Design Project

Alex Young
Shengmei Hu
Bryson Goto

Spring 2020

Table of Contents

1 Introduction

The purpose of this project is to implement a logic design that uses an NES controller input to control three different outputs: a seven segment display, RGB lights, and DC Motor. The inputs used in this project are not normally used in conjunction with the outputs chosen, but through application of digital logic design concepts, we are able to convert inputs made on the NES controller function with the seven segment display, RGB LEDs, and DC motor.

Figure 1: NES Controller [1]

1.1 Project Goals

- Support communication between NES controller and RGB LEDs, 7 segment display, and DC motor
- Use an adder controlled with the arrow keys and display the values onto the 7 segment display
- Run the motor back and forth through the select and start button
- Change the color of the RGB light through the A and B button

2 High Level Description

2.1 Top Level Hardware Diagram

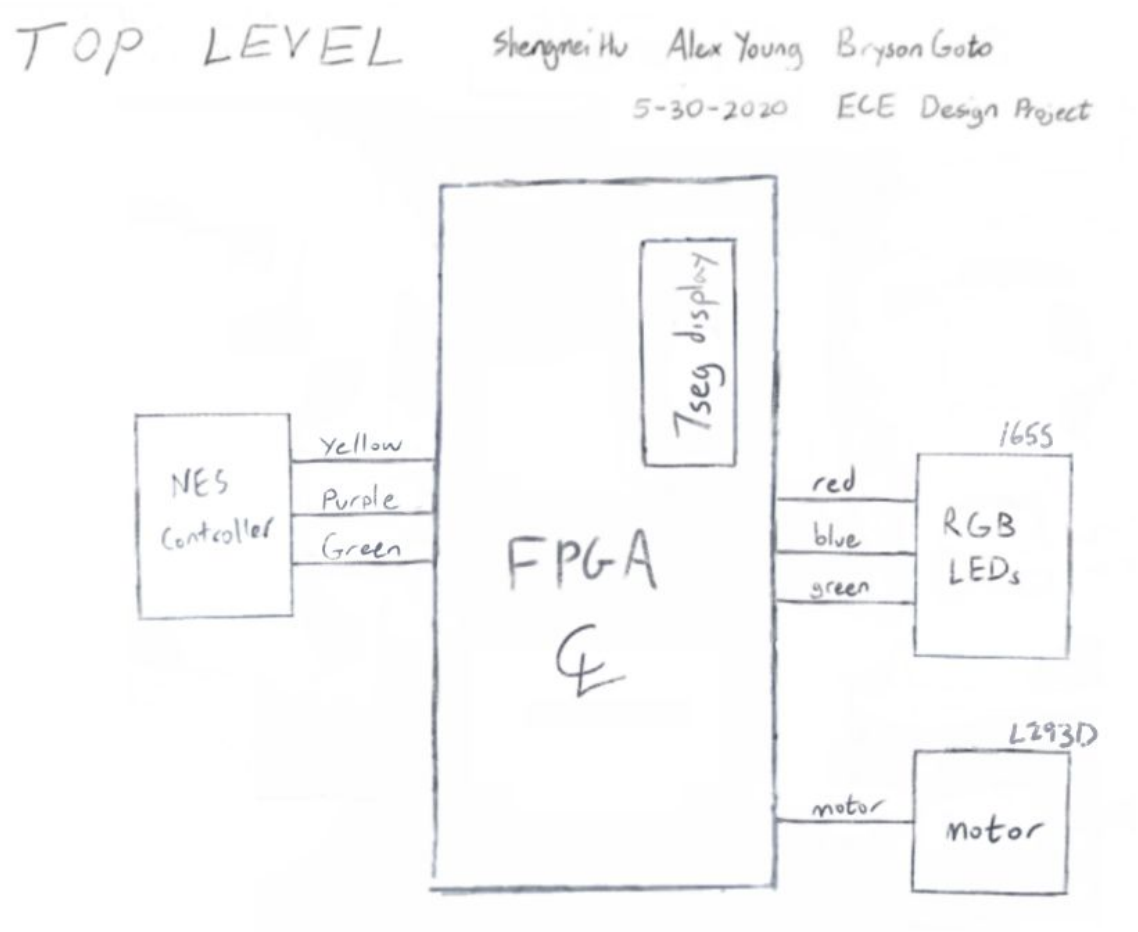

Figure 2: Top Level Hardware Diagram

Inputs: Signals from the NES controller which has 8 different button inputs, as well as a clock and reset signal from the FPGA.

Outputs: Converted signals from NES controller inputs that lead to the RGB LED, DC motor, and 7 segment display (7 bits). The 7 segment display is already integrated on the FPGA.

Description: The NES controller would be controlling 3 different outputs: the DC motor, RGB LEDs, and 7 segment display. Using combinational logic on the FPGA, the input buttons would be translated into signals that the various outputs can use. The inputs and outputs would be connected through pins on the FPGA had we implemented everything with actual hardware.

2.2 Top Level HDL Schematic

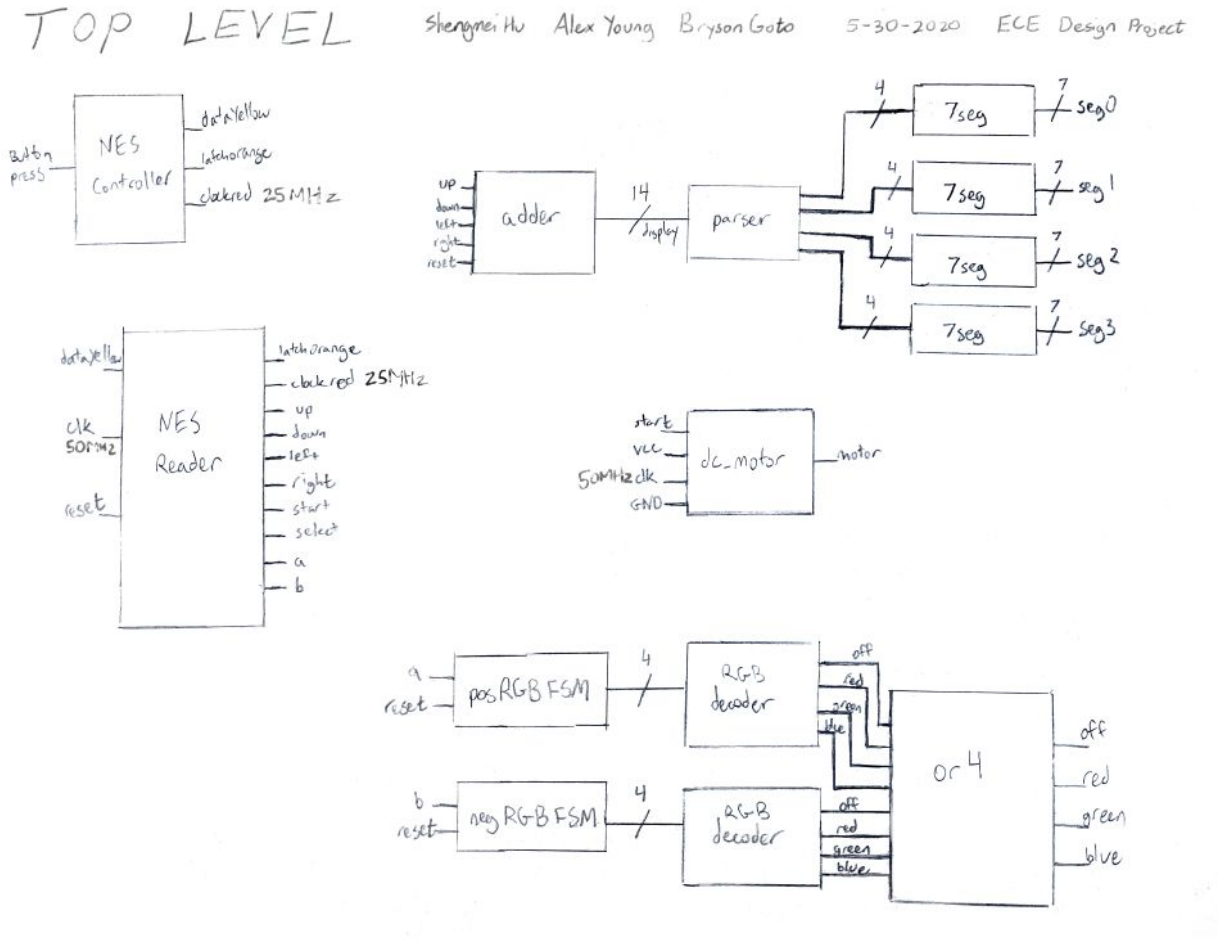

Figure 3: Top Level HDL Schematic

Inputs: Signals from the NES controller, clock and reset signals from the FPGA.

Outputs: Converted signals from NES controller inputs that lead to the RGB LED, DC motor, and 7 segment display (7 bits)

Description: Shown in the diagram above, the NES controller goes into the NES reader that processes inputs from the controller. These are then output as directional, a, b, start, and select buttons. Each button group controls a different output type. The directional buttons on the NES controller are connected to an adder / subtractor module. Each button will add and subtract a set value and the result will be displayed on the 7 segment display, up to four digits. The values set to each direction are:

Up → +1
Down → -1
Left → +10
Right → -10

The A and B buttons on the NES controller control the RGB light. These functions act similar to a finite state machine where the flow of the inputs would cycle around, changing the output accordingly. In our case, pressing the A button cycles the light through: red, green, blue, and then off. Pressing the b button cycles it in reverse, the result being: blue, green, red, off.

The final output is the DC motor which is driven by the start button on the NES controller. Pressing the start button will drive the motor and it will continue to drive until the button is no longer pressed.

3 Controller Descriptions

3.1 NES Controller Description

The NES controller is an active low controller which has 8 different buttons that lead into the main shift register. Each button is connected to a pull up resistor leading to a 5V source. The other inputs for this controller are the clock and latch pins which all lead into a single data pin (labelled NES Data below in Figure X). The 8 bit input is ordered from top to bottom in Figure X below with the right button as the most significant bit and the a button as the least significant bit.

