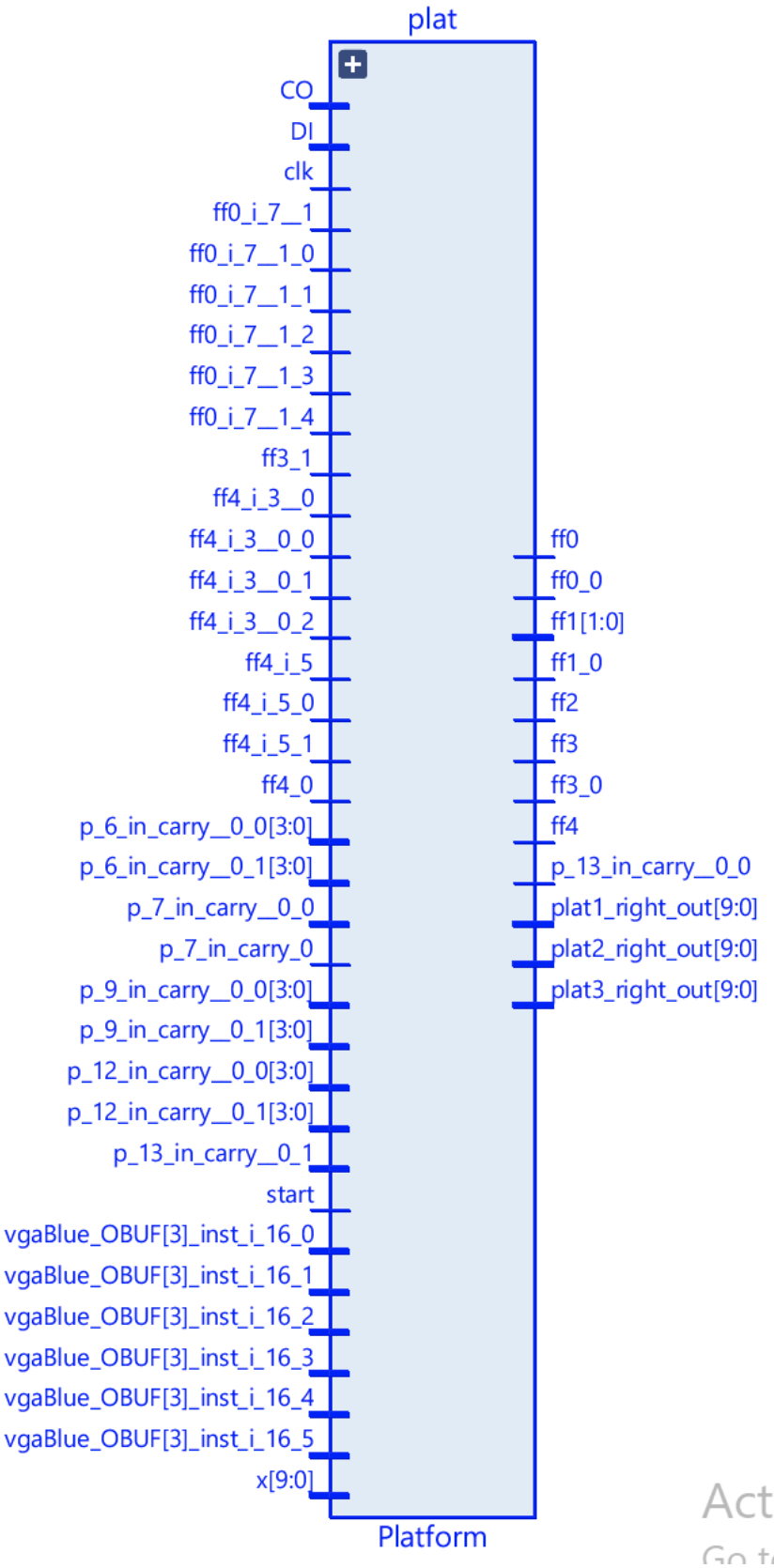
5 Appendix

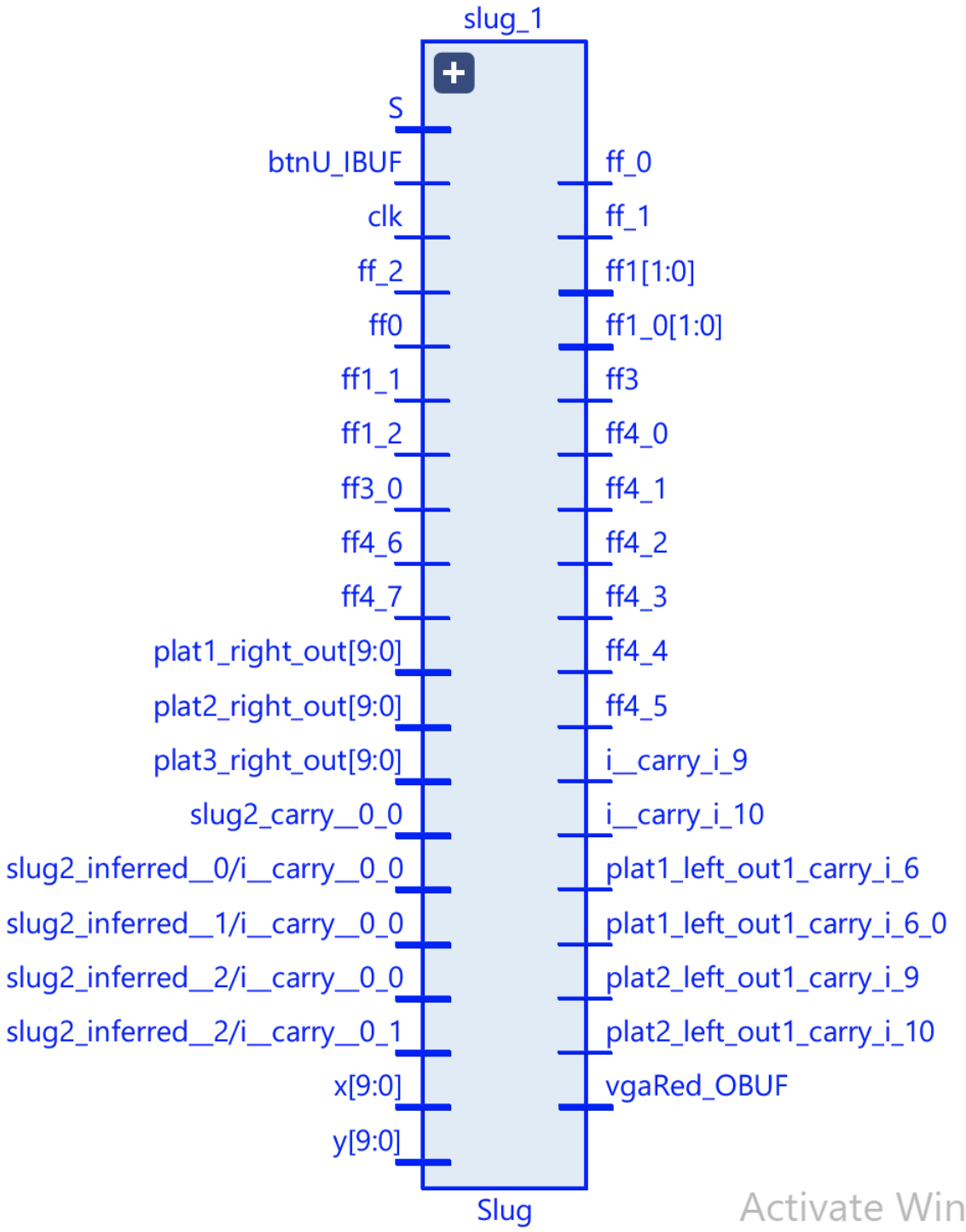
5.1 Schematics

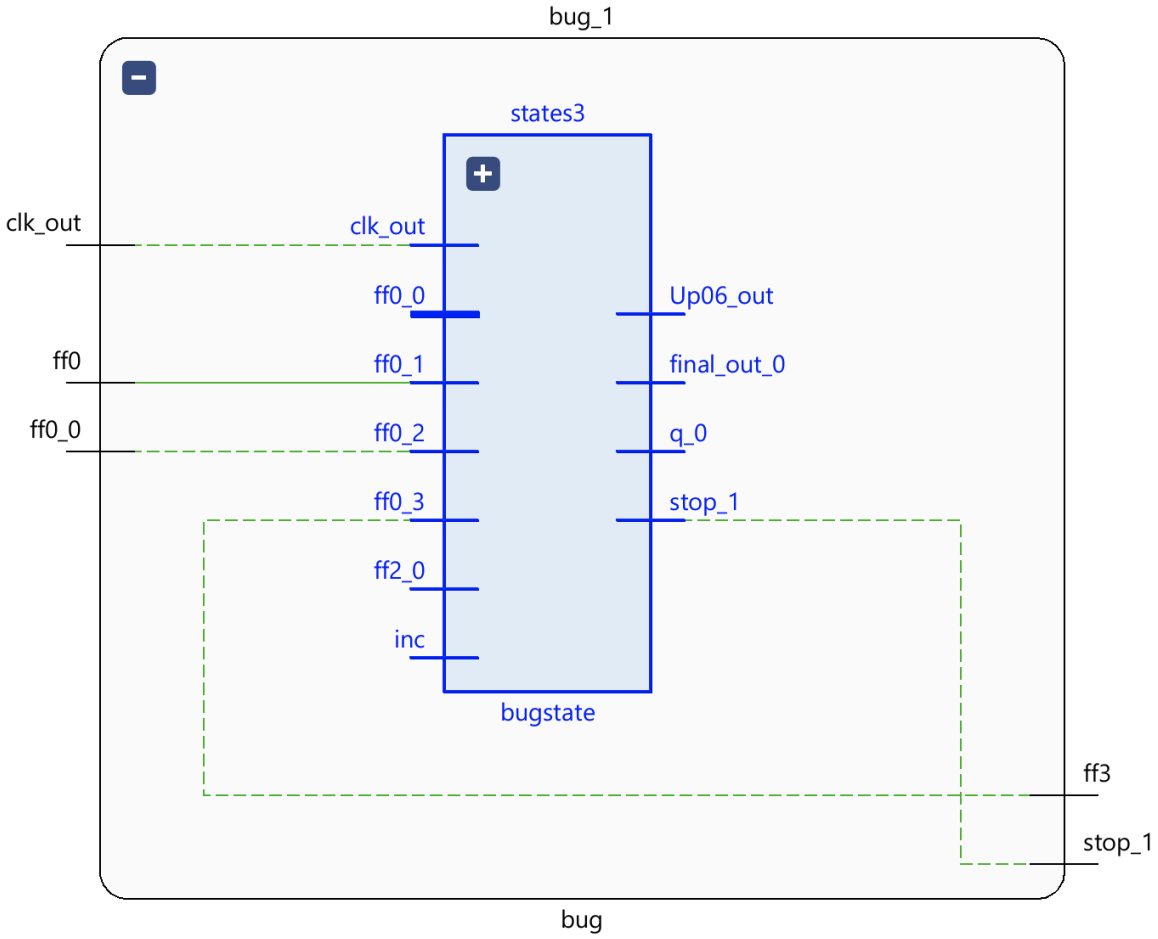





control
DI
Hsync_OBUF
S
Up0
Vsync_OBUF
ff
ff0
ff0_0
ff0_1
ff0_2
ff0_3
ff0_4
ff0_5
ff0_6
ff1
ff1_0
ff2
ff2_0
ff2_1
ff2_2
ff2_3[3:0]
ff2_4[1:0]
ff2_5[3:0]
ff2_6[3:0]
ff2_7[3:0]
ff2_8[3:0]
ff2_9[3:0]
ff2_10[3:0]
ff3
ff3_0
ff3_1
ff4
ff4_0
ff4_1
ff4_2
ff4_3
ff4_4
ff4_5
ff4_6
ff4_7
final_out_0
toggleHeightFF[3:0]
toggleHeightFF_0[3:0]
vgaBlue_OBUF
vgaGreen_OBUF
x[9:0]
y[9:0]
bug2_carry[1:0]
clk_out
counter[1:0]
counter2[1:0]
ff3_2
ff4_i_5
p_7_in_carry
p_7_in_carry_i_7
p_10_in_carry
p_10_in_carry_0
platform1[1:0]
plat1_right_out[9:0]
plat2_right_out[9:0]
plat3_right_out[9:0]
start
stop_1
toggle_height
vgaBlue_OBUF[3]_inst_i_2
vgaGreen[0]
vgaGreen[0]_0
VGAcontrol







5.2 Verilog Source Code

TOP LEVEL

```
module top(
    input btnU,
    input btnD,
