Oregon State University

ECE 271

Final Design Project

Benjamin Geyer

Christien Hotchkiss

Phi Luu

November 30th, 2018

Contents

1 Introduction

The purpose of the final project is to fully design and simulate a VGA controller. A VGA, or video graphics array, can be used to display a wide range of resolutions where each pixel is controlled by 3 analog pins corresponding to Red, Green, and Blue. Using system verilog, an extremely powerful hardware description language, we created modules to decode data transmitted through SNES/NES, PS/2, and Infrared, and outputted the data to a VGA controller that displays an image. Our team tested our modules using both ModelSim and the DE10-Lite. We were able to simulate and robustly test our modules using ModelSim. ModelSim creates a simulation that exercises all possible states of the design to ensure that each unique input scenario is handled properly according to the system verilog code. Using this program is also a much more efficient way to debug issues in the design and is far more practical than testing each scenario using a board such as an FPGA. While not specifically required for this project, our team also decided to test our design using the DE10-Lite, which is a FPGA provided to us through Oregon State University. By connecting the DE10-Lite with a monitor using a VGA cable, we were able to determine that our design correctly outputs VGA images, which aligned with our results from simulation. The main goal of this project is to successfully handle different types of inputs that are wired to control the output as a VGA image.

1.1 Project Goals

- Effectively decode data transmitted through SNES/NES, PS/2, and Infrared
- Control a VGA image output for each type of input
- VGA image of a box should stop moving and change color when it hits the edge of the display

Figure 1.1.1: NES controller

Figure 1.1.2: PS/2 keyboard

Figure 1.1.3: PS/2 port

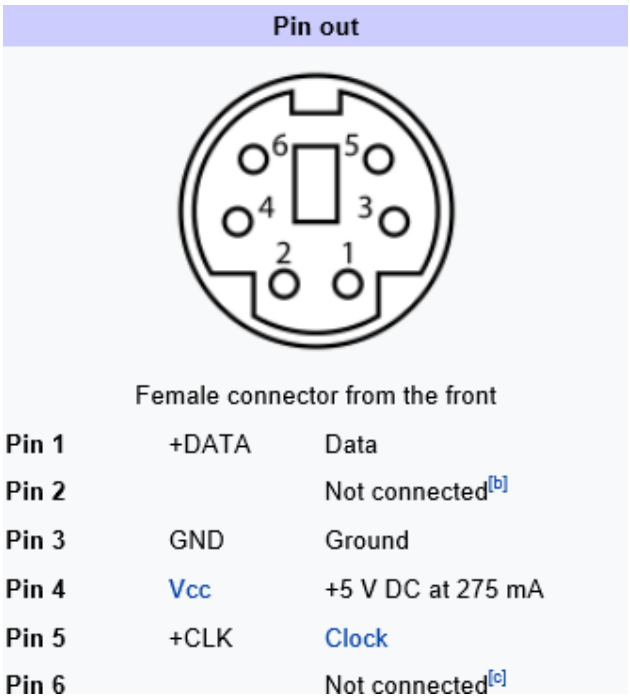

Figure 1.1.4: PS/2 port 6-pin mini-DIN connector

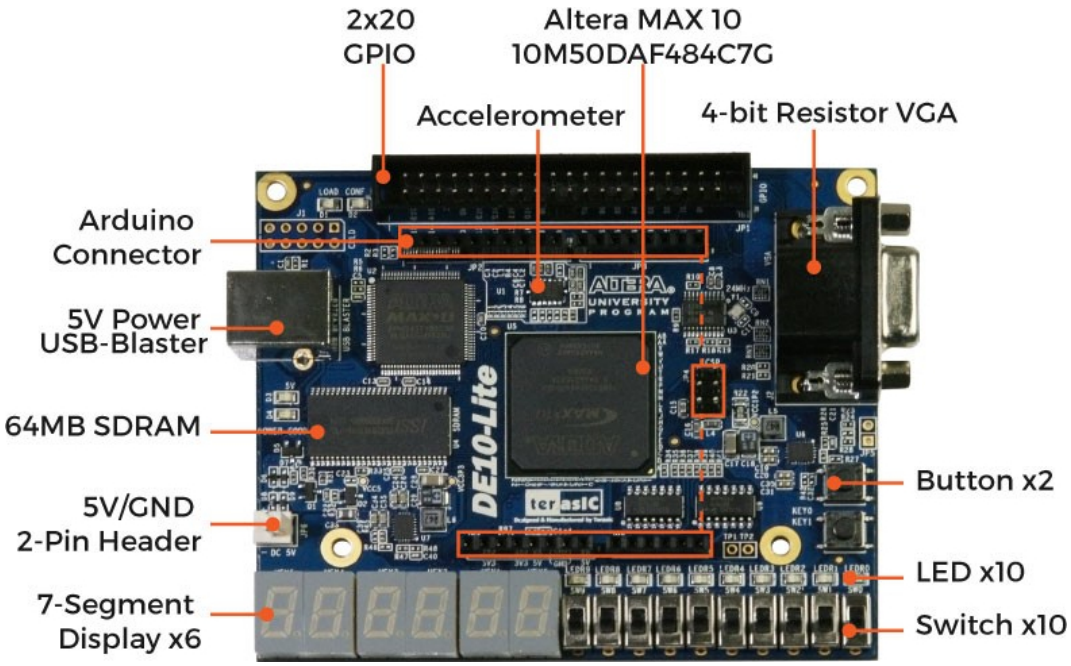

Figure 1.1.5: ECE 272 DE10-Lite FPGA courtesy of the EECS department at OSU

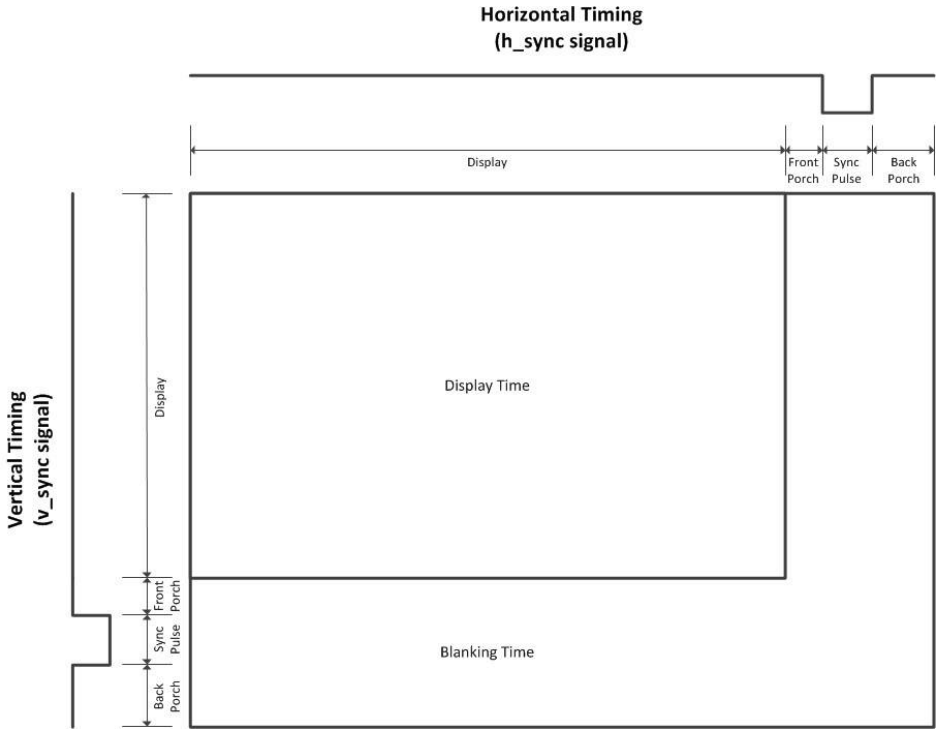

Figure 1.1.6: Timing specifications and pixel range of the VGA output

2 High-Level Description

2.1 Top-Level Hardware Diagram

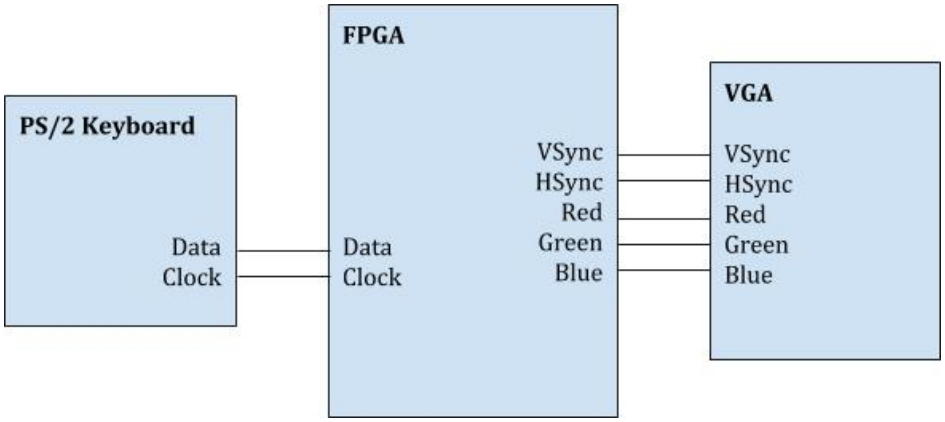

Figure 2.1.1: Top-level hardware diagram for PS/2 keyboard

Inputs: The FPGA reads in data and clock signals from the PS/2 keyboard.

Outputs: The FPGA outputs VSync, HSync signals as well as Red, Green, and Blue values to display as a VGA image.

Description: The top-level hardware diagram above shows that the DE10-Lite takes in

a Data and Clock signal from the PS/2 Keyboard. Then, it manipulates that data to form five outputs required for a VGA image.

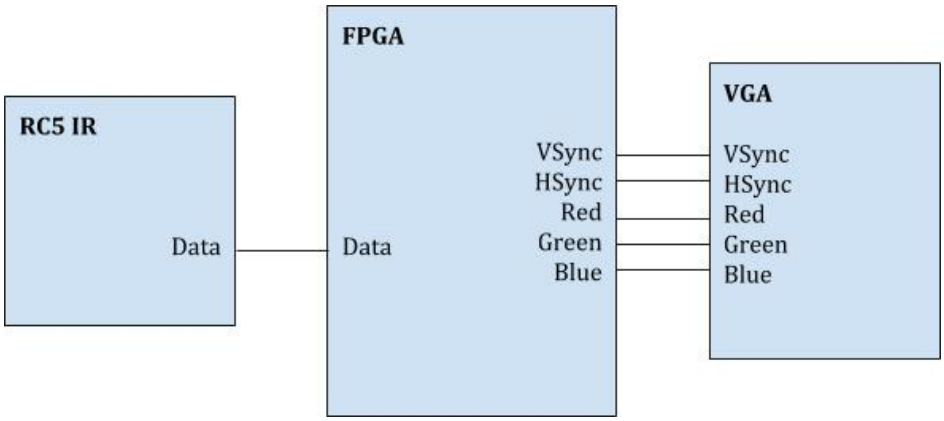

Figure 2.1.2: Top-level hardware diagram for RC5 IR

Inputs: The FPGA reads in a data signal from RC5 IR.

Outputs: The FPGA outputs VSync, HSync signals as well as Red, Green, and Blue values to display as a VGA image.

Description: The top-level hardware diagram above shows that the DE10-Lite takes in a Data signal in the form of RC5 IR. Then, the FPGA manipulates that data to form five outputs required for a VGA image.

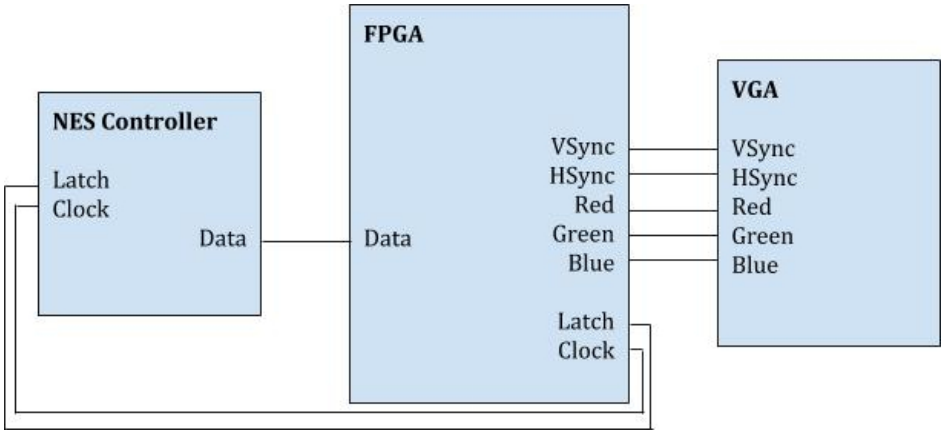

Figure 2.1.3: Top-level hardware diagram for NES controller

Inputs: The FPGA reads Latch, Clock, and Data signals from the NES Controller.

Outputs: The FPGA outputs VSync, HSync signals as well as Red, Green, and Blue values to display as a VGA image.

Description: The top-level hardware diagram above shows that the DE10-Lite takes in a Data, Latch, and Clock signal from the NES Controller. Then, the FPGA manipulates that data to form five outputs required for a VGA image.

