ECE 271, Design Project

Nicholas Broce, Caden Friesen, Trevor Horine

March 4, 2021

Contents

1 Project Description

Inputs: The inputs for the entire project are clk50mhz, reset, nesdata, keyboarddata, clock-keyboard, resetkeyboard, and irdata. This project is split into three parts with three total input systems, those being the NES Controller, the PS/2 keyboard, and the IR remote signal, and each of those systems use the same 50MHz clock signal as a base input. This clock signal is either used as is or is altered with clock dividers in each of the sub-projects own files. Each of the three systems also share a common reset signal called reset. Nesdata and keyboard data operate very similarly to each other. Both are binary signals that are really a long signal of binary digits that denote a specific button or key depending on the controller inputted. The data going through the nesdata and keyboarddata inputs are incremented with their specific clock signals off a shift register inside the inputted system. For nesdata specially this is done when the output neslatch goes high and it driven by the output nesclk. For the keyboarddata input a similar latch like signal is handled inside the keyboard_press_driver and keyboard_inner_driver as seen in figures 55 and 57. As discussed already the clockkeyboard is a clock cycle coming directly from the PS/2 keyboard. Unlike the other two inputs irdata is not driven by a clock signal but it is still however a binary signal that is constantly changing in a pattern to denote specific infrared signals. The IRreader block then takes that stream a parses through it as it comes in to ensure it can deliver the correct outputs for the given input stream.

Outputs: From top to bottom the outputs for this entire project are nesclk, neslatch, outcw, outccw, leddata, audioout, dig0, dig1, dig2, dig3, seg1, seg2, seg3, seg4, seg5, seg6, seg7, and seg8. As previously discussed in the inputs section above newlatch and newclk help driver the inputted data from the NES controller. This is done when neslatch is high and the newclk cycles, these two in combination allows for nesdata to receive a string of binary data that denotes which button is pressed. Neslatch goes high periodically to ensure that any button press it captured properly. Outcw and outccw work very similarly to each other. Together the driver DC motor that acts as one of the outputs for the overall project. When either go high the DC motor receives power to turn in either a clockwise direction or a counter clockwise direction depending on which output goes high. Due to controller limitations it is not possible for both to go high at the same time. Leddata acts as another waveform binary signal to demote specific output colors. To create specific colors on the LEDs twenty four bits of data are passed through the output leddata at specific timing frequencies. Similarly to leddata audioout is a waveform created with 1's and 0's at specific frequencies to give an audio device musical notes based on the key pressed on the PS/2 keyboard. Dig0 through dig3 are collections of 7 LEDs built into the FPGA that make up the 7-segment display units. Depending on the inputs given different LEDs are lit up to showcase the address and command code of the IR signal provided. Seg1 through seg 8 are more LEDs on the 7-segment display units that are used for error checking. When something goes wrong or an incorrect IR signal is taken through the irdata input, two of the 7-segment displays at the left end light with with a NE for "not equal".

The design is shown at figure 1 on the following page.