    input btnC,
    input btnR,
    input btnL,
    input clkin,
    output [6:0]seg,
    output dp,
    output [15:0] led,
    output [3:0] an,
    output [3:0] vgaBlue,
    output [3:0] vgaRed,
    output [3:0] vgaGreen,
    output Vsync ,
    output Hsync

    );

    wire [7:0] Output;
    assign led[7:0] = Output;
    wire clk, digsel, start,block,stop,stop_1;
    labVGA_clks not_so_slow (.clkin(clkin), .greset(btnR), .clk(clk), .digsel(digsel));
   FDRE #(.INIT(1'b0)) ff2 (.C(clk), .R(1'b0), .CE(1'b1), .D(btnC|start), .Q(start));
    wire [9:0] x,y;
    wire [3:0] color;
    wire [3:0] pond, border,vgaB,vgaR,vgaG, platform, slug,bug;
    VGAcontrol control
(.vgaBlue(vgaG),.vgaRed(vgaR),.vgaGreen(vgaG),.clock(clk),.x_coord(x),.y_coord(y),.Hsync(Hsync),
.Vsync(Vsync));
    Pond pond_1(
    .clock(clk),
    .x(x),.y(y),
    .Pond(pond)
    );
    Slug slug_1(
    .clock(clk),
    .btnU(btnU),
    .stop(stop),
    .x(x),.y(y),
    .slug(slug),
    .block(block),
    .platform(color)
    );
```

```
bug bug_1(
.clock(clk),
.start(start),
.x(x),.y(y),
.bug(bug),
.slug(slug),
.increment(Output),
.stop(stop_1)
);

Border Border_1(
.clock(clk),
.x(x),.y(y),
.border(border)
);


Platform plat (
    .clk(clk),
    .start(start),
    .pause(stop|stop_1),
    .x(x),
    .y(y),
    .platform(block),
    .color_platform(color)
);


wire [3:0] rcOUT;
ring_counter rc1 (
    .clk(clk),
    .Advance(digsel),
    .Q(rcOUT)
);

wire [3:0] sOut;
selector selec (
    .sel(rcOUT),
    .N(Output),
    .H(sOut)
);

hex7seg segg(
    .n(sOut),
    .seg(seg)
);
assign an[2] = 1'b1;
assign an[3] = 1'b1;
assign an[1] = ~rcOUT[1];
```

```
    assign an[0] = ~rcOUT[0];

    //assign dp = |bug & |slug;
    assign vgaGreen = pond | slug | bug ;
    assign vgaBlue = pond | border | color;
    assign vgaRed = slug | color;

endmodule
```