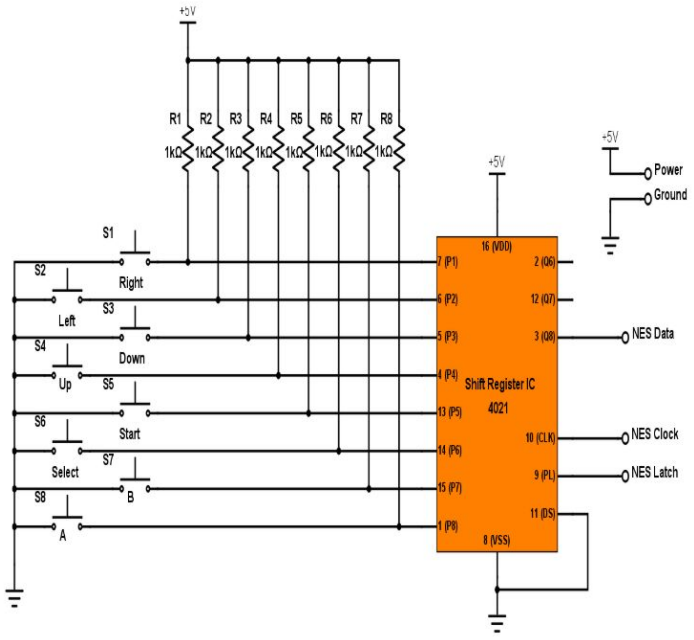

Figure 4: NES Controller Diagram [2]

4 HDL Components

4.1 Top Module Components

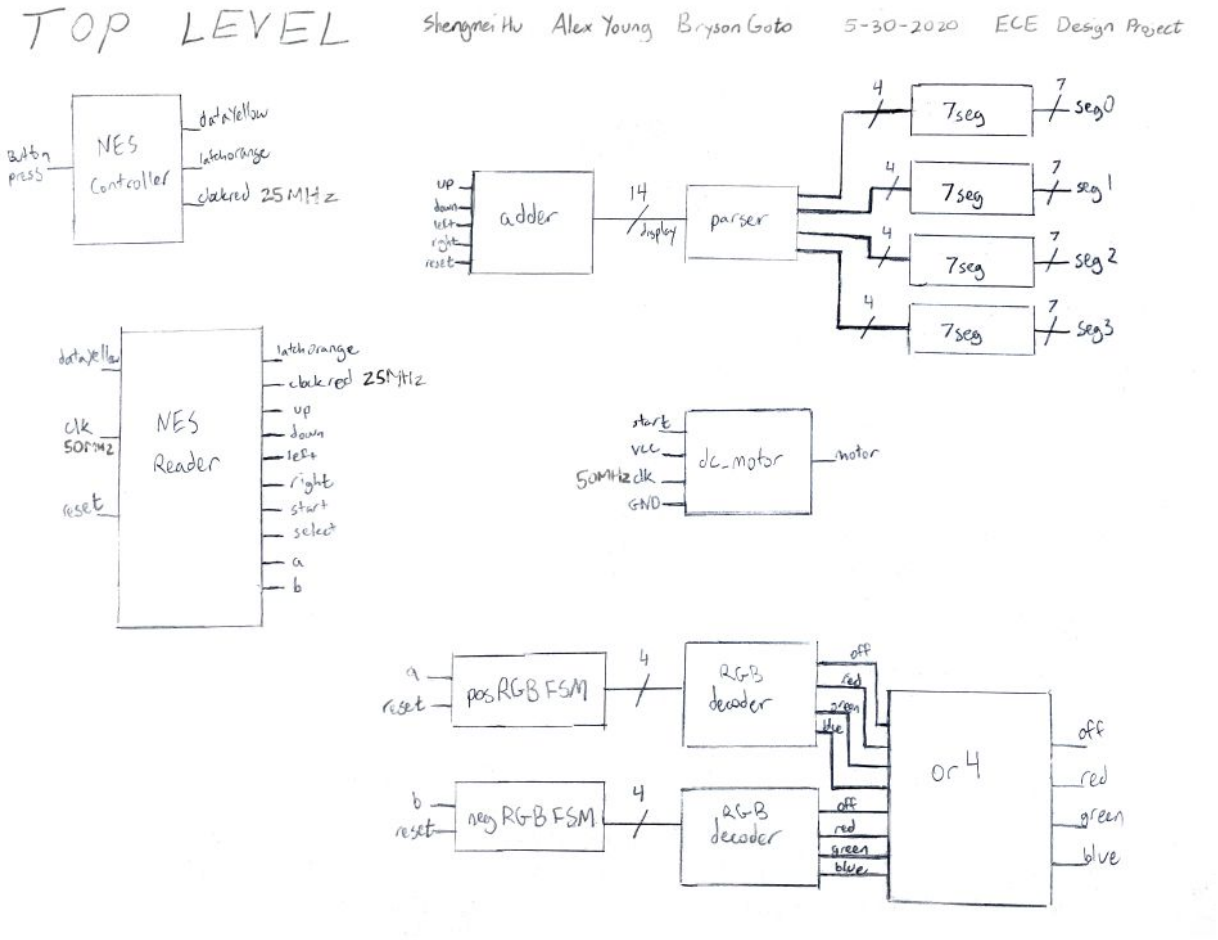

Figure 5: Top Module for HDL schematic

Inputs: The inputs of this top design are the button signals from the NES controller as well as a clock and reset signal from the FPGA.

Outputs: The output signals vary depending on the module it's sent to. The 7 segment display is sent a 7-bit signal, the DC motor a 1 bit signal, and the RGB 3 1-bit signals.

Description: The top level schematic shows all of the modules and how each component is connected. The NES controller reads in input from the user and by sending those signals to the NES reader, the rest of the modules are able to take those changed signals and translate them to outputs.

4.2 NES Reader

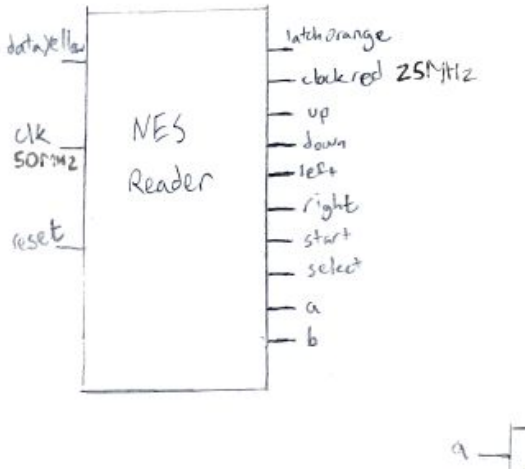

Figure 6: NES Reader Module

Inputs: The NES reader takes in the input from the NES controller as well as the clock and reset signals from the FPGA. The data yellow is an 8 bit signal sent from the NES controller.

Outputs: The output is the individual buttons that are on the NES controller as well as the clock and latch signals. The latch orange output signals to the modules that a new button was pressed.

Description: The NES reader is from the given files from previous years. The module takes in the 8 bit input from the NES controller and separates each bit into eight 1 bit outputs that correspond to the buttons on the controller. When the orange latch goes high, it's a signal for the other modules that a new input may be entered. The orange latch will go high every 16 counts (a full hexadecimal cycle).

4.3 Seven Segment Decoder

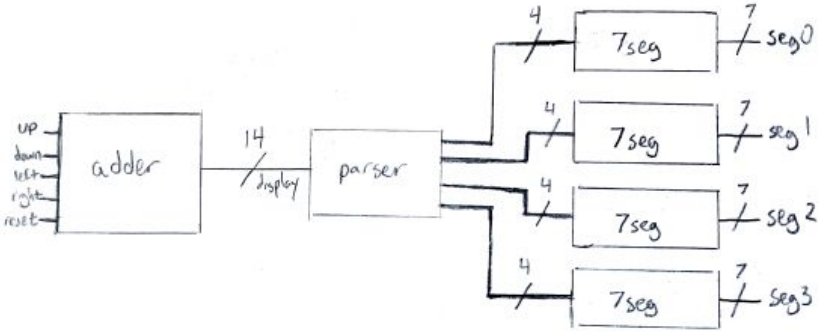

Figure 7: Seven Segment Decoder

Inputs: The inputs of the seven segment decoder correspond to the directional buttons of the NES controller.

Outputs: The output of the parser is four 7 bit binary signals that are output to addressable LEDs on the 7 segment display on the FPGA.

Description: The adder will add or subtract a value depending on the directional button from the NES controller. The parser will then separate the value into the thousands, hundreds, tens, and ones digits. These four values will then be sent to the 7 segment display driver to be converted to a code that corresponds to the value to be displayed on the 7 segment display.

4.3.1 Adder

Figure 8: Adder

Inputs: The adder takes in the arrow button signals from the NES controller as well as the reset signal from the FPGA. Each signal is 1 bit and the rising edge of each signal is used to change the output.

Output: The output is a 14-bit signal that leads to the parser module. This output is the value of the number that will be displayed on the 7 segment display.

Description: The adder will add or subtract a value depending on the button that is pushed on the NES controller. The module is looking for a rising edge signal, meaning that users are not able to hold down the button to increase the values displayed. The reason the output is 14 bits is because on a 4 digit segment display, 9,999 is the largest value that can be displayed. 14 bits is the lowest amount of bits that can be used to display every value possible for four 7 segments displays.

4.3.2 Parser

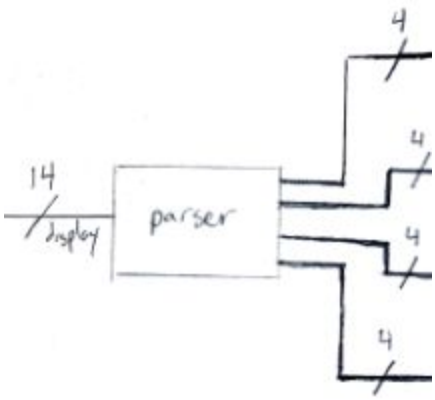

Figure 9: Parser Module

Inputs: The input of the parser is the 14 bit signal from the adder module. The signal indicates what the value that should be displayed is in binary.

Outputs: The output of the parser is four 4 bit binary signals that are sent to the 7 segment display.

Description: The input signal is a 14 bit signal that will be broken down into a 4 bit signal for each segment display, separating into the thousands, hundreds, tens, and ones place. The 4 bits are for the signals to determine which numeric value each digit is in the number.

