ECE 271, Design Project, Group 05

Jazmin Cartagena, Brieanna Jeibmann, Ekaterina Rott, Shelby Westerberg

May 8th, 2020

Contents

1 Project Description

This design reads both a NES controller and a VCR remote control inputs. These inputs are later processed through out various hardwares described throughout the following sections in detail. These inputs are later processed out into three output signals. One is a square audio wave, the next is a RGB indicator, and the last is a seven segment display. The audio wave creates an audible sound when a button is depressed on one of the controllers. The RGB indicator creates a visual signal that a button has been depressed on one of the controllers and the seven segment display indicates how many times a button was depressed.

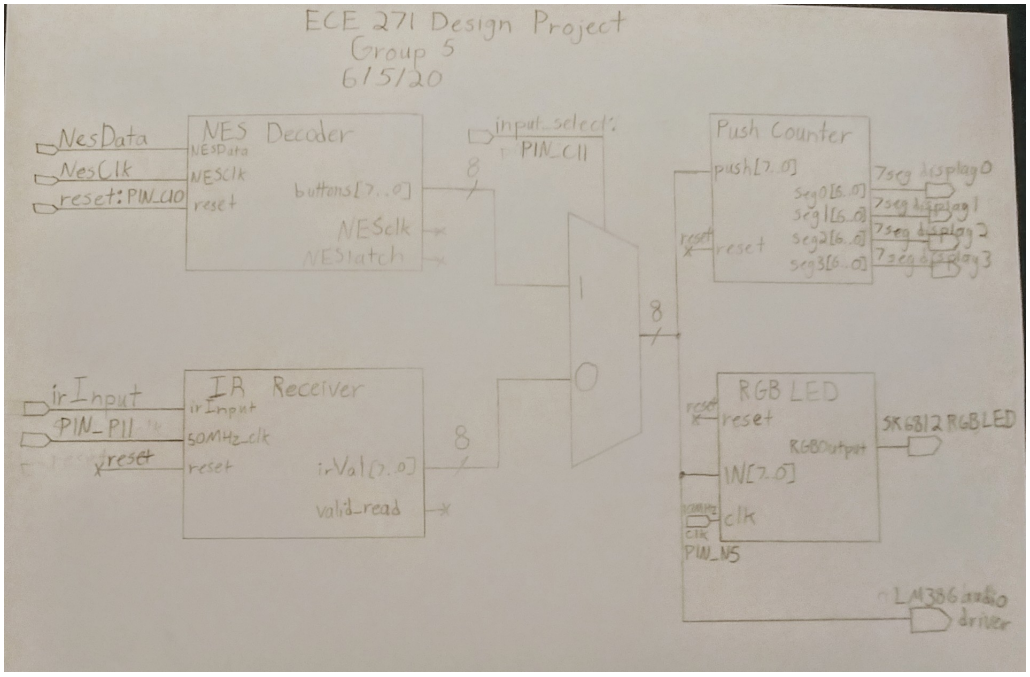

Figure 1: Project Block Diagram

2 High Level Description

Inputs: This reads an NES controller and the IR signals from a VCR remote.

Outputs: This displays a 4 digit value on the seven segment display, generates a square audio output, and changes the colors on a RGB LED.

2.1 NES Decoder

The NES controller is an 8-bit controller that transmits 8-bits of data, one bit for each of the buttons: A, B, up, down, right, left, select, and start. The NES controller contains a latch to store the state of the buttons as well as a clock. For every 60Hz, the NES sends a 12 microsecond signal to the latch indicating the latch to store the state of the buttons. After the latch signal, eight clocks signals are emitted—one for each button on the controller. If a button is pressed on the rising edge of a pulse the data is asserted to ground making the NES controller active low. [2]

The NES decoder for this unit is a modified version of the NESreader.sv provided by Matthew Shuman from his 2016 example file. It was modified to by separating all the modules into there own SystemVerilog files and put together in a top level diagram. Some of the naming conventions where also changed along with all hexadecimal values being converted to binary. This changes were made to improve the understanding and readability of the unit as a whole.

Inputs: Takes in a data wire that delivers eight bits, one bit per button, one bit at a time. It will also take in a clk signal to increment thought the buttons. The reset button initializes the the values in the decoder and resets the the data stored in the decoder.

Outputs: The NES decoder three outputs: an 8-bit bus, a clk out, and a latch. The 8-bit bus with 1-bit per button– right, left, down, up, start, select, b, and a. The clk output is a clock signal for the 8-bit bus, and in essence, is half the frequency as the input clk. The latch output is used as an indicator of the state of the latch.

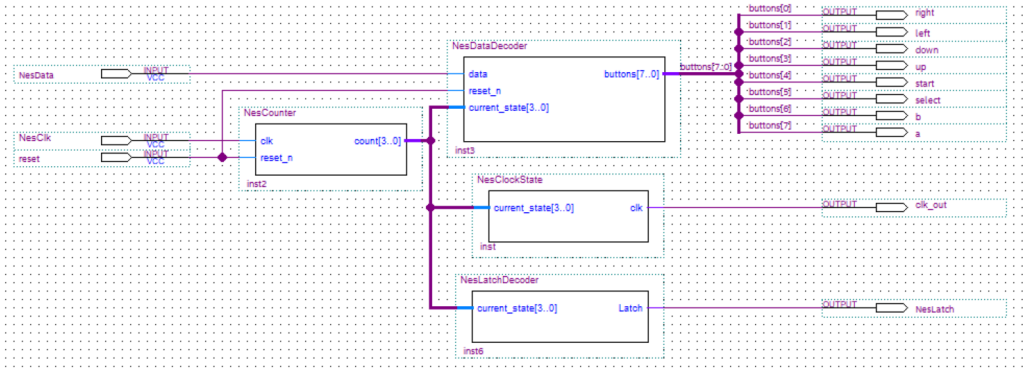

Figure 2: NES Decoder Top Level Diagram

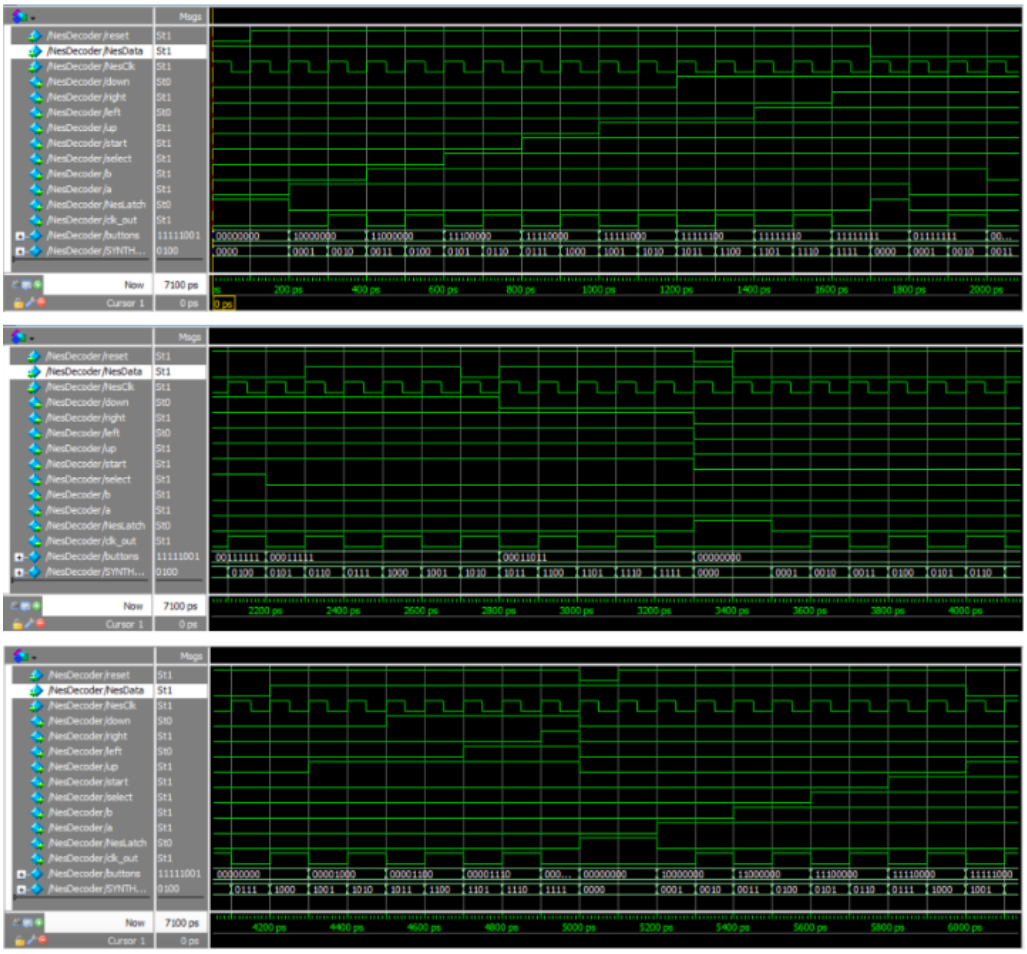

Figure 3: NES Decoder simulation

2.1.1 NES Counter

Inputs: The NES counter has two inputs:a clock signal and a reset signal. The reset signal is used to initialize the variables in the counter and to reset it.