2.2 Top-Level Hardware Diagram

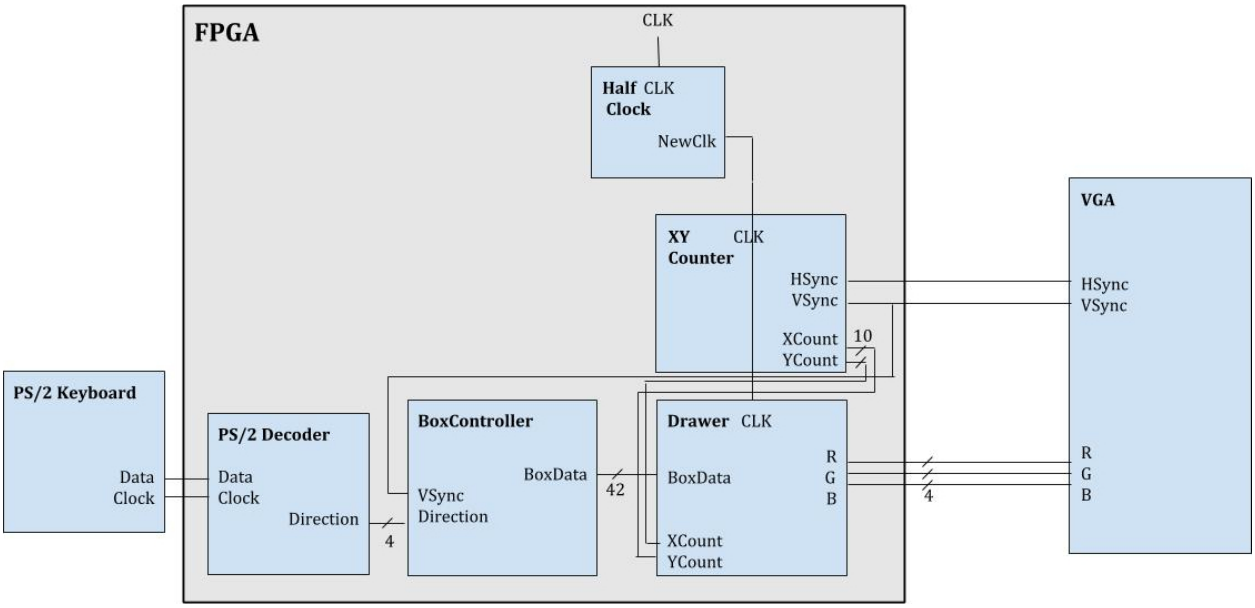

Figure 2.2.1: Top-level HDL diagram for PS/2 keyboard

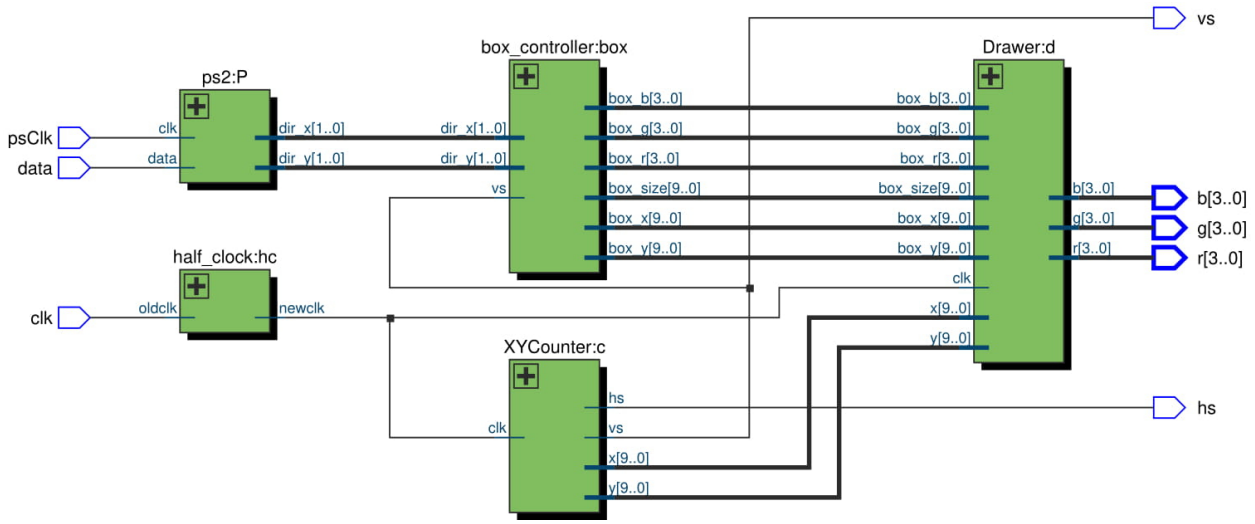

Figure 2.2.2: Top-level diagram for PS/2 keyboard

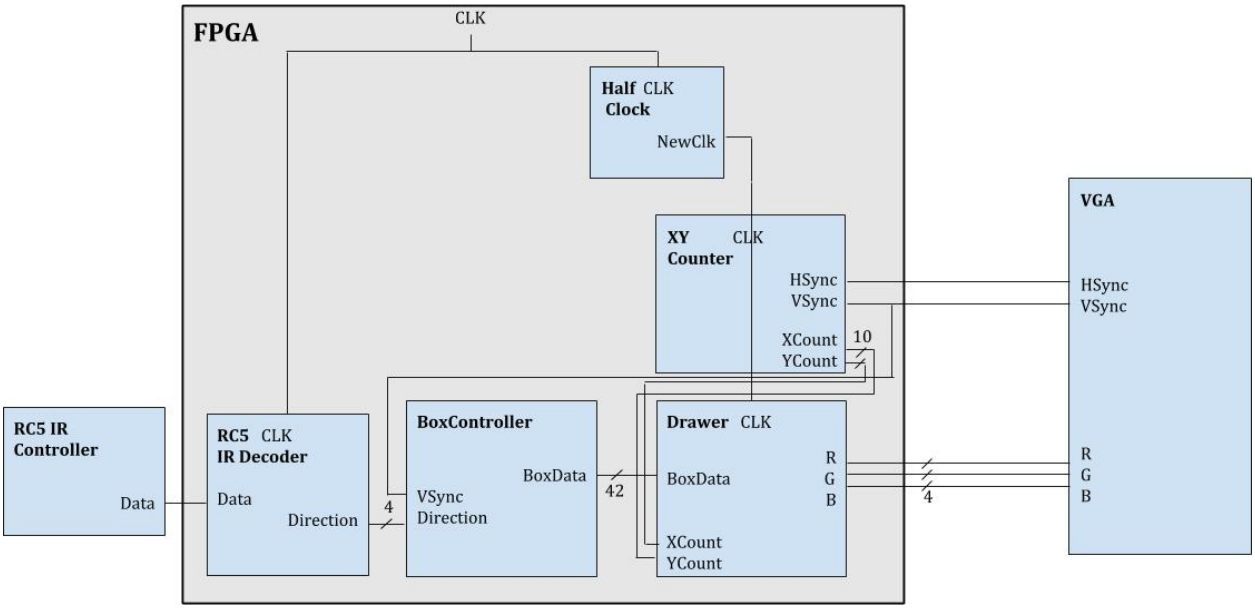

Figure 2.2.3: Top-level HDL diagram for RC5 IR

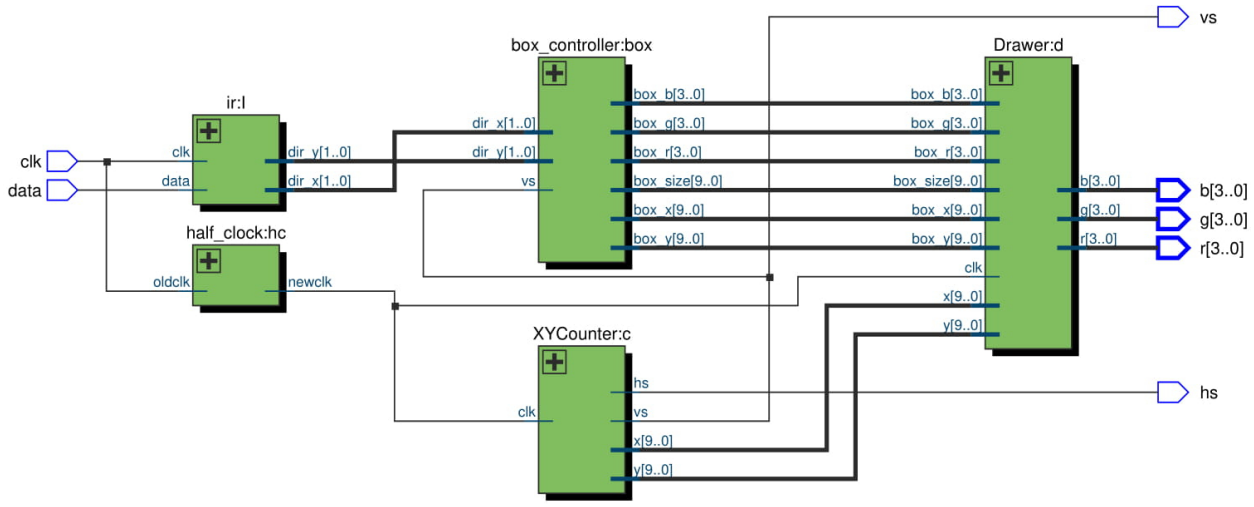

Figure 2.2.4: Top-level diagram for RC5 IR

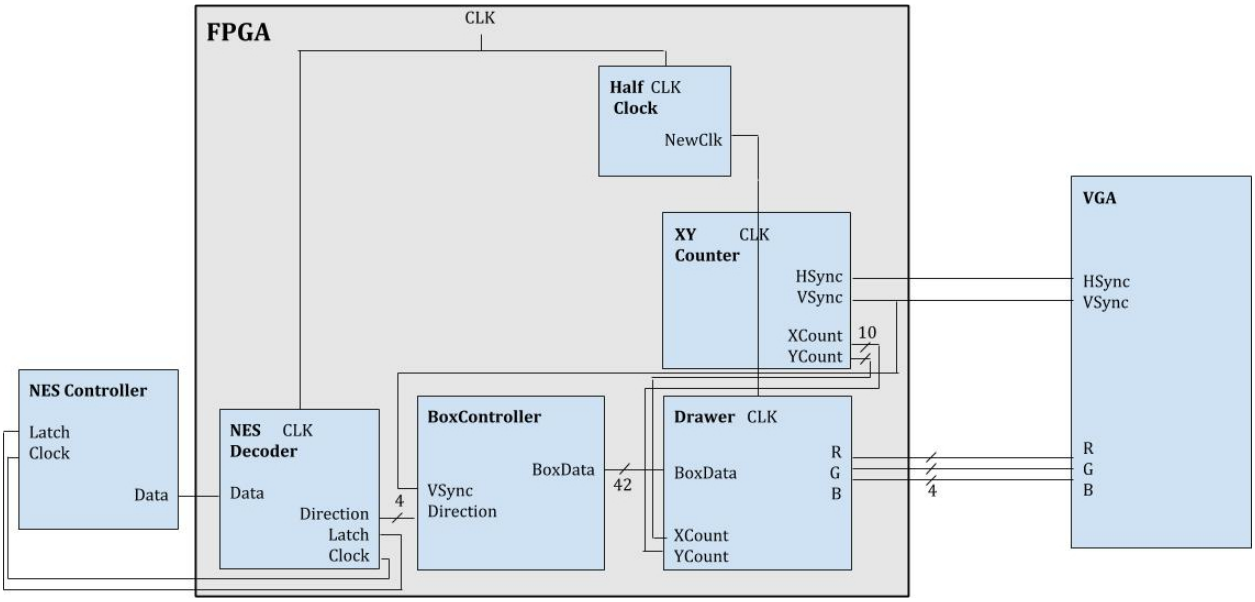

Figure 2.2.5: Top-level HDL diagram for NES controller

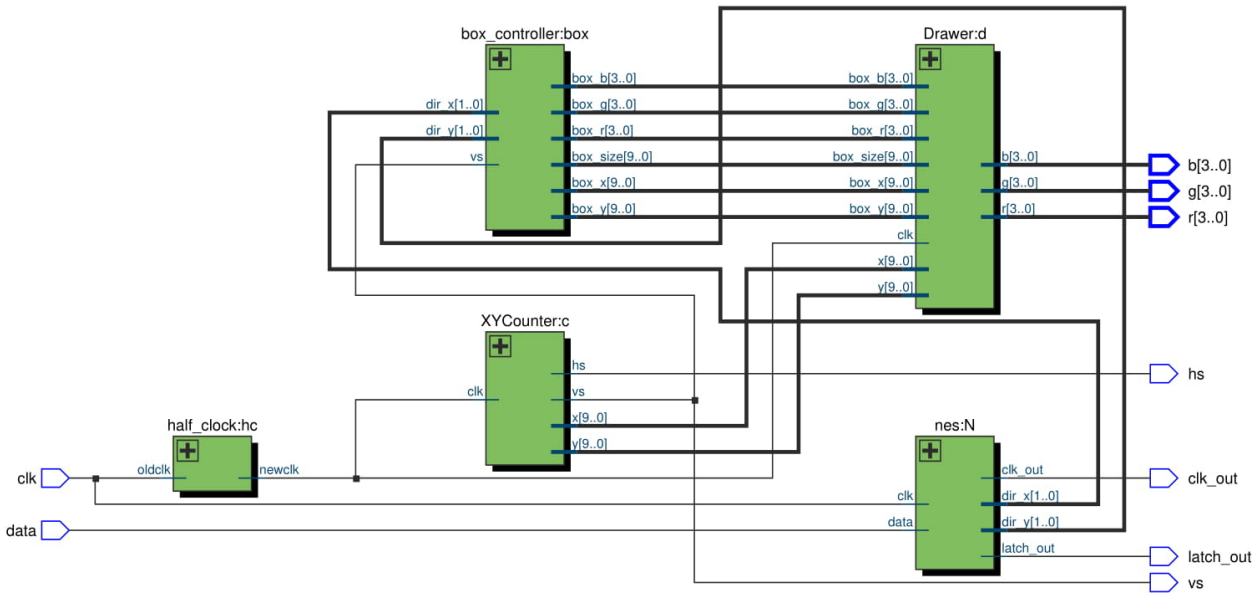

Figure 2.2.6: Top-level diagram for NES controller

3 Controllers

3.1 PS/2

The PS/2 port is a 6-pin mini-DIN connector used for connecting keyboards or mice to a computer system. It was first designed by IBM in 1987. A mini-DIN connector refers to a family of multi-pin electrical connectors used in a variety of environments. They are 9.5 millimeters in diameter and come in seven patterns ranging from three to nine pins. PS/2 ports have serial, synchronous, and bidirectional communication in which the attached device generates a clock signal while the host controls communication along the clock line. When the clock is LOW, communication from the PS/2 port is prevented. The assignments of each pin on the 6-pin connector are illustrated clearly in Figure 1.1.4, which can be found on page 4.

3.2 RC5 IR

RC5 IR refers to a protocol developed by Phillips in the late 1980's as a consumer infrared remote control communication code for consumer electronics. The RC5 uses a bi-phase coding with a frequency of 36 kHz and is comprised of a keypad and a transmitter integrated circuit, which drives an infrared LED. The transmission of data starts with two bits followed by a toggle bit, which changes value upon a each key-press. There are five address bits, which indicate the device that is being controlled. Lastly, the six commands bits represent the data that is to be transmitted from the RC5 to the device.

3.3 NES Controller

The NES Controller is an 8-bit controller developed by Nintendo in the 1980's. The device transmits 8-bits of data, which equates to one bit for each of the buttons on the controller: A, B, Select, Start, Up, Down, Left, and Right. The NES controller also contains a latch to store the state of buttons internally as well as a clock. Every 60 Hz, the NES sends a 12 microsecond signal to the latch, indicating that it must store the state of all eight buttons. Following the Latch signal, 8 clock pulses are emitted, one for each button on the controller. If the button is pressed upon the rising edge of the pulse, data is asserted to ground. This means that in the NES controller, the buttons are active low because Data asserts to ground when the button is pressed.

4 HDL Modules

4.1 PS/2 Modules

4.1.1 PS/2 Decoder

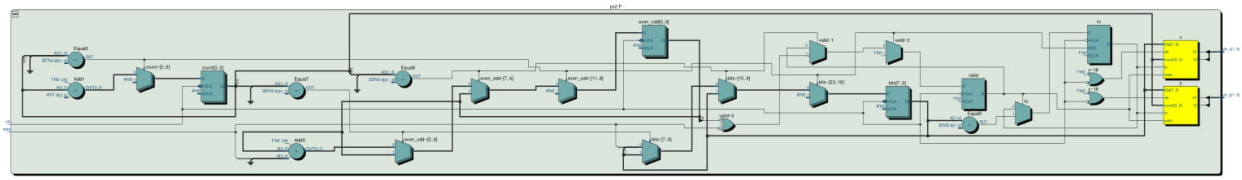

Figure 4.1.1: PS/2 keyboard decoder

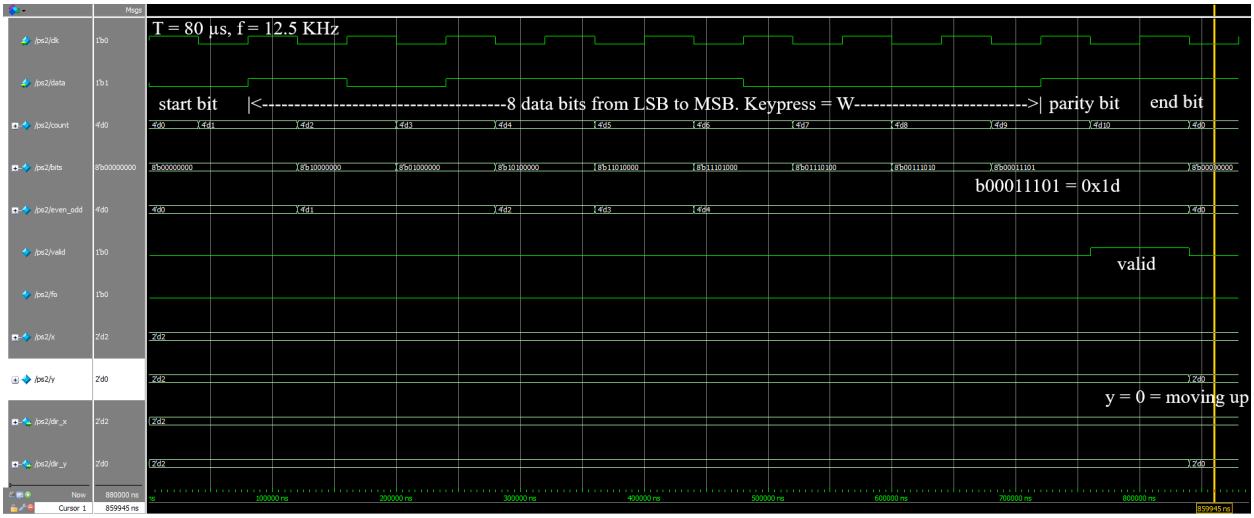

Figure 4.1.2: PS/2 module's simulation testing - Keypress W (up)