PLATFORM

```
module Platform(
    input clk,
    input start,
    input pause,
    input [3:0] color_platform,
    input [9:0] x,y,
    output platform
);

    wire frame;
    assign frame = (x == 10'd799) & (y == 10'd524);

    wire [10:0] platform1;
    wire [10:0] platform2;
    wire [10:0] platform3;

    wire [7:0] platform_x;
    wire [7:0] lfsr_output;
    wire [7:0] flip_out;

    wire edge_detected;
    EdgeDetector e1(.clk(clk), .btn(start), .edge_detected(edge_detected));


    wire [9:0] plat1_space_left, plat1_space_right, plat1_left_out, plat1_right_out;
    counterUD15L coutn1 (.clk(clk), .Up(start & frame & ~pause), .LD(edge_detected), .Dw(1'd0), .Din(10'd80),
.R(plat1_space_left <= 10'd8), .Q(platform1));
    assign plat1_space_left = 10'd640;
    assign plat1_space_right = plat1_space_left + platform_x;
    assign plat1_left_out = {10{plat1_space_right > platform_x}} & plat1_space_left - platform1|
{10{plat1_right_out <= platform_x}} & 10'd8;
    assign plat1_right_out = plat1_space_right - platform1;

    wire [9:0] plat2_space_left, plat2_space_right, plat2_left_out, plat2_right_out;
    counterUD15L couint2 (.clk(clk), .Up(start & frame & ~pause), .LD(edge_detected), .Dw(1'd0), .Din(10'd380),
.R(plat2_space_left <= 10'd8), .Q(platform2));
    assign plat2_space_left = 10'd640;
    assign plat2_space_right = plat2_space_left + platform_x;
```

```
    assign plat2_left_out ={10{plat2_right_out > platform_x}} & plat2_space_left - platform2 | {10{plat2_right_out
<= platform_x}} & 10'd8;
    assign plat2_right_out = plat2_space_right - platform2;

    wire [9:0] plat3_space_left, plat3_space_right, plat3_left_out, plat3_right_out;
    counterUD15L count3 (.clk(clk), .Up(start & frame & ~pause), .LD(edge_detected), .Dw(1'd0), .Din(10'd640),
.R(plat3_space_left <= 10'd8), .Q(platform3));
    assign plat3_space_left = 10'd640;
    assign plat3_space_right = plat3_space_left + platform_x;
    assign plat3_left_out = {10{plat3_right_out > platform_x}} & plat3_space_left - platform3 | {10{plat3_right_out
<= platform_x}} & 10'd8;
    assign plat3_right_out = plat3_space_right - platform3;


    LFSR random (.clk(clk), .reset(1'b0), .lsfr_out(lfsr_output));
    FDRE #(.INIT(1'b0)) w0 (.C(clk), .CE((plat1_space_right <= 10'd8)), .D(lfsr_output[0]), .Q(flip_out[0]));
    FDRE #(.INIT(1'b0)) w1 (.C(clk), .CE((plat1_space_right <= 10'd8)), .D(lfsr_output[1]), .Q(flip_out[1]));
    FDRE #(.INIT(1'b0)) w2 (.C(clk), .CE((plat1_space_right <= 10'd8)), .D(lfsr_output[2]), .Q(flip_out[2]));
    FDRE #(.INIT(1'b0)) w3 (.C(clk), .CE((plat1_space_right <= 10'd8)), .D(lfsr_output[3]), .Q(flip_out[3]));
    FDRE #(.INIT(1'b0)) w4 (.C(clk), .CE((plat1_space_right <= 10'd8)), .D(lfsr_output[4]), .Q(flip_out[4]));
    FDRE #(.INIT(1'b0)) w5 (.C(clk), .CE((plat1_space_right <= 10'd8)), .D(lfsr_output[5]), .Q(flip_out[5]));
    //FDRE #(.INIT(1'b0)) w6 (.C(clk), .CE((plat1_space_right <= 10'd9)), .D(lfsr_output[6]), .Q(flip_out[6]));
    //FDRE #(.INIT(1'b0)) w7 (.C(clk), .CE((plat1_space_right <= 10'd9)), .D(lfsr_output[7]), .Q(flip_out[7]));

    assign platform_x = 8'd128+flip_out[5:0];

    assign platform = (plat1_left_out < 10'd200 & plat1_right_out > 10'd216) | (plat2_left_out < 10'd200 &
plat2_right_out > 10'd216) | (plat3_left_out < 10'd200 & plat3_right_out > 10'd216);

    assign color_platform = {4{ ((x < plat1_left_out | x > plat1_right_out) & (x < plat2_left_out | x > plat2_right_out)
& (x < plat3_left_out | x > plat3_right_out)) &
    (y >= 10'd200 & y <= 10'd207) & ((start & (x >= 10'd8 & x <= 10'd631))|(~start & (((x >= 10'd120 & x <=
10'd250) | (x >=10'd380  & x <= 10'd520))))))}};


endmodule
```