Top Level

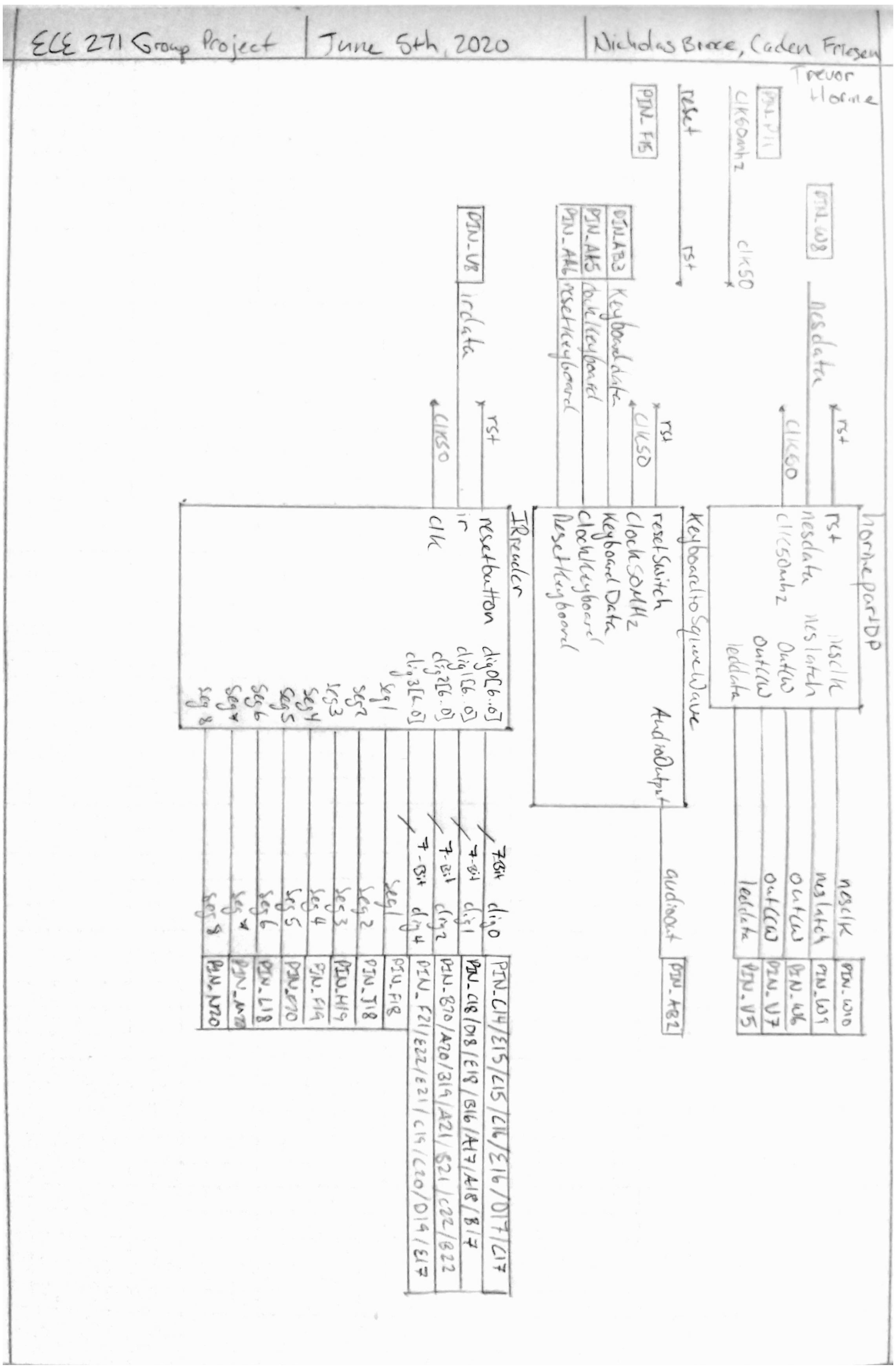
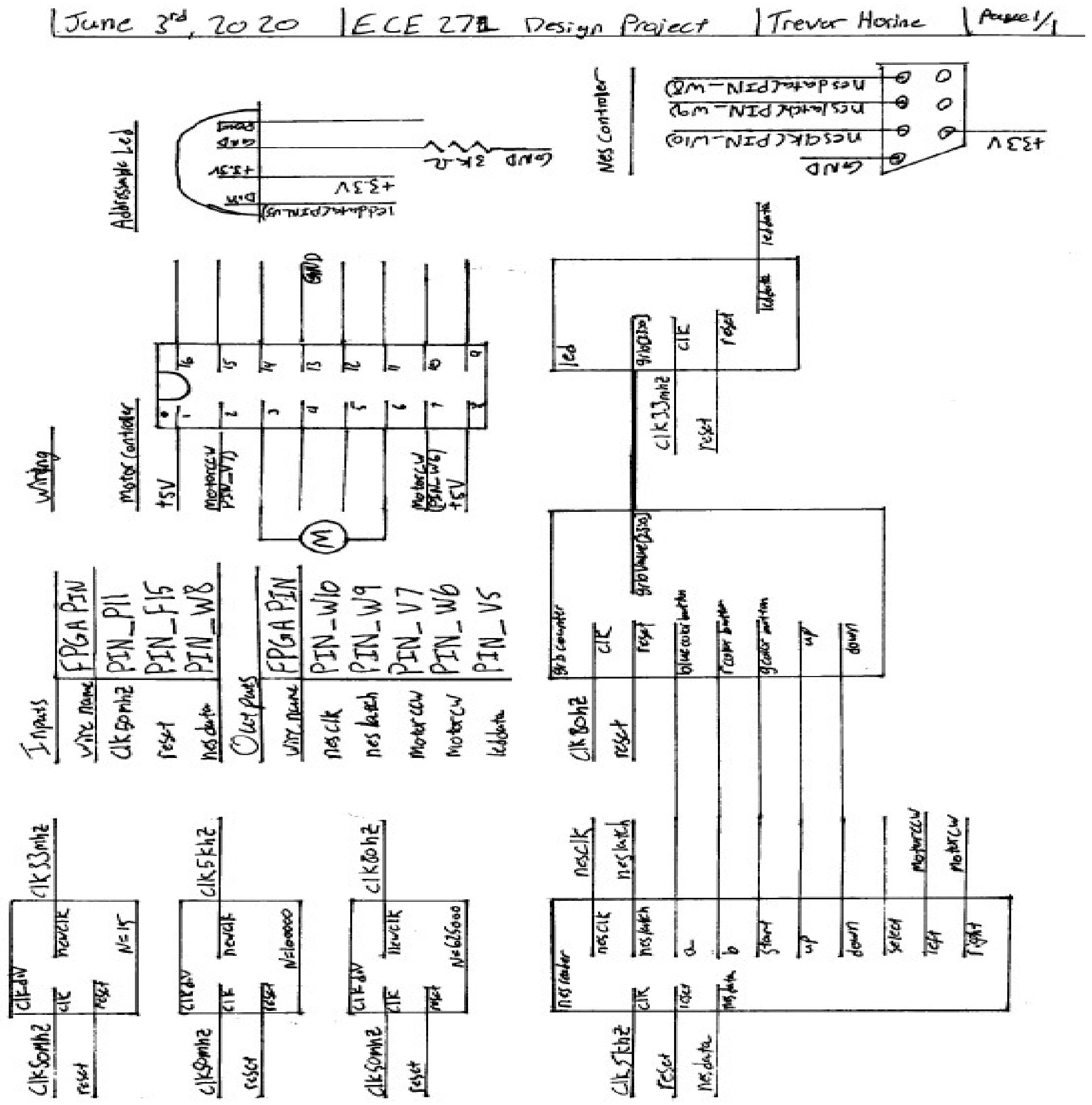