4.3.3 Seven Segment Display

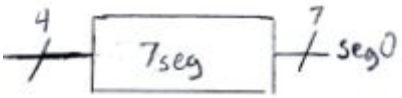

Figure 10: 7 Segment Display

Inputs: The input for the 7 segment display is a 4 bit input coming from the parser. The 4 bits indicates the numeric value the corresponding digit is in the displayed number.

Outputs: The output for the 7 segment display is the 7 bit output that leads into the LEDs for the display.

Description: As stated before, the 4 bit input signals to the segment display what digit should be displayed to the lights. The 7 outputs of the module describe to the lights which lights should be

on or off based on the digit that should be displayed. There are a total of four 7 segment display modules in the design.

4.4 DC Motor

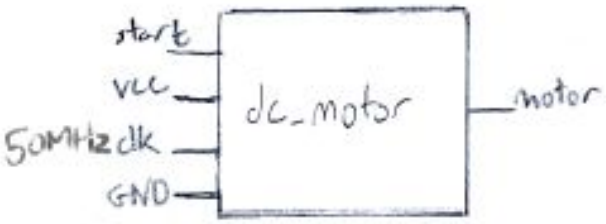

Figure 11: DC Motor

Inputs: The DC motor module takes in 4 inputs: the start button from the NES controller, voltage from a power source, a clock signal, and ground. The clock signal is given from the FPGA and the voltage is given from the module that powers the DC motor.

Outputs: The output is a 1 bit output that signals the motor to turn on or off. A high signal will indicate the motor is on and a low signal will show the motor is turned off.

Description: The DC motor requires power to move which is why the Vcc and ground signals are required for this module. Without power, the component can't operate. The start button is the input that signals the DC motor to turn on or off while the rising clock edge keeps the module updated on what the signal is. The output is the high or low signal that when sent to the motor, will turn on or off accordingly.

4.5 Addressable RGB LEDs

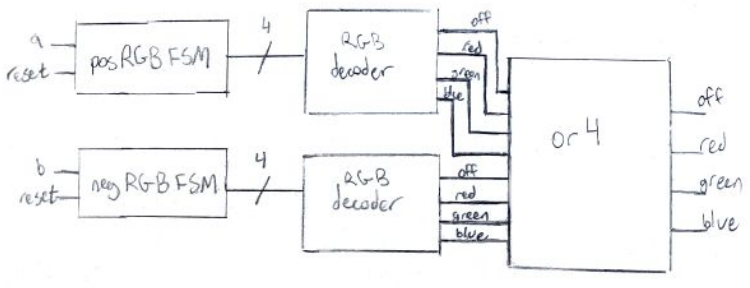

Figure 12: RGB Top Module

Inputs: The inputs of the RGB Top Module are the a and b buttons, and reset.

Outputs: The outputs of the RGB Top Module are the red, green, and blue LEDs, as well as the off state.

Description: The positive and negative RGB finite state machines will take inputs from a and b, which will cycle (in different directions) through the different LED colors by outputting a 4 bit signal that indicates what color will be turned on in the RGB decoder. The OR_4 Module will combine both the a and b inputs to determine whether each color will be on or not, and this will be shown in the final output, the LEDs.

4.5.1 Positive and Negative RGB Finite State Machines

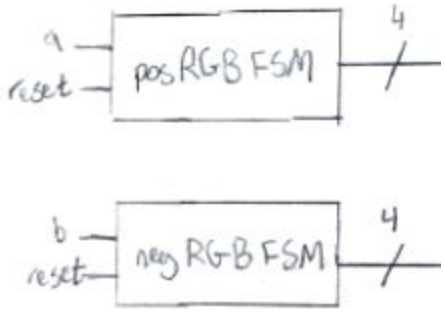

Figure 13: RGB Finite State Machines

Inputs: The inputs of the positive and negative RGB state machines are the a button and reset, and the b button and reset signals respectively. Each input is a 1 bit signal that is used to determine the light shown on the RGB.

Outputs: The output of the RGB finite state machine is a 4 bit signal that shows what state the RGB is in. The states are as followed: off, red, green, and blue.

Description: The finite state machines are used to determine the color that is shown on the RGB. The a button will spin the cycle clockwise with the states being off, red, green, and blue. The b button will spin the cycle counterclockwise with the states being blue, green, red, and off.

4.5.2 RGB Decoder

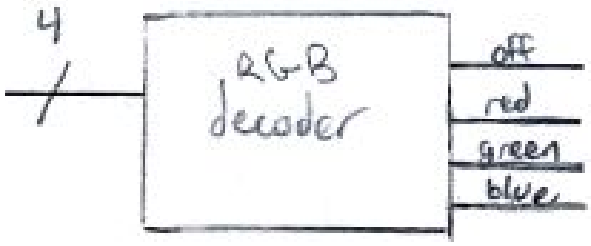

Figure 14: RGB Decoder

Inputs: The RGB decoder takes in a 4 bit signal from the finite state machines which signal what the state is.

Outputs: The output of the decoder is the different states that lead to an OR module.

Description: Both the positive and negative RGB finite state machine modules have a respective decoder. The decoder is used to determine what the state is for the a button and b button cycle individually. Later, the two signals are OR'd together in order to determine what the final color should look like.

4.5.3 OR_4 Module

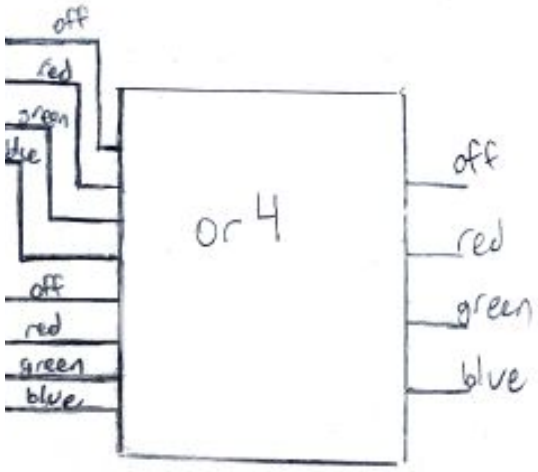

Figure 15: OR_4 Module

Inputs: The inputs to the OR_4 Module are the off, red, green, and blue states from both the positive and negative RGB decoders.

Outputs: Off, red, green, or blue signals as a 1 bit output to indicate whether the state is on or not, thus determining the final color.

Description: Each state type from both the positive and negative decoders will be fed through an OR gate to determine whether the state is true or not.