Outputs: The NES counter outputs a variable that increments by one per clock cycle and resets to zero when reset is high when reset is pushed or the counter reaches 15.

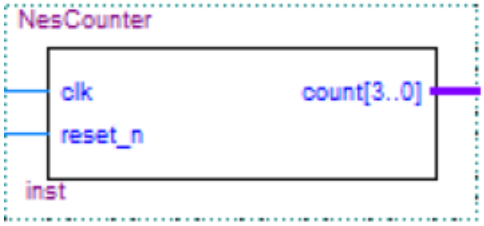

Figure 4: NES Counter Block

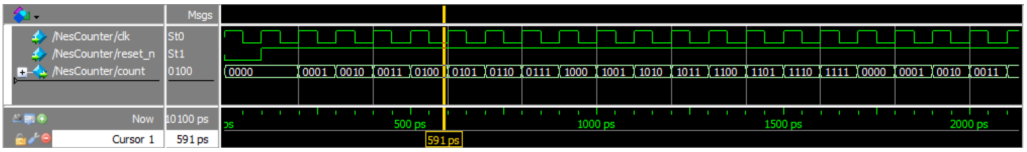

Figure 5: NES Counter simulation

2.1.2 NES Data Decoder

Inputs: The NES data decoder has three inputs: data, reset, and current state of the controller. The data input sends eight signals in a cycle to indicate if each button is high or low which is then stored. The reset input resets the data stored in the decoder back to zero and initializes the module to zero. The current state input is an 4-bit bus that has a variable used to tract which button data the data line is delivering.

Outputs: The data decoder outputs an 8-bit bus, 1-bit per button, that indicates which button is being pushed.

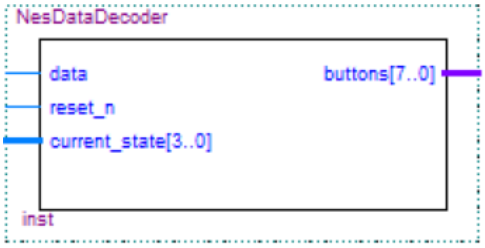

Figure 6: NES Data Decoder Block

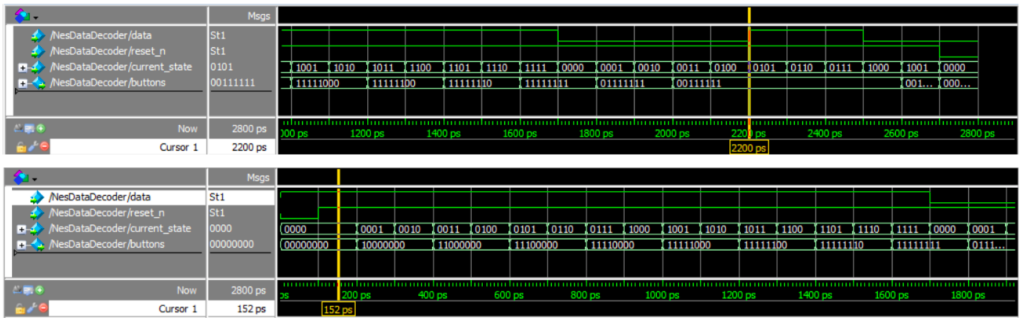

Figure 7: NES Decoder simulation

2.1.3 NES Clock State

Inputs: The NES clock state takes in a 4-bit bus input for the current state of the controller.

Outputs: The NES clock state outputs a clk signal for when using the data from the NES decoder.

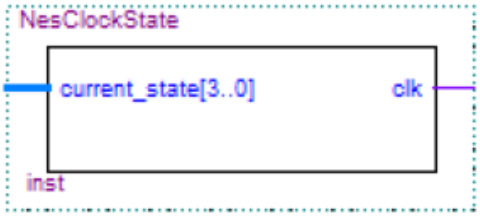

Figure 8: NES Clock State Block

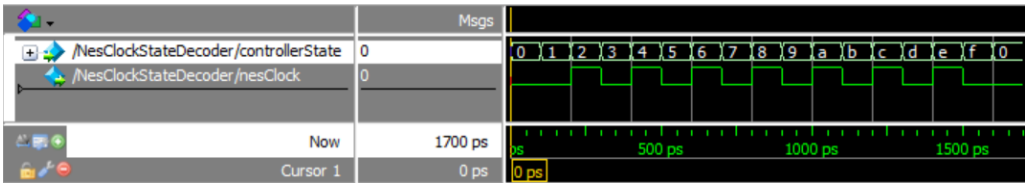

Figure 9: NES Counter simulation

2.1.4 NES Latch Decoder

Inputs: The NES latch decoder takes in a 4-bit bus input for the current state of the controller.
Outputs: The latch decoder output indicates the state of the latch of the NES controller.

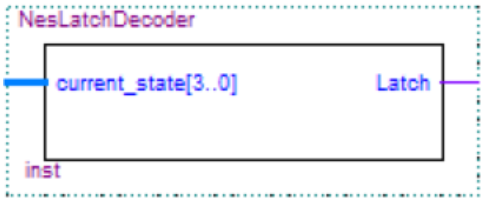

Figure 10: NES Latch Decoder Block

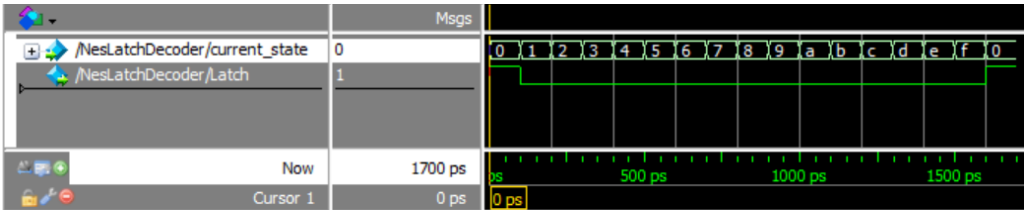

Figure 11: NES Counter simulation

2.2 Push Counter

The push counter is designed to count the number of times the buttons on the input controllers are pushed. The entire unit is based off of Lab 5 from ECE 272.

Inputs: The push counter takes in two inputs: an 8-bit bus and a reset. the reset input initializes the unit to 0 and will reset the unit to 0. The 8-bit bus represents if a button is being pushed. If one or more of the bits is high then that means a button is being pushed. Only one should be high for the NES controller but multiple for the VCR remote. The input is then fed into an 8-bit OR gate to create a clock, if any wire is high then the output of the OR gate is high. This works since between each button press there will be a period where all the bits of the bus are zero so a pulse for the clock can be generated.

Outputs: The push counter has four 7-bit busses: Seg0, Seg1, Seg2, and Seg3. The outputs go from least significant to most significant, Seg0 to Seg3, to represent the unsigned decimal value of how many times the buttons are pushed.

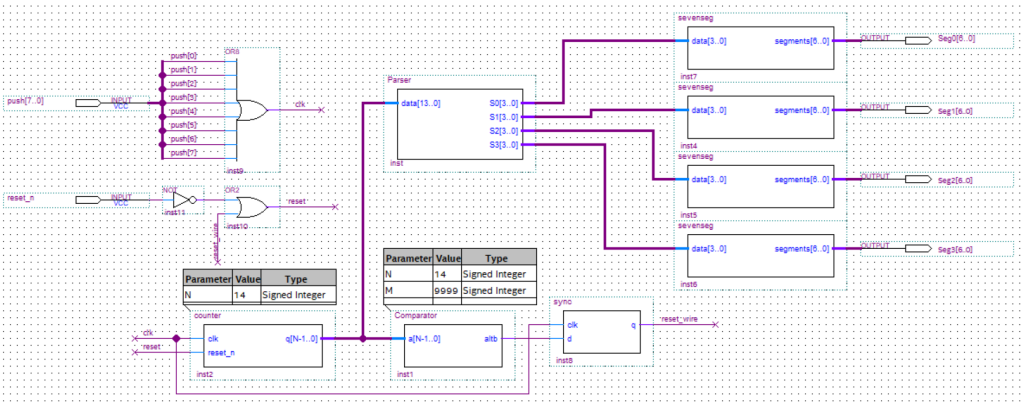

Figure 12: Push Counter Diagram

Figure 13: Push Counter simulation

2.2.1 Counter [1]

Inputs: The counter takes in two inputs: clock and reset. The clock value is generated when a button is pushed on the input controller. So it will increment the value of the counter by one per push of a button. The reset input resets the counter back to zero and initialized the register inside the counter to zero.