Figure 1

2 NES Controller Input - Motor and Addressable LED Output

This section was implemented with a NES controller, addressable LED, and DC motor. The wiring diagrams can be found in figure 59 and the video of it in action can be found at the following link, https://youtu.be/PWSKQtm6lo0. Inputs: This part of the design project reads inputs from a NES controller and uses a switch and the 50MHz clock from the DE10-Lite FPGA.

Outputs: This module allows the user to control a motor and have it rotate clockwise when the right button on the d-pad of the NES controller is pressed and counter clockwise when the left button of the controller is pressed. In addition the addressable LED's RGB value can be controlled to a level of 0 to 255 for each color, red, green and blue. In order to increase a colors value hold that colors button, A for blue, B for red, and START for green, and use the up button on the d-pad. to decrease the color value for a color follow the same process but use the down button instead. The color values do roll over the top and bottom so the user can go from 0 to 255 by going down and from 255 to 0 by going up.



Inputs

wire name	FPGA PIN
clk50mhz	PIN_P11
reset	PIN_F15
nesdata	PIN_W8

Outputs

wire name	FPGA PIN
nesclk	PIN_W10
neslatch	PIN_W9
motorccw	PIN_V7
motorcw	PIN_W6
leddata	PIN_V5

Figure 2: The top level design for the NES controller to addressable LED and motor.

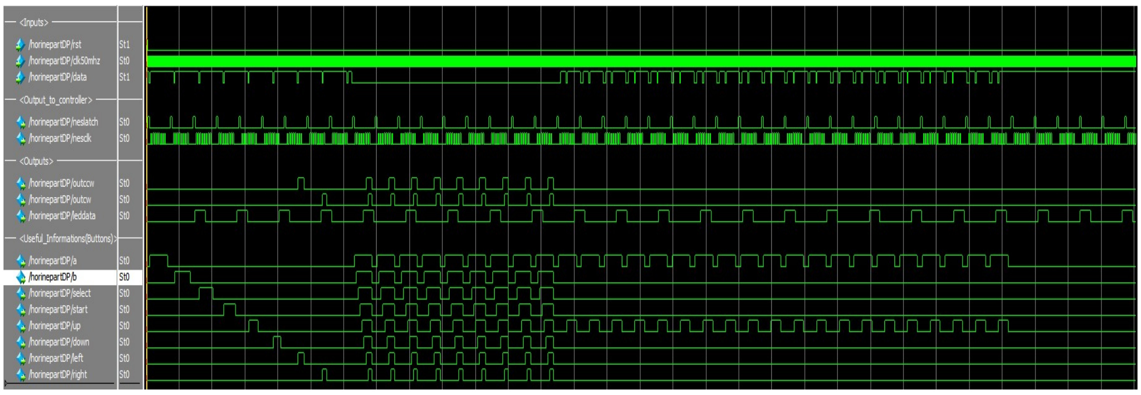


Figure 3: The top level simulation for the NES controller to addressable LED and motor.

2.1 Clock Divider

Inputs: This block takes in a clock signal, 50MHz clock was used in the is project but this block will work with any. This block also has a active high reset. In addition this block has a parameter N.

Outputs: This block counts positive edges of the input clock until the counter matches the parameter N, then toggles the output clock. parameters N = 15, 100000, and 625000 are used in this sections of the project to obtain 3.3MHz, 5KHz, and 80Hz output clock signals respectively.

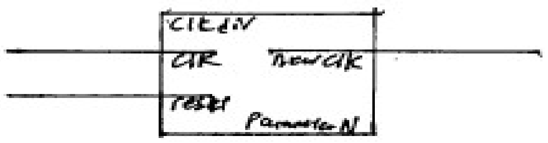


Figure 4: This is an block diagram for the clock divider.