5 Appendix

5.1 Design Synthesis and Analysis

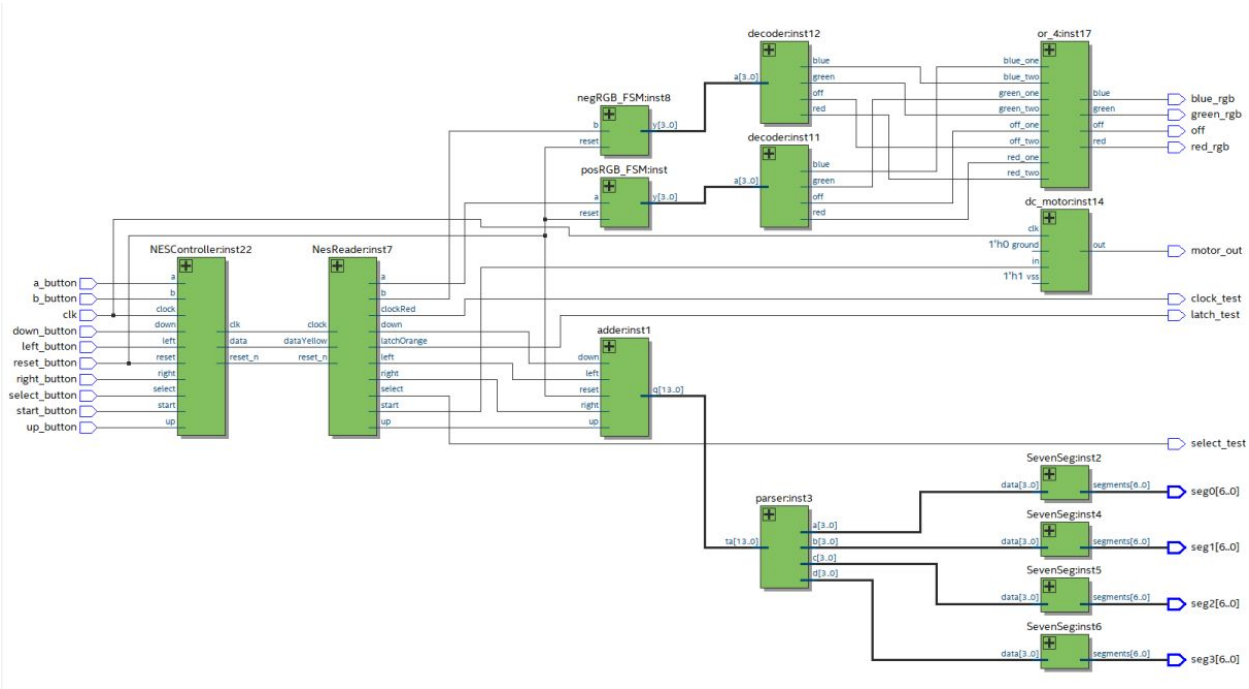

Figure 16: Quartus RTL Viewer of Top Module

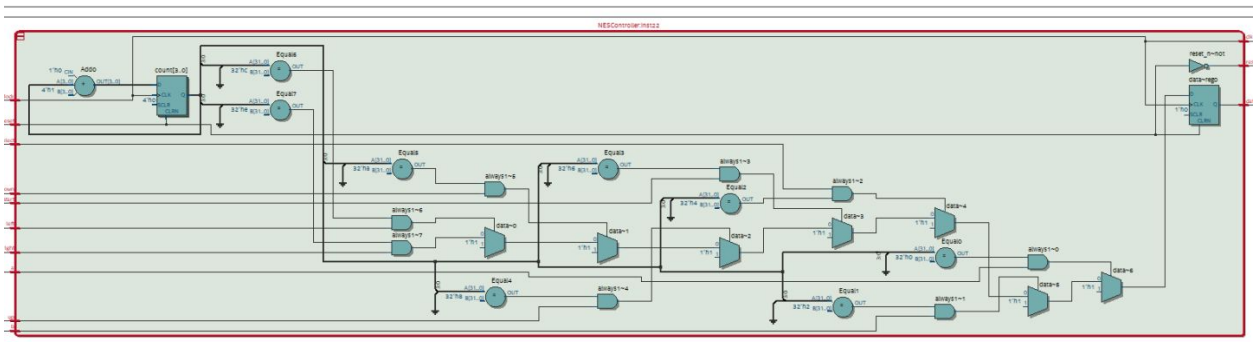

Figure 17: Quartus RTL Viewer of NES Controller

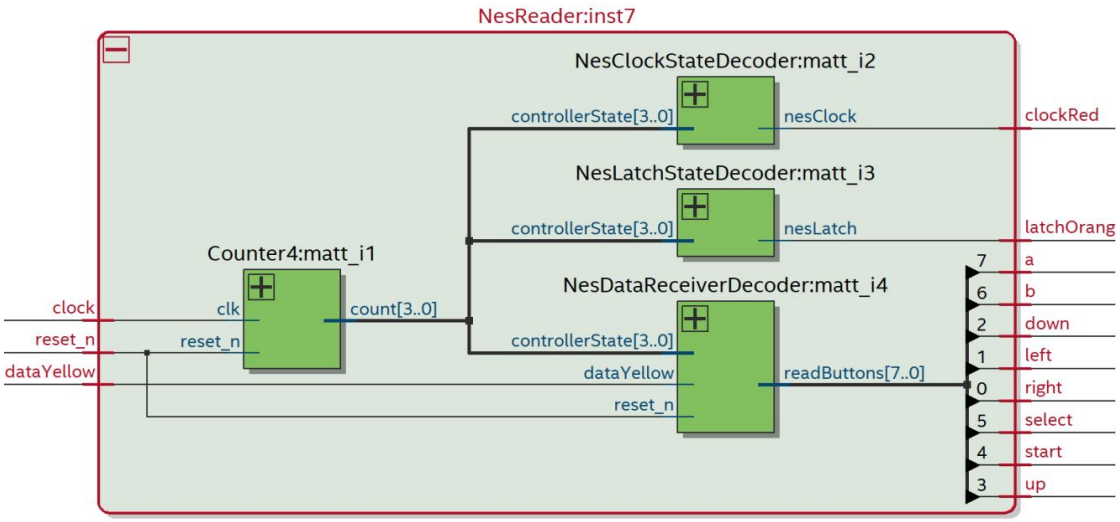

Figure 18: Quartus RTL Viewer of NES Reader

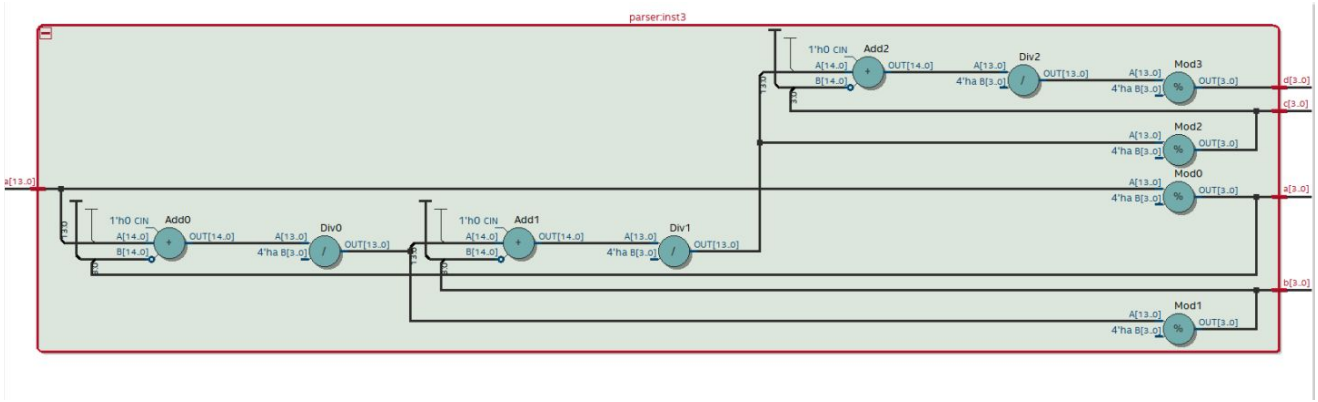

Figure 19: Quartus RTL Viewer of Parser

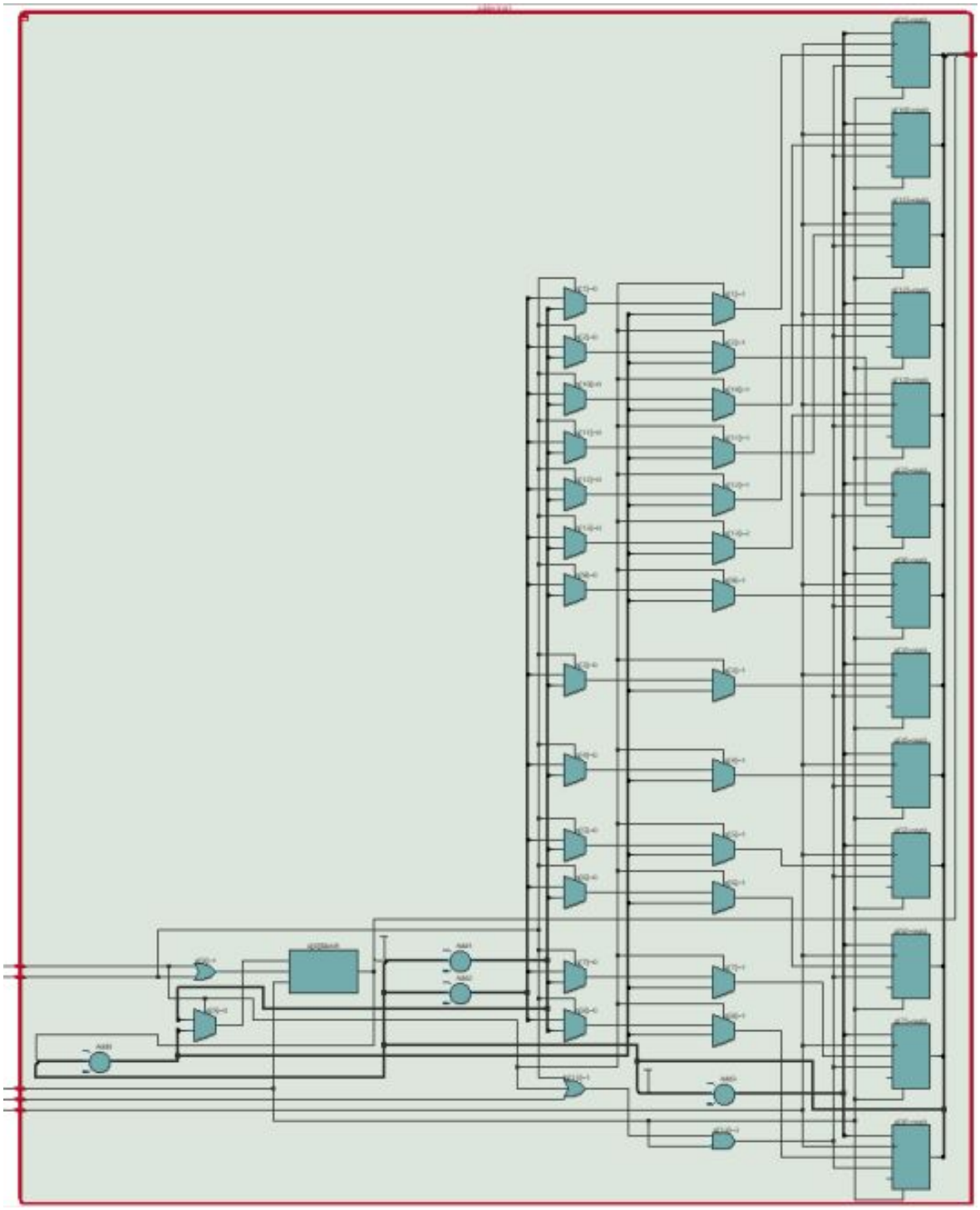

Figure 20: Quartus RTL Viewer of Adder

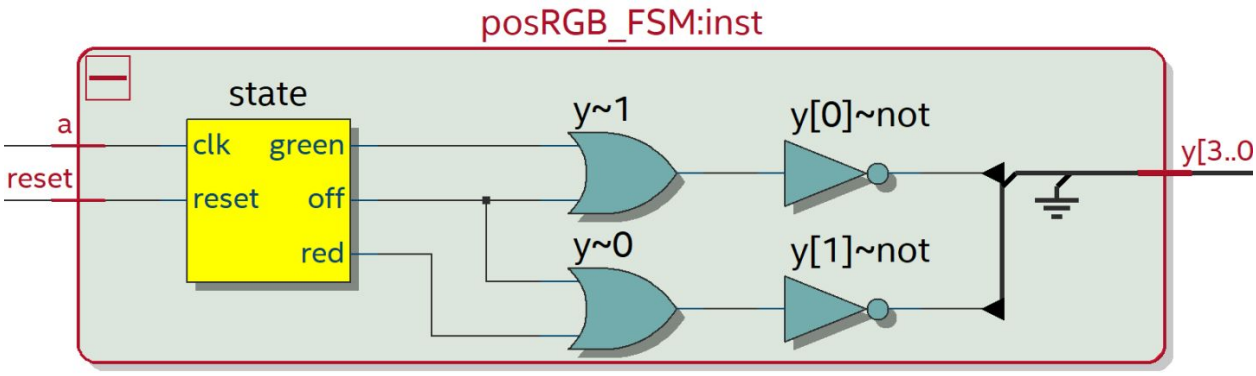

Figure 21: Quartus RTL Viewer of Positive RGB Finite State Machine

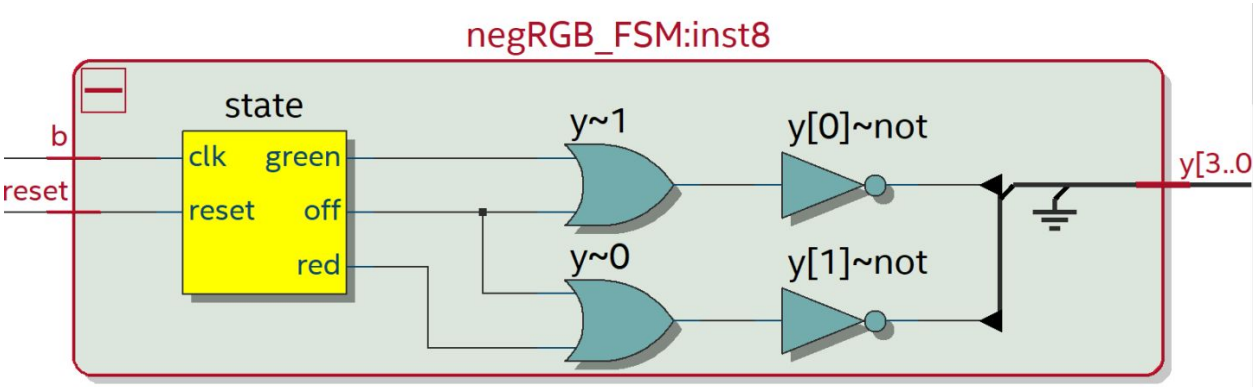

Figure 22: Quartus RTL Viewer of Negative RGB Finite State Machine

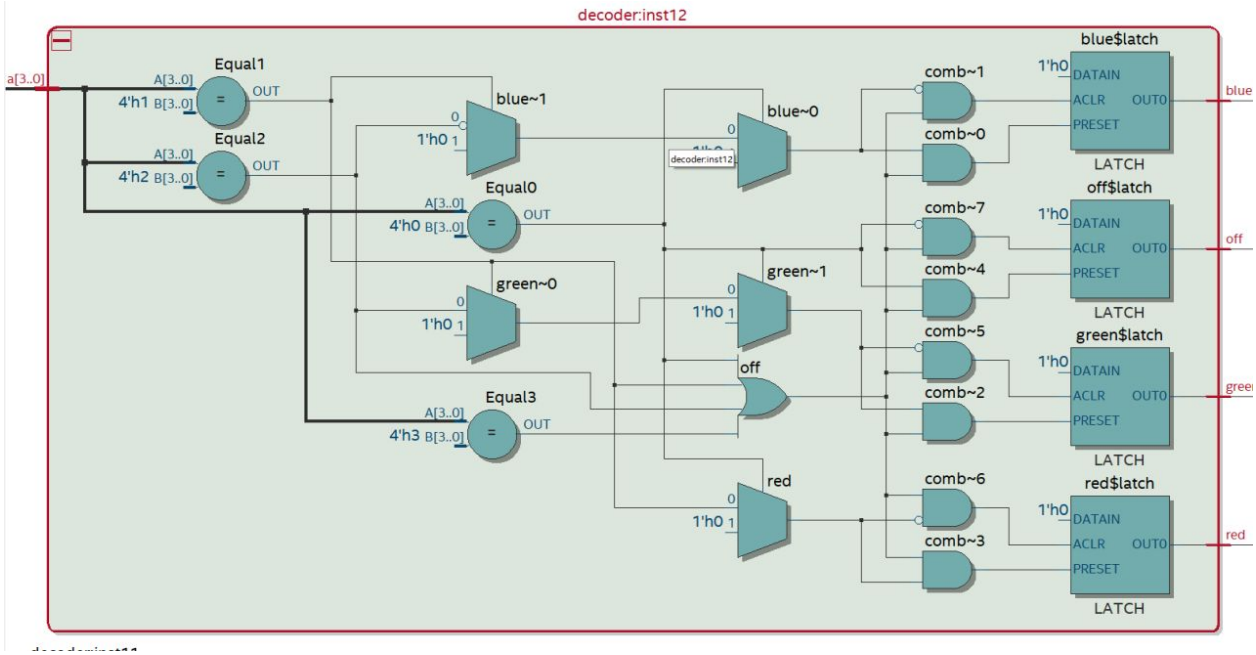

Figure 23: Quartus RTL Viewer of RGB Decoder