Output: The counter outputs a 14-bit bus that represents the current count of the counter in binary form.

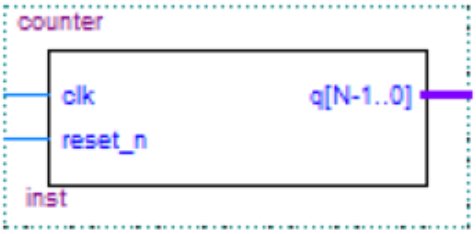

Figure 14: Counter Block

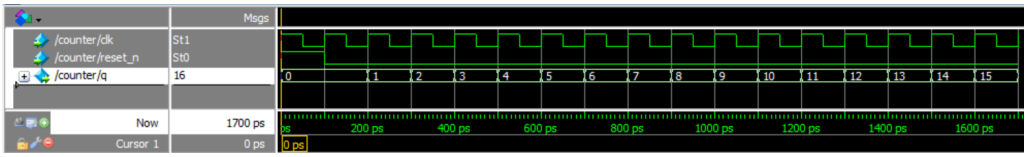

Figure 15: Counter simulation

2.2.2 Parser

Inputs: The parser takes in only one input, a 14-bit bus which can be represented in a decimal value. The parser then takes this input and splits it into individual parts that are between 0-9.

Output: The parser has four outputs: S0, S1, S2, S3. The outputs go from least significant to most significant, starting from S0 represents the ones place to S3 representing the thousands place.

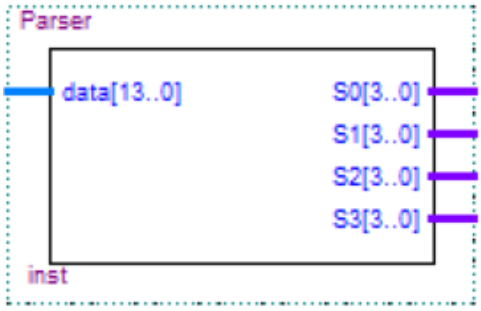

Figure 16: Parser Block

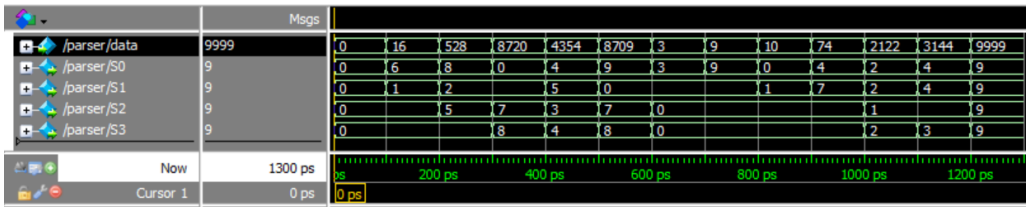

Figure 17: Parser simulation

2.2.3 Seven Segment Decoder

Inputs: The seven segment decoder takes in 4-bit bus that has a value that ranges from 0-15.

Output: The decoder outputs a 7-bit wire that would then lead to a seven segment display. The bus values determine which LED's on the display should be lit up to show a number. A 0 turns on the LED and a 1 turns off the LED

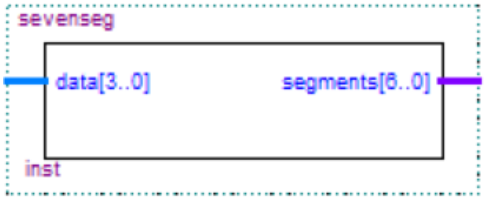

Figure 18: Seven Segment Decoder Block

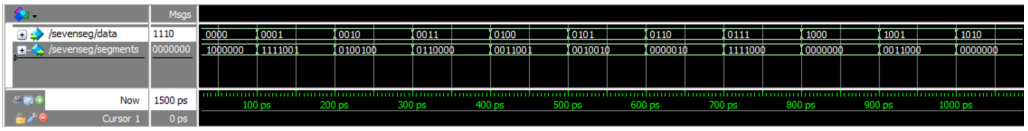

Figure 19: Seven Segment Decoder simulation

2.2.4 Comparator [1]

Inputs: The comparator takes in a 14-bit bus from the counter to comparator to a parameter, in this case it's testing whether the the decimal value of the 14-bit bus is greater than 9999.

Output: The comparator outputs a 1-bit value that states whether if the comparison is true or false. If the input is greater then or equal to 9999 the comparator outputs a 1, else it outputs a 0.

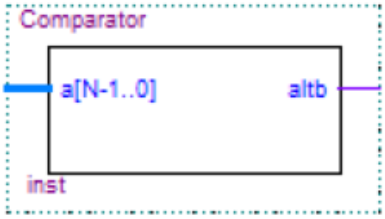

Figure 20: Comparator Block (N=14)

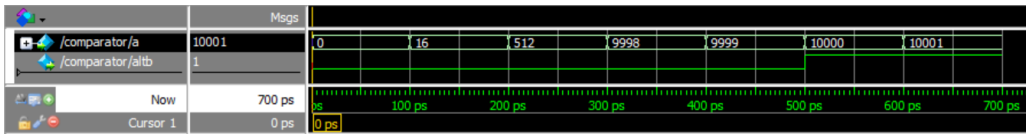

Figure 21: Comparator simulation

2.2.5 Synchronizer [1]

Inputs: The synchronize takes in two inputs: a clock signal and d. The synchronizer synchronizes the d with the clock signal.

Output: The output of the synchronizer is the same as the input d but is now in sync with the clock signal to ensure there are no timing issues.

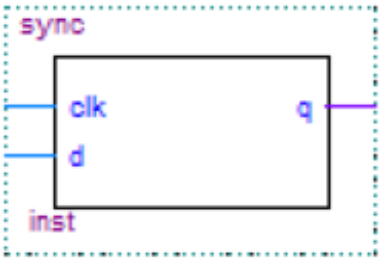

Figure 22: Synchronizer Block

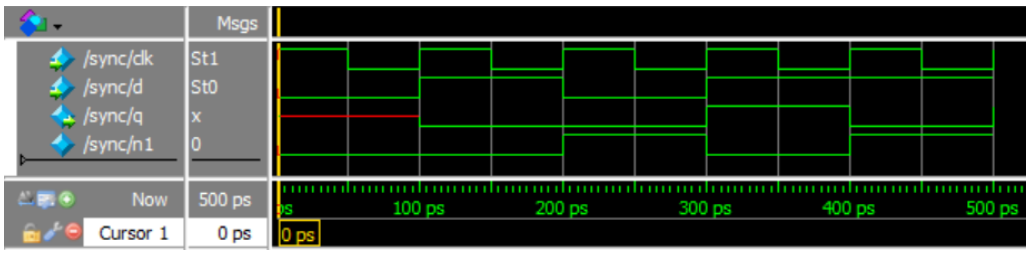

Figure 23: Synchronizer Simulation

2.3 Square Audio Output

This audio multiplexer is using the outputs signals from the NES and VCR controllers. The multiplexer is able to distinguish between the two controllers by using switch 0, which is on the FPGA. The switch is designed so that if the output is in the 1s position it is therefore designated for the NES controller, if the switch is in the 0s position than it is designated for the VCR controller. Once the switch position has been selected the 8 bits will travel to the audio multiplexer. The muliplexer uses these inputs to create an audible signal for the designated controller. All the buttons make the same audible noise.

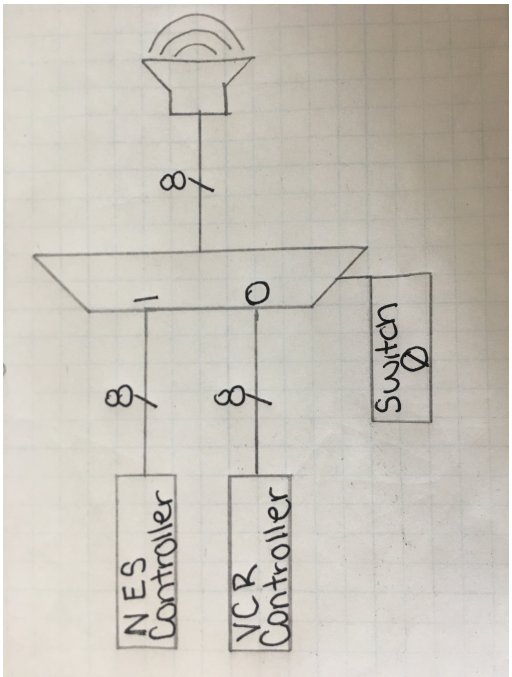

Figure 24: Audio output block Diagram

2.3.1 Audio Multiplexer

Inputs: NES Controller and VCR (IR) controller button depression with the designation between controllers by switch 0

Outputs: Audible noise for button depressed.

