ECE 271 Group 2 Design Project

Taylor Brennan, Marcus Wheeler, Tyler Walker

June 4th, 2021

Contents

1 Project Description **3**
1.1 Project Deliverables . . . 3

2 IR TV Remote Description **4**
2.1 Timing and Power . . . 4

3 High Level Description **5**
3.0.1 Top Level Simulation . . . 5
3.1 IR Receiver . . . 5
3.1.1 IR Receiver Simulation . . . 6
3.2 IR Decoder . . . 7
3.2.1 First Multiplexer . . . 7
3.2.2 Second Multiplexer . . . 8
3.2.3 Comparator . . . 9
3.2.4 Shift Register . . . 9
3.2.5 Address Swapper . . . 10
3.2.6 Tristate . . . 11
3.2.7 IR Decoder Simulation . . . 11
3.3 VGA Display Output . . . 12
3.3.1 Top Level Diagram . . . 12
3.3.2 Counter . . . 12
3.3.3 Sync Count . . . 12
3.3.4 Color Mux . . . 14
3.4 Sprite VGA . . . 15
3.4.1 Top Level Diagram . . . 15
3.4.2 Address Modifier . . . 16
3.4.3 Sprite Comparator . . . 16
3.4.4 Address Converter . . . 17
3.4.5 Sprite ROM . . . 17
3.4.6 Display Mux . . . 18
3.4.7 VGA Display . . . 18
3.4.8 VGA Display Output Simulation . . . 18

A SystemVerilog Files **19**
A.1 IR Receiver . . . 19
A.1.1 Comparator . . . 19
A.1.2 Clock Counter . . . 19
A.1.3 Enable Counter . . . 19
A.1.4 Multiplexer . . . 19
A.1.5 Shift Register . . . 19
A.1.6 Tristate . . . 19
A.1.7 Synchronizer . . . 20
A.2 IR Decoder . . . 20
A.2.1 Comparator . . . 20
A.2.2 First Multiplexer . . . 20
A.2.3 Second Multiplexer . . . 20

A.2.4 Shift Register . . . 20
A.2.5 Address Swapper . . . 20
A.2.6 Tristate . . . 21
A.3 VGA Display Output . . . 21
A.3.1 Counter . . . 21
A.3.2 Sync Count . . . 21
A.3.3 Color Mux . . . 22
A.4 Sprite VGA . . . 22
A.4.1 Address Modifier . . . 22
A.4.2 Sprite Comparator . . . 22
A.4.3 Address Converter . . . 22
A.4.4 Display Mux . . . 22

B Do files **23**
B.0.1 RC-5 Encoder . . . 23
B.0.2 IR Decoder . . . 23

1 Project Description

The goal of this project is to design logic hardware to read an infrared (IR) signal from a TV remote and move a sprite on a VGA display. The TV remote follows the RC-5 IR transmission protocol, which sends a series of time based pulses that correspond with a logic 1 or 0. By using an IR receiver module, decoders to interpret the signals, and registers to convert the signals into sprite movement, the pulses can be used to change where a sprite appears on a VGA display.

Figure 1: RC-5 IR TV Remote (Aliradar ©2021)

In this project, pressing buttons 2, 4, 6, and 8 shown in Figure 1 will move the sprite up, down, left, and right, respectively. The other buttons will not have an effect on the sprite display. By implementing this ability, the sprite movement can be used for other projects, like moving a character in a game or adjusting where a picture shows up on a screen.

1.1 Project Deliverables

This project will be able to:

1. Read a pulse signal from an IR receiver and decode it
2. Use the information from the signal to determine which button was pressed
3. Move a sprite in the direction the button corresponds to, if the button corresponds to one

2 IR TV Remote Description

The RC-5 IR Transmission Protocol was developed by the manufacturer Phillips and is often used for projects, due to the TV remote controls being cheap and compatible with many device types [1]. RC-5 uses bi-phase coding, where a logic "0" is represented by a half-bit pulse burst and a half-bit amount of space, and a logic "1" is represented by the opposite. Figure 2 below shows the bi-phase pulses.

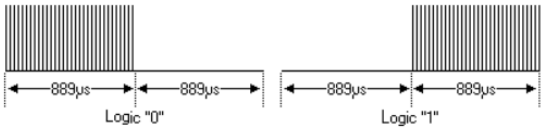

Figure 2: Pulses for Logic "0" and "1" (SB-Projects ©2001)

There are 14 bits in a complete RC-5 signal. The first two bits are start pulses, which always have the value of "1". It is important to note that the IR receiver won't recognize there's a signal being sent until halfway through the first bit. The next bit is the toggle, which inverts every time a key is released. The next five bits are the address, which starts with the most significant bit. The last six bits are the command, which also start with the most significant bit. This data will be repeated as long as the key is pressed. Figure 3 below shows the complete signal for command 35 going to address 05.

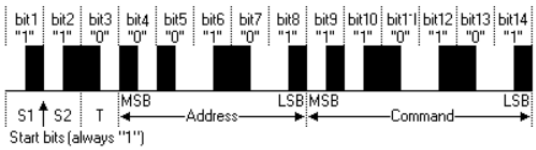

Figure 3: Signal for command 35 (SB-Projects ©2001)

2.1 Timing and Power

Each bit has a constant time of 1.778ms, which means there are 889μs per half bit. The entire signal takes about 25ms to complete, and there are 64 bits passed in a carrier frequency of 36kHz. Figure 4 shows the timing for the same signal described above.

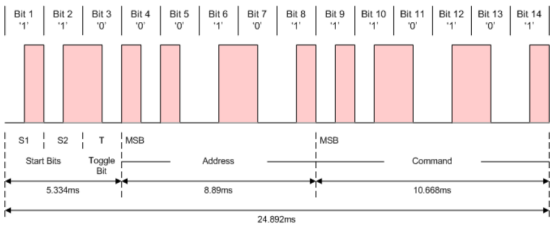

Figure 4: Timing for IR Signal (Altium ©2021)

In terms of power, the pulse-to-pulse ratio of the carrier frequency is 1/3 to 1/4, which reduces power consumption [2]. One of the largest consumers of power in a circuit is the IR transmitter, so reducing the power consumption is crucial [3].

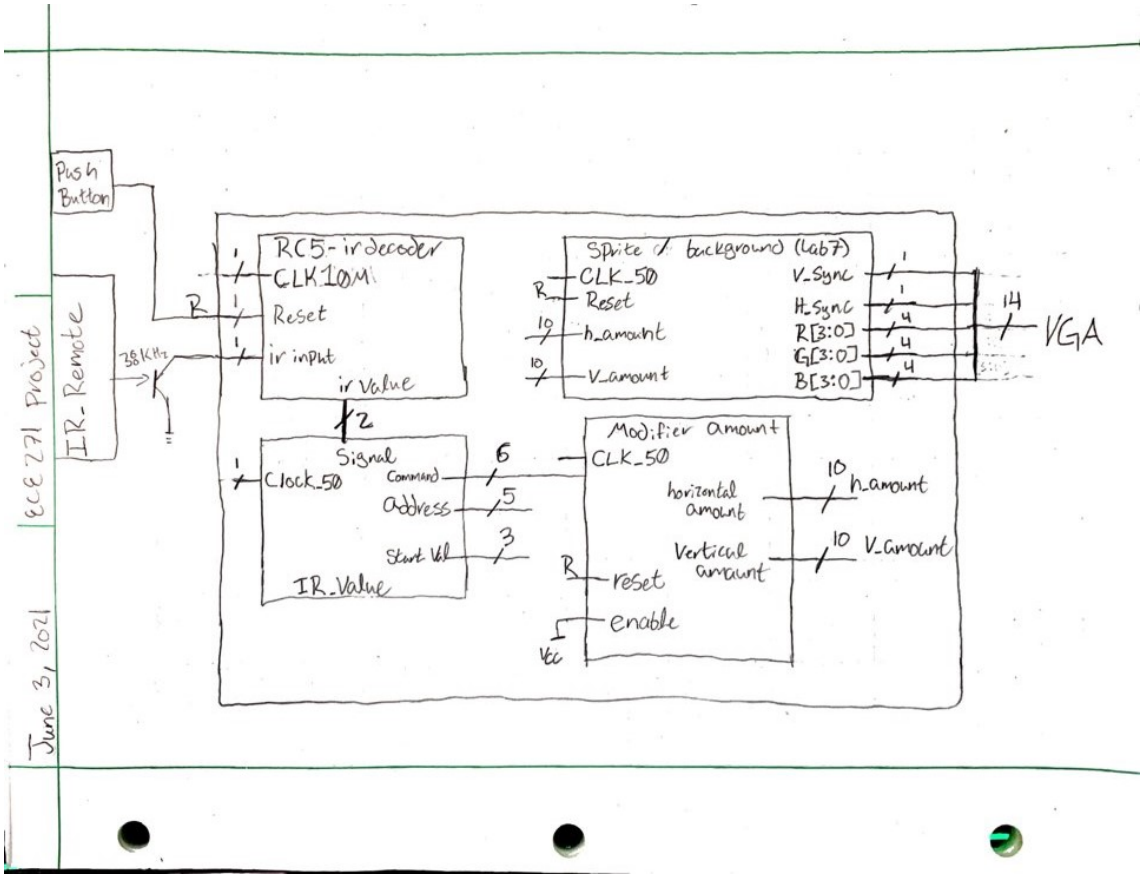

Figure 5: Top level Diagram

Description	FPGA Pin
CLK (50M Hz)	P11
CLK (10M Hz)	N5
Reset	B8
IR Senor Signal	AB5

3 High Level Description

3.0.1 Top Level Simulation

From here, each functional unit of this design will be broken down into its components, and will have a description for what each piece does. If an individual block in more than one functional unit, it will only be described once.