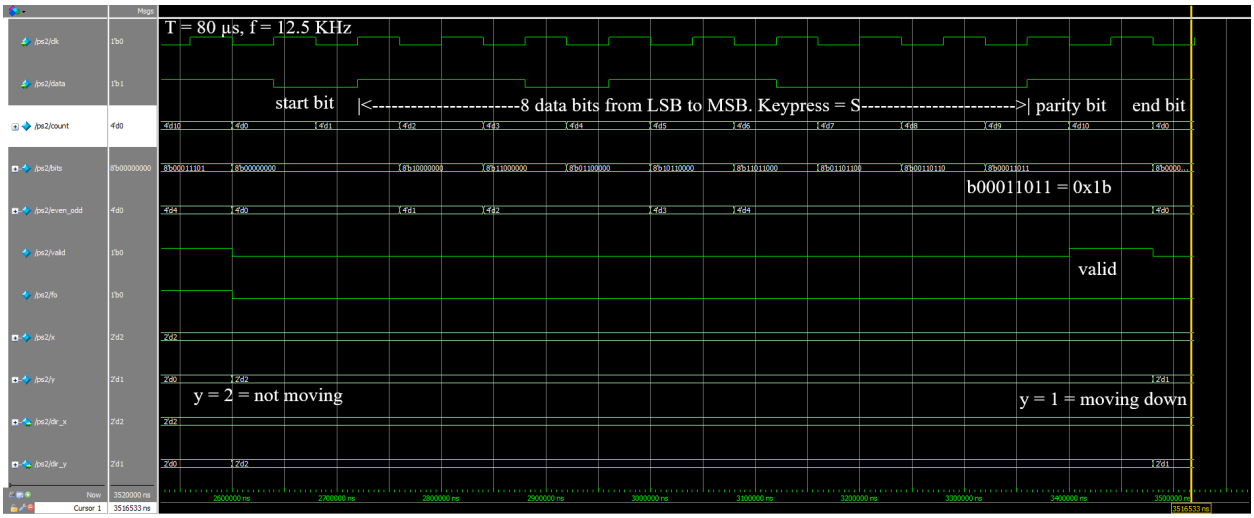

Figure 4.1.3: PS/2 module's simulation testing - Keypress S (down)

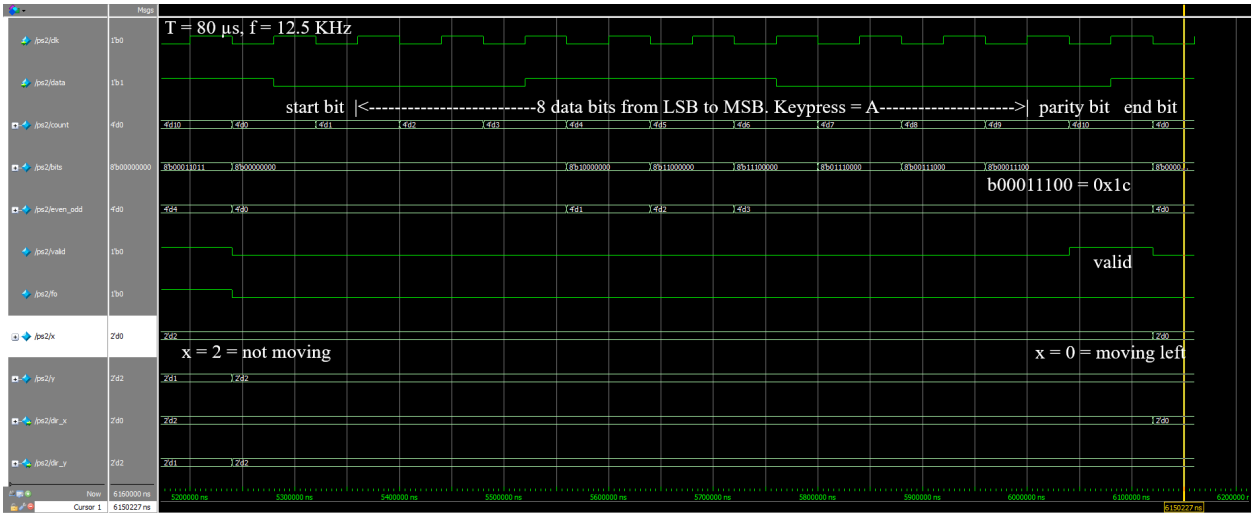

Figure 4.1.4: PS/2 module's simulation testing - Keypress A (left)

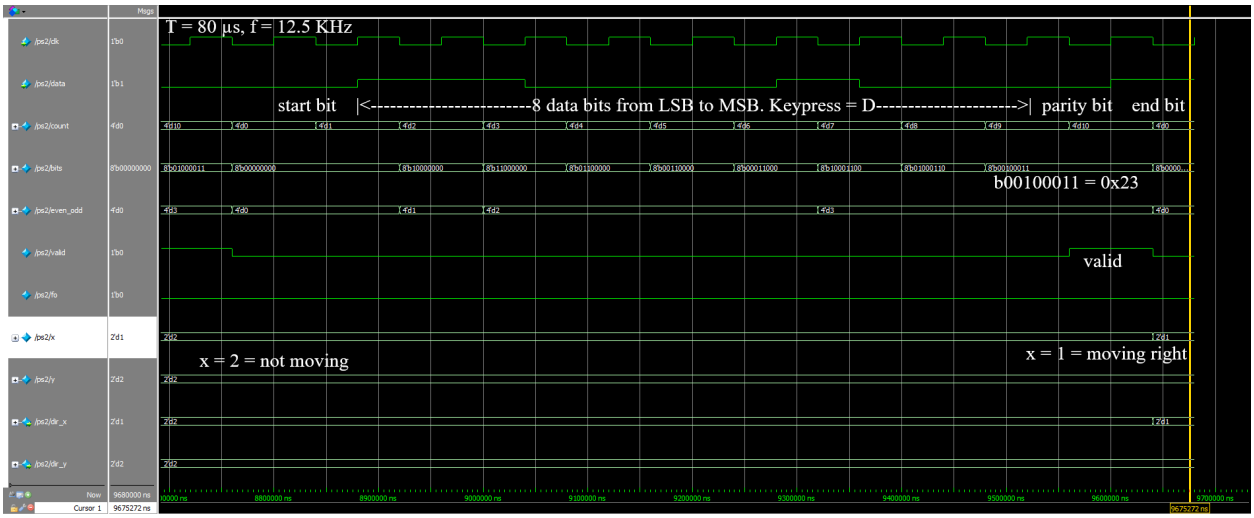

Figure 4.1.5: PS/2 module's simulation testing - Keypress D (right)

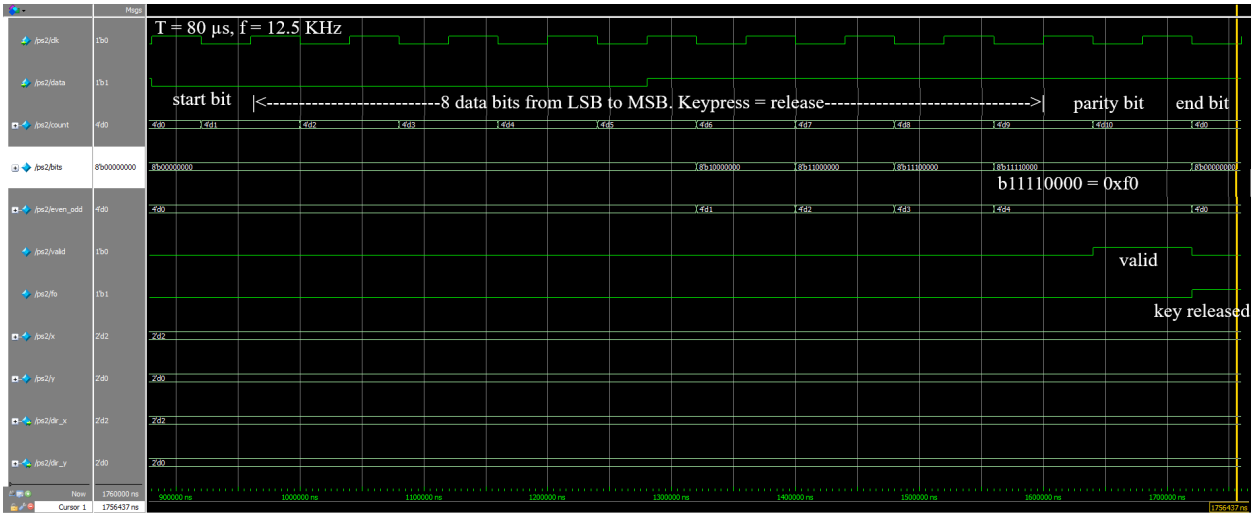

Figure 4.1.6: PS/2 module's simulation testing - Key release signal (0xF0)

Inputs: Clock and Data signal.

Outputs: Two buses to represent the box's movement in the x and y direction.

Description: The PS/2 keyboard decoder module begins with an always flip flop on the negative edge of the clock signal, indicating that the data values are generated only after the clock has gone to low. The module then determines the value of `count`, which continuously increments between 0 and 10. If the value of `count` is 10, then the keyboard inputs, recognized as A, W, S, and D, and translated into variables representing the movement and direction of the box image. If the `count` is between 1 and 8, then data from the PS/2 keyboard is simply stored until `count` returns to 10, in which the values are then outputted to directional values stored in the variables `dir_x` and `dir_y`.

4.2 RC5 IR Modules

4.2.1 RC5 IR Decoder

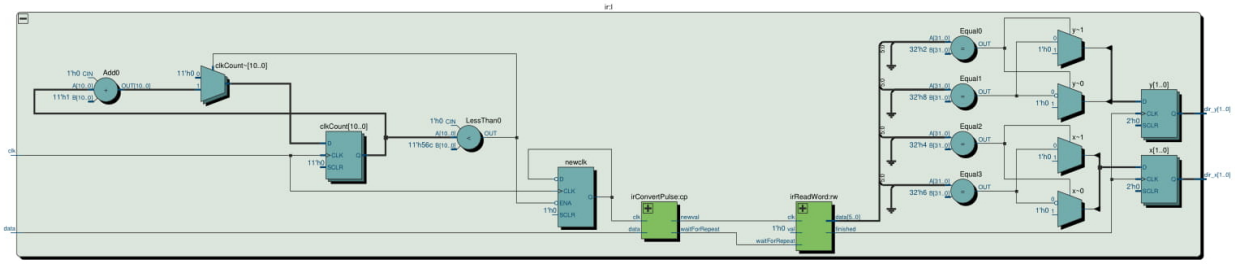

Figure 4.2.1: RC5 IR decoder

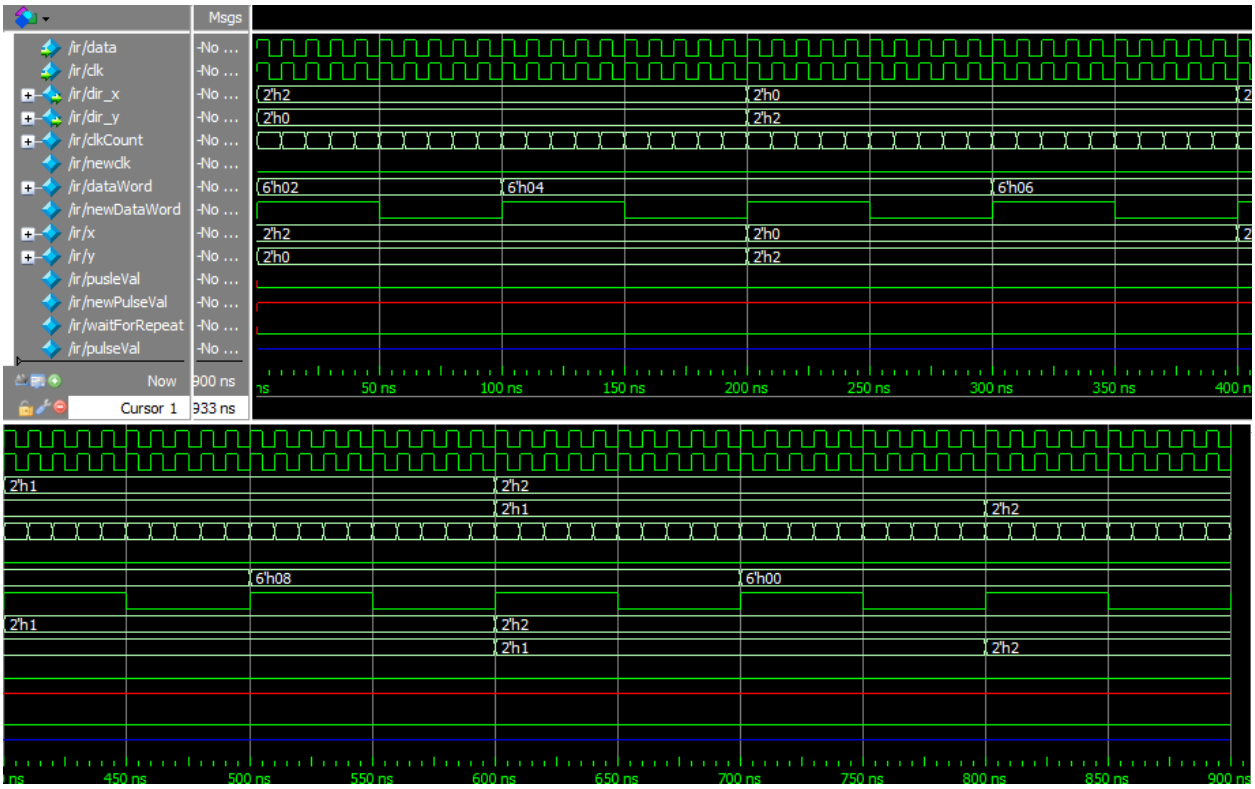

Figure 4.2.2: RC5 IR decoder decoding up, down, left, right, and neutral direction controls from data word

Inputs: Clock and Data signal.

Outputs: Two buses to represent the box's movement in the x and y direction.