SLUG

```
module Slug(
    input clock,
    input btnU,
    input block,
    input [9:0] x, y,
    input [3:0] platform, pond,
    output [3:0] slug,
    output stop
    );
```

```
    wire frame,dead2,dead3,frame2;
    wire [9:0] height, counter,counter2,counter3;
    wire [3:0] hitPlatform, hitPond, above, below, collision;
    assign frame = (~|(x^10'd799)&~|(y^10'd524));
    assign frame2 = (~|(x^10'd798)&~|(y^10'd523));
    FDRE #(.INIT(1'b0)) ff (.C(clock), .R(1'b0), .CE(1'b1), .D((|slug & |platform)|(dead2)), .Q(dead2));
    FDRE #(.INIT(1'b0)) ff4 (.C(clock), .R(1'b0), .CE(1'b1), .D((height>=10'd360)|(dead2)|dead3), .Q(dead3));
    assign height = 10'd10-counter;
    counterUD15L ate (
        .clk(clock),
        .Up(btnU&(frame|frame2)&~(height<=10'd8)&~below&~(height>=10'd360)&~dead2),
        .Dw(~btnU&(frame|frame2)&~(height>=10'd360)&~(above|below)&~dead2),
        .R(1'b0),
        .LD(1'b0),
        .Q(counter)
    );
        counterUD15L ate1 (
        .clk(clock),
        .Up(frame&dead2&~(counter2==10'd192)),
        .Dw(1'b0),
        .R(1'b0),
        .LD(1'b0),
        .Q(counter2)
    );
    counterUD15L ate4 (
        .clk(clock),
        .Up(frame&dead3),
        .Dw(1'b0),
        .R(1'b0),
        .LD(1'b0),
        .Q(counter3)
    );
    assign slug ={4{~dead3|(dead3&counter3[4])}}&{4{(((y >= height) & (y <= height+10'd16)) & (x >=
10'd200-counter2 & x <= 10'd216-counter2))}};
    assign stop = (dead2&counter2==10'd192);
    assign hitPlatform = |slug & |platform;
    assign hitPond = slug & pond;
    assign above = ~block & (height == 10'd183);
    assign below = ~block & (height == 10'd208);
    //assign collision = hitPlatform & ((height >= 10'd221) | (height + 10'd16 <= 10'd227));

endmodule
```

BUG

```
module bug(
    input clock,
    input start,
    input [9:0] x, y,
```

```
input [3:0] slug,
output [3:0] bug,
output stop,
output [7:0] increment
);

wire frame, frame2, collision;
assign frame = ~|(x^10'd799)&~|(y^10'd524);
assign frame2 = ~|(x^10'd798)&~|(y^10'd523);
wire [9:0] width, counter, counter2;
assign collision = |bug & |slug;
wire reset, inc, runtime, flash, timeup;

bugstate states3(
 .clock(clock),
 .slug(slug),
 .bug(bug),
 .Timeup(timeup),
 .reset(reset),
 .Flash(flash),
 .Inc(inc),
 .Runtime(runtime)
);

wire inc2;
counterUD15L ate (.clk(clock), .Up(start&(frame|frame2)&~flash), .Dw(1'b0), .R(reset),.LD(1'b0), .Q(counter));
assign stop = flash;
assign width = 10'd700-counter;
counterUD15L ate2 (.clk(clock), .Up(flash&frame), .Dw(1'b0), .R(reset),.LD(1'b0), .Q(counter2));
EdgeDetector up1( //prevents button staying high if we hold it down
.clk(clock),
.btn(inc),
.edge_detected(inc2)
);
counterUD15L ate3 (.clk(clock), .Up(inc2), .Dw(1'b0), .R(1'b0),.LD(1'b0), .Q(increment));
assign timeup = (counter2 == 10'd240);

wire [7:0] height;
wire toggle_height;

wire [7:0] lsfr_out;
LFSRbug  lol (.clk(clock), .reset(reset), .lsfr_out(lsfr_out));

FDRE toggleHeightFF (.C(clock), .R(1'b0), .CE(reset), .D(lsfr_out[7]), .Q(toggle_height));



MUX2to1_8bit height_mux (.s(toggle_height), .f(height));
```

```
   assign bug ={4{~flash|(flash&counter2[3])}}&{4{(((y >= height) & (y <= height+10'd8)) & (x >= width & x <= width+10'd8))}};

endmodule
```