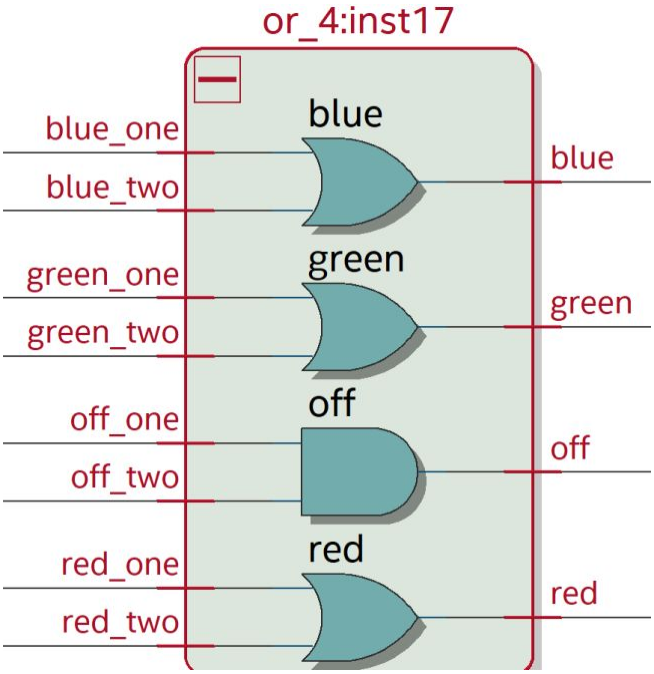

Figure 24: Quartus RTL Viewer of OR_4 Module

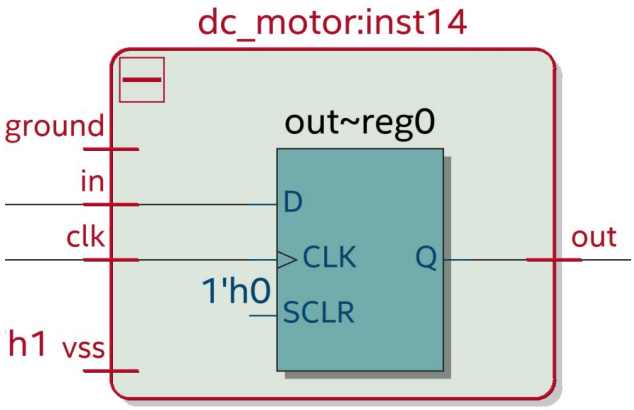

Figure 25: Quartus RTL Viewer of DC Motor Module

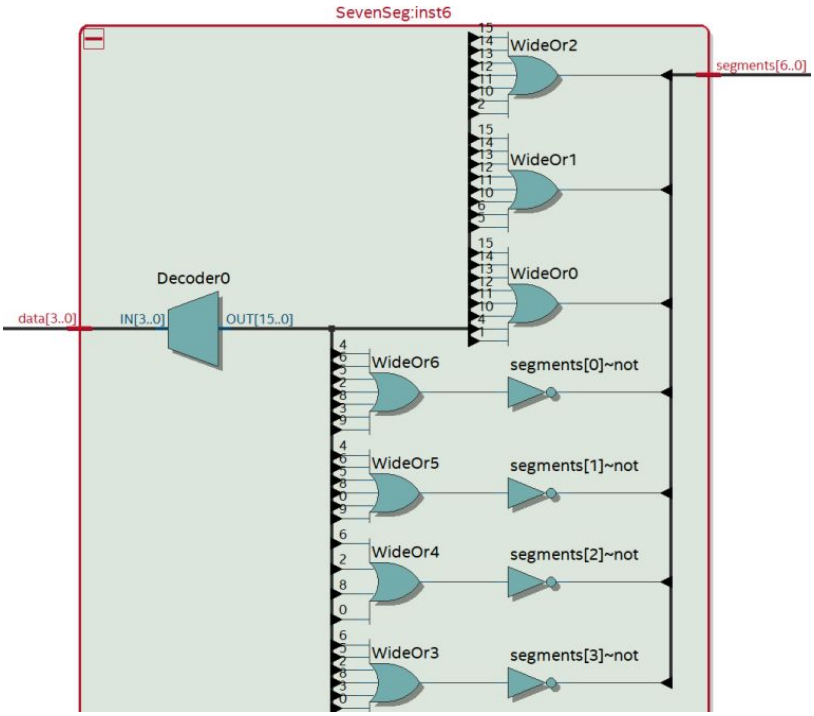

Figure 26: Partial Quartus RTL Viewer of Seven Seg Display Driver

Flow Summary

<<Filter>>

Flow Status	Successful - Fri Jun 05 15:14:03 2020
Quartus Prime Version	19.1.0 Build 670 09/22/2019 SJ Lite Edition
Revision Name	DesignProject
Top-level Entity Name	Top
Family	MAX 10
Device	10M08DAF484C8G
Timing Models	Final
Total logic elements	1,122 / 8,064 (14 %)
Total registers	39
Total pins	46 / 250 (18 %)
Total virtual pins	0
Total memory bits	0 / 387,072 (0 %)
Embedded Multiplier 9-bit elements	0 / 48 (0 %)
Total PLLs	0 / 2 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 1 (0 %)