Description: The first step in designing the RC5 IR module was slowing the normal FPGA clock frequency of 50 MHz to 36 KHz, the frequency used in RC5 data transmission. RC5 works by using bi-phase encoding, meaning that during each clock period, there is either a 1 or 0, where 1 represents a rising edge and 0 represents a falling edge. Next, the

ConvertPulse and ReadData modules were instantiated. Convert Pulse takes in the 32-bit pulse emitted by the RC5 and treats the entire pulse as HIGH and the absence of a pulse as LOW. Then, when the pulse signal encoding and bi-phase encoding are decoded, a 14-bit result follows. The first 2-bits are start bits and represent when a new data transmission occurs. The toggle is unaffected if no new button is pressed. However, if a new data line is activated, the toggle bit changes. The next 5-bits are address bits to indicate the path to a device that is being transmitted to. Lastly, 6-bits are used to represent the transmitted data. In our module, the encodings used to represent the direction and

4.2.2 RC5 IR Convert Pulse

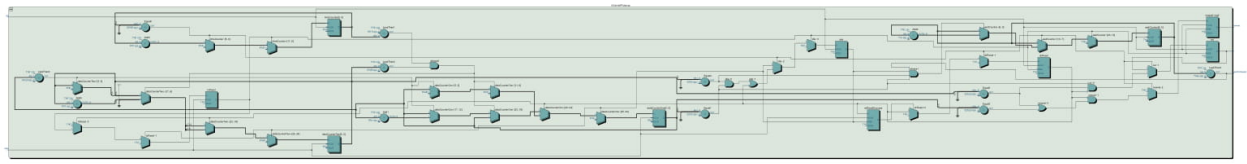

Figure 4.2.3: IR convert pulse

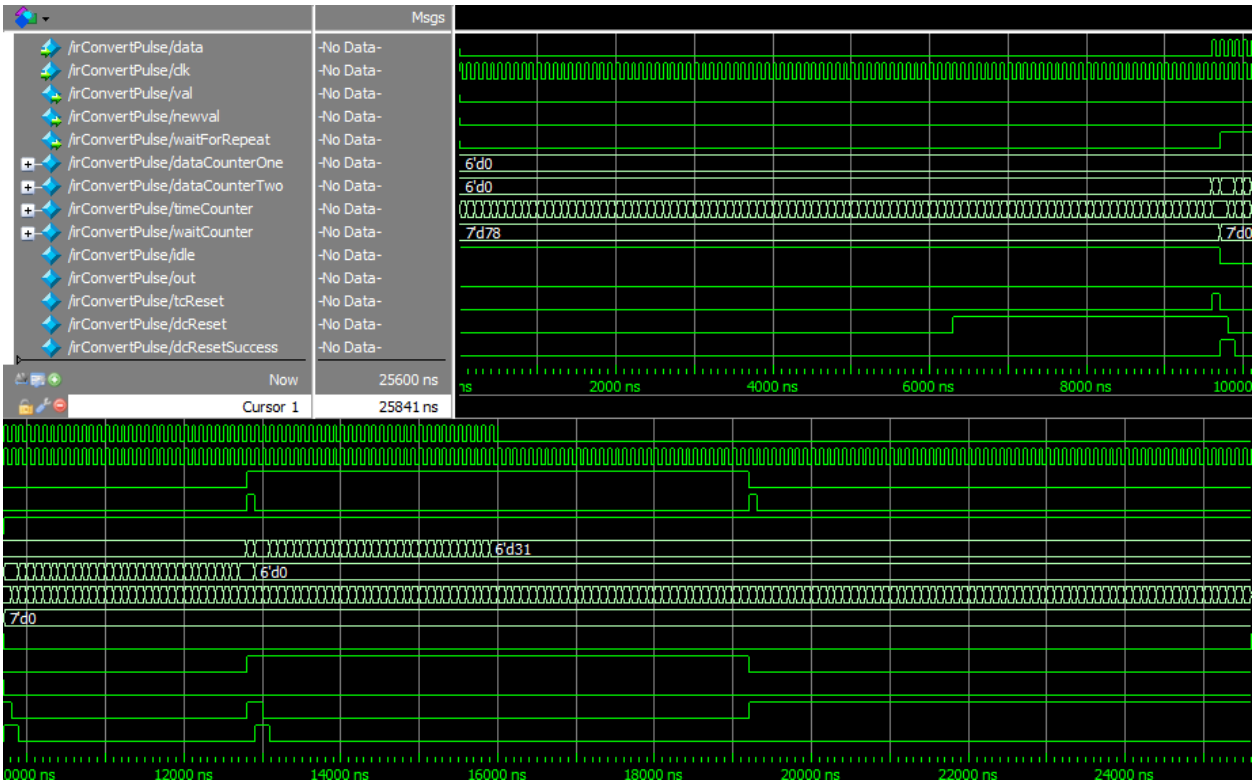

Figure 4.2.4: Testing conversion of inputs from idle to HIGH to LOW

Description: The module above converts the encoded pulse outputted in bi-phase encoding by RC5 IR into binary form.

4.2.3 RC5 IR Read Word

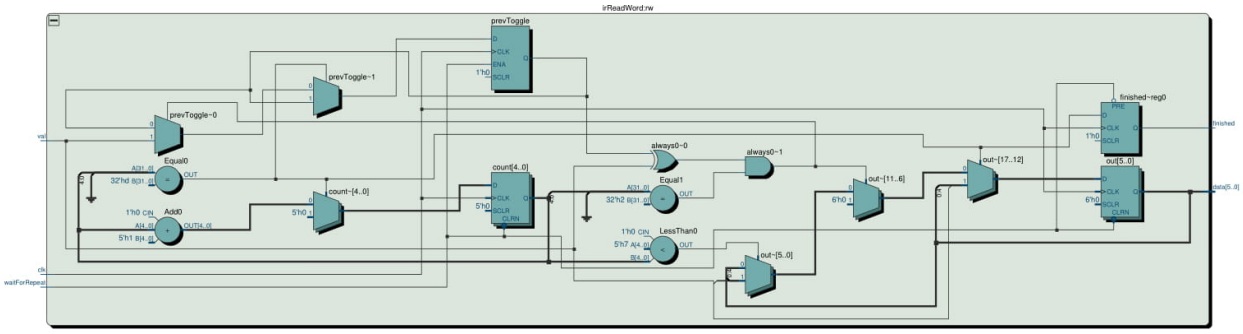

Figure 4.2.5: IR read word

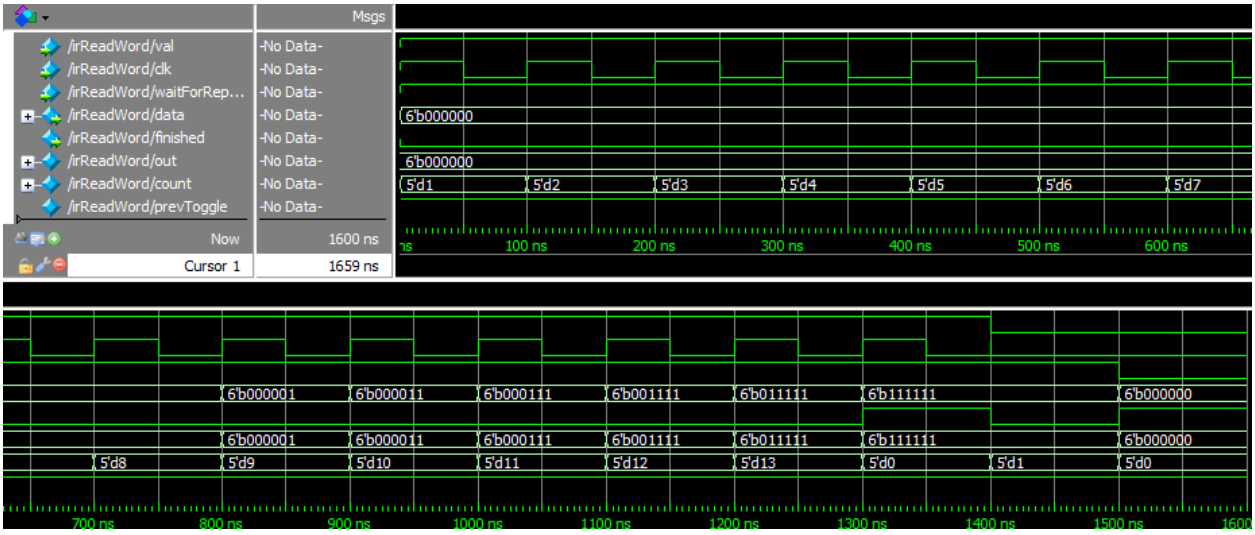

Figure 4.2.6: Readind data word “111111” then resetting

Description: The module above converts encoded binary into data.

4.3 NES Modules

4.3.1 NES Decoder

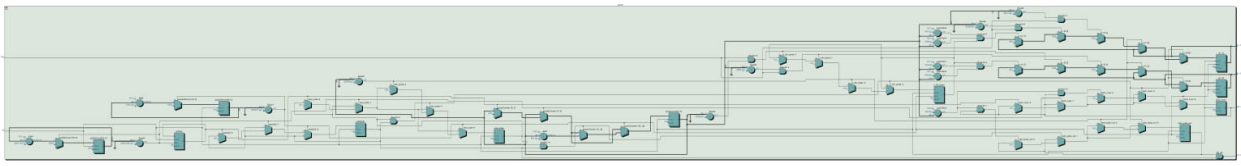

Figure 4.3.1: NES controller decoder

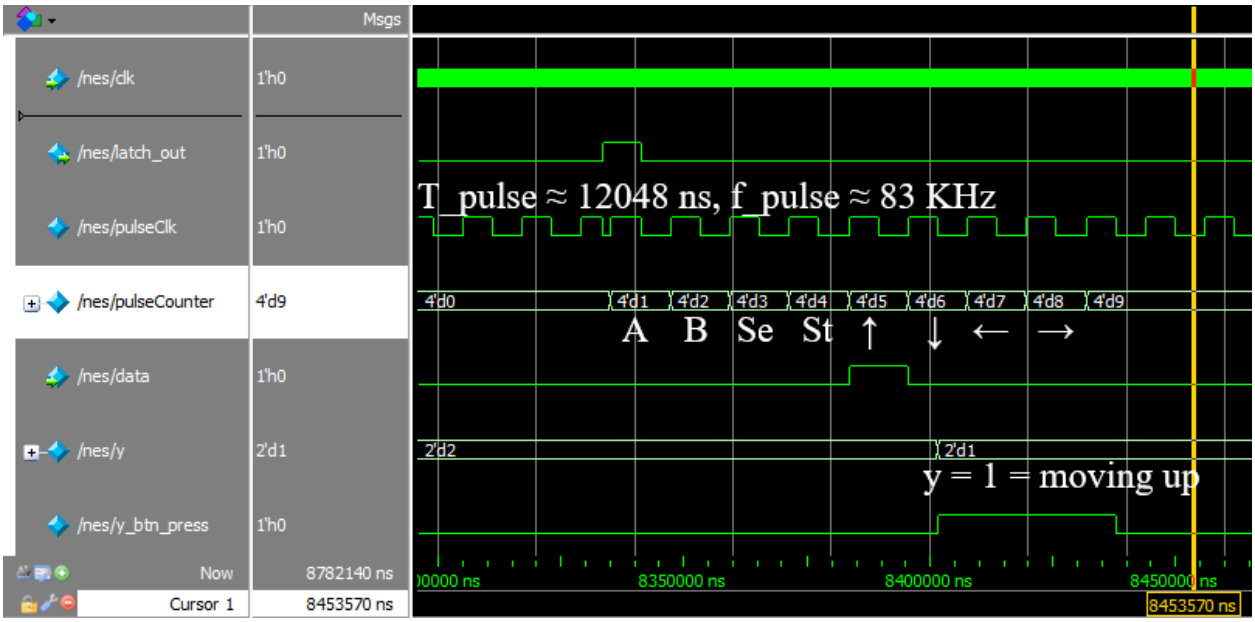

Figure 4.3.2: NES module's simulation testing - Moving up keypress

Inputs: Clock and Data signal.

Outputs: Two buses to represent the box's movement in the x and y direction. Also outputs new clock signal and latch data.

Description: Initially, the decoder sends a latch output to the controller. Every 60Hz, the decoder asserts the latch for 12 microseconds, which stabilizes each buttons state. For this module, a clock slowing mechanism was inserted so that the internal clock of the FPGA, which is naturally 50MHz, can be used. On the falling edge of the latch, the decoder reads the first value, which is A. Six microseconds after, it begins sending 8 clock pulses which have a period of 12 microseconds. On every rising edge of the clock, the decoder cycles to the next button value, for a total of eight button values. The value is then read on the falling edge of that cycle and stored in module variables. Since the only button actions which affect our box image are the up, down, left, and right arrows, those are the only ones that are stored and sent to different modules.

4.4 General Modules Used in Each Design

4.4.1 Clock Frequency Module

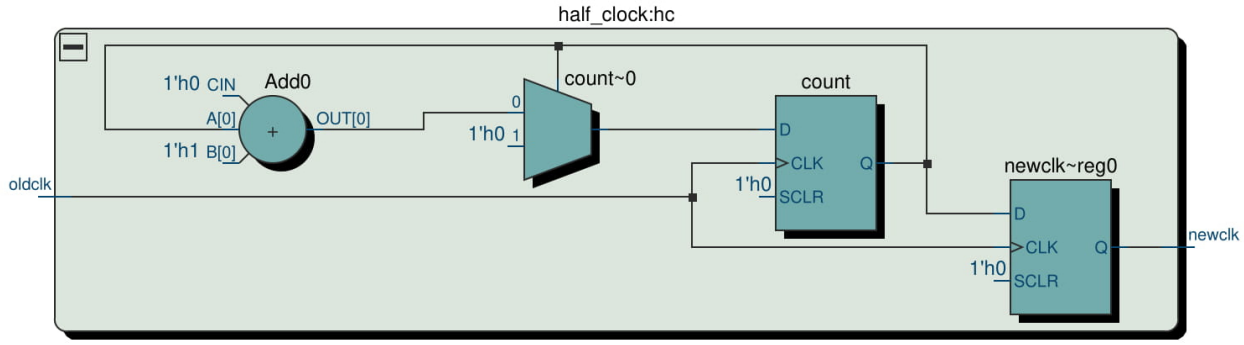

Figure 4.4.1: Half clock module

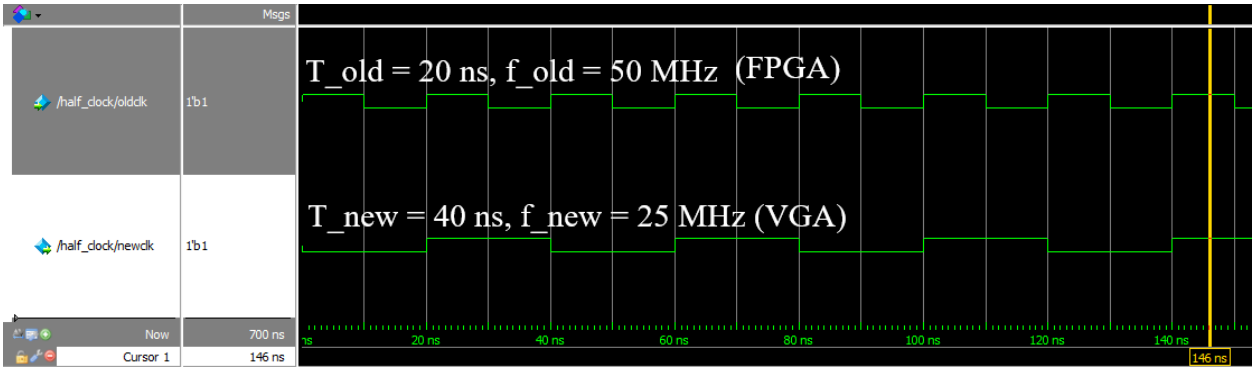

Figure 4.4.2: Half clock module's simulation testing - Making a half-as-fast clock

Inputs: `oldClock` signal.

Outputs: `newClock` signal representing a clock with half the frequency of `oldClock`.

Description: The clock module plays an essential role in allowing the output to be displayed as VGA. The natural frequency of the DE10-Lite is 50 MHz. However, the pixel clode on the VGA mode timing specifications is 25 MHz. Because of this difference, a half clock module is required to half the clock frequency of the FPGA so that it corresponds with the pixel clock at 25 MHz.