POND

```
module Pond(
   input clock,
   input [9:0] x,y,
   output [3:0]Pond

   );

   assign Pond = {4{(y <= 10'd471) && (y >= 10'd375) & (x >= 10'd8) && (x <= 10'd632)}};

endmodule
```

BORDER

```
module Border(
   input clock,
   input [9:0] x, y,
   output [3:0]border
   );



   assign border = {4{((((y >= 10'd471) & (y <= 10'd479)) & ((x>= 10'd0)&(x<=10'd639)))|(((x >= 10'd0) & (x <= 10'd7)) & ((y >= 10'd0) & (y <= 10'd479))))|((((y >= 10'd0) & (y <= 10'd479)) & ((x>= 10'd632)&(x<=10'd639)))|(((x >= 10'd0) & (x <= 10'd639)) & ((y >= 10'd0) & (y <= 10'd7)))))}};


endmodule
```

VGA CONTROL

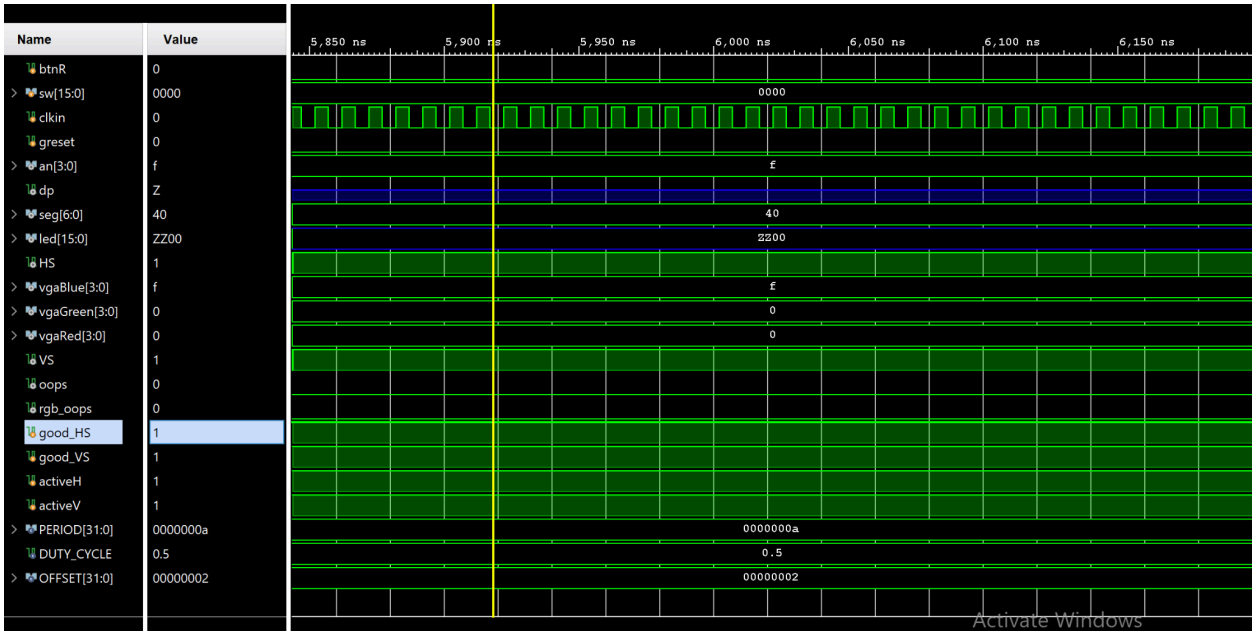
```
module VGAcontrol(
   output [3:0] vgaBlue,
   output [3:0] vgaRed,
   output [3:0] vgaGreen,
   input clock,
   output [9:0] x_coord,
   output [9:0] y_coord,
   output Hsync, Vsync

   );
```

```
    wire [9:0] hsyncStart, hsyncEnd,vsyncStart, vsyncEnd;

    assign hsyncStart = 10'd655; //Hsync Low Region
    assign hsyncEnd = 10'd750;
    assign vsyncStart = 10'd489; //Vsync Low Region
    assign vsyncEnd = 10'd490;

    wire active_region;
    wire [14:0] x_pixel_addr, y_pixel_addr;
    wire active_x, active_y;
    assign active_x = (x_pixel_addr < 10'd640); //doesnt include 640
    assign active_y = (y_pixel_addr < 10'd480); //doesnt include 480

    assign active_region = active_x && active_y;
    assign vgaBlue = 4'b1111 & {4{active_region}};
    assign vgaRed = 4'b1111 & {4{active_region}};
    assign vgaGreen = 4'b1111 & {4{active_region}};


    wire [9:0] lastpixel_x, lastpixel_y;
    assign lastpixel_x = 10'd799;
    assign lastpixel_y = 10'd524;

    counterUD15L x (
        .clk(clock),
        .Up(1'b1),
        .Dw(1'b0),
        .LD(x_pixel_addr ==lastpixel_x), //t after each horizontal line of 800 pixels (from 0 to 799), the x counter will
be reset to 0.
        .Din(15'b000000000000000),
        .Q(x_pixel_addr));

    counterUD15L y (
        .clk(clock),
        .Up(lastpixel_x == x_pixel_addr),
        .Dw(1'b0),
        .LD(lastpixel_y == y_pixel_addr & x_pixel_addr ==lastpixel_x),
        .Din(15'b000000000000000),
        .Q(y_pixel_addr));

    assign Hsync = (x_pixel_addr < hsyncStart) || (x_pixel_addr > hsyncEnd);
    assign Vsync = (y_pixel_addr < vsyncStart) || (y_pixel_addr > vsyncEnd);
    assign x_coord = x_pixel_addr [9:0];
    assign y_coord = y_pixel_addr [9:0];



endmodule
```