Figure 27: Top Level Design Analysis

5.2 Source Code

5.2.1 top_module

```
module Top(
        clk,
        up_button,
        down_button,
        left_button,
        right_button,
        reset_button,
        a_button,
        b_button,
        select_button,
        start_button,
        red_rgb,
        green_rgb,
        blue_rgb,
        motor_out,
        off,
        latch_test,
        clock_test,
        select_test,
        seg0,
        seg1,
        seg2,
        seg3
);


input wire      clk;
input wire      up_button;
input wire      down_button;
input wire      left_button;
input wire      right_button;
input wire      reset_button;
input wire      a_button;
input wire      b_button;
input wire      select_button;
input wire      start_button;
output wire     red_rgb;
output wire     green_rgb;
```

```
output wire     blue_rgb;
output wire     motor_out;
output wire     off;
output wire     latch_test;
output wire     clock_test;
output wire     select_test;
output wire     [6:0] seg0;
output wire     [6:0] seg1;
output wire     [6:0] seg2;
output wire     [6:0] seg3;

wire    a;
wire    b;
wire    blue_wire1;
wire    blue_wire2;
wire    clk_wire;
wire    clock;
wire    data_wire;
wire    [13:0] display;
wire    down;
wire    [3:0] fsm1;
wire    [3:0] fsm2;
wire    GND;
wire    green_wire1;
wire    green_wire2;
wire    [3:0] hundreds;
wire    latch;
wire    left;
wire    off_wire1;
wire    off_wire2;
wire    [3:0] ones;
wire    red_wire1;
wire    red_wire2;
wire    reset_n_wire;
wire    reset_wire;
wire    right;
wire    select;
wire    start;
wire    [3:0] tens;
wire    [3:0] thousands;
wire    up;
wire    VCC;
```

```
posRGB_FSM b2v_inst(
        .a(a),
        .reset(reset_wire),
        .y(fsm1));

adder   b2v_inst1(
        .up(up),
        .down(down),
        .left(left),
        .right(right),
        .reset(reset_wire),
        .q(display));
        defparam        b2v_inst1.N = 14;

decoder         b2v_inst11(
        .a(fsm1),
        .off(off_wire1),
        .red(red_wire1),
        .green(green_wire1),
        .blue(blue_wire1));

decoder         b2v_inst12(
        .a(fsm2),
        .off(off_wire2),
        .red(red_wire2),
        .green(green_wire2),
        .blue(blue_wire2));

dc_motor        b2v_inst14(
        .in(start),
        .vss(VCC),
        .clk(clk),
        .ground(GND),
        .out(motor_out));

or_4    b2v_inst17(
        .off_one(off_wire1),
        .red_one(red_wire1),
        .green_one(green_wire1),
        .blue_one(blue_wire1),
        .off_two(off_wire2),
        .red_two(red_wire2),
        .green_two(green_wire2),
```

```
        .blue_two(blue_wire2),
        .off(off),
        .red(red_rgb),
        .green(green_rgb),
        .blue(blue_rgb));

SevenSeg        b2v_inst2(
        .data(ones),
        .segments(seg0));

NESControllerb2v_inst22(
        .up(up_button),
        .down(down_button),
        .left(left_button),
        .right(right_button),
        .start(start_button),
        .select(select_button),
        .a(a_button),
        .b(b_button),
        .clock(clk),
        .reset(reset_wire),
        .data(data_wire),
        .clk(clk_wire),
        .reset_n(reset_n_wire));

parser  b2v_inst3(
        .ta(display),
        .a(ones),
        .b(tens),
        .c(hundreds),
        .d(thousands));


SevenSeg        b2v_inst4(
        .data(tens),
        .segments(seg1));

SevenSeg        b2v_inst5(
        .data(hundreds),
        .segments(seg2));

SevenSeg        b2v_inst6(
        .data(thousands),
```

```
        .segments(seg3));

NesReader      b2v_inst7(
        .dataYellow(data_wire),
        .clock(clk_wire),
        .reset_n(reset_n_wire),
        .latchOrange(latch),
        .clockRed(clock),
        .up(up),
        .down(down),
        .left(left),
        .right(right),
        .start(start),
        .select(select),
        .a(a),
        .b(b));

negRGB_FSM b2v_inst8(
        .b(b),
        .reset(reset_wire),
        .y(fsm2));

assign  reset_wire = reset_button;
assign  latch_test = latch;
assign  clock_test = clock;
assign  select_test = select;
assign  GND = 0;
assign  VCC = 1;

endmodule
```

5.2.2 adder

```
module adder #(parameter N = 14)
              (input logic up, down, left, right,
               input logic reset,
               output logic [N-1:0] q);
  always_ff @(posedge up, posedge down, posedge left, posedge right, posedge reset)
    begin
      if (reset) q <= 14'b0;
      else if (up) q <= q + 14'b0001;
      else if (down) q <= q - 14'b0001;
```

```
        else if (left) q <= q + 14'b1010;
        else if (right) q <= q - 14'b1010;
      end
endmodule
```

5.2.3 parser

```
module parser (input logic [13:0] ta,
               output logic [3:0] a, b, c ,d);
  always_comb
    begin
      a = ta % 4'b1010;
      b = ((ta - (a)) / 4'b1010) % 4'b1010;
      c = ((((ta - (a)) / 4'b1010) - b) / 4'b1010) % 4'b1010;
      d = ((((((ta - (a)) / 4'b1010) - b) / 4'b1010) - c) / 4'b1010) % 4'b1010;
    end
endmodule
```

5.2.4 aRGB

```
module posRGB_FSM(input logic a,
                  input logic reset,
                  output logic [3:0]y);
  typedef enum logic [1:0] {off, red, green, blue} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge a, posedge reset)
    if (reset) state <= off;
    else state <= nextstate;

  // next state logic
  always_comb
    case (state)
      off:     nextstate <= red;
      red:     nextstate <= green;
      green:   nextstate <= blue;
      blue:    nextstate <= off;
      default: nextstate <= off;
    endcase
  // output logic
```

```
  always_comb
    case (state)
      off:    y <= 4'b00;
      red:    y <= 4'b01;
      green:  y <= 4'b10;
      blue:   y <= 4'b11;
    endcase
endmodule
```

5.2.5 bRGB

```
module negRGB_FSM(input logic b,
                  input logic reset,
                  output logic [3:0]y);
  typedef enum logic [1:0] {off, red, green, blue} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge b, posedge reset)
    if (reset) state <= off;
    else state <= nextstate;

  // next state logic
  always_comb
    case (state)
      off:    nextstate <= blue;
      blue:   nextstate <= green;
      green:  nextstate <= red;
      red:    nextstate <= off;
      default: nextstate <= off;
    endcase
  // output logic
  always_comb
    case (state)
      off:    y <= 4'b00;
      red:    y <= 4'b01;
      green:  y <= 4'b10;
      blue:   y <= 4'b11;
    endcase
endmodule
```

5.2.6 RGBdecoder

```
module decoder(input logic [3:0] a,
          output logic off, red, green, blue);

      always@(*)
        if (a == 4'b00)
          begin
            off <= 1;
            red <= 0;
            green <= 0;
            blue <= 0;
          end
        else if (a == 4'b01)
          begin
            off <= 0;
            red <= 1;
            green <= 0;
            blue <= 0;
          end
        else if (a == 4'b10)
          begin
            off <= 0;
            red <= 0;
            green <= 1;
            blue <= 0;
          end
        else if (a == 4'b11)
          begin
            off <= 0;
            red <= 0;
            green <= 0;
            blue <= 1;
          end

endmodule
```

5.2.7 OR_4

```
module OR_4(input logic off_one, red_one, green_one, blue_one, off_two, red_two,
green_two, blue_two,
                        output logic off, red, green, blue);

    assign off = off_one && off_two;
    assign red = red_one || red_two;
    assign green = green_one || green_two;
    assign blue = blue_one || blue_two;

endmodule
```

5.2.8 sevSeg

```
module SevenSeg(input logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case(data)
      // abc_defg
      0: segments = 7'b000_0001;
      1: segments = 7'b100_1111;
      2: segments = 7'b001_0010;
      3: segments = 7'b000_0110;
      4: segments = 7'b100_1100;
      5: segments = 7'b010_0100;
      6: segments = 7'b010_0000;
      7: segments = 7'b000_1111;
      8: segments = 7'b000_0000;
      9: segments = 7'b000_1100;
    default: segments = 7'b111_1111;
  endcase
endmodule
```

5.2.9 NES_reader

```
module NesReader(
 input logic dataYellow,
 input logic clock,
 input logic reset_n,
 output logic latchOrange,
 output logic clockRed,
 output logic up,
```

```
  output logic down,
  output logic left,
  output logic right,
  output logic start,
  output logic select,
  output logic a,
  output logic b
  );
  logic [3:0] count;

  Counter4 matt_i1(
    .clk              (clock),
    .reset_n          (reset_n),
    .count            (count)
  );

  NesClockStateDecoder matt_i2(
    .controllerState  (count),
    .nesClock         (clockRed)
  );

  NesLatchStateDecoder matt_i3 (
    .controllerState  (count),
    .nesLatch         (latchOrange)
  );

  NesDataReceiverDecoder matt_i4 (
    .dataYellow       (dataYellow),
    .reset_n          (reset_n),
    .controllerState  (count),
    .readButtons      ({a, b, select, start, up, down, left, right})
  );
endmodule


module Counter4(
  input logic clk, reset_n,
  output logic [3:0] count);

  always_ff @ (posedge clk, negedge reset_n)
    if(!reset_n) count <= 4'b0;
    else count <= count + 1;
endmodule
```

```
module NesLatchStateDecoder(
  input logic [3:0] controllerState,
  output logic nesLatch);

  always_comb
    case(controllerState)
      4'h0: nesLatch = 1;
      default: nesLatch = 0;
    endcase
endmodule


module NesClockStateDecoder(
  input logic [3:0] controllerState,
  output logic nesClock);

  always_comb
    case (controllerState)
      4'h2: nesClock = 1;
      4'h4: nesClock = 1;
      4'h6: nesClock = 1;
      4'h8: nesClock = 1;
      4'ha: nesClock = 1;
      4'hC: nesClock = 1;
      4'hE: nesClock = 1;
      default: nesClock = 0;
    endcase
endmodule


module NesDataReceiverDecoder(
  input logic dataYellow,
  input logic reset_n,
  input logic [3:0] controllerState,
  output logic [7:0] readButtons);

  always_ff @ (posedge controllerState[0], negedge reset_n)
    if(!reset_n) readButtons <= 8'b0;
    else case(controllerState[3:0])
      4'h1: readButtons[7] <= dataYellow;        //a button
      4'h3: readButtons[6] <= dataYellow;        //b button
```

```
    4'h5: readButtons[5] <= dataYellow;       //select button
    4'h7: readButtons[4] <= dataYellow;       //start button
    4'h9: readButtons[3] <= dataYellow;       //up button
    4'hB: readButtons[2] <= dataYellow;       //down button
    4'hD: readButtons[1] <= dataYellow;       //left button
    4'hF: readButtons[0] <= dataYellow;       //right button
    default: readButtons <= readButtons;
  endcase
endmodule
```

5.2.10 NES_controller

```
module NESController(input logic up, down, left, right, start, select, a, b, clock, reset,
                     output logic data, clk, reset_n);
                     logic [3:0] count;

  always_comb
  begin
    clk <= clock;
    reset_n <= !reset;
  end

  always_ff @ (posedge clock, posedge reset)
  begin
    if(reset || count == 16)
    begin
      count <= 4'b0;
      data <= 0;
    end
    else
    begin
      count <= count + 1;
        if (count == 0 && a == 1) data <= 1;
        else if (count == 2 && b == 1) data <= 1;
        else if (count == 4 && select == 1) data <= 1;
        else if (count == 6 && start == 1) data <= 1;
        else if (count == 8 && up == 1) data <= 1;
        else if (count == 10 && down == 1) data <= 1;
        else if (count == 12 && left == 1) data <= 1;
        else if (count == 14 && right == 1) data <= 1;
        else data <= 0;
      end
```

```
    end

endmodule
```