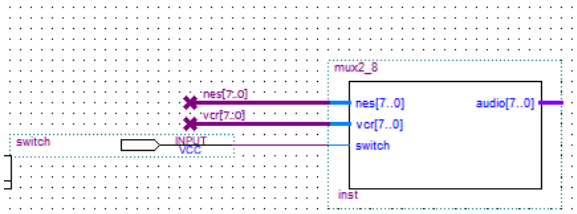

Figure 25: Multiplexer decoder block

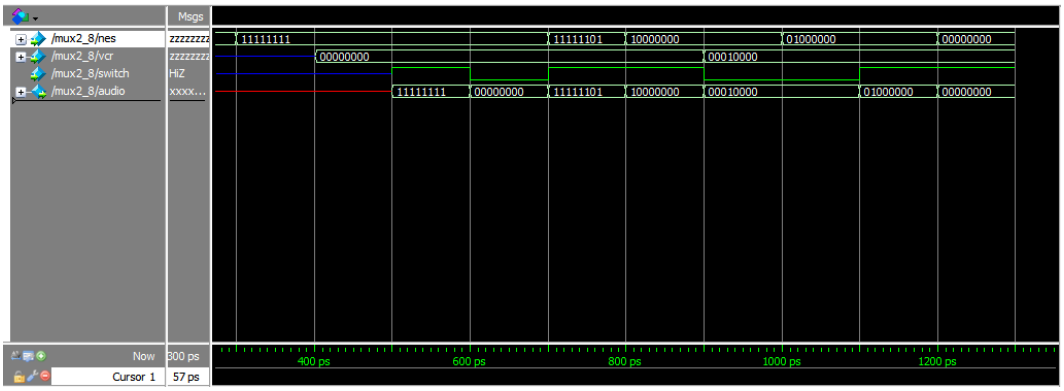

Figure 26: Audio Multiplexer

2.4 RGB LEDs

The SK6812 RGB Module has 3 inputs and 1 output. An input for power, ground, and data in, and an output of data out. Data in and data out are one bit each, and the RGB LED reads color serially, where 24 bits of information are read one by one. One bit is 9 microseconds of time, and resetting the color is 24 full bits of pulling the LED to ground. The block diagram created for the RGB LED Module is shown in figure 27

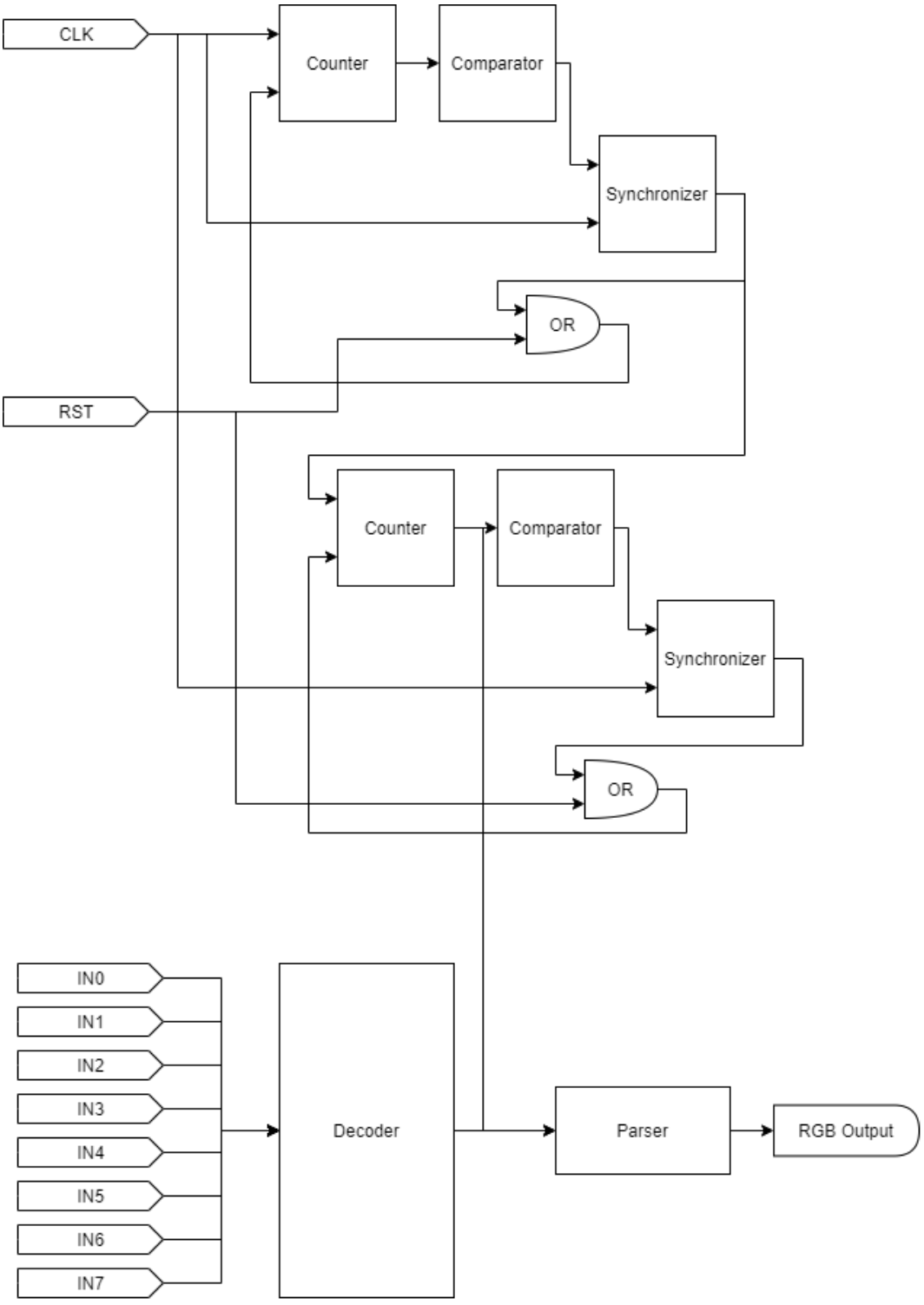

Figure 27: Block Diagram for the RGB LED Module

2.4.1 Clock Module

The first part of the RGB LED output is two counters linked together. The first takes the 10MHz clock on the DE10-Lite and counts 9 clock cycles and increments one, the second counts 24 of those 9 clock cycles. The first clock is used to make sure everything else in the RGB module only changes every 9 microseconds, the amount of time required for the RGB LED to read one bit of information. The second counter cycles through the 24 bits of information that gets output from the decoder block.

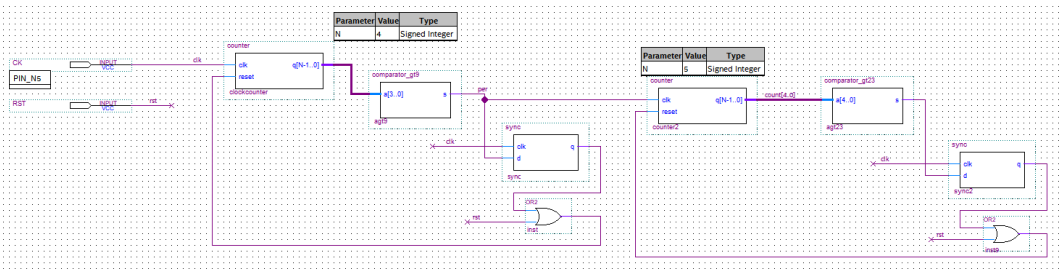

Figure 28: Schematic for the Clock Modules

One clock module is created with a counter, comparator, synchronizer, and an or gate. The counter, who's SystemVerilog is shown in figure 48, increments its output on every rising edge of the clock. The counter then gets fed into a comparator, either checking the counter has counted to 9 (as shown in figure 47) or to 24 (as shown in figure 46). When the comparator finds the clock has counted to the right number, it outputs a 1, which gets sent to a synchronizer, shown in figure 51. The output of the synchronizer and the output of a reset button both get sent to an OR gate, so the clock modules can be reset either by hand or by the counters.

Figure 28 shows how the two clocks are created and linked together in the top level diagram of the RGB module.

2.4.2 Decoder and Parser

The decoder and parser modules handle the output of any input module and the output to the RGB LED. The decoder takes any input and outputs 24 bit output corresponding to a color, as shown in figure 49. The decoder was directly connected to the parser, the SystemVerilog for which is shown in figure 50, which outputs the bit of the color input corresponding to a clock count. The schematic of the decoder and the parser is shown in figure 29.