BUG STATE

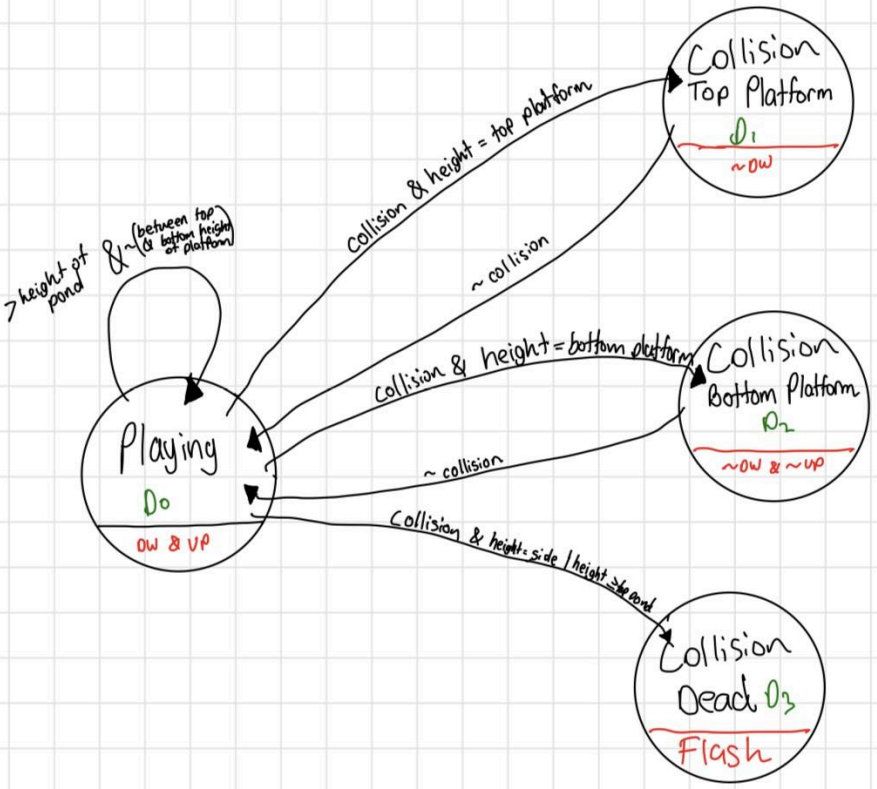
```
module bugstate(
    input clock,
    input [3:0] slug,
    input [3:0]bug,
    input Timeup,
    output reset,
    output Flash,
    output Inc,
    output Runtime

    );

    wire [2:0] d;
    wire [2:0] q;
    FDRE #(.INIT(1'b1)) ff0 (.C(clock), .CE(1'b1), .R(1'b0), .D(d[0]), .Q(q[0]));
    FDRE #(.INIT(1'b0)) ff1 (.C(clock), .CE(1'b1), .R(1'b0), .D(d[1]), .Q(q[1]));
    FDRE #(.INIT(1'b0)) ff2 (.C(clock), .CE(1'b1), .R(1'b0), .D(d[2]), .Q(q[2]));

    wire collision;
    assign collision = |bug & |slug;
    assign d[0] = (q[0] & ~collision)|q[2];
    assign d[1] = (q[0]&collision)| (q[1] & ~Timeup);
    assign d[2] = (q[1] & Timeup);

    assign Flash = q[1];
    assign Runtime = q[1];
    assign reset = q[1]&Timeup;
    assign Inc = d[1];
    //assign Runtime = collision;


endmodule
```