5.2.11 DC motor

```
module dc_motor(input logic in,
                input logic vss,
                input clk,
                input ground,
                output logic out);

    always@(posedge clk)
      if (in == 1) out <= 1;
      else out <= 0;

endmodule
```

5.3 Simulation Results

5.3.1 top_module

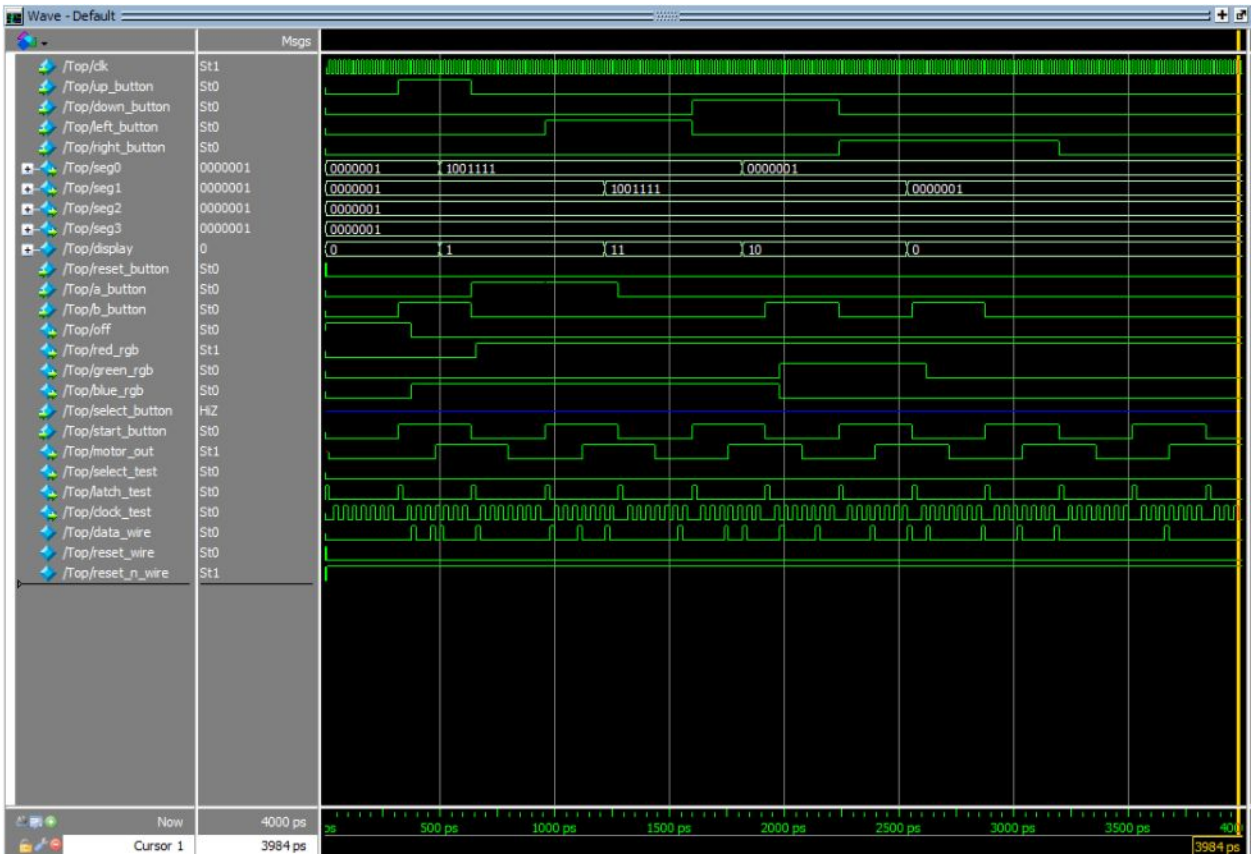

Figure 28: Simulation of top design with every output tested

Do file:

```
vsim work.Top
add wave *
force reset_button 0 @ 0, 1 @ 5, 0 @ 10
force up_button 0 @ 0, 1 @ 320, 0 @ 640
force down_button 0 @ 0, 1 @ 1600, 0 @ 2240
force left_button 0 @ 0, 0 @ 640, 1 @ 960, 0 @ 1600
force right_button 0 @ 0, 1 @ 2240, 0 @ 3200
force a_button 0 0, 1 640, 0 960, 1 960, 0 1280
force b_button 0 0, 1 320, 0 640, 1 1920, 0 2240, 1 2560, 0 2880
force clk 0 10, 1 20 -r 20
force start_button 0 0, 1 320 -r 640
run 4000
```

5.3.2 adder

Figure 29: Test to see value when arrow keys are pushed

Do file:

vsim work.adder
add wave *
force up 0 0, 1 10, 0 20, 1 30, 0 40, 1 50, 0 60, 0 70, 0 80, 0 90, 0 100
force down 0 0, 0 10, 0 20, 0 30, 0 40, 0 50, 0 60, 1 70, 0 80, 0 90, 1 100
force left 0 0, 0 10, 1 20, 0 30, 1 40, 0 50, 0 60, 0 70, 0 80, 1 90, 0 100
force right 0 0, 0 10, 0 20, 0 30, 0 40, 0 50, 1 60, 0 70, 1 80, 0 90, 0 100
force reset 1 0, 0 5
run 300

5.3.3 parser

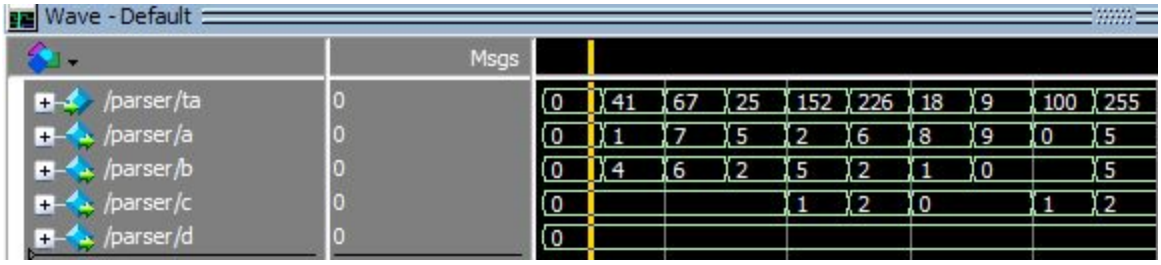

Figure 30: Simulation of what number is parsed to each segment

Do file:

vsim work.parser
add wave *
force ta 0 0, 00101001 10, 01000011 20, 00011001 30, 10011000 40, 11100010 50, 00010010 60, 1001 70, 01100100 80, 11111111 90
run 100

5.3.4 aRGB

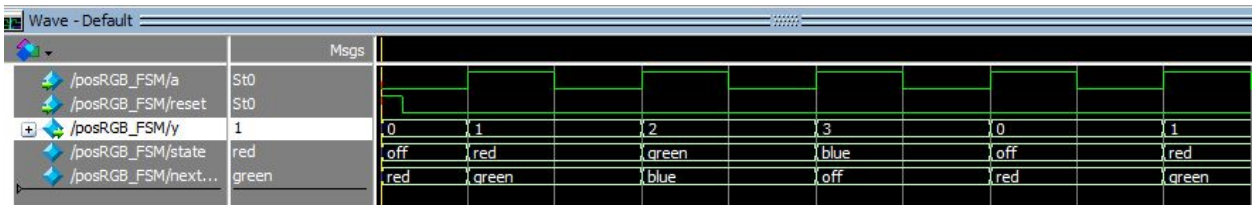

Figure 31: Test to see what color is lit up from pressing A button

<u>Do file:</u>

```
vsim work.posRGB_FSM
add wave *
force reset 1 @ 0, 0 @ 5
force a 0 @ 0, 1 @ 20 -r 40
run 200
```

5.3.5 bRGB

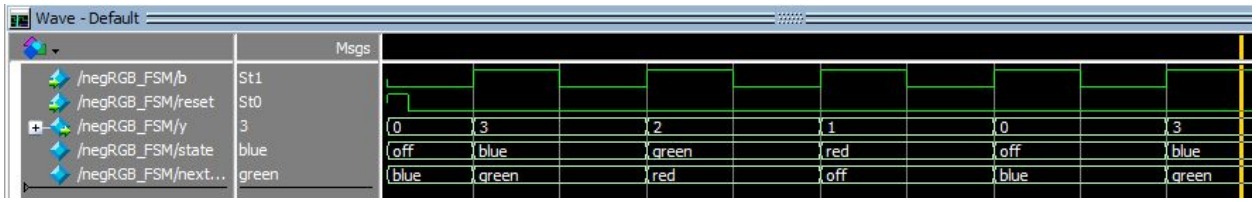

Figure 32: Test to see what color is lit up from pressing B button

<u>Do file:</u>

```
vsim work.negRGB_FSM
add wave *
force reset 1 @ 0, 0 @ 5
force b 0 @ 0, 1 @ 20 -r 40
run 200
```