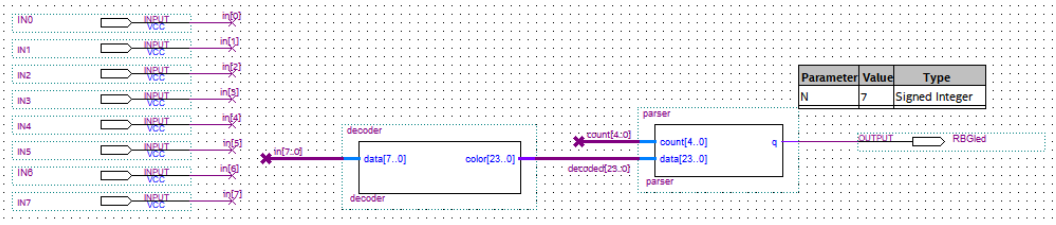

Figure 29: Schematic for the Decoder and Parser Modules

2.4.3 Testing

The testing for the decoder, parser, and clock modules are shown in figures 30, 31, 32, 33. Figure 30 shows the ModelSim simulation for asserting and un-asserting the reset button successfully. Figure 31 shows the simulation that takes an input and successfully changes the color output to a new number. Figure 32 shows the output of the parser module successfully cycling through the 24 bits output from the decoder module. Finally, figure 33 shows the decoder repeatedly changing its output when the input changes. Together, these four tests show that the RGB design successfully implements the requirements for the SK6812.

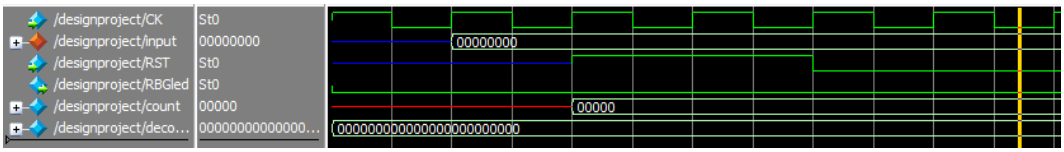

Figure 30: Reset Test

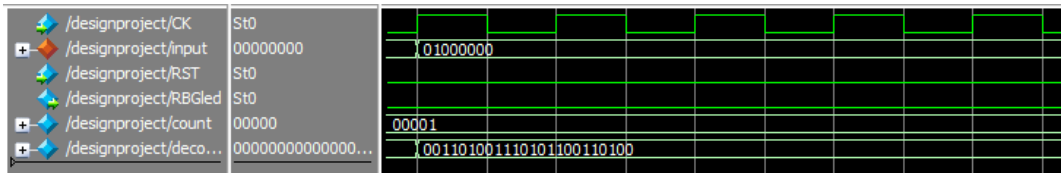

Figure 31: Decoder Test

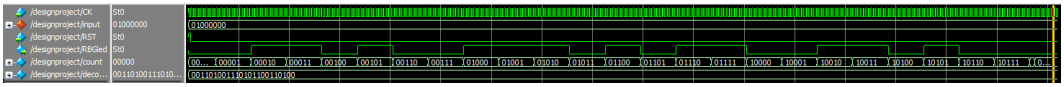

Figure 32: Output Test

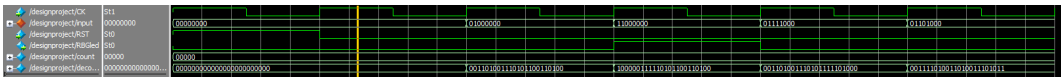

Figure 33: Changing Inputs Test

2.5 IR Receiver

Inputs: reset, clk_50MHz, irInput Outputs: irVal[7..0], valid_read

Infrared remotes and receivers see common usage in technologies such as television remotes. The remote outputs a particular frequency of IR light, commonly 38kHz, and the reader on the television or other device has a circuit such that a HIGH is outputted when this light is detected, LOW when it is not. This is where this IR receiver module picks up this input as irInput for the design project. What is described so far is a function of electrical circuits, and the resistance of particular components given different lighting conditions, and is beyond the scope of this project. Once this HIGH or LOW is detected, it must be decoded.

IR equipment uses a variety of different encoding schemes, however this project focuses on decoding inputs in the NEC code. This code comprises of a 9ms burst, called a leader code, then uses pulse distance coding to communicate the address and data bits. A short pause between bursts indicates a LOW, about 1.125ms between the end of one burst and the beginning of another. A long pause indicates a HIGH, about 2.25ms between the end of one burst and the beginning of another. The NEC code sends 8 address bits, then follows them up with their inverse, then sends 8 data bits, and finally their inverse. There is one final additional burst at the end to show when the end of the last data bit is.[3] Figure 34 shows an example of this pattern. The module created for this section ignores the address bits, and does not check the inverted data bits, though this could be added for error checking in some receivers.

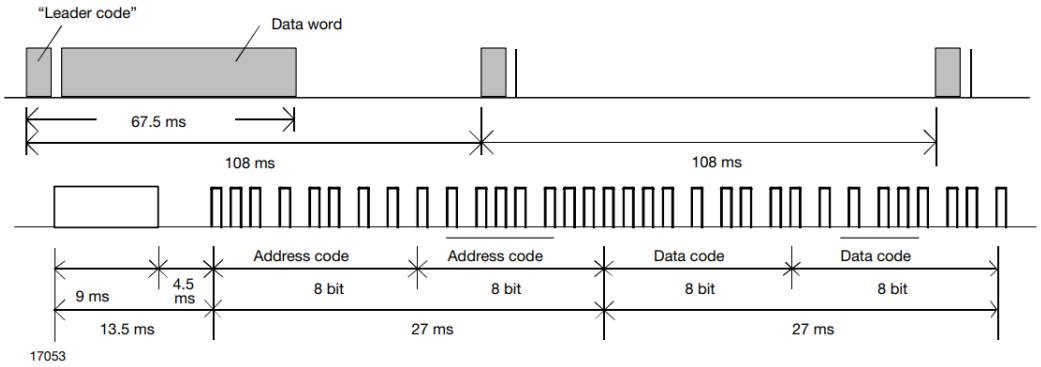

Figure 34: NEC Code Example Signal[3]

The following image shows a possible encoding scheme for an actual IR remote. Each button, shown on the left, is represented by an eight bit data output of the decimal number on the right. This one in particular is a Comcast/Panasonic remote. Although the hardware implementation was not done for this project, a lookup table of buttons such as this one would allows particular functions in other modules to be assigned to particular buttons on the remote.

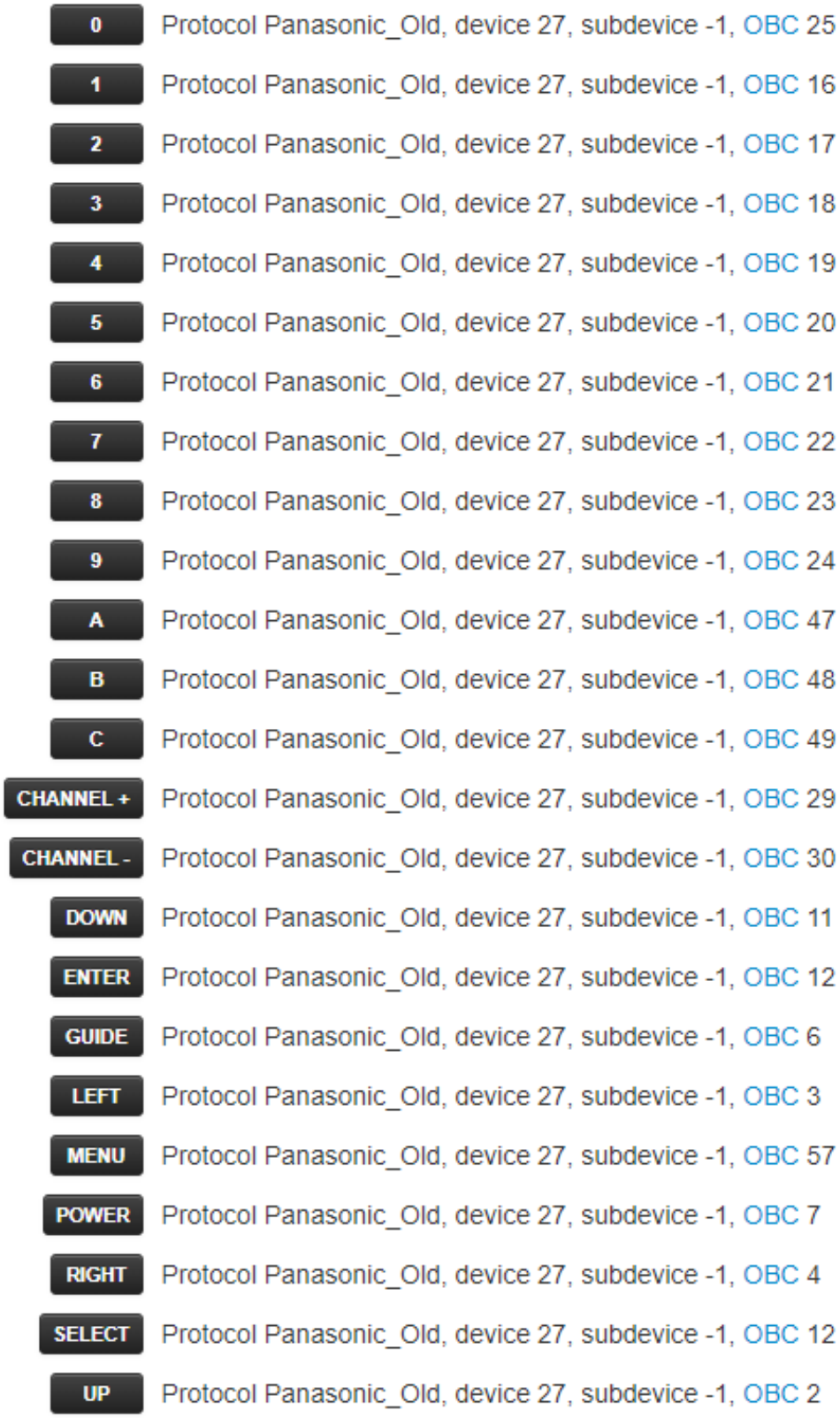

Figure 35: Comcast Old Panasonic Protocol[4]

To accomplish the decoding of such a pattern, the following module was developed. It is split into three main parts: the IR Decoder, which decides whether the most recent pause was long enough to generate a HIGH signal, the Toggle Start, which determines whether a sufficiently long leader code has been detected, and Input Counter, which keeps track of what point in the signal it is currently at. Notice that the data from the IR Decoder is only passed along if Toggle Start is outputting HIGH value, meaning that the receiver is in the middle of reading a valid signal because a leader code had been detected. The Input Counter is used for two purposes: as an index for storing the output of the IR Decoder, and as a reset for Toggle Start once the signal has ended. A clock divider is also included in order to split the 50MHz clock into a slower signal such that the counters need not keep track of and compare to unnecessarily large values. The eight bit binary value represented by the IR data is outputted via irVal. The output valid_read indicates whether the value stored in irVal is current, i.e. it is not in the process of being changed.

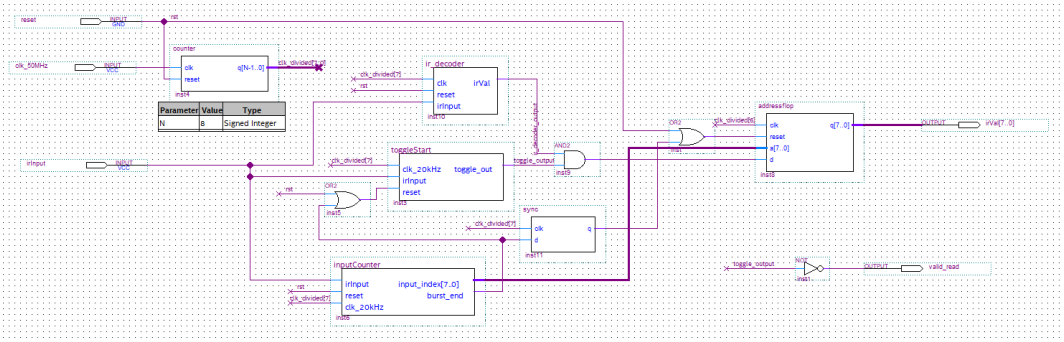

Figure 36: IR Receiver Design in Quartus

The number of divisions necessary for the clock cycle was determined by considering that in order to count clock cycles to measure the time between pulses, the number of clock cycles expected to go by should have a wide enough range to allow some variability in the length, but be narrow enough to fit easily on a small register. The FPGA's input clock cycle of 50MHz was divided with the goal of a clock with a period around 50us, which would allow for dozens of clock cycles for the HIGH/LOW test, and just under 200 for the leader code. Thus, $\frac{1}{50*10^(-6)us} = \frac{50MHz}{2^N}$, where N is the number of divisions needed. This gives an N of 8, so an 8 bit counter was used to divide the clock, and the most significant bit of that counter was used as the new clock, with a frequency of around 20kHz.

2.5.1 IR Decoder

Inputs: clk, reset, irInput Outputs: irVal

The purpose of the IR Decoder module is to determine whether the current low period has been long enough to count as a HIGH, and continuously output that value to irVal. The counter increases with each 20kHz clock cycle until the irInput is high, or reset is high at which point the counter is set back to zero. This count is fed into a comparator and checked against a hardcoded cutoff value. This cutoff value was calculated by taking halfway between the typical LOW and HIGH pulse distances, 1.125ms and 2.25ms respectively, and finding how many 20kHz clock cycles should go by: $\frac{N}{20*10^3} = 1.68 * 10^{-3}$, therefore $N = 33$.

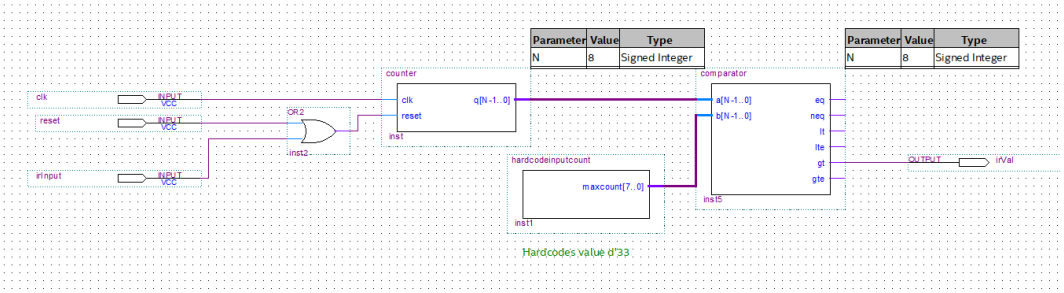

Figure 37: IR Input Decoder Design in Quartus

2.5.2 Toggle Start

Input: clk_20kHz, irInput, reset Output: toggle_out

The purpose of the Toggle Start module is to check whether there has been a leader code since the last time it was reset. Similar to the IR Decoder, a cutoff was determined by comparing the clock period to the desired time, just under 9ms. A key difference between the IR Decoder and Toggle Start is that Toggle start stores a HIGH value in an SR Flip-Flop, and outputs it to toggle_out, if this cutoff requirement is met, and does not revert to zero unless it is reset. This is because Toggle Start must continue outputting a HIGH value for the entire length of the data sequence in order for the data to be read in the top level.

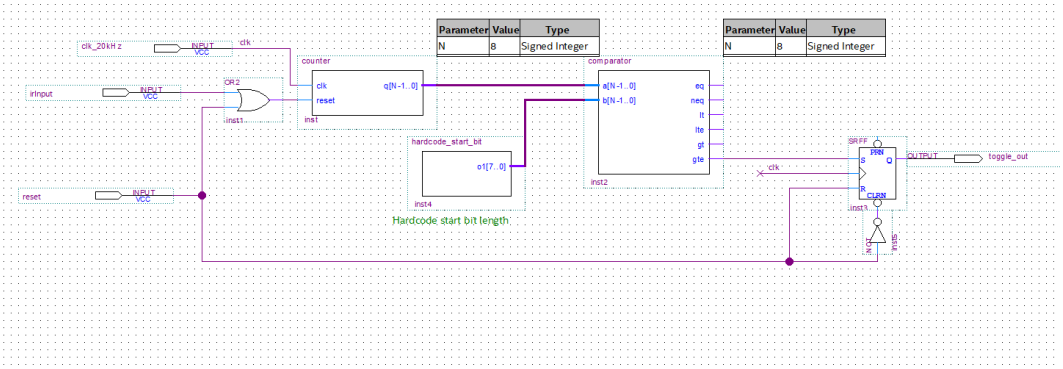

Figure 38: Toggle Start Design in Quartus

2.5.3 Input Counter

Inputs: irInput, reset Outputs: input_index[7..0], burst_end

The purpose of the Input Counter is to keep track of where in the signal the receiver is. Its counter is incremented on every rising edge of the IR input. This count is then compared to the expected number of rising edges in a single signal. When this evaluates to HIGH, burst_end is high for one clock cycle, then the input counter resets its counter. The current value of the counter is always outputted via input_index.

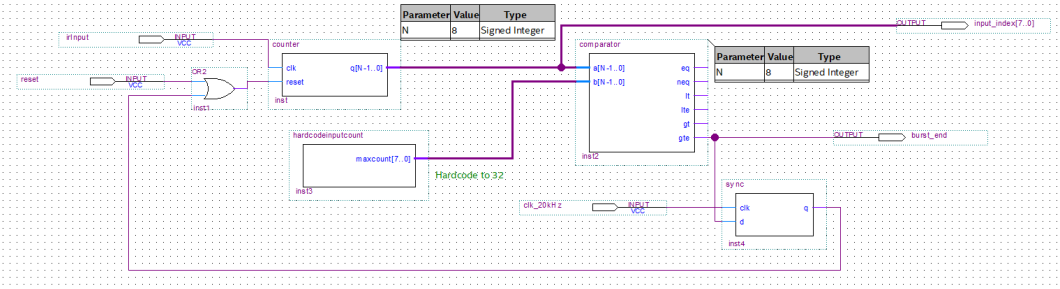

Figure 39: Input Counter Design in Quartus

2.5.4 Testing

The IR receiver was tested in ModelSim using the do-file in Appendix B. Additionally, the divided clock cycles were forced directly in ModelSim, rather than simulating 256 times as many clock cycles using the 50MHz clock. The following ModelSim screenshot of this simulation shows the end of the read. Just to the left of the yellow indicator the input goes HIGH for the final burst in the sequence, which causes the toggle read to go LOW, and in turn valid_read goes HIGH. The valid_read output was not needed for the outputs of this project, but was included to allow versatility should the IR receiver be used for outputs which cannot fluctuate before settling on the correct value. The value of irVal, the main data output for the module, then goes back to all zeros after two rising edges of the clock. This allows the Push Counter to differentiate between button pushes. The do-file for this simulation was created to mimic the IR input given by the example

NEC coded signal in Figure 34. The value outputted at irVal at the time of the yellow indicator matches the eight bit binary value given by that signal, indicating a successful test.

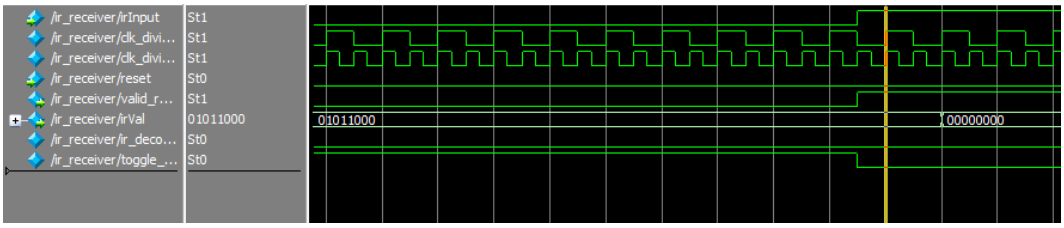

Figure 40: IR Receiver Functional Unit Test

The following ModelSim screenshot shows the test of the IR Decoder unit. It shows that the output irVal goes HIGH only once the decoder has counted at least 33 clock cycles, as calculated in section 2.5.1. Once the irInput is asserted, the count returns to zero, and irVal returns to LOW.

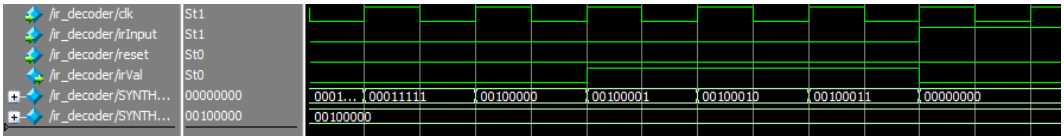

Figure 41: IR Decoder Module Test

Below shows the ModelSim results from testing the Toggle Start module. The first test shows the toggle_out going HIGH as soon as toggle_count hits 160, the cutoff for finding a leader code. This was simulated by forcing the clk_20kHz, and forcing the irInput to LOW. The second test shows toggle_out staying HIGH even after irInput is asserted again, which resets the toggle_count to zero. The second test also shows that asserting reset does set toggle_out to LOW.

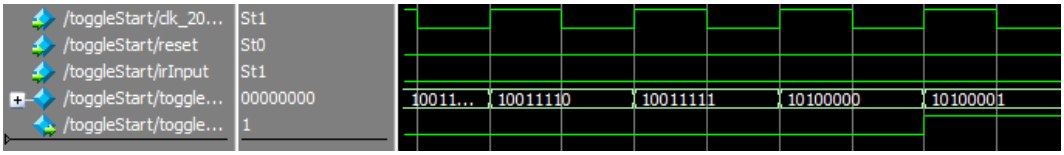

Figure 42: Toggle Start Module Test One

Figure 43: Toggle Start Module Test Two

Below shows the ModelSim test results from testing the Input Counter module. This was done by forcing both the clk and irInput to act as clocks. The burst end output only goes HIGH when the count is 32 or higher, showing that the signal is at the end of its burst. This then causes a reset, so count starts over at zero on the next rising clock edge. The burst end signal also returns to LOW at this point.

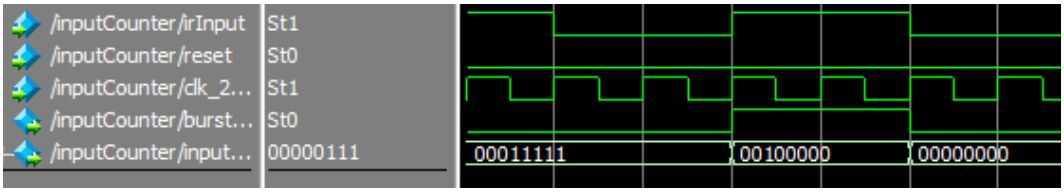

Figure 44: Input Counter Module Test

A SystemVerilog Files

A.1 NES Decoder

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

        always_ff@(posedge clk, negedge reset_n)
                if(!reset_n) count <= 4'b0000;
                else count <= count + 1;

endmodule
```

```
module NesDataDecoder(input logic data,
                                                        input logic reset_n,
                                                                input logic [3:0] current_state,
                                                                output logic [7:0] buttons);

        always_ff@(posedge current_state[0], negedge reset_n)
                if(!reset_n) buttons <= 8'b00000000;
                        else case(current_state[3:0])
                                4'b0001: buttons[7] <= data;     //a button
                                4'b0011: buttons[6] <= data;     //b button
                                4'b0101: buttons[5] <= data;     //select button
                                4'b0111: buttons[4] <= data;     //start button
                                4'b1001: buttons[3] <= data;     //up button
                                4'b1011: buttons[2] <= data;     //down button
                                4'b1101: buttons[1] <= data;     //left button
                                4'b1111: buttons[0] <= data;     //right button
                                default: buttons <= buttons;
                        endcase
endmodule
```

```
module NesClockState(input logic [3:0] current_state,
                                                        output logic clk);

        //generating clock
        always_comb
                case (current_state)
                        4'b0010: clk = 1;
                        4'b0100: clk = 1;
                        4'b0110: clk = 1;
                        4'b1000: clk = 1;
                        4'b1010: clk = 1;
                        4'b1100: clk = 1;
                        4'b1110: clk = 1;
                        default: clk = 0;
                endcase
endmodule
```

```
module NesLatchDecoder(input logic [3:0] current_state,
                                                                output logic Latch);

        always_comb
                case(current_state)
                        4'b0000: Latch = 1;
                        default: Latch = 0;
                endcase
endmodule
```

A.2 Push Counter

```
module counter #(parameter N = 14)
                                        (input logic clk,
                                         input logic reset_n,
                                         output logic [N-1:0] q);

        always_ff@(posedge clk, posedge reset_n)
                if(reset_n) q <= 0;
                else        q <= q + 1;
endmodule
```

```
module parser(input logic [13:0] data,
                                  output logic [3:0] S0, S1, S2, S3);

        always_comb
                begin
                        S0 = data % 10;                         //ones
                        S1 = (data / 10) % 10;                  //tens
                        S2 = (data / 100) % 10;                 //hundreds
                        S3 = (data / 1000) % 10;                //thousands
                end
endmodule
```

```
module sevenseg(input logic [3:0] data,
                                        output logic [6:0] segments);
        always_comb
                case(data)
                        //                                                      gfe_dcba
                        0:                      segments = 7'b100_0000;
                        1:                      segments = 7'b111_1001;
                        2:                      segments = 7'b010_0100;
                        3:                      segments = 7'b011_0000;
                        4:                      segments = 7'b001_1001;
                        5:                      segments = 7'b001_0010;
                        6:                      segments = 7'b000_0010;
                        7:                      segments = 7'b111_1000;
                        8:                      segments = 7'b000_0000;
                        9:                      segments = 7'b001_1000;
                        default: segments = 7'b000_0000;
                endcase
endmodule
```

```
module comparator #(parameter N = 14, parameter M = 9999)
                                           (input logic [N-1:0] a,
                                           output logic altb);

        always_comb
                altb = a > M;

endmodule
```

```
module sync(input logic clk,
                                input logic d,
                                output logic q);

        logic n1;

        always_ff@(posedge clk)
                begin
                        n1 <= d; //nonblocking
                        q <= n1; //nonblocking
                end
endmodule
```

A.3 Audio Multiplexer

```
module mux2_8(input logic [7:0] nes,vcr,
              input logic       switch,
              output logic [7:0] audio);


  assign audio = switch ? nes : vcr;
  endmodule
```

Figure 45: Multiplexer decoder block

A.4 RGB LED

```
1  module comparator_gt23(input logic[4:0] a,
2            output logic s);
3
4     assign s = (a > 5'b10111);
5
6  endmodule
7
```

Figure 46: SystemVerilog for the first Comparator Block

```
1  module comparator_gt9(input logic[3:0] a,
2            output logic s);
3
4     assign s = (a > 4'b1001);
5
6  endmodule
7
```

Figure 47: SystemVerilog for the second Comparator Block

```
1   module counter #(parameter N=7)
2             (input logic clk,
3             input logic reset,
4             output logic [N-1:0] q);
5
6      always_ff@(posedge clk, posedge reset)
7         if(reset) q <= 0;
8         else q <= q + 1;
9
10  endmodule
11
```

Figure 48: SystemVerilog for the Counter Module

```
 1  module decoder(input logic [7:0]data, output logic [23:0]color);
 2
 3      always_comb
 4          case(data)
 5          // GG RR BB
 6          8'd2  : color = 24'h34_eb_34; // IR up
 7          8'd3  : color = 24'h83_eb_34; // IR left
 8          8'd4  : color = 24'hc3_eb_34; // IR right
 9          8'd6  : color = 24'he5_eb_34; // IR guide
10          8'd7  : color = 24'hd5_eb_34; // IR power
11          8'd11 : color = 24'heb_ab_34; // IR down
12          8'd12 : color = 24'heb_89_34; // IR enter
13          8'd16 : color = 24'heb_34_55; // IR 1
14          8'd17 : color = 24'heb_34_ae; // IR 2
15          8'd18 : color = 24'heb_eb_34; // IR 3
16          8'd19 : color = 24'hc3_34_eb; // IR 4
17          8'd20 : color = 24'h89_34_eb; // IR 5
18          8'd21 : color = 24'h5e_34_eb; // IR 6
19          8'd22 : color = 24'h3d_34_eb; // IR 7
20          8'd23 : color = 24'h34_43_eb; // IR 8
21          8'd24 : color = 24'h34_67_eb; // IR 9
22          8'd25 : color = 24'h34_9e_eb; // IR 0
23          8'd29 : color = 24'h34_cc_eb; // IR channel +
24          8'd30 : color = 24'h34_eb_e8; // IR channel -
25          8'd47 : color = 24'h34_eb_ab; // IR A
26          8'd48 : color = 24'h34_eb_77; // IR B
27          8'd49 : color = 24'h34_eb_55; // IR C
28          8'd57 : color = 24'h00_80_00; // IR menu
29          default: color = 24'h00_00_00;
30          endcase
31
32  endmodule
33
```

Figure 49: SystemVerilog for the Decoder Module

```
 1  module parser #(parameter N=7)
 2            (  input logic [4:0] count,
 3               input logic [23:0] data,
 4               output logic q);
 5
 6     always_comb
 7        case(count)
 8        0 :   q = data[23];
 9        1 :   q = data[22];
10        2 :   q = data[21];
11        3 :   q = data[20];
12        4 :   q = data[19];
13        5 :   q = data[18];
14        6 :   q = data[17];
15        7 :   q = data[16];
16        8 :   q = data[15];
17        9 :   q = data[14];
18        10:   q = data[13];
19        11:   q = data[12];
20        12:   q = data[11];
21        13:   q = data[10];
22        14:   q = data[9];
23        15:   q = data[8];
24        16:   q = data[7];
25        17:   q = data[6];
26        18:   q = data[5];
27        19:   q = data[4];
28        20:   q = data[3];
29        21:   q = data[2];
30        22:   q = data[1];
31        23:   q = data[0];
32        default :   q = 0;
33        endcase
34
35  endmodule
36
```

Figure 50: SystemVerilog for the Parser Module

```
 1  module sync(input logic clk, d, output logic q);
 2
 3      logic n1;
 4
 5      always_ff@(posedge clk)
 6          begin
 7              n1 <= d;
 8              q <= n1;
 9          end
10  endmodule
11
```

Figure 51: SystemVerilog for the Synchronizer Module

A.5 IR Decoder

```
module addressflop(input logic clk, input logic reset, input logic [7:0]a, input logic d, output logic
    [7:0]q);
        always_ff@(posedge clk, posedge reset)
                begin
                 if(reset)
                        q <= 8'b0000_0000;
                 else
                  begin
                   case(a)
                        17: q[0] <= d;
                        18: q[1] <= d;
                        19: q[2] <= d;
                        20: q[3] <= d;
                        21: q[4] <= d;
                        22: q[5] <= d;
                        23: q[6] <= d;
                        24: q[7] <= d;
                        default: q <= q;
                   endcase
                  end
                end


endmodule
```

```
module comparator #(parameter N = 8) (input logic[N-1:0] a, b,
output logic eq, neq, lt, lte, gt, gte);

        assign eq = (a==b);
        assign neq = (a!=b);
        assign lt = (a<b);
        assign lte = (a<=b);
        assign gt = (a>b);
        assign gte = (a>=b);
endmodule
```

```
module counter#(parameter N=8)
(input logic clk, input logic reset, output logic [N-1:0] q);

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

```
module hardcode_start_bit(output logic[7:0] o1);
        always_comb
                begin
                        o1 = 8'b1010_0000;
                end
endmodule
```

```
module hardcodeinputcount(output logic [7:0]maxcount);
        always_comb
                begin
                        maxcount = 8'b0010_0000;
                        //this is 32, the # of expected rising edges in one burst
                end

endmodule
```

```
module sync(input logic clk, input logic d, output logic q);
        logic n1;

        always_ff@(posedge clk)
                begin
                        n1 <= d;
                        q <= n1;
                end
endmodule
```

B Simulation Files (Do scripts)

```
force reset 1 @ 0
force reset 0 @ 60

force irInput 1 @ 0
force irInput 0 @ 9000
force irInput 1 @ 13500

force irInput 0 @ 14625
force irInput 1 @ 15750
force irInput 0 @ 16875
force irInput 1 @ 18000
force irInput 0 @ 19125
force irInput 1 @ 21375
force irInput 0 @ 22500
force irInput 1 @ 24750
force irInput 0 @ 25875
force irInput 1 @ 27000
force irInput 0 @ 28125
force irInput 1 @ 30375
force irInput 0 @ 31500
force irInput 1 @ 33750
force irInput 0 @ 34875
force irInput 1 @ 37125

force irInput 0 @ 38250
force irInput 1 @ 40500
force irInput 0 @ 41625
force irInput 1 @ 43875
force irInput 0 @ 45000
force irInput 1 @ 46125
force irInput 0 @ 47250
force irInput 1 @ 48275
force irInput 0 @ 49500
```

```
34  force irInput 1 @ 51750
35  force irInput 0 @ 52875
36  force irInput 1 @ 54000
37  force irInput 0 @ 55125
38  force irInput 1 @ 56250
39  force irInput 0 @ 57375
40  force irInput 1 @ 58500
41
42  force irInput 0 @ 59625
43  force irInput 1 @ 60750
44  force irInput 0 @ 61875
45  force irInput 1 @ 63000
46  force irInput 0 @ 64125
47  force irInput 1 @ 65250
48  force irInput 0 @ 66375
49  force irInput 1 @ 68625
50  force irInput 0 @ 69750
51  force irInput 1 @ 72000
52  force irInput 0 @ 73125
53  force irInput 1 @ 74250
54  force irInput 0 @ 75375
55  force irInput 1 @ 77625
56  force irInput 0 @ 78750
57  force irInput 1 @ 79875
58
59  force irInput 0 @ 81000
60  force irInput 1 @ 83250
61  force irInput 0 @ 84375
62  force irInput 1 @ 86625
63  force irInput 0 @ 87750
64  force irInput 1 @ 90000
65  force irInput 0 @ 91125
66  force irInput 1 @ 92250
67  force irInput 0 @ 93375
68  force irInput 1 @ 94500
69  force irInput 0 @ 95625
70  force irInput 1 @ 97875
71  force irInput 0 @ 99000
72  force irInput 1 @ 100125
73  force irInput 0 @ 101250
74  force irInput 1 @ 103500
75
76  force irInput 0 @ 104625
77  run 110000
```

References

[1] D. Harris and S. Harris, *Digital Design and Computer Architecture*. Elsevier Science & Technology, 2012.

[2] J. Corleto, "Nes controller interface with an arduino uno." https://www.allaboutcircuits.com/projects/nes-controller-interface-with-an-arduino-uno/, 2016.

[3] Vishay, "Data formats for ir remote control." https://www.vishay.com/docs/80071/dataform.pdf, 2019.

[4] irdb, "Find ir codes." http://irdb.tk/find/, 2014.