3.1 IR Receiver

Here is the schematic diagram for the IR Receiver:

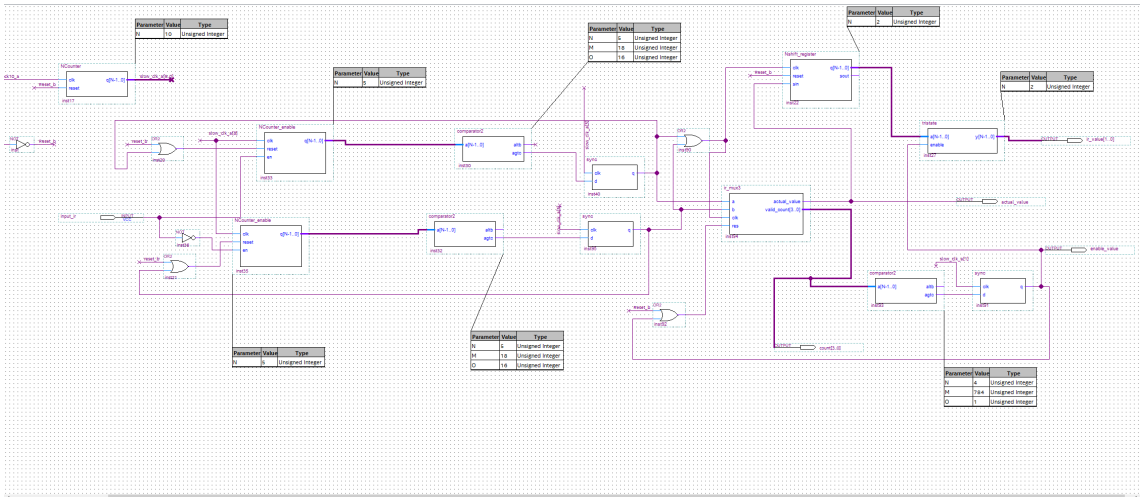

Figure 6: IR Receiver Schematic Diagram

3.1.1 IR Receiver Simulation

Below is the simulation for the IR Receiver.

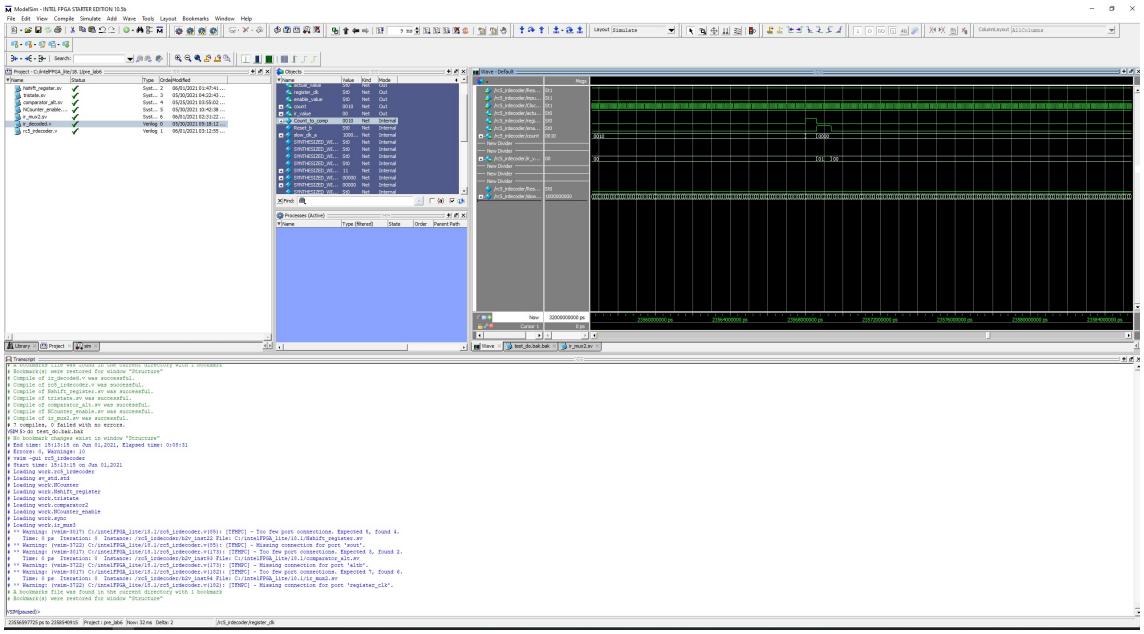

Figure 7: IR Receiver Simulation

Inputs: This reads a IR signal.

Outputs: This outputs a 2-bit binary number that relates to the input.

Description: The input for the IR comes from the from the Ardinuo circuit pin. It counts how long a value has been set to high or low, and stores a value into a register to be outputted when the count of values is 2.

3.2 IR Decoder

Here is a schematic diagram for the IR Decoder:

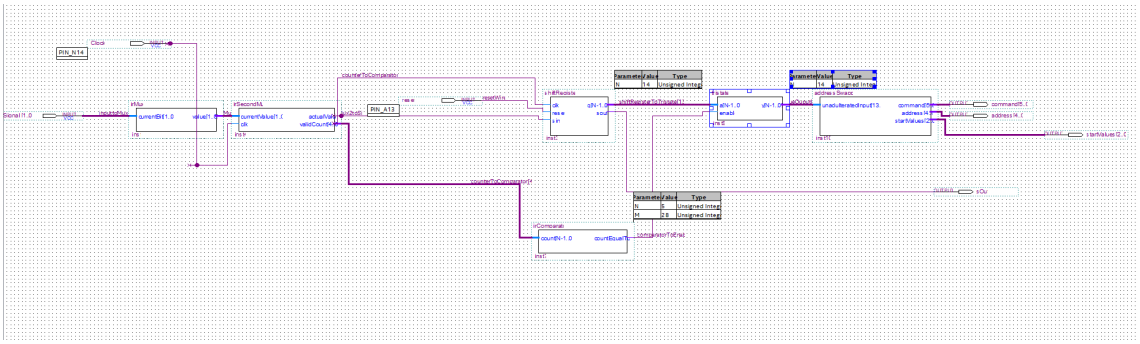

Figure 8: IR Decoder Schematic Diagram

3.2.1 First Multiplexer

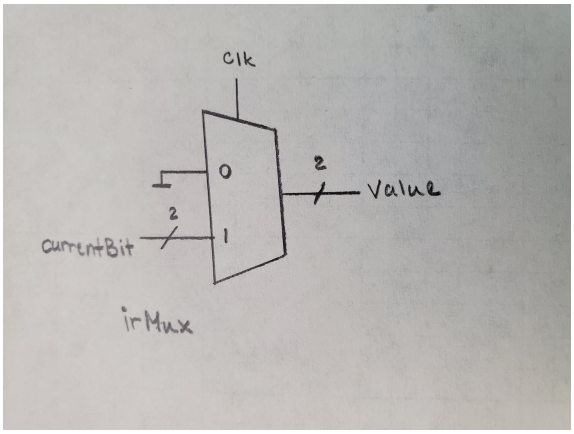

Figure 9: IR Multiplexer 1 Block Diagram

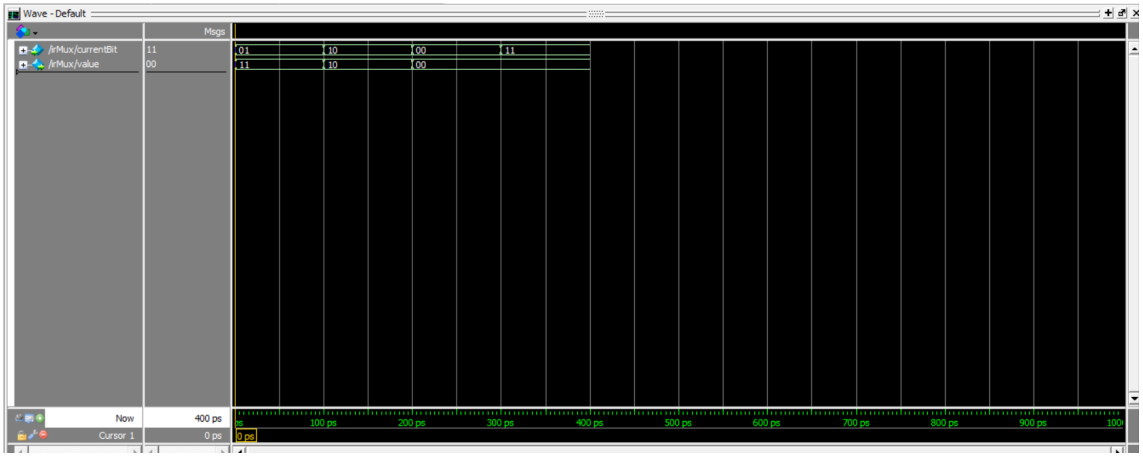

Figure 10: IR Multiplexer 1 simulation

Inputs: This reads a 2-bit number.

Outputs: This outputs a 2-bit binary number that relates to the input.

Description: The input currentBit comes from the IR Receiver output irValue, and the only inputs that are accepted are a 01 (which passes a HIGH value) or a 10 (which passes a LOW value). All other inputs will pass a value of 00, which will do nothing. If passing a HIGH value, we pass 11, and if LOW, we pass 10.

3.2.2 Second Multiplexer

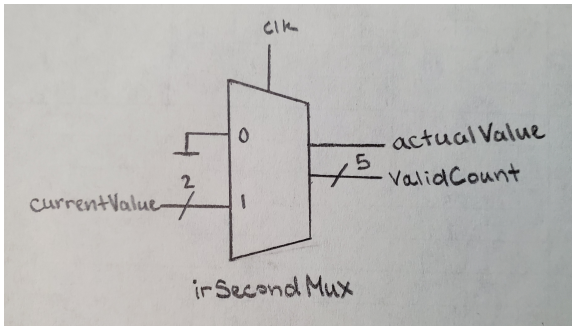

Figure 11: Second Multiplexer Block Diagram

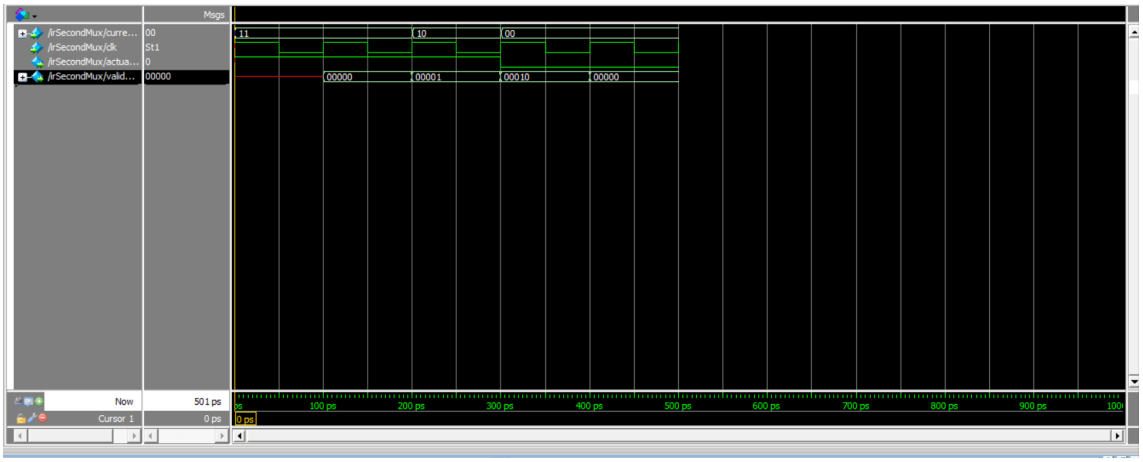

Figure 12: Second Multiplexer Simulation

Inputs: currentValue, clk

Outputs: actualValue, validCount

Description: This module depends on the inputs from our first multiplexer - if I get a 11, I know that I have a 1 - a 10 then a 0 - so I pass that out in actualValue. Additionally, if I get a valid input, I increment my count.

3.2.3 Comparator

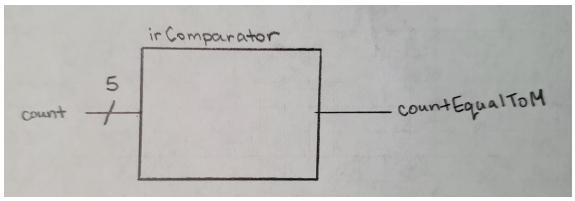

Figure 13: Comparator Block Diagram

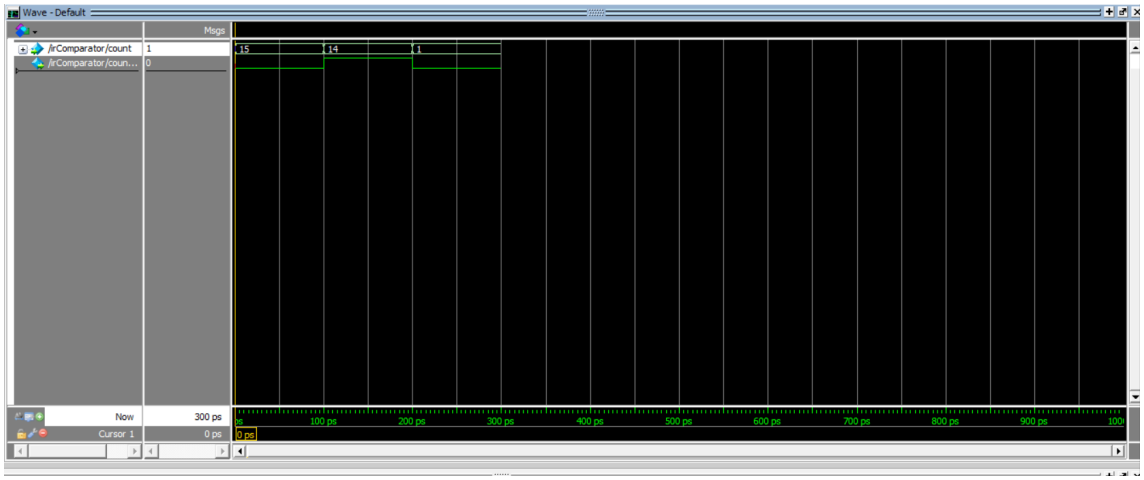

Figure 14: Comparator simulation

Inputs: count

Outputs: countEqualToM

Description: Basic comparator - if my count is equal to my parameter, send a pulse. This pulse lets my values through my tristate and resets my counter. Additionally, because I know that any actual inputs will give me 14 bits, but I'm going to get them in clock cycles of two, so when I get to my 28th increment I can reset my counter and enable my tristate to let my values through because I know for sure my shift register is full.

3.2.4 Shift Register

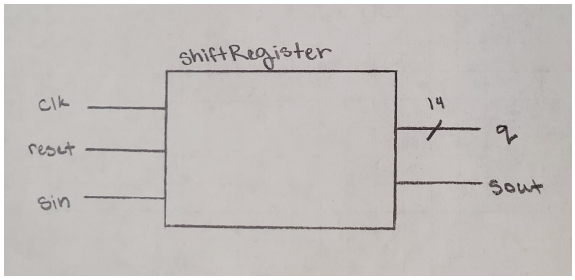

Figure 15: Shift Register Block Diagram

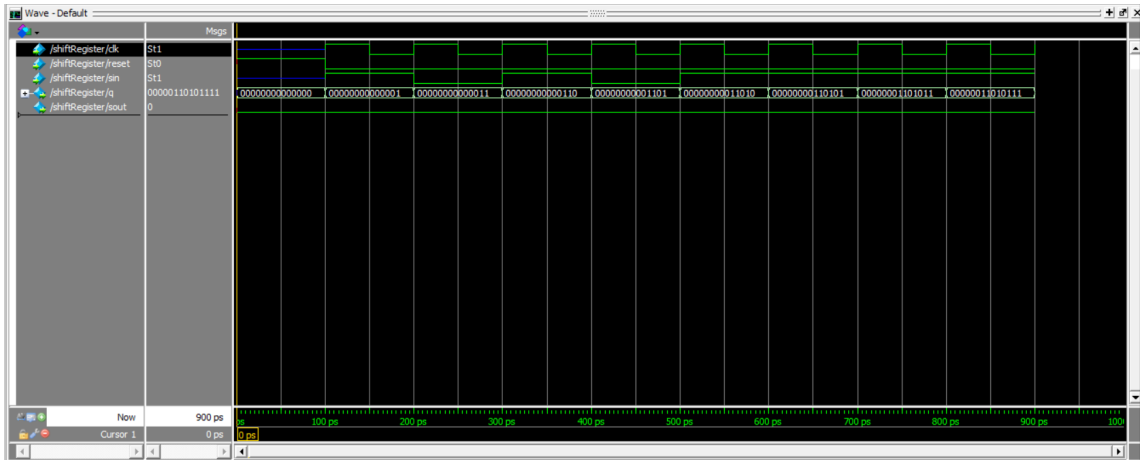

Figure 16: Shift Register simulation

Inputs: clk, reset, sin

Outputs: q, sout

Description: This is a shift register - at every cycle it accepts a new value in and increases all other values' positions in line. For our clock we're using the valid input count from our second multiplexer - that way when we know we got a valid input, we can store it in our shift register and not store noise in the meantime.

3.2.5 Address Swapper

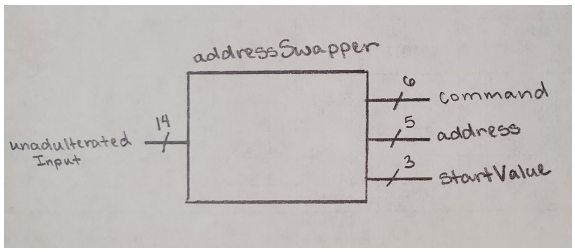

Figure 17: Address Swapper Block Diagram

Figure 18: AddressSwapper simulation

Inputs: unadulteratedInput

Outputs: command, address, startValues

Description: This is an ugly (but guaranteed correct) way of filtering out the values that we need from our shift register. After we've gotten all 14 bits that we need and the tristate passes them through, the address swapper tells us what bits are what and passes them out of the module.

3.2.6 Tristate

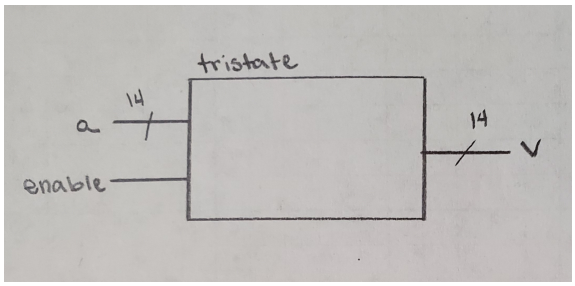

Figure 19: Tristate Block Diagram

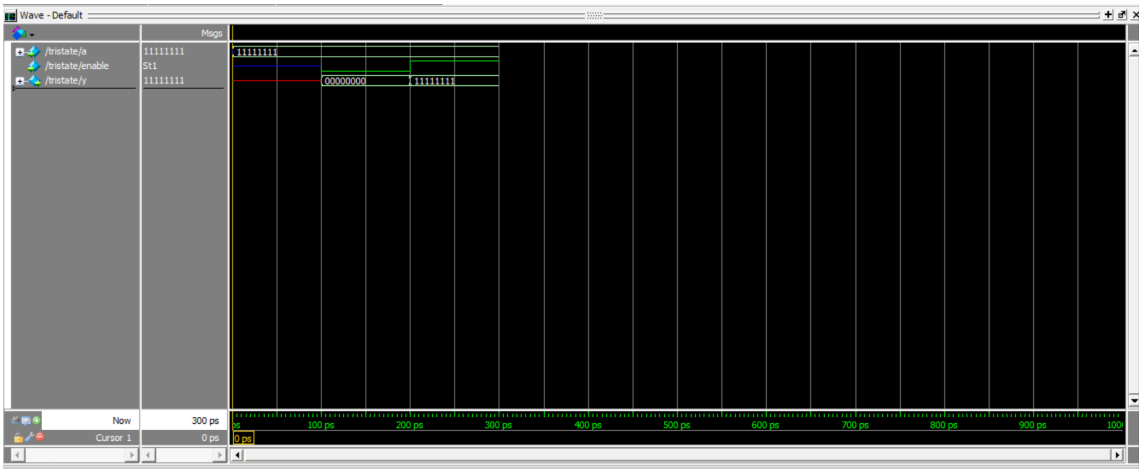

Figure 20: IR Tristate simulation

Inputs: a, enable

Outputs: y

Description: Just a filter for our Shift Register. It makes it so that we don't constantly send nonsense - only when we know that our register is filled with values.

3.2.7 IR Decoder Simulation

Below is the simulation for the IR Decoder.

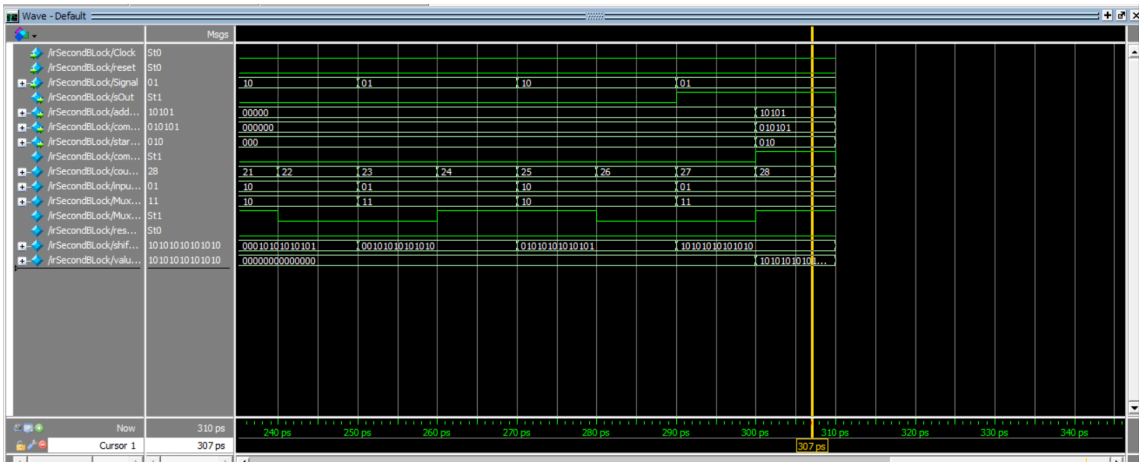

Figure 21: IR Decoder Simulation

3.3 VGA Display Output

3.3.1 Top Level Diagram

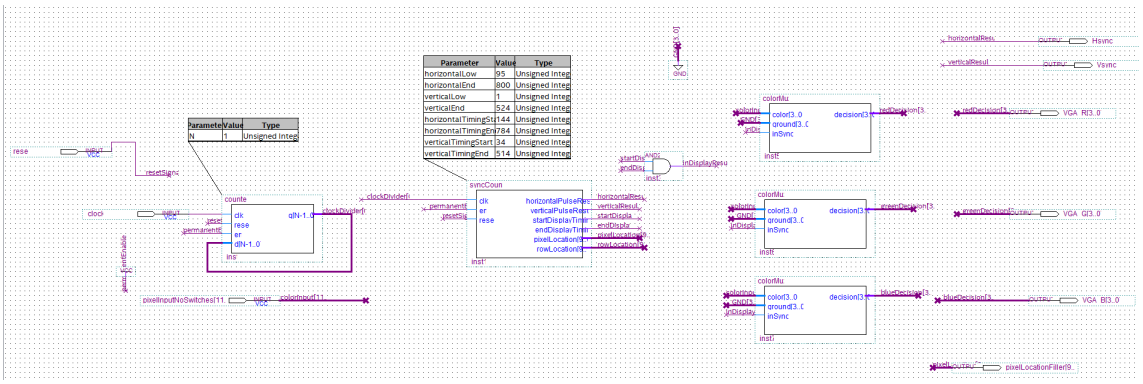

Figure 22: VGA Block Diagram

3.3.2 Counter

Inputs: Clock, en, reset, d

Outputs: q

Description: A basic counter - we're using it as a clock divider here in order to get to 25 mHz.

3.3.3 Sync Count

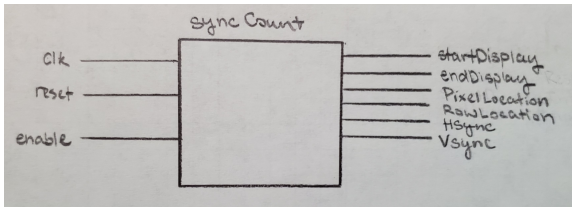

Figure 25: Sync Count Block Diagram

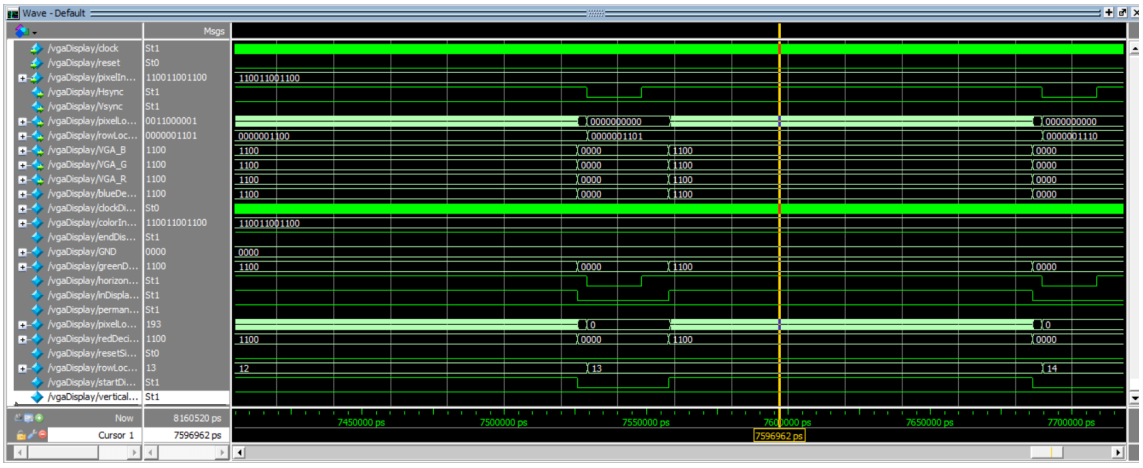

Figure 23: VGA simulation

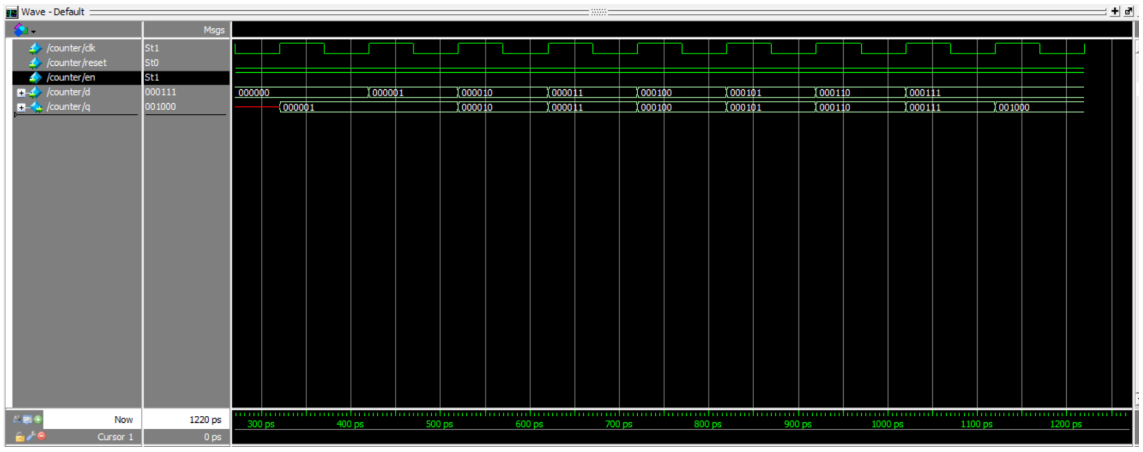

Figure 24: VGACounter simulation

Inputs: Clock, en, reset

Outputs: horizontalPulseResult (hsync), verticalPulseResult(vsync), startDisplayTiming, endDisplayTiming, pixelLocation, rowLocation

Description: The idea here is to increment our current pixel until we hit the max length of our row - then we reset our current pixel and increment our current row by one. If our horizontal value is within our hsync bound, it's then true, and the same is true with our vertical value. In addition, our display timings tell us when we are actually "in the screen" and able to write pixels to the screen.

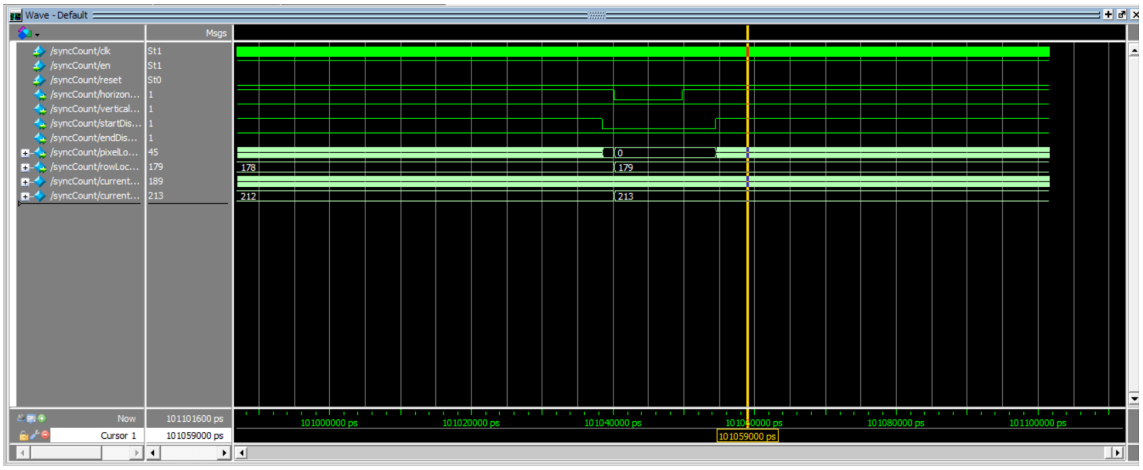

Figure 26: Sync Count simulation

3.3.4 Color Mux

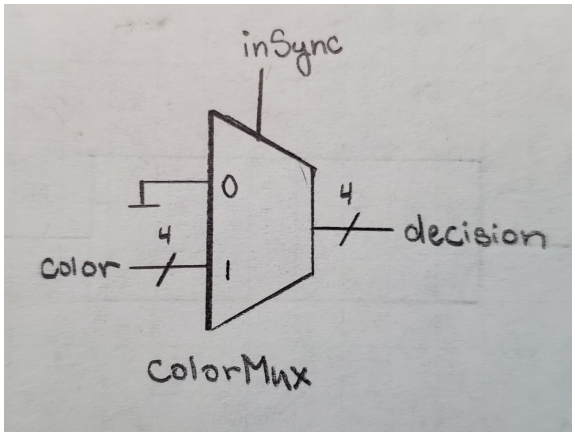

Figure 27: Color Multiplexer Block Diagram

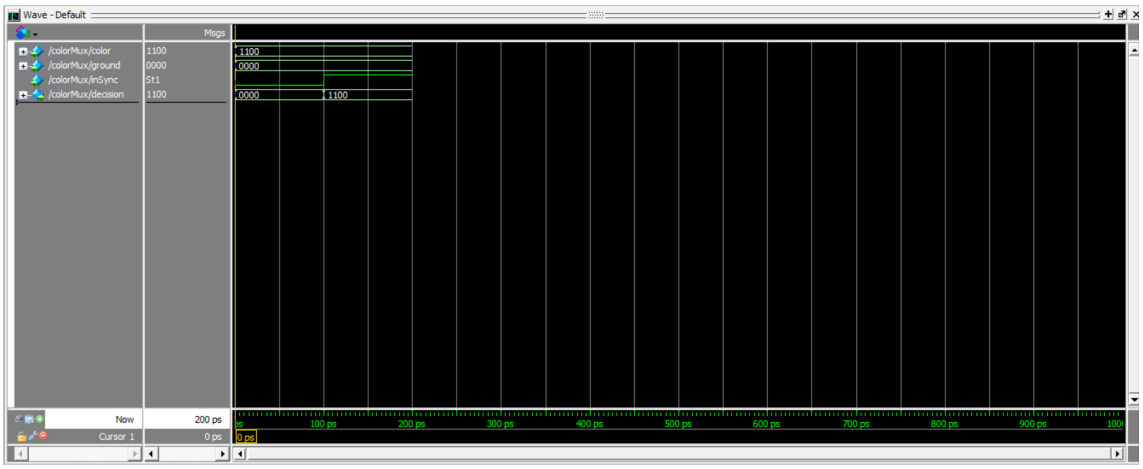

Figure 28: VGAColorMux simulation

Inputs: Color, Ground, inSync

Outputs: Decision

Description: This is a basic mux - the only notable thing about it is that our inSync value is provided to us from Sync Count and an and gate. If both Vsync and Hsync are high, then our color is let through. Otherwise, we get routed to ground.

3.4 Sprite VGA

3.4.1 Top Level Diagram

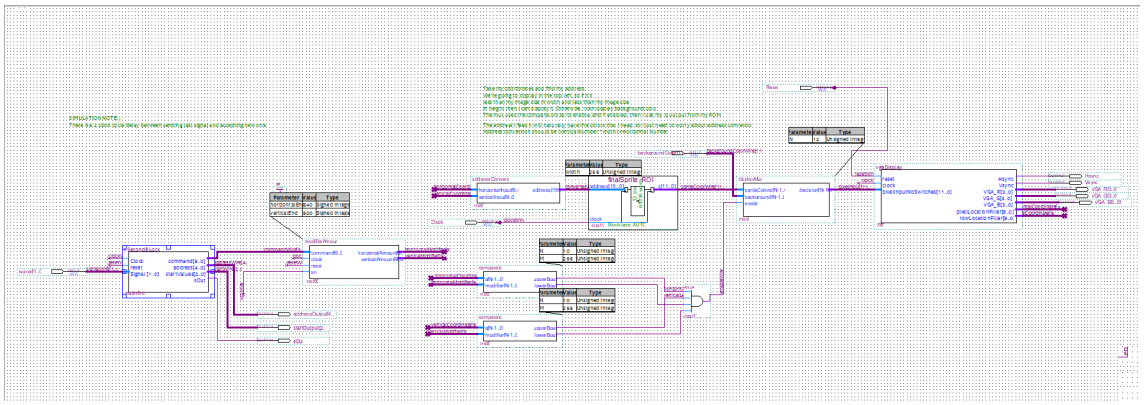

Figure 29: Sprite with movement and VGA

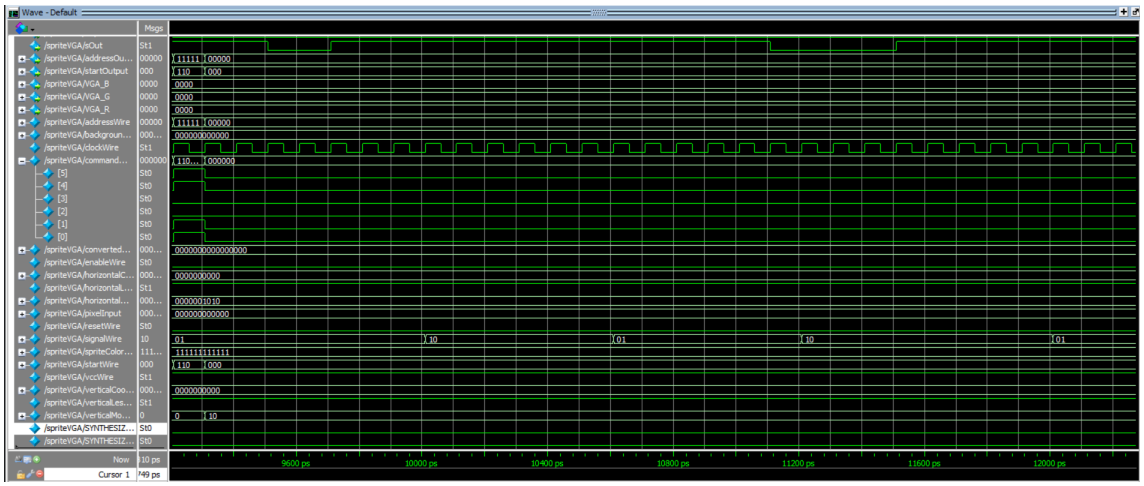

Figure 30: Simulation of a correct input and the values we get from it

Inputs: Signal, Clock, Reset, backgroundColor

Outputs: Hsync, Vsync, VGA RED, VGA BLUE, VGA GREEN, Address, Start

Description: This module takes in two bits representing the value that we got from IR receiver. It then passes it through our IR Decoder in order to find what our command, address, and start values are. Currently start values and address are disregarded, but they're saved and we could do verification with them if wanted. Once the command is recieved, it goes through the modifierAmount block which determines how much to move our image bounds by. We then pass the modified amount to the comparators that determine whether the sprite is being drawn or not. By shifting the bounds of where the comparators say to draw, we can give the sprite the appearance of movement. When enabled, the displayMux allows the spriteColors stored in our ROM through - otherwise it passes the background colors, those colors are then passed to the VGA for it to determine whether we're in the display bounds or not.

3.4.2 Address Modifier

Figure 31: Address Modifier Simulation

Inputs: Command, Clock, Reset, en(able)

Outputs: horizontalAmount, verticalAmount

Description: Address Modifier takes in a 6 bit command received from the decoder and matches it against specific values for up, down, left, and right. The module restricts movement outside of the bounds of our display by checking whether the new location is in bounds before assigning it - if we get a nonsense command or something that moves us out of bounds, we simply ignore it. The output increments and decrements in tens, and won't go above 640x480 or below 0x0 (the start value).

3.4.3 Sprite Comparator

Inputs: A(our coordinate), Modifier

Outputs: UpperBound, LowerBound

Description: The Sprite Comparator is simple - it just keeps track of both lower and upper bounds so that we can increment or decrement either. The modifier has already been checked by the previous module to make sure we can't move out of bounds, so we're able to just assign our outputs and not worry about error checking in this module.

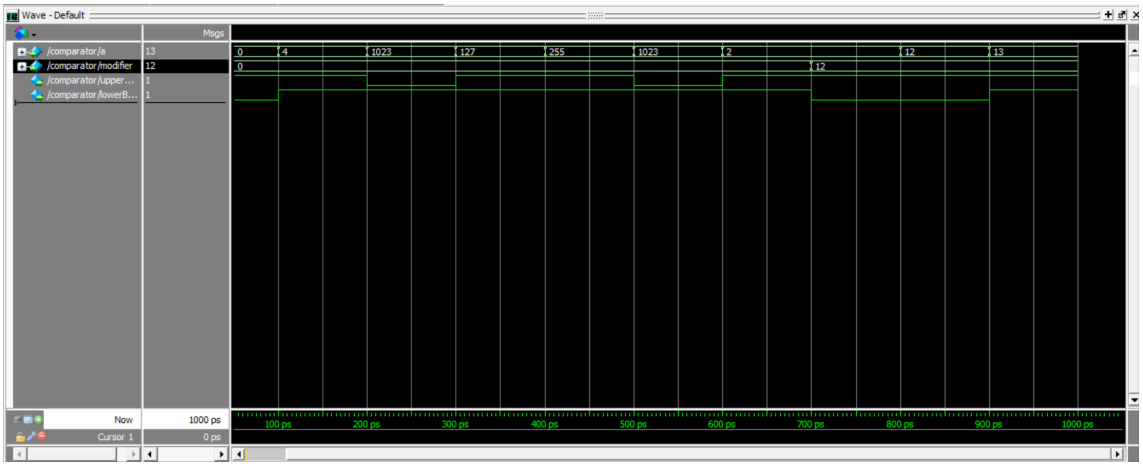

Figure 32: Comparator Simulation

3.4.4 Address Converter

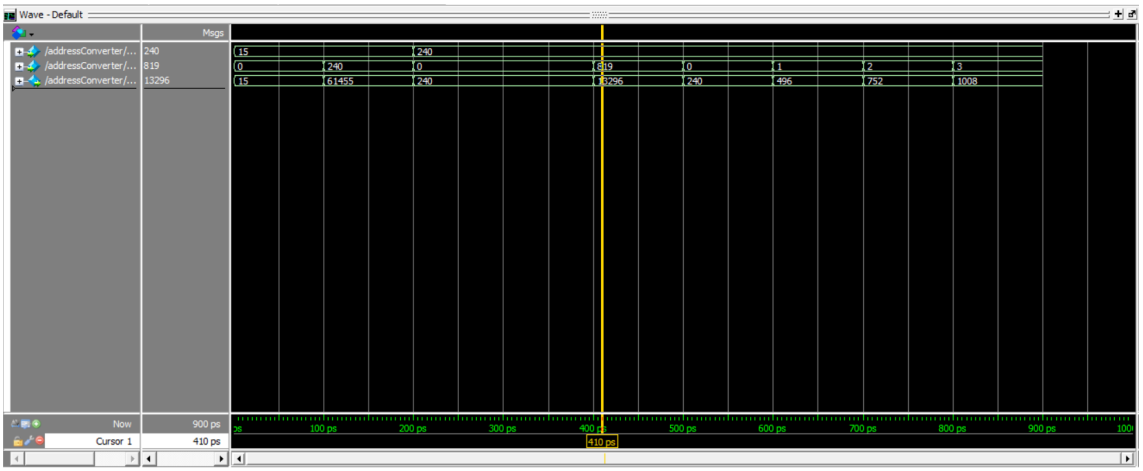

Figure 33: Address Converter Simulation

Inputs: horizontalInput, verticalInput

Outputs: address

Description: This is a simple module - we take the vertical input, multiply it by the width of the row, and add our horizontal input. This process gives us the address of any given x and y combination.

3.4.5 Sprite ROM

Inputs: Address, Clock

Outputs: spriteColors

Description: This used Quartus's built in IP stuff to make a ROM our size. The basic idea is that we now take in the address and read whatever is at that line out into our spriteColors.

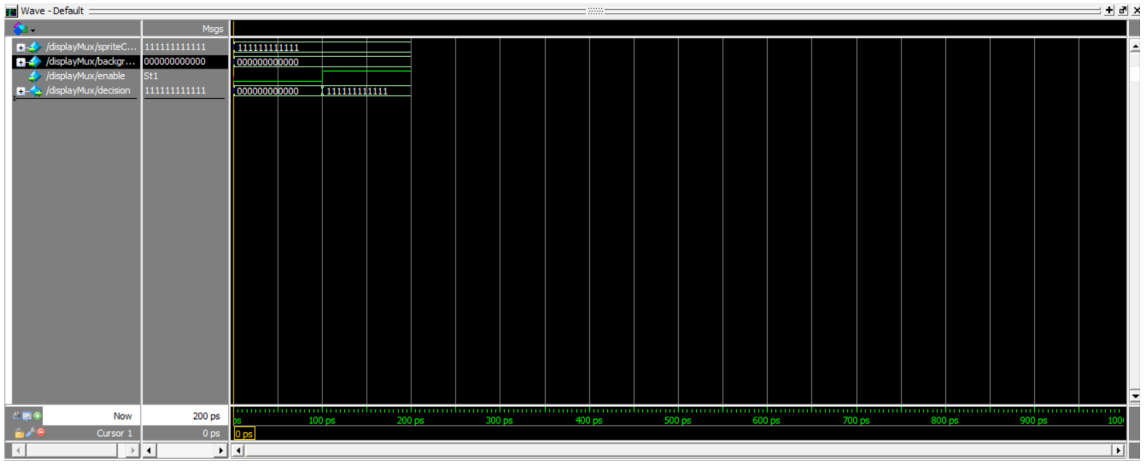

Figure 34: Color Mux Simulation

3.4.6 Display Mux

Inputs: spriteColors, background, enable

Outputs: Decision

Description: This module takes in both the color options that could be displayed and, depending on the enable, only allows one to break through. The enable is based on a 4-wide and gate with all of the bounds from our comparators. If we're inside of both our horizontal and vertical bounds, then the spriteColors pass through - otherwise the background does.

3.4.7 VGA Display

Inputs: reset, clock, pixelInputNoSwitches

Outputs: Hsync, Vsync, VGA R, VGA G, VGA B, horizontalCoordinate, verticalCoordinate

Description: Functions as described earlier - This takes in 25 mHz clock pulses and counts pixels. The Hsync and Vsync tell the television whether it's within the necessary bounds or not, and the VGA series tell the television what colors to write. The coordinates are also how far into the actual display we are - not just from the start of the clock cycles. It only starts incrementing when display is valid, and that's how we can start displaying our sprite at (0,0) to begin with. For a simulation, see Figure 14.

3.4.8 VGA Display Output Simulation

A SystemVerilog Files

A.1 IR Receiver

A.1.1 Comparator

```
//IR Receiver Comparator
module comparator2 #(parameter N = 10, M = 784, O = 143)
                                                    (input logic [N-1:0] a,
                                                     output logic altb, agtc);

                reg [N-1:0] b = M;
                reg [N-1:0] c = O;
                assign altb = (a < b);
                assign agtc = (a > c);
endmodule
```

A.1.2 Clock Counter

```
//Counter
module counter1 #(parameter N = 10) //25 because 2^10 = 10
                                        (input logic clk,
                                         input logic reset,
                                         input logic [N-1:0] d, //stores data
                                         output logic[N-1:0]q);

                always_ff@(posedge clk, posedge reset) //on the rising edge of the clock and reset
                        if (reset) q <= 0; //reset to zero
                        else        q <= d + 1; //iterate up by one
endmodule
```

A.1.3 Enable Counter

```
//IR Receiver Counter
module NCounter_enable #(parameter N = 8)
                                                                    (input logic clk, input logic reset,
                                                                    input logic en,
                                                                    output logic [N-1:0] q);

                        always_ff @(posedge clk, posedge reset)
                                if(reset)
                                        q <= 0;
                                else if(en)
                                        q <= (q + 1);
                                else
                                        q <= 4'b0;
endmodule
```

A.1.4 Multiplexer

```
//IR Receiver Mux
module ir_mux3 (input logic a, b,
                                        input logic clk, res,
                                        output logic actual_value, register_clk,
                                        output logic [3:0] valid_count);

                        always_ff @(posedge clk or posedge res)
                                if(res)
                                        valid_count <= 0;
                                else if(a)
                                        begin
                                                actual_value = 1;
                                                valid_count <= valid_count + 1;
                                                register_clk = 1;
                                        end
                                else if(b)
                                        begin
                                                actual_value = 0;
                                                valid_count <= valid_count + 1;
                                                register_clk = 1;
                                        end
                                else register_clk = 0;
endmodule
```

A.1.5 Shift Register

```
//IR Receiver Shift Register
module Nshift_register #(parameter N = 8)
                                                                    (input logic clk,
                                                                     input logic reset, sin,
                                                                     output logic [N-1:0] q,
                                                                     output logic sout);

                        always_ff @(posedge clk, posedge reset)
                                if(reset)
                                        q <= 0;
                                else
                                        q <= {q[N-2:0], sin};
                        assign sout = q[N-1];
endmodule
```

A.1.6 Tristate

```
//IR Receiver Tristate
module irv_tristate #(parameter N = 8)
                                        (input logic [N-1:0] a,
                                         input logic enable,
                                              output logic [N-1:0] y);

                reg b = (N-1);

                always_comb
```

```
                        if(enable)
                                y = a;
                        else
                                y = 4'b0;
endmodule
```

A.1.7 Synchronizer

```
//IR Receiver Synchronizer
module sync  (input logic clk,
                                input logic d,
                                output logic q);

        logic n1;

        always_ff@(posedge clk) //flip flop at rise of clock
                begin
                        n1 <= d;
                        q <= n1;
                end
endmodule
```

A.2 IR Decoder

A.2.1 Comparator

```
//IR Decoder Comparator
module irComparator #(parameter N = 4, M = 14)
                                                        (input logic [N-1:0] count,
                                                         output logic countEqualToM);

                assign countEqualToM = (count == M);
endmodule
```

A.2.2 First Multiplexer

```
//IR Decoder Mux
module irMux(input logic [1:0] currentBit,
                                output logic [1:0] value);

                //input 2 bit value. All values ignored except 01 high and 10 low
                always_comb
                        case(currentBit)
                                1:                      value = 11;
                                2:                      value = 10;
                                default value = 00;
                        endcase
endmodule
```

A.2.3 Second Multiplexer

```
//IR Decoder Second Mux
module irSecondMux(input logic [1:0] currentValue,
                                                input logic clk,
                                                output logic actualValue,
                                                output logic [4:0] validCount);

                always_ff @(posedge clk)
                        case(currentValue)
                                3:
                                begin
                                        actualValue = 1;
                                        validCount <= validCount + 1;
                                end

                                2:
                                begin
                                        actualValue = 1;
                                        validCount <= validCount + 1;
                                end

                                0: validCount <= 0;
                        endcase
endmodule
```

A.2.4 Shift Register

```
//IR Decoder Shift Register
module shiftRegister #(parameter N = 14)
                                                        (input logic clk,
                                                         input logic reset,
                                                         input logic sin,
                                                         output logic [N-1:0] q,
                                                         output logic sout);

                        always_ff @(posedge clk, posedge reset)
                                if(reset) //if reset = 1
                                        q <= 0;
                                else //if reset = 0
                                        q <= {q[N-2:0], sin};
                        assign sout = q[N-1];

endmodule
```

A.2.5 Address Swapper

```
module addressSwapper(input [13:0] unadulteratedInput,
                        output [5:0] command,
                                                        output [4:0] address,
                                                        output [2:0] startValues);
   assign command[5] = unadulteratedInput[5];
   assign command[4] = unadulteratedInput[4];
```

```
    assign command[3] = unadulteratedInput[3];
    assign command[2] = unadulteratedInput[2];
    assign command[1] = unadulteratedInput[1];
    assign command[0] = unadulteratedInput[0];

    assign address[4] = unadulteratedInput[10];
    assign address[3] = unadulteratedInput[9];
    assign address[2] = unadulteratedInput[8];
    assign address[1] = unadulteratedInput[7];
    assign address[0] = unadulteratedInput[6];

    assign startValues[0] = unadulteratedInput[11];
    assign startValues[1] = unadulteratedInput[12];
    assign startValues[2] = unadulteratedInput[13];


endmodule
//input 110 11111 100011
//I get my values one at a time ->
//1
//->1
//11
//->0
//011
//->1
//1011
//backwards
//so, input[0] = shiftRegister[13]
```

A.2.6 Tristate

```
//IR Decoder Tristate
module tristate #(parameter N = 8)
                                        (input logic [N-1:0] a,
                                         input logic enable,
                                         output logic [N-1:0] y);

                //if enabled, input goes through to output. Otherwise, 0
                assign y = enable ? a : 0;
endmodule
```

A.3 VGA Display Output

A.3.1 Counter

```
module counter #(parameter N = 6)
  (input logic clk,
   input logic reset,
        input logic en,
        input logic [N-1:0] d,
        output logic [N-1:0] q);
          always_ff @(posedge clk, posedge reset)
            if (reset) q <= 0;
            else if(en) q <= d + 1;
endmodule
```

A.3.2 Sync Count

```
module syncCount #(parameter horizontalLow = 95, horizontalEnd = 799, verticalLow = 1, verticalEnd =
     524,
                            horizontalTimingStart = 143, horizontalTimingEnd = 784,
                                                                    verticalTimingStart = 34,
                                                                         verticalTimingEnd = 514)
          (input logic clk,
           input logic en,
                           input logic reset,
                           output logic horizontalPulseResult,
                           output logic verticalPulseResult,
                           output logic startDisplayTiming,
                           output logic endDisplayTiming,
                           output logic [9:0] pixelLocation,
                           output logic [9:0] rowLocation);
  logic [9:0] currentPixel = 10'b0000000000;
  logic [9:0] currentRow = 10'b0000000000;

  always_ff @(posedge clk)
    begin
      if(reset)
                begin
                  pixelLocation <=10'b0000000000;
                  rowLocation <=10'b0000000000;
                  currentPixel <=10'b0000000000;
                  currentRow <=10'b0000000000;
                end

            else if(en)
              begin
                      //if I'm below my max, add one
                      if(currentPixel < horizontalEnd)
                        begin
                      currentPixel <= currentPixel + 1;
                        end
                      //if my horizontal pixel is at the max
                      else if(currentPixel >= horizontalEnd)
                        begin
                                    //and if my vertical row is at the max, reset my row
                           if(currentRow >= verticalEnd)
                             begin
                                        currentRow <= 10'b0000000000;
                                      end
                           //otherwise, add one to your row.
                               else if(currentRow < verticalEnd)
                                 begin
                             currentRow <= currentRow + 1;
                                      end
                           //and then reset your horizontal pixel no matter what.
                          currentPixel <= 10'b0000000000;
                        end
```

```
                    pixelLocation <= currentPixel;
                    rowLocation <= currentRow;
                    end

          end
    assign horizontalPulseResult = !(currentPixel <= horizontalLow);
    assign verticalPulseResult = !(currentRow <= verticalLow);
    assign startDisplayTiming = ((currentPixel >= horizontalTimingStart) && (currentPixel <
        horizontalTimingEnd));
    assign endDisplayTiming = (currentRow >= verticalTimingStart) && (currentRow < verticalTimingEnd);
endmodule
```

A.3.3 Color Mux

```
module colorMux(input logic [3:0] color,
                input logic [3:0] ground,
                                  input logic inSync,
                                  output logic [3:0] decision);
  assign decision = inSync ? color : ground;
endmodule
```

A.4 Sprite VGA

A.4.1 Address Modifier

```
//Take in my command and determine what I'm adding to the comparators.
//((width * row) + height) = address
//(address - height)/width = row
//height = address - width * row;
module modifierAmount #(parameter horizontalEnd = 640, verticalEnd = 500)
                       (input logic [5:0] command,
                        input logic clock,
                                                        input logic reset,
                                                        input logic en,
                                                        output logic [9:0] horizontalAmount,
                                                        output logic [9:0] verticalAmount);

always_ff @(posedge clock, posedge reset)
    if(reset)
    begin
            verticalAmount <= 6'b000000;
            horizontalAmount <= 6'b000000;
    end
         else if(en)
          case(command)
                  //Down
                  6'b110011:
                  if(verticalAmount < (verticalEnd - 'd10))
                    verticalAmount <= (verticalAmount + 'd10);
                  //Up
                  6'b001100 :
                    if(verticalAmount >= 10)
                            verticalAmount <= (verticalAmount - 'd10);
                  //Left
                  6'b111000 :
                    if(horizontalAmount >= 10)
                            horizontalAmount <= (horizontalAmount - 'd10);
                  //Right
                  6'b000111 :
                    if(horizontalAmount < (horizontalEnd- 'd10))
                            horizontalAmount <= (horizontalAmount + 'd10);
      endcase
endmodule
```

A.4.2 Sprite Comparator

```
module comparator #(parameter N = 10, M = 256)
                   (input logic [N-1:0] a,
                                                   input logic [N-1:0] modifier,
                                                   output logic upperBound,
                                                   output logic lowerBound);
  assign lowerBound = (a > (0 + modifier));
  assign upperBound = ((a + modifier) < M);
endmodule
```

A.4.3 Address Converter

```
module addressConverter #(parameter width = 256)
                         (input logic [9:0] horizontalInput,
                                                                 input logic [9:0] verticalInput,
                                                                 output logic [15:0] address);
assign address = ((verticalInput * width) + horizontalInput);
endmodule
```

A.4.4 Display Mux

```
module displayMux #(parameter N = 12)
                   (input logic [N-1:0] spriteColors,
                                                   input logic [N-1:0] background,
                                                   input logic enable,
                                                   output logic [N-1:0] decision);

assign decision = ( enable ? spriteColors : background);
endmodule
```

B Do files

B.0.1 RC-5 Encoder

```
vsim -gui rc5_irdecoder
add wave *
force -freeze sim:/rc5_irdecoder/Clock10_a 1 0, 0 {50000 ps} -r 100000 ps
force Reset_a 0 0, 1 10
run 20 ms
force input_ir 0 0, 1 .889 ms
run 1.778 ms
force input_ir 1 0, 0 .889 ms
run 1.778 ms
force input_ir 0 0, 1 .889 ms
run 1.778 ms
force input_ir 1 0, 0 .889 ms
run 12 ms
```

B.0.2 IR Decoder

```
add wave *
force -freeze sim:/spriteVGA/Reset 1 0
run 10
force -freeze sim:/spriteVGA/Reset 0 0
force -freeze sim:/spriteVGA/backgroundColor 000000000000 0
force -freeze sim:/spriteVGA/spriteColorWire 111111111111 0
force -freeze sim:/spriteVGA/Clock 1 0, 0 {50 ps} -r 100
force -freeze sim:/spriteVGA/signal 01 0
run 400
force -freeze sim:/spriteVGA/signal 10 0
run 600
force -freeze sim:/spriteVGA/signal 01 0
run 600
force -freeze sim:/spriteVGA/signal 10 0
run 600
force -freeze sim:/spriteVGA/signal 01 0
run 600
```

References

[1] S. Bergmans, “Phillips RC-5 Protocol,” published 1, Aug. 2020. [Online]. Available: https://www.sbprojects.net/knowledge/ir/rc5.php

[2] Altium, “RC-5 Transmission Protocol,” published 13 Sept. 2017. [Online]. Available: https://techdocs.altium.com/display/FPGA/Philips+RC5+Infrared+Transmission+Protocol

[3] Military Analysis Network, “Basic Radar Systems,” accessed 3, June 2021. [Online]. Available: https://fas.org/man/dod-101/navy/docs/es310/radarsys/radarsys.htm