5.3.6 RGBdecoder

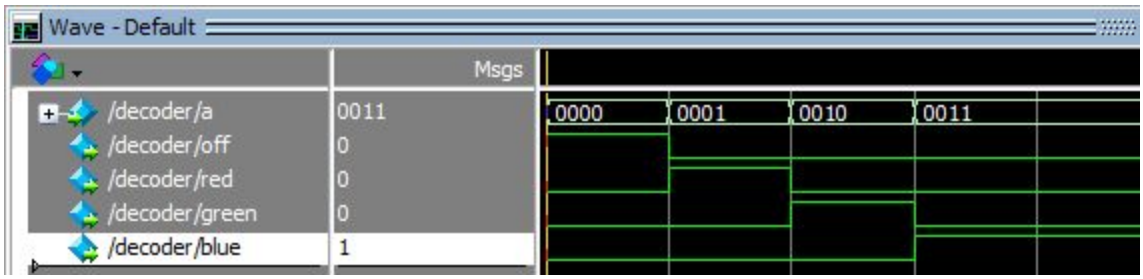

Figure 33: Decoder value test for A button value

Do file:

vsim work.decoder
add wave *
force a 00 @ 0, 01 @ 20, 10 @ 40, 11 @ 60
run 100

5.3.7 OR_4

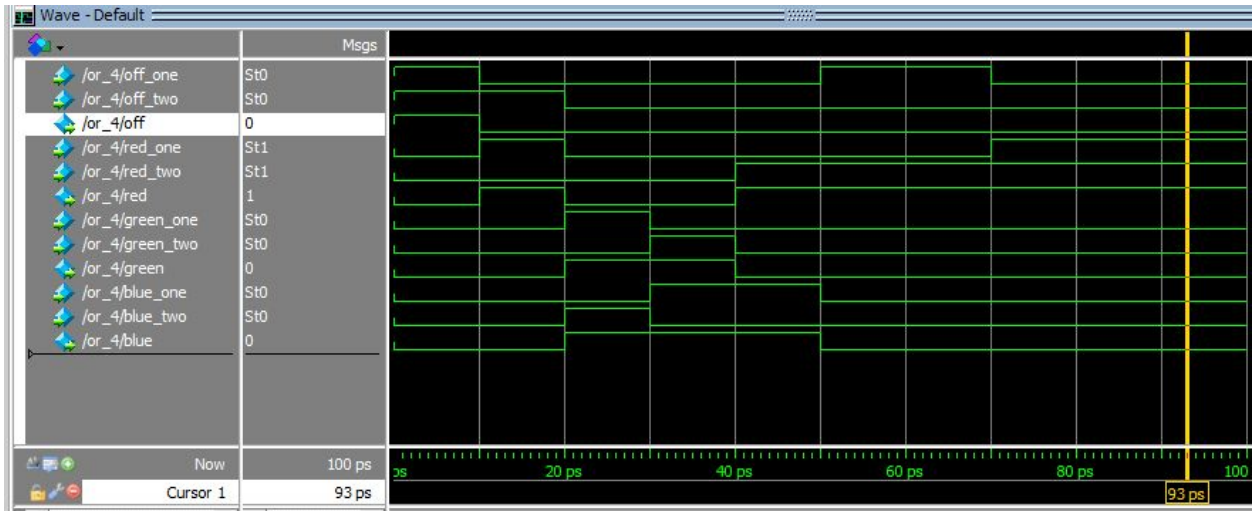

Figure 34: Test to check RGB color after pressing A and B buttons

Do file:

vsim work.or_4
add wave *
force off_one 1 0, 0 10, 0 20, 0 30, 0 40, 1 50, 1 60, 0 70
force red_one 0 0, 1 10, 0 20, 0 30, 0 40, 0 50, 0 60, 1 70
force green_one 0 0, 0 10, 1 20, 0 30, 0 40, 0 50, 0 60, 0 70

force blue_one 0 0, 0 10, 0 20, 1 30, 1 40, 0 50, 0 60, 0 70
force off_two 1 0, 1 10, 0 20, 0 30, 0 40, 0 50, 0 60, 0 70
force red_two 0 0, 0 10, 0 20, 0 30, 1 40, 1 50, 1 60, 1 70
force green_two 0 0, 0 10, 0 20, 1 30, 0 40, 0 50, 0 60, 0 70
force blue_two 0 0, 0 10, 1 20, 0 30, 0 40, 0 50, 0 60, 0 70
run 100

5.3.8 sevSeg

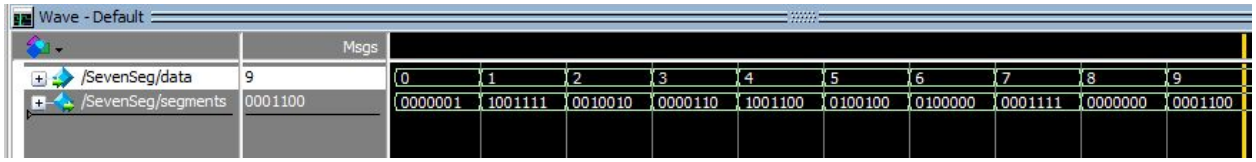

Figure 35: Simulation of 7 segment value displayed

Do file:

vsim work.SevenSeg
add wave *
force data 0000 0, 0001 10, 0010 20, 0011 30, 0100 40, 0101 50, 0110 60, 0111 70, 1000 80, 1001 90
run 100

5.3.9 NESreader

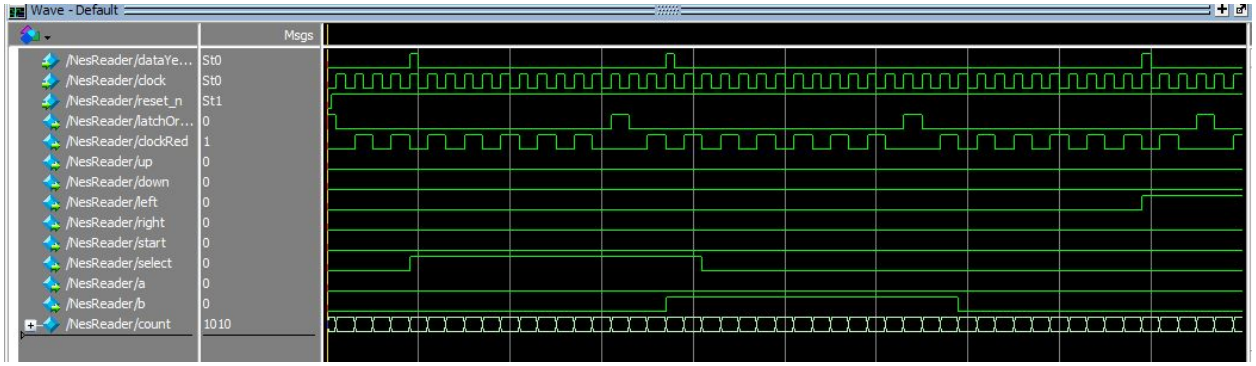

Figure 36: Test to see if NES reader correctly reads what buttons are pressed

Do file:

vsim work.NesReader
add wave *
force reset_n 0 @ 0, 1 @ 5
force clock 0 0, 1 10 -r 20
force dataYellow 0 0, 1 90, 0 100, 1 370, 0 380, 1 890, 0 900

run 5000

5.3.10 NEScontroller

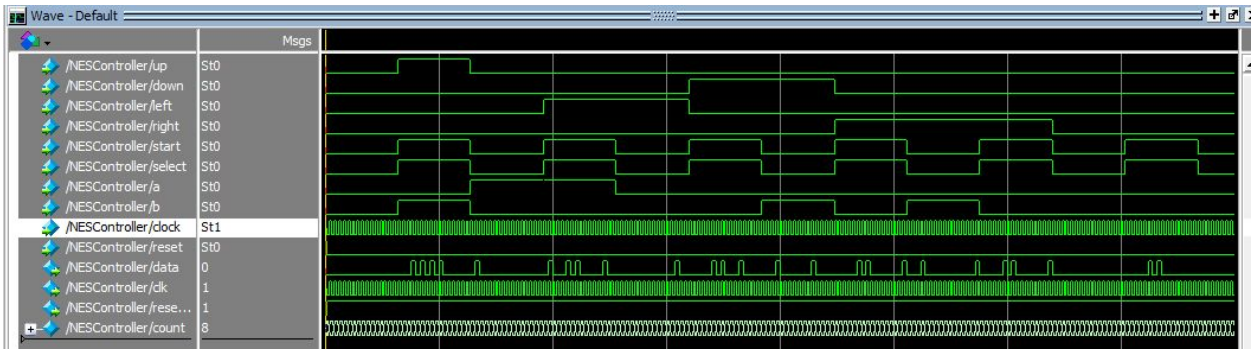

Figure 37: Test for NES Controller Inputs

Do file:

```
vsim work.NESController
add wave *
force reset 0 @ 0, 1 @ 5, 0 @ 10
force up 0 @ 0, 1 @ 320, 0 @ 640
force down 0 @ 0, 1 @ 1600, 0 @ 2240
force left 0 @ 0, 0 @ 640, 1 @ 960, 0 @ 1600
force right 0 @ 0, 1 @ 2240, 0 @ 3200
force a 0 0, 1 640, 0 960, 1 960, 0 1280
force b 0 0, 1 320, 0 640, 1 1920, 0 2240, 1 2560, 0 2880
force clock 0 10, 1 20 -r 20
force start 0 0, 1 320 -r 640
force select 0 0, 1 320 -r 640
run 4000
```

5.3.11 DC motor

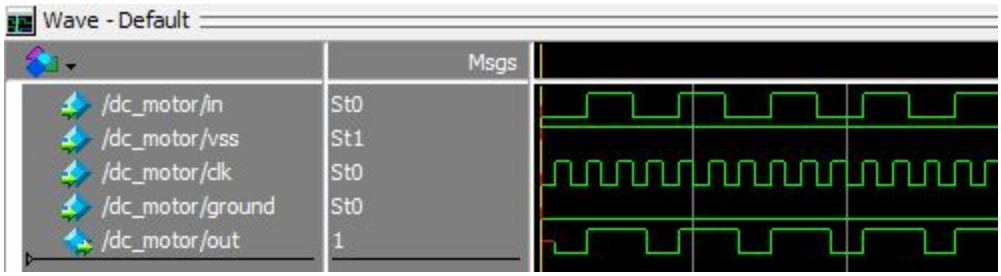

Figure 38: Test to show motor is moving when the Start button is pressed

Do file:

```
vsim work.dc_motor
add wave *
force in 0 @ 0, 1 @ 30 -r 60
force clk 0 0, 1 10 -r 20
force vss 1 0
force ground 0 0
run 300
```

6 References

[1] "Raspberry USB Nintendo NES Gamepad Controller for Raspberry Pi", *Pi Supply*, 2020. [Online]. Available: https://uk.pi-supply.com/products/raspberry-usb-nintendo-nes-gamepad-controller-for-raspberry-pi. [Accessed: 05- Jun- 2020].

[2] J. Corleto, "NES Controller Interface with an Arduino UNO - Projects", *Allaboutcircuits.com*, 2016. [Online]. Available: https://www.allaboutcircuits.com/projects/nes-controller-interface-with-an-arduino-uno/. [Accessed: 05- Jun- 2020].