4.4.2 Box Controller Module

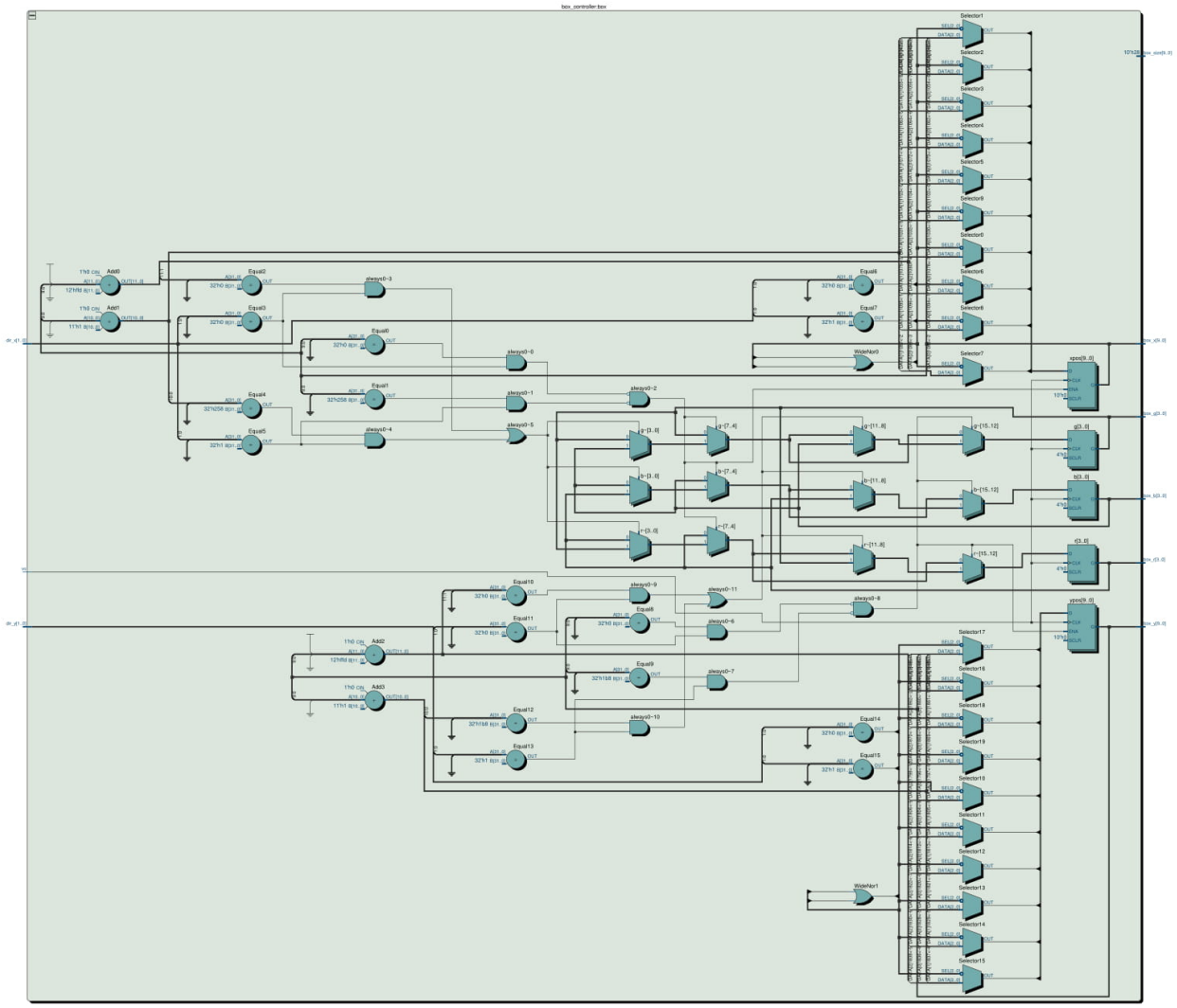

Figure 4.4.3: Box controller module

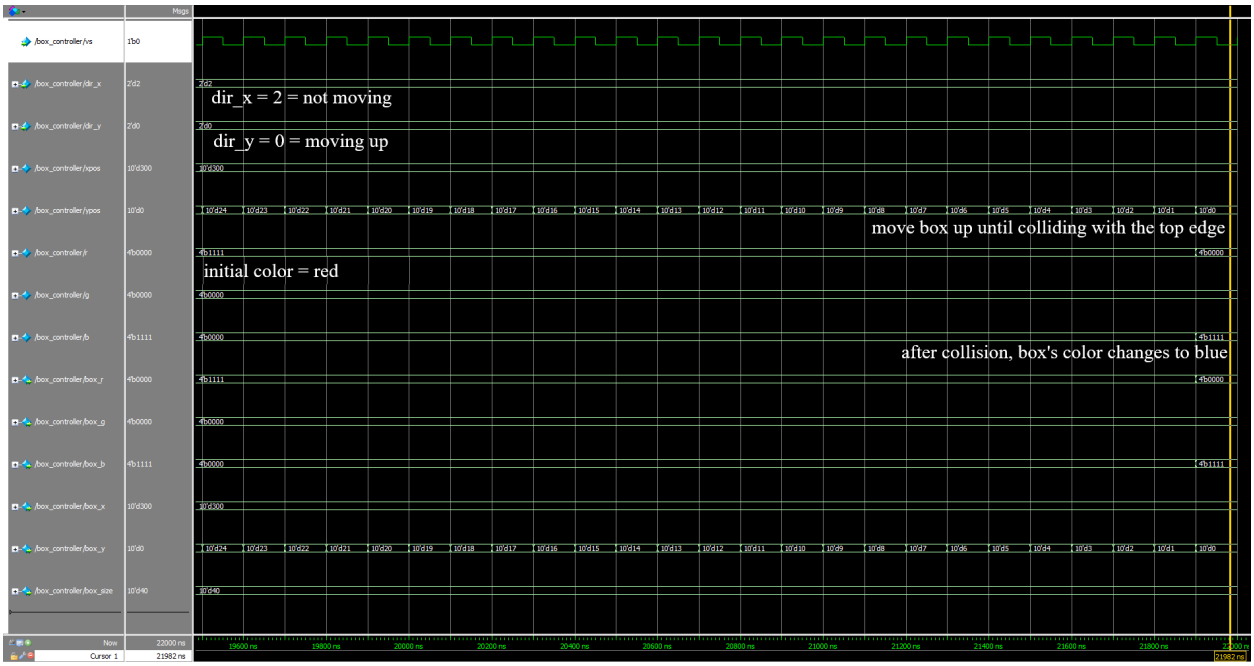

Figure 4.4.4: Box controller module's simulation testing - Box collides with the top edge of the screen

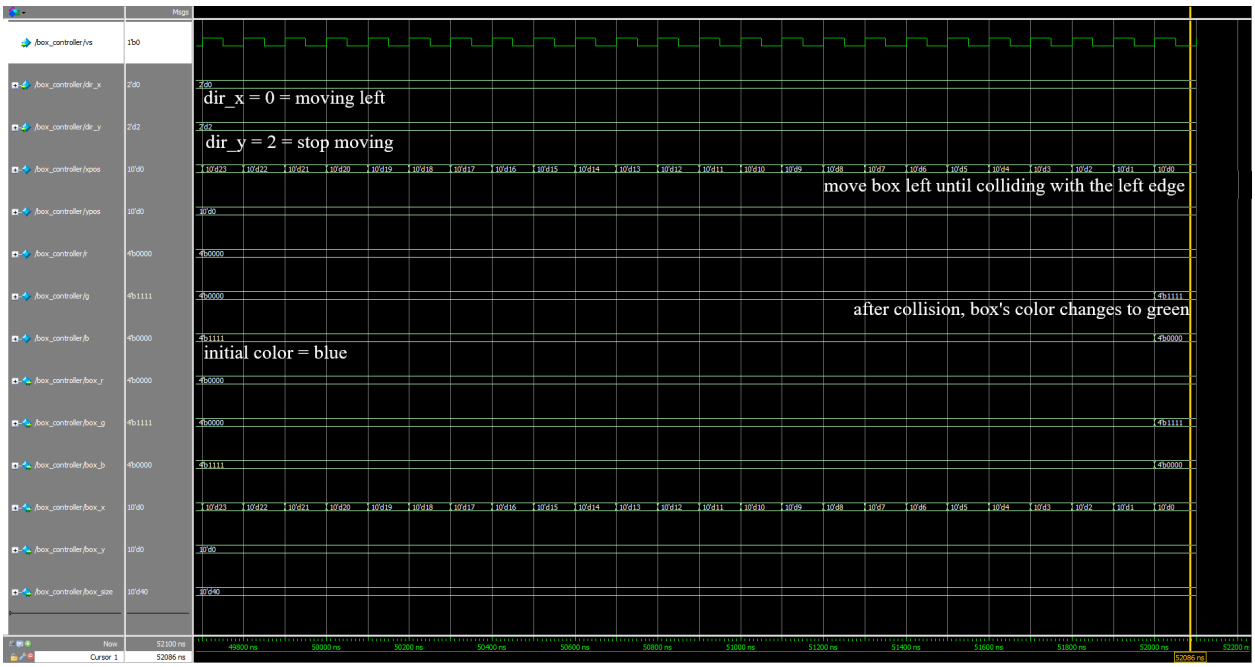

Figure 4.4.5: Box controller module's simulation testing - Box collides with the left edge of the screen

Inputs: Vertical Sync. X and Y direction in which the box is moving.

Outputs: Three 4-bit buses representing red, green, and blue. Two 10-bit buses representing the and y location of the box.

Description: The Box Controller Module controls the action and animations of the box image. The module takes in 3 inputs representing vertical sync, X direction, and Y direction. X direction and Y specify which direction the box is moving, where 0 represents left or up and 1 represents right or down. The Box Controller checks to ensure that the box will remain in the active pixel range. It then checks to see if a collision with a vertical or horizontal boundary has occurred. If this is true, then the red, green, and blue data lines corresponding to the box are switched, forcing the image to change color. In addition, the box stops moving. For example, if a red box were heading to the right and encounters a collision with the rightmost part of the screen, then the box will change colors to blue, and also stop moving so it does not continue past the edge of the screen. The current state of the box, including its position and color, are outputted to Drawer module.

4.4.3 XY Counter Module

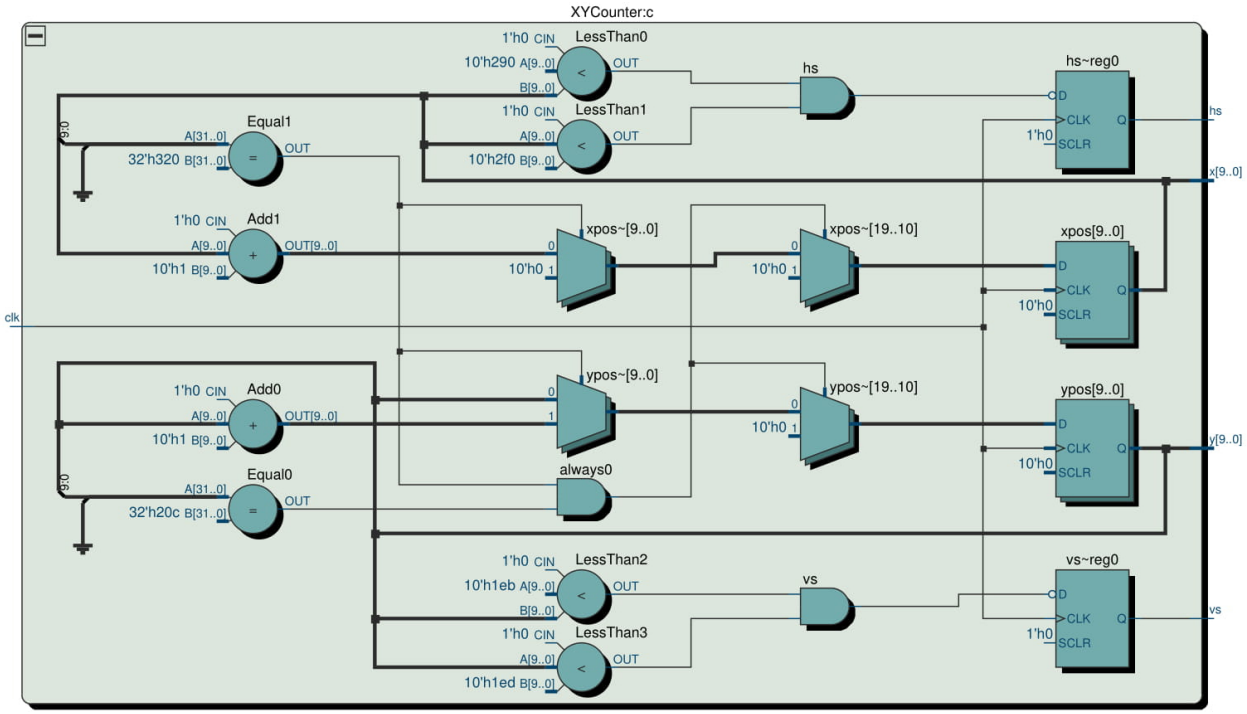

Figure 4.4.6: XY Counter module

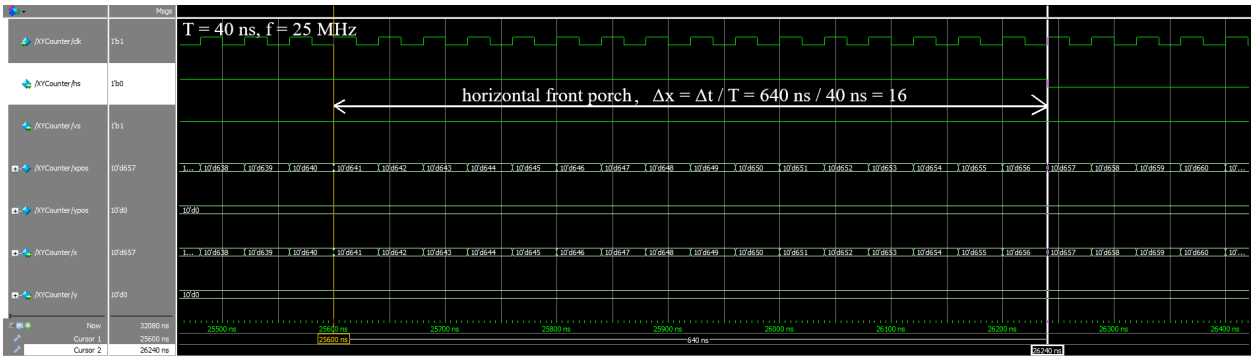

Figure 4.4.7: XY counter module's simulation testing - Horizontal front porch

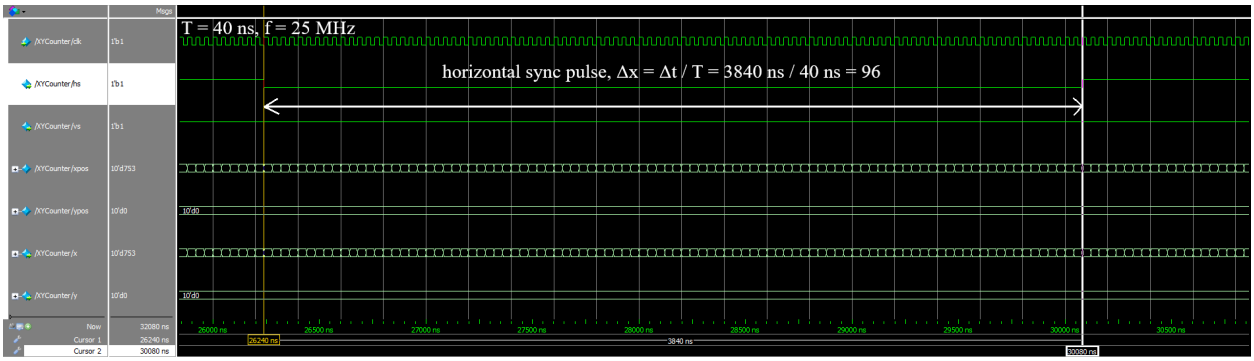

Figure 4.4.8: XY counter module's simulation testing - Horizontal sync pulse

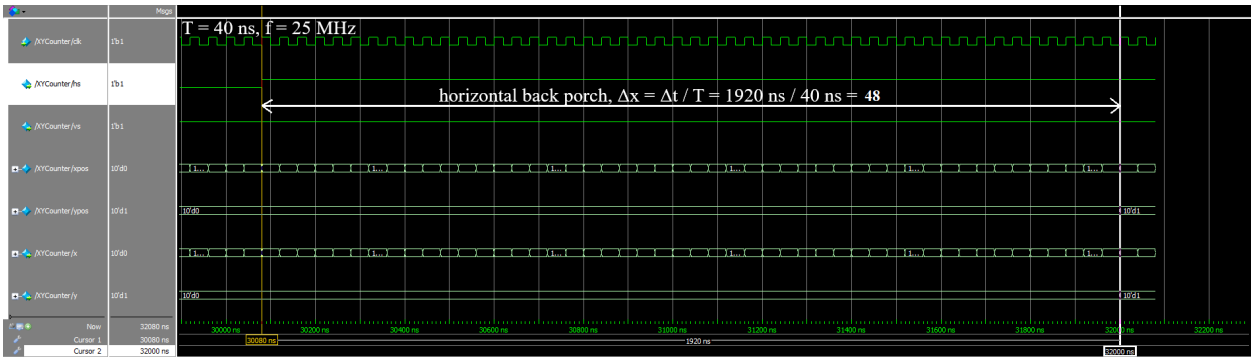

Figure 4.4.9: XY counter module's simulation testing - Horizontal back porch and counter completes one line

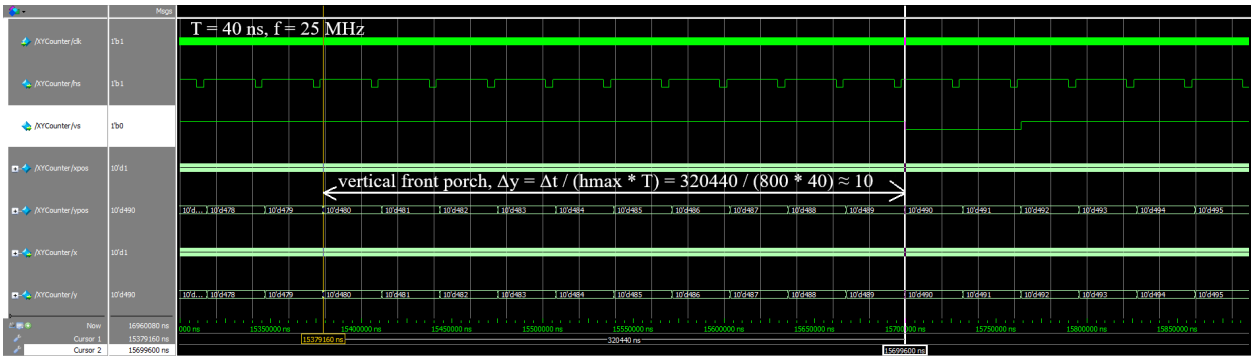

Figure 4.4.10: XY counter module's simulation testing - Vertical front porch

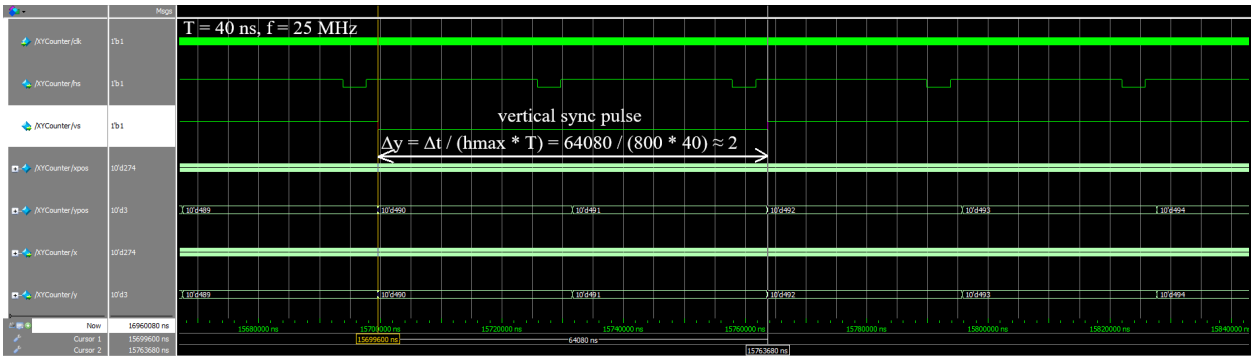

Figure 4.4.11: XY counter module's simulation testing - Vertical sync pulse

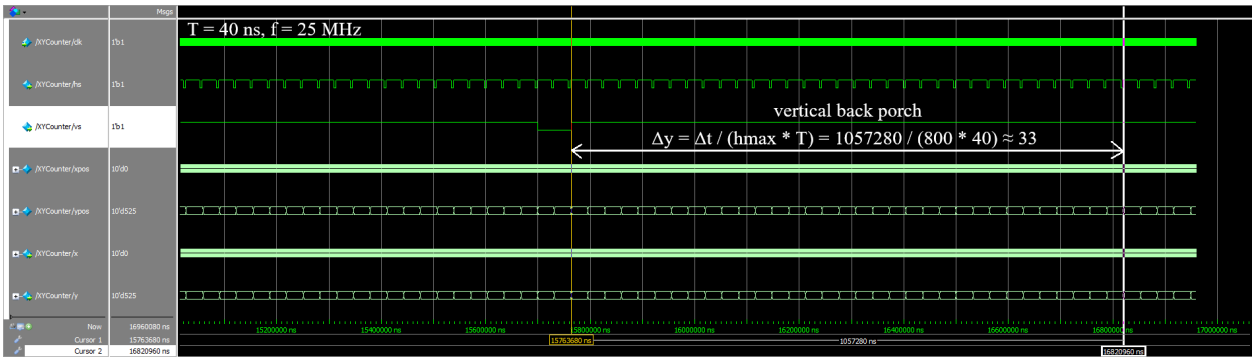

Figure 4.4.12: XY Counter module's simulation testing - Vertical back porch and counter completes one screen

Inputs: Clock

Outputs: Vertical Sync and Horizontal Sync. Two 10-bit buses representing the current x and y pixel location.

Description: The XY Counter module plays a critical role in outputting data as a VGA image. Its purpose is to keep track of horizontal (x) and vertical (y) values to track the pixel position. The module takes only one input, which is the clock, and outputs values for

HSync, VSync, and x and y positions. By looking at Figure 4.4.6, HSync and VSync values are calculated and then outputted directly to the VGA. The x representing the pixel position is continuously incremented until it reaches 800, which signifies a new frame, during which y is initialized and incremented by one. At the same time, x is reset to 0 and incremented again to 800. This process repeats until y reaches 524, which signifies the end of the screen. At this moment, both x and y are reset to 0. The XY Counter module outputs the x and y location to the Drawer module.

4.4.4 Drawer Module

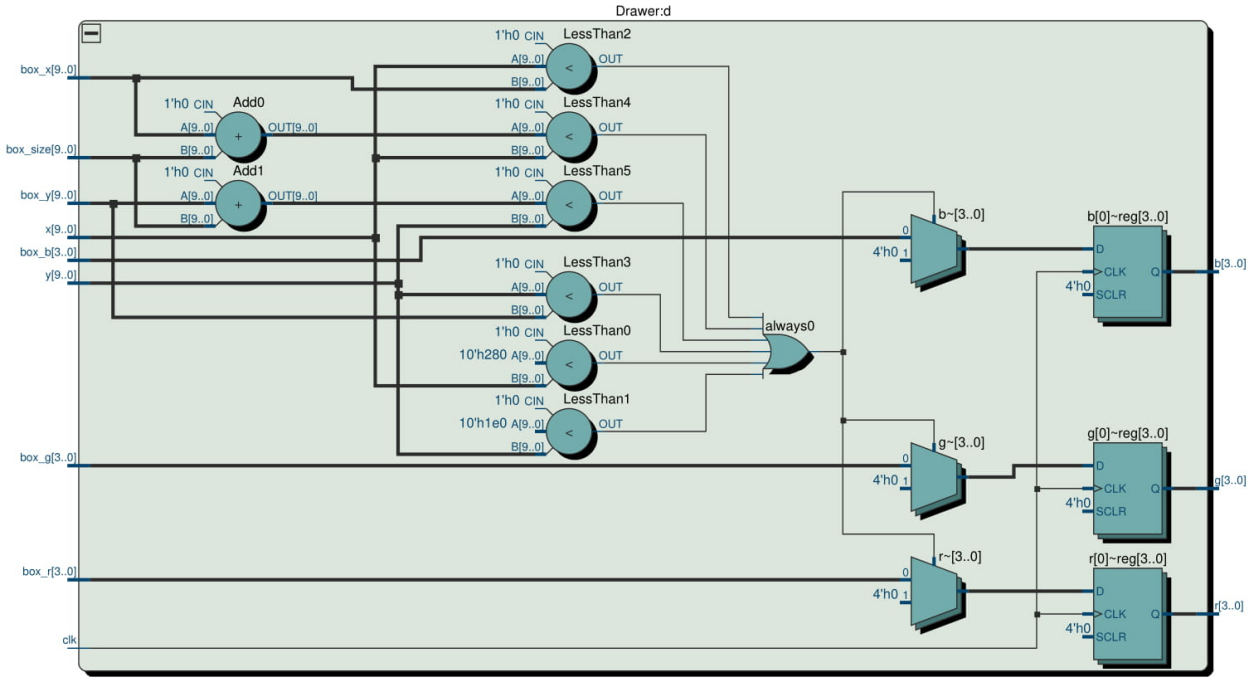

Figure 4.4.13: Drawer module

Inputs: Three 4-bit buses representing red, green, and blue. X and Y counter as well as x and y locations of the box.

Outputs: Three 4-bit buses representing red, green, and blue.

Description: The Drawer module shown above is the last step in displaying the output as a VGA image. This module reads in data corresponding to the x and y locations that are outputted from the XY Counter module as well as x and y values indicating the size of the box to be displayed. If the x and y locations outputted from the counter module are not within the active pixel range specified by the VGA timing specifications, then a black box is drawn, effectively showing nothing. However, if the location of x and y are within the active pixel range, then the outputs: red, green, and blue, are assigned values so that a colorful box image appear on the monitor. This box will appear at the specified box x and y values,

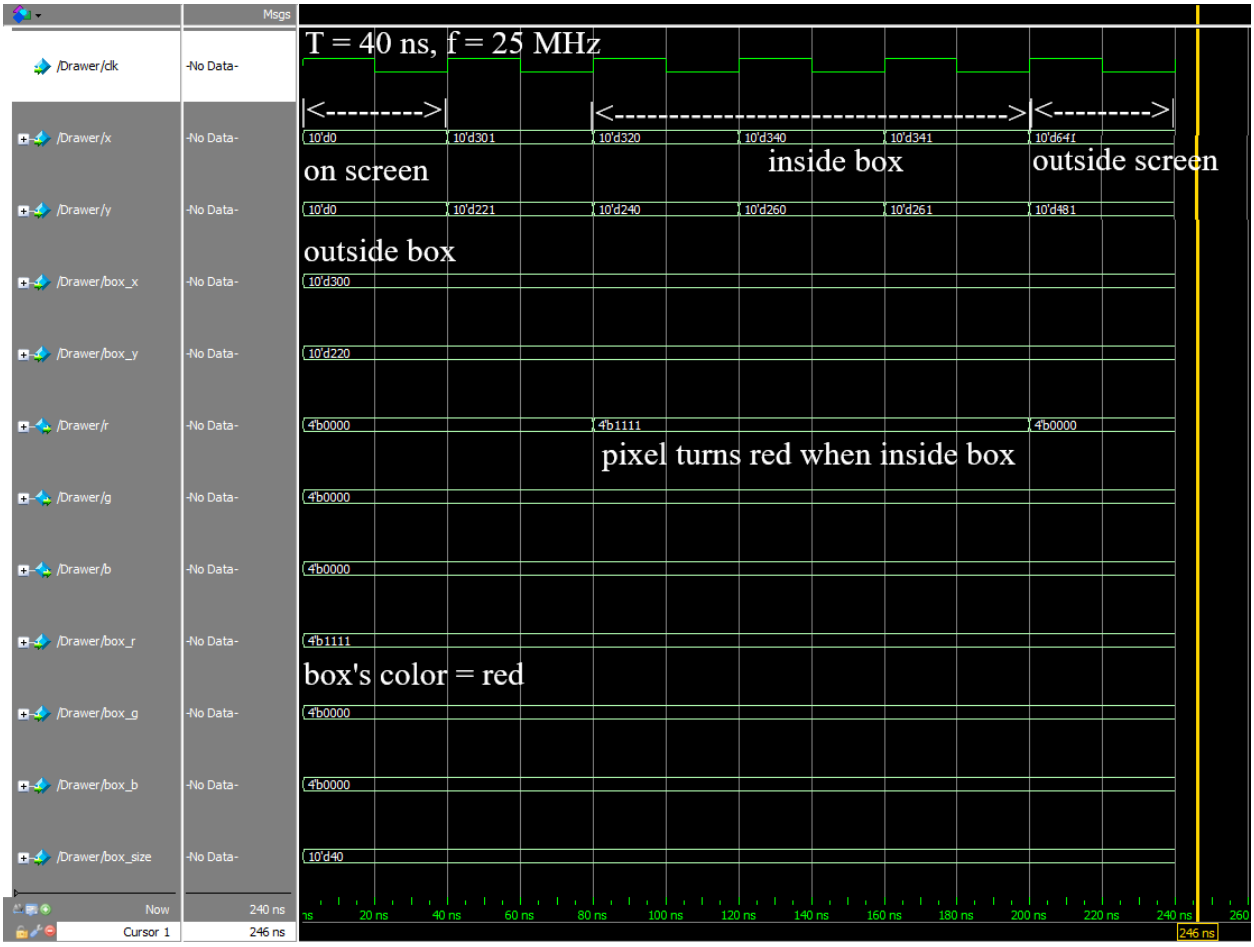

Figure 4.4.14: Drawer module's simulation testing - Different pixel colors at different coordinates

and will have a size specified by the inputs. If the current pixel is outside the location of the box, black is displayed, otherwise either red, green, or blue are displayed to form the box.

5 Putting It All Together

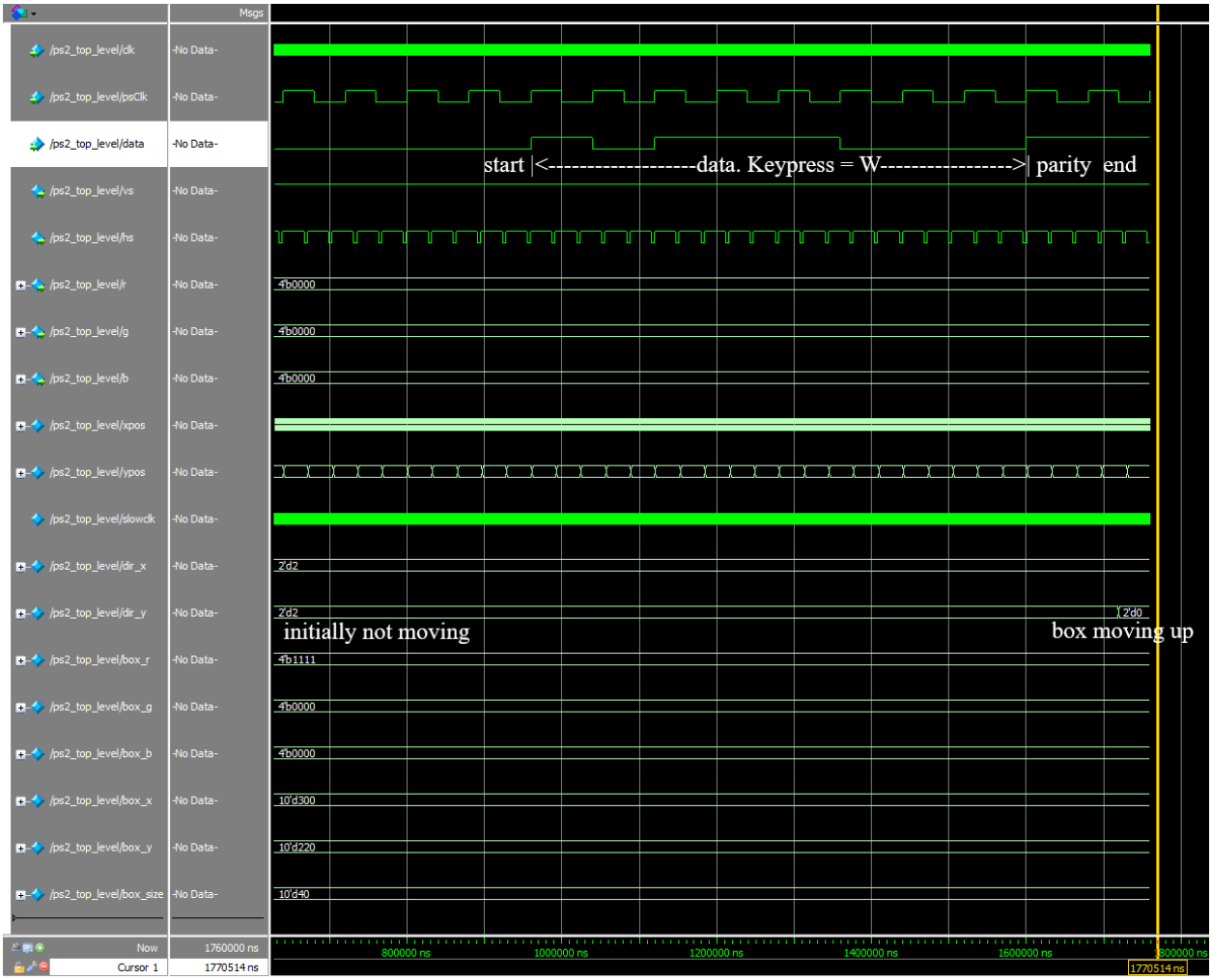

Figure 5.0.1: PS/2 Top Level module's simulation testing - Box starts moving up after receiving a W keypress

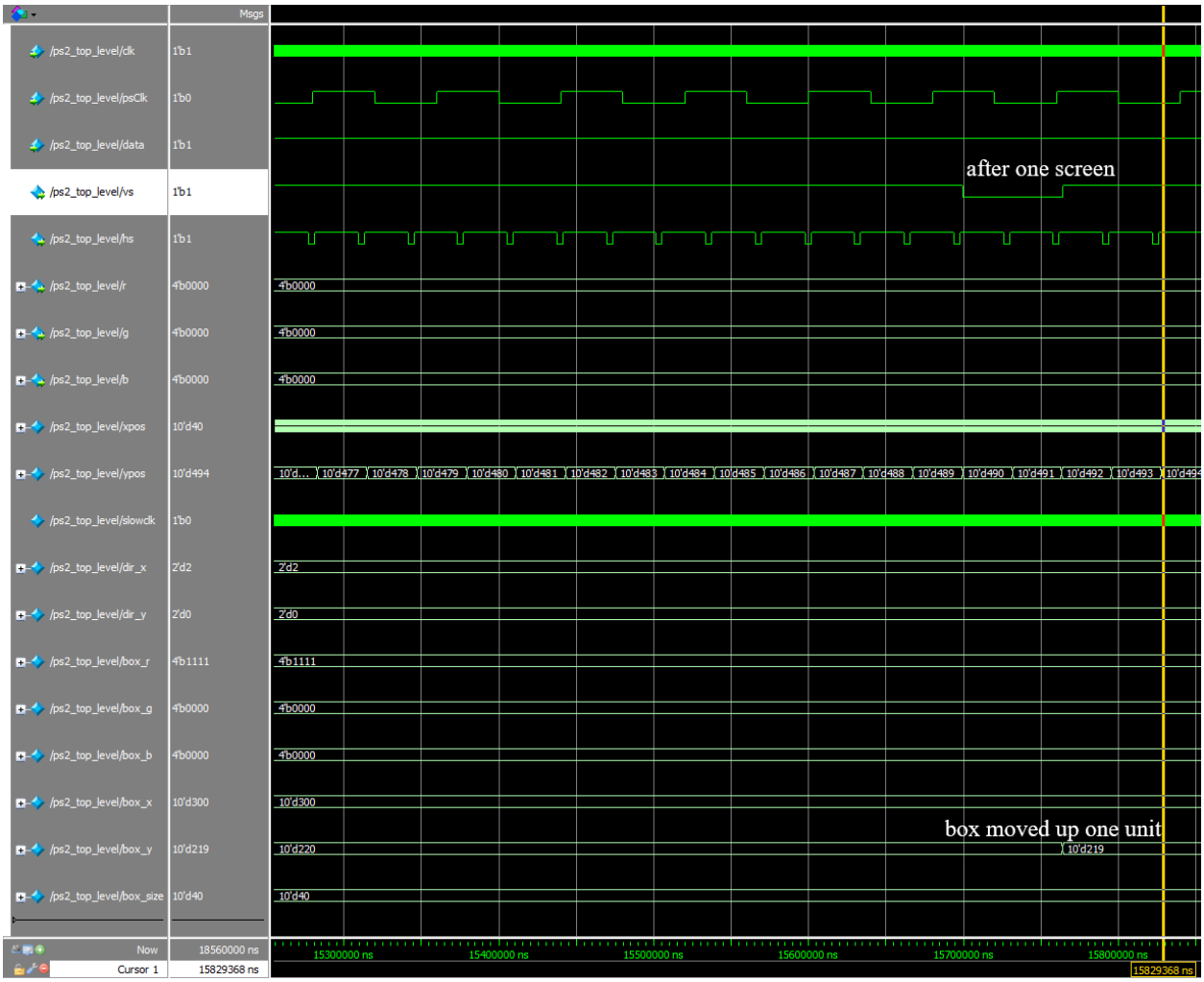

Figure 5.0.2: PS/2 Top Level module's simulation testing - Box has moved up 1 pixel. For the remaining 219 pixels the box has left before colliding with the screen, it moves in the same way

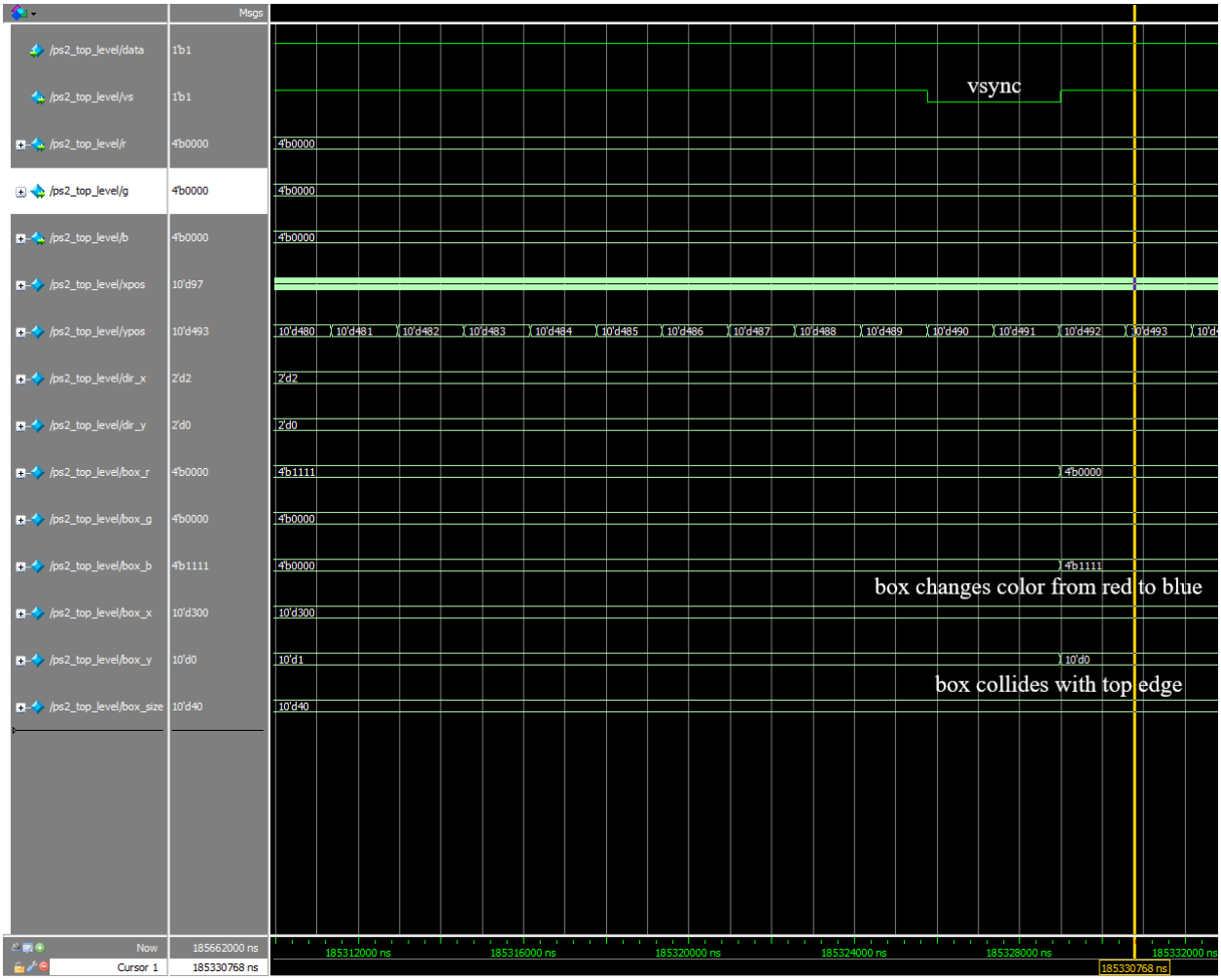

Figure 5.0.3: PS/2 Top Level module's simulation testing - Box changes color when colliding with the screen

Since our group decided to implement all three modes of input, constructing this project was essentially like creating three designs with several commonalities. In each design, the XY Counter, Half Clock, Box Controller, and Drawer modules were all used in transferring data from the FPGA to a VGA output. The one difference between each design was implementation of the decoder module. Each unique form of input needed a separate decoder module that was able to read and transmit data correctly from the controller to the other modules within the FPGA. A top level HDL file was made for all three modes of input and tested thoroughly using ModelSim.

6 Above and Beyond

After looking at the rubric, our group decided to take it one step further by accomplishing all of the tasks listed on the rubric. We successfully implemented VGA output using all three

forms of input: PS/2, RC5 IR, and NES. In addition, we attempted to animate the behavior of the outputted image, rather than having a static and plain figure. By adding the box controller module, we included logic that would force the image to change color and direction upon one condition. If the box hit the edge of the active pixel range, whether in the horizontal or vertical direction, then it would change color. In addition to changing color, the box stops moving, creating a collision.

7 Appendix

7.1 SystemVerilog Source Code

7.1.1 Top Level

```
module top_level(
    input logic clk,
    input logic[1:0] dir_x, dir_y,
    output logic vs, hs,
    output logic[3:0] r, g, b
);

logic[9:0] xpos, ypos;
logic slowclk;

logic[3:0] box_r, box_g, box_b;
logic[9:0] box_x, box_y, box_size;

half_clock hc (
    .oldclk(clk),
    .newclk(slowclk)
);

XYCounter c (
    .clk(slowclk),
    .vs(vs),
    .hs(hs),
    .x(xpos),
    .y(ypos)
);

box_controller box (
```

```
28      .vs(vs),
29      .dir_x(dir_x),
30      .dir_y(dir_y),
31      .x(xpos),
32      .y(ypos),
33      .box_r(box_r),
34      .box_g(box_g),
35      .box_b(box_b),
36      .box_x(box_x),
37      .box_y(box_y),
38      .box_size(box_size)
39  );
40
41  Drawer d (
42      .clk(slowclk),
43      .box_r(box_r),
44      .box_g(box_g),
45      .box_b(box_b),
46      .x(xpos),
47      .y(ypos),
48      .box_x(box_x),
49      .box_y(box_y),
50      .box_size(box_size),
51      .r(r),
52      .g(g),
53      .b(b)
54  );
55
56  endmodule
```

7.1.2 PS/2 Top Level

```
1   module ps2_top_level(
2       input logic clk, psClk, data,
3       output logic vs, hs,
4       output logic[3:0] r, g, b
5   );
6
7   // Track x and y and slow clock
```

```
8   logic[9:0] xpos, ypos;
9   logic slowclk;
10
11  // Get direction input
12  logic[1:0] dir_x, dir_y;
13
14  // Track box info
15  logic[3:0] box_r, box_g, box_b;
16  logic[9:0] box_x, box_y, box_size;
17
18  // Slow clock
19  half_clock hc (
20      .oldclk(clk),
21      .newclk(slowclk)
22  );
23
24  // Generate position and vs/hs signals
25  XYCounter c (
26      .clk(slowclk),
27      .vs(vs),
28      .hs(hs),
29      .x(xpos),
30      .y(ypos)
31  );
32
33  // Get keyboard input for direction
34  ps2 P (
35      .clk(psClk),
36      .data(data),
37      .dir_x(dir_x),
38      .dir_y(dir_y)
39  );
40
41  // Move box
42  box_controller box (
43      .vs(vs),
44      .dir_x(dir_x),
45      .dir_y(dir_y),
46      .box_r(box_r),
47      .box_g(box_g),
48      .box_b(box_b),
```

```
49      .box_x(box_x),
50      .box_y(box_y),
51      .box_size(box_size)
52  );
53
54  // Draw box
55  Drawer d (
56      .clk(slowclk),
57      .box_r(box_r),
58      .box_g(box_g),
59      .box_b(box_b),
60      .x(xpos),
61      .y(ypos),
62      .box_x(box_x),
63      .box_y(box_y),
64      .box_size(box_size),
65      .r(r),
66      .g(g),
67      .b(b)
68  );
69
70  endmodule
```

7.1.3 PS/2 Controller

```
1   module ps2 #(parameter
2       UP = 8'h1D, DOWN = 8'h1B, LEFT = 8'h1C, RIGHT = 8'h23  // PS2 key code of W,
        ↪  S, A, and D, repsectively
3   )(
4       input logic clk, data,
5       output logic[1:0] dir_x, dir_y
6   );
7
8   // Track data input
9   logic[7:0] bits = 8'b0;
10
11  // Track count of bits coming in and number of ones
12  logic[3:0] count = 0;
13  logic[3:0] even_odd = 0;
```

```
14
15  // Hold output values
16  // 0 = up/left
17  // 1 = down/right
18  // 2 = stop
19  logic[1:0] x = 2;
20  logic[1:0] y = 2;
21
22  // Track valid input and whether past input was F0 (indicating key release)
23  logic valid = 0;
24  logic fo = 0;
25
26  always_ff @(negedge clk)
27      begin
28          if (count == 10) // Ending bit
29              begin
30                  if (valid)
31                      // Translate directions to movement
32                      begin
33                          // not released && is not moving && to move up -> move up
34                          if (~fo && y == 2 && bits == UP) y <= 0;
35                          // released && is moving up && to move up -> stop
36                          else if (fo && y == 0 && bits == UP) y <= 2;
37                          // not released && is not moving && to move down -> move
                            ↪  down
38                          else if (~fo && y == 2 && bits == DOWN) y <= 1;
39                          // released && is moving down && to move down -> stop
40                          else if (fo && y == 1 && bits == DOWN) y <= 2;
41
42                          // not released && is not moving && to move left -> move
                            ↪  left
43                          if (~fo && x == 2 && bits == LEFT) x <= 0;
44                          // released && is moving left && to move left -> stop
45                          else if (fo && x == 0 && bits == LEFT) x <= 2;
46                          // not released && is not moving && to move right -> move
                            ↪  right
47                          else if (~fo && x == 2 && bits == RIGHT) x <= 1;
48                          // released && is moving right && to move right -> stop
49                          else if (fo && x == 1 && bits == RIGHT) x <= 2;
50
51                          // Get code for key release
```

```
52                          if (bits == 8'hF0) fo <= 1;
53                          else fo <= 0;
54                      end
55
56                  even_odd <= 0;
57                  count <= 0;
58                  bits <= 8'd0;
59                  valid <= 0;
60              end
61          else if (count == 9)  // Check parity bit
62              begin
63                  count <= count + 1;
64
65                  // Check validity based on number of ones and parity bit (odd
                    ↪  total)
66                  valid <= even_odd[0] == ~data;
67              end
68          else if (count == 0)  // Start bit -- do nothing but advance count
69              count <= count + 1;
70          else
71              begin
72                  // Track count and number of odd
73                  even_odd <= even_odd + data;
74                  count <= count + 1;
75
76                  // Load data
77                  bits <= (bits >> 1) + (data * 128);
78              end
79      end
80
81  // Set output
82  assign dir_x = x;
83  assign dir_y = y;
84
85  endmodule
```

7.1.4 RC5 IR Top Level

```
module ir_top_level(
    input logic clk, data,
    output logic vs, hs,
    output logic[3:0] r, g, b
);

// Track x and y and slow clock
logic[9:0] xpos, ypos;
logic slowclk;

// Get direction input
logic[1:0] dir_x, dir_y;

// Track box info
logic[3:0] box_r, box_g, box_b;
logic[9:0] box_x, box_y, box_size;

// Slow clock
half_clock hc (
    .oldclk(clk),
    .newclk(slowclk)
);

// Generate position and vs/hs signals
XYCounter c (
    .clk(slowclk),
    .vs(vs),
    .hs(hs),
    .x(xpos),
    .y(ypos)
);

// Get IR input for direction
ir I (
    .clk(clk),
    .data(data),
    .dir_x(dir_x),
    .dir_y(dir_y)
```

```
);

// Move box
box_controller box (
    .vs(vs),
    .dir_x(dir_x),
    .dir_y(dir_y),
    .box_r(box_r),
    .box_g(box_g),
    .box_b(box_b),
    .box_x(box_x),
    .box_y(box_y),
    .box_size(box_size)
);

// Draw box
Drawer d (
    .clk(slowclk),
    .box_r(box_r),
    .box_g(box_g),
    .box_b(box_b),
    .x(xpos),
    .y(ypos),
    .box_x(box_x),
    .box_y(box_y),
    .box_size(box_size),
    .r(r),
    .g(g),
    .b(b)
);

endmodule
```

7.1.5 RC5 IR Controller

```
// IR module for RC5 code
module ir #(parameter
    UP = 2, DOWN = 8, LEFT = 4, RIGHT = 6
)(
```

```
5       input logic data, clk,
6       output logic[1:0] dir_x, dir_y
7   );
8
9   // Count for slowing clock and new clock
10  logic[10:0] clkCount = 0;
11  logic newclk = 0;
12
13  // Complete data word and its update status
14  logic[5:0] dataWord = 0;
15  logic newDataWord = 0;
16
17  // Internal x/y movement
18  logic[1:0] x = 2;
19  logic[1:0] y = 2;
20
21  // Used for handling pulses
22  logic pusleVal = 0;
23  logic newPulseVal = 0;
24  logic waitForRepeat = 0;
25
26  // Slow clock from 50MHz to 36KHz as is used for RC5
27  always_ff @(posedge clk)
28      if (clkCount < 1388)
29              clkCount <= clkCount + 1;
30      else
31          begin
32              clkCount <= 0;
33              newclk = ~newclk;
34          end
35
36  // Converts from encoded pulse to binary
37  irConvertPulse cp(
38      .data(data),
39      .clk(newclk),
40      .val(pusleVal),
41      .newval(newPulseVal),
42      .waitForRepeat(waitForRepeat)
43  );
44
45  // Converts from encoded binary to data word
```

```
46  irReadWord rw(
47      .val(pulseVal),
48      .clk(newPulseVal),
49      .waitForRepeat(waitForRepeat),
50      .data(dataWord),
51      .finished(newDataWord)
52  );
53
54  // Translate from data word to movement value
55  always_ff @(posedge newDataWord)
56      begin
57          if (dataWord == UP) y <= 0;
58          else if (dataWord == DOWN) y <= 1;
59          else y <= 2;
60
61          if (dataWord == LEFT) x <= 0;
62          else if (dataWord == RIGHT) x <= 1;
63          else x <= 2;
64      end
65
66  assign dir_x = x;
67  assign dir_y = y;
68
69  endmodule
70
71
72  module irReadWord(
73      input logic val, clk, waitForRepeat,
74      output logic[5:0] data,
75      output logic finished
76  );
77
78  // Stores output and counts bits
79  logic[5:0] out = 0;
80  logic[4:0] count = 0;
81
82  // Tracks previous toggle -- not used in this implementation
83  // but necessary for certain controls so it will be tracked
84  // anyway
85  logic prevToggle = 1;
86
```

```
87  always_ff @(posedge clk, negedge waitForRepeat)
88      begin
89          if (~waitForRepeat)  // Reset if time limit for repeating signal is up
90              begin
91                  out <= 0;
92                  count <= 0;
93                  finished <= 1;
94              end
95          else if (count == 13)  // Reset count, store final value, and signal
            ↪  finished transmission
96              begin
97                  out <= (out << 1) + val;
98                  finished <= 1;
99                  count <= 0;
100             end
101         else if (count == 2 && val != prevToggle)  // Set toggle and reset value
            ↪  if different button press
102             begin
103                 prevToggle <= val;
104                 out <= 0;
105                 count <= count + 1;
106                 finished <= 0;
107             end
108         else if (count > 7)  // If in data word, add to output
109             begin
110                 out <= (out << 1) + val;
111                 count <= count + 1;
112                 finished <= 0;
113             end
114         else
115             begin
116                 count <= count + 1;
117                 finished <= 0;
118             end
119     end
120
121 assign data = out;
122
123 endmodule
124
125
```

```
126 module irConvertPulse(
127     input logic data, clk,
128     output logic val, newval, waitForRepeat
129 );
130 
131 // Tracks two sections with potential to have pulses
132 logic[5:0] dataCounterOne = 0;
133 logic[5:0] dataCounterTwo = 0;
134 
135 // Counts time
136 logic[5:0] timeCounter = 0;
137 
138 // Counts time from
139 logic[6:0] waitCounter = 78;
140 logic idle = 1;
141 logic out = 0;
142 
143 logic tcReset = 0;
144 logic dcReset = 0;
145 logic dcResetSuccess = 0;
146 
147 always_ff @(posedge data)
148     begin
149         if (idle) // Reset time counter if start of a transmission
150             begin
151                 dataCounterOne <= 0;
152                 dataCounterTwo <= 1;
153                 tcReset <= 1;
154             end
155         else if (dcReset)
156             begin
157                 if (timeCounter > 31 && dataCounterTwo < 32)
158                     begin
159                         dataCounterOne <= 0;
160                         dataCounterTwo <= 1;
161                     end
162                 else if (dataCounterOne < 32)
163                     begin
164                         dataCounterOne <= 1;
165                         dataCounterTwo <= 0;
166                     end
```

```
167
168                 dcResetSuccess <= 1;
169             end
170         else  // Increment data
171             begin
172                 if (timeCounter > 31 && dataCounterTwo < 32) dataCounterTwo <=
                    ↪  dataCounterTwo + 1;
173                 else if (dataCounterOne < 32) dataCounterOne <= dataCounterOne +
                    ↪  1;
174
175                 tcReset <= 0;
176                 dcResetSuccess <= 0;
177             end
178     end
179
180 always_ff @(posedge clk)
181     begin
182         if (tcReset)
183             begin
184                 timeCounter <= 33;
185                 waitCounter <= 0;
186                 idle <= 0;
187             end
188         else if (timeCounter == 63)  // Set output and reset
189             begin
190                 // Increment counter if still in window for additional
191                 if (idle && dataCounterOne == 0 && dataCounterTwo == 0 &&
                    ↪  waitCounter < 78)
192                     waitCounter <= waitCounter + 1;
193
194                 out <= dataCounterOne == 0 && dataCounterTwo == 32 && ~dcReset;
195                 newval <= (dataCounterOne == 32 || dataCounterTwo == 32) &&
                    ↪  ~dcReset;
196                 idle <= (dataCounterOne == 0 && dataCounterTwo == 0) || ~dcReset;
197                 dcReset <= 1;
198                 timeCounter <= 0;
199             end
200         else  // Increment data
201             begin
202                 timeCounter <= timeCounter + 1;
203                 newval <= 0;
```

```
204
205                 if (dcResetSuccess)
206                     dcReset <= 0;
207             end
208     end
209
210 assign val = out;
211
212 // Do not wait if outside of continued transmission window
213 assign waitForRepeat = waitCounter < 78;
214
215 endmodule
```

7.1.6 NES Top Level

```
1  module nes_top_level(
2      input logic clk, data,
3      output logic vs, hs, latch_out, clk_out,
4      output logic[3:0] r, g, b
5  );
6
7  // Track x and y and slow clock
8  logic[9:0] xpos, ypos;
9  logic slowclk;
10
11 // Get direction input
12 logic[1:0] dir_x, dir_y;
13
14 // Track box info
15 logic[3:0] box_r, box_g, box_b;
16 logic[9:0] box_x, box_y, box_size;
17
18 // Slow clock
19 half_clock hc (
20     .oldclk(clk),
21     .newclk(slowclk)
22 );
23
24 // Generate position and vs/hs signals
```

```
25  XYCounter c (
26      .clk(slowclk),
27      .vs(vs),
28      .hs(hs),
29      .x(xpos),
30      .y(ypos)
31  );
32
33  // Get nes input for direction
34  nes N (
35      .clk(clk),
36      .data(data),
37      .dir_x(dir_x),
38      .dir_y(dir_y),
39      .latch_out(latch_out),
40      .clk_out(clk_out)
41  );
42
43  // Move box
44  box_controller box (
45      .vs(vs),
46      .dir_x(dir_x),
47      .dir_y(dir_y),
48      .box_r(box_r),
49      .box_g(box_g),
50      .box_b(box_b),
51      .box_x(box_x),
52      .box_y(box_y),
53      .box_size(box_size)
54  );
55
56  // Draw box
57  Drawer d (
58      .clk(slowclk),
59      .box_r(box_r),
60      .box_g(box_g),
61      .box_b(box_b),
62      .x(xpos),
63      .y(ypos),
64      .box_x(box_x),
65      .box_y(box_y),
```

```
66      .box_size(box_size),
67      .r(r),
68      .g(g),
69      .b(b)
70  );
71
72  endmodule
```

7.1.7 NES Controller

```
1   module nes #(parameter
2       UP = 5, DOWN = 6, LEFT = 7, RIGHT = 8
3   )(
4       input logic data, clk,
5       output logic latch_out, clk_out,
6       output logic[1:0] dir_x, dir_y
7   );
8
9   // Track status of x and y direction and if a directional button was pressed
10  logic[1:0] x = 2;
11  logic[1:0] y = 2;
12  logic x_btn_press = 0;
13  logic y_btn_press = 0;
14
15  // Track when to poll controller for button presses
16  logic[19:0] pollClkCount = 0;
17  logic pollClk = 0;
18
19  // Track when to pulse clock
20  logic[9:0] pulseClkCount = 0;
21  logic pulseClk = 0;
22
23  // Track info about latch and clock pulse
24  logic latch_inner = 0;
25  logic new_pulse = 0;
26  logic start_pulse_out = 0;
27  logic[3:0] pulseCounter = 0;
28
29  // Convert clk from 50MHz to 60Hz for polling controller
```

```
30  // and to 83kHz to produce clock pulses
31  always_ff @(posedge clk)
32      begin
33          if (pollClkCount == 416666)
34              begin
35                  // Start polling controller by resetting pulse data, starting
                    ↪  latch,
36                  // and signaling start of new pulse
37                  if (~pollClk)
38                      begin
39                          latch_inner <= 1;
40                          new_pulse <= 1;
41                          pulseClkCount <= 0;
42                          pulseClk <= 0;
43                          pulseCounter <= 0;
44                      end
45
46                  pollClkCount <= 0;
47                  pollClk <= ~pollClk;
48              end
49          else
50              pollClkCount <= pollClkCount + 1;
51
52          if (pulseClkCount == 301)
53              begin
54                  if (new_pulse && ~pulseClk)  // Only update if controller poll
                    ↪  has started
55                      begin
56                          // Track beginning of clock pulse to controller (after
                            ↪  latch)
57                          if (pulseCounter > 0 && latch_inner == 0)
58                              start_pulse_out <= 1;
59
60                          // Count pulse
61                          pulseCounter <= pulseCounter + 1;
62                      end
63
64                  if (new_pulse && pulseClk)  // Only update if controller poll has
                    ↪  started
65                      begin
66                          // Reset latch once its 12us have passed
```

```
67                      if (pulseCounter > 0 && latch_inner)
68                          latch_inner <= 0;
69
70                      // Reset pulse data if all buttons have been accounted
71                      if (pulseCounter == 9)
72                          begin
73                              new_pulse <= 0;
74                              start_pulse_out <= 0;
75                              y_btn_press <= 0;
76                              x_btn_press <= 0;
77                          end
78                      else
79                          begin
80                              // Check for UP or DOWN buttons being pressed and
                                ↪  track that
81                              // a button in y direction has been pressed
82                              if (pulseCounter == UP && data == 0)
83                                  begin
84                                      y <= 0;
85                                      y_btn_press <= 1;
86                                  end
87                              else if (pulseCounter == DOWN && data == 0)
88                                  begin
89                                      y <= 1;
90                                      y_btn_press <= 1;
91                                  end
92                              else if (pulseCounter >= UP && pulseCounter >=
                                ↪  DOWN && y_btn_press == 0)  // Reset if no y
                                ↪  button press
93                                  y <= 2;
94
95                              // Check for RIGHT or LEFT buttons being pressed
                                ↪  and track that
96                              // a button in x direction has been pressed
97                              if (pulseCounter == LEFT && data == 0)
98                                  begin
99                                      x <= 0;
100                                     x_btn_press <= 1;
101                                 end
102                             else if (pulseCounter == RIGHT && data == 0)
103                                 begin
```

```
                                        x <= 1;
                                        x_btn_press <= 1;
                                    end
                                else if (pulseCounter >= LEFT && pulseCounter >=
                                ↪  RIGHT && y_btn_press == 0)  // Reset if no x
                                ↪  button press
                                    x <= 2;
                            end
                        end

                    pulseClkCount <= 0;
                    pulseClk <= ~pulseClk;
                end
            else
                pulseClkCount <= pulseClkCount + 1;
        end

    // Assign outputs, and only set clock if part of pulse output
    assign latch_out = latch_inner;
    assign clk_out = start_pulse_out && pulseClk;
    assign dir_x = x;
    assign dir_y = y;

    endmodule
```

7.1.8 Box Controller

```
module box_controller #(parameter
    SIZE = 40
)(
    input logic vs,
    input logic[1:0] dir_x, dir_y,
    output logic[3:0] box_r, box_g, box_b,
    output logic[9:0] box_x, box_y, box_size
);

// Track x and y
logic[9:0] xpos = 300;
logic[9:0] ypos = 220;
```

```
13
14  // Track box color
15  logic[3:0] r = 4'b1111;
16  logic[3:0] g = 4'b0;
17  logic[3:0] b = 4'b0;
18
19  always_ff @(posedge vs)
20      begin
21          if(~(dir_x == 0 && xpos == 0) && ~(dir_x == 1 && xpos == 640 - SIZE))  //
            ↪  Check that box will remain in screen bounds
22              begin
23                  if((xpos - 1 == 0 && dir_x == 0) || (xpos + 1 == 640 - SIZE &&
                    ↪  dir_x == 1))  // Check for collision
24                      begin
25                          // Swap colors if collision occurs
26                          r <= g;
27                          g <= b;
28                          b <= r;
29                      end
30
31                  // Increment position
32                  case(dir_x)
33                      0: xpos <= xpos - 1;
34                      1: xpos <= xpos + 1;
35                  endcase
36              end
37
38          if(~(dir_y == 0 && ypos == 0) && ~(dir_y == 1 && ypos == 480 - SIZE))  //
            ↪  Check that box will remain in screen bounds
39              begin
40                  if((ypos - 1 == 0 && dir_y == 0) || (ypos + 1 == 480 - SIZE &&
                    ↪  dir_y == 1))  // Check for collision
41                      begin
42                          // Swap colors if collision occurs
43                          r <= g;
44                          g <= b;
45                          b <= r;
46                      end
47
48                  // Increment position
49                  case(dir_y)
```

```
                    0: ypos <= ypos - 1;
                    1: ypos <= ypos + 1;
                endcase
            end
    end

assign box_x = xpos;
assign box_y = ypos;
assign box_size = SIZE;
assign box_r = r;
assign box_g = g;
assign box_b = b;

endmodule
```

7.1.9 Half Clock

```
module half_clock(
    input logic oldclk,
    output logic newclk
);

logic count = 0;

always_ff @(posedge oldclk)
    if (count == 1)
        begin
            newclk <= 1;
            count <= 0;
        end
    else
        begin
            newclk <= 0;
            count <= count + 1;
        end

endmodule
```

7.1.10 XY Counter

```
module XYCounter(
    input logic clk,
    output logic vs, hs,
    output logic[9:0] x, y
);

// Track current x and y
logic[9:0] xpos = 0;
logic[9:0] ypos = 0;

always_ff @(posedge clk)
    begin
        // Generate HSync and VSync signals when in sync section
        hs <= ~((xpos >= 640 + 16) && (xpos < 640 + 16 + 96));
        vs <= ~((ypos >= 480 + 10) && (ypos < 480 + 10 + 2));

        if(xpos == 800 && ypos == 525)  // Reset when at end of screen
            begin
                ypos <= 0;
                xpos <= 0;
            end
        else if(xpos == 800)  // Increment y and reset x at end of line
            begin
                ypos <= ypos + 1;
                xpos <= 0;
            end
        else  // Otherwise increment x
            xpos <= xpos + 1;
    end

assign x = xpos;
assign y = ypos;

endmodule
```

7.1.11 Drawer

```
module Drawer(
    input logic clk,
    input logic[3:0] box_r, box_g, box_b,
    input logic[9:0] x, y, box_x, box_y, box_size,
    output logic[3:0] r, g, b
);

always_ff @(posedge clk)
    // Draw black if not in box pixels
    if (x > 640 || y > 480 || x < box_x || y < box_y || x > box_x + box_size || y
    ↪  > box_y + box_size)
        begin
            r <= 4'b0;
            g <= 4'b0;
            b <= 4'b0;
        end
    else  // Draw box
        begin
            r <= box_r;
            g <= box_g;
            b <= box_b;
        end

endmodule
```