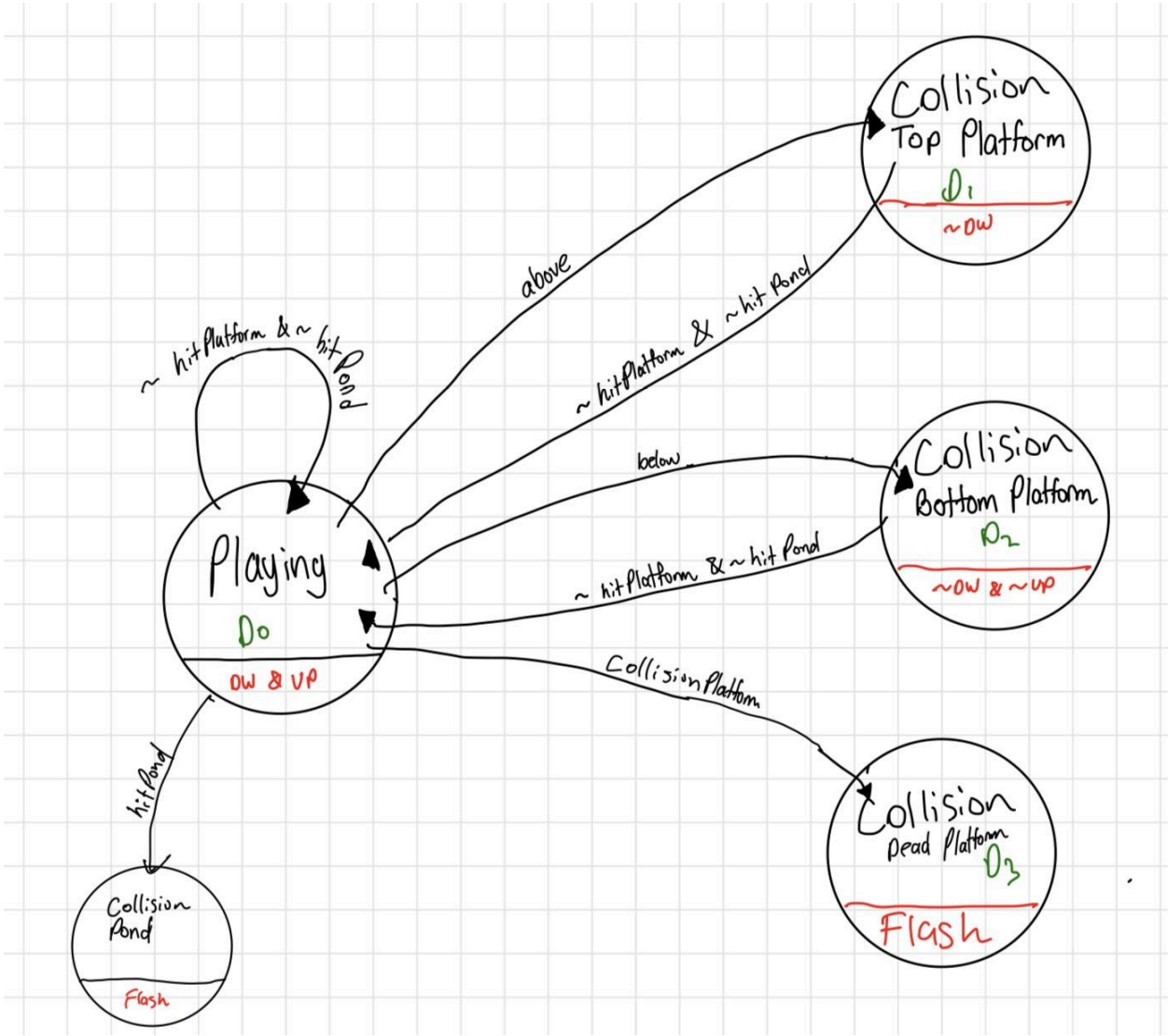
Hsync and Vsync Simulation



Other State Machines that did not work

Slug


Collision
Top Platform
D1
~OW
Collision & height = top platform
~ collision
> height of pond
& ~(between top & bottom height of platform)
Collision & height = bottom platform
Collision
Bottom Platform
D2
~OW & ~UP
Playing
D0
OW & UP
~ collision
Collision & height = side / height = bottom
Collision
Dead D3
Flash


Collision
Top Platform
D1
~OW
above
~ hitPlatform & ~ hitPond
~ hitPlatform & ~ hitPond
below
Collision
Bottom Platform
D2
~OW & ~UP
Playing
D0
OW & UP
~ hitPlatform & ~ hitPond
Collision Platform
hitPond
Collision
Dead Platform
D3
Flash
Collision
Pond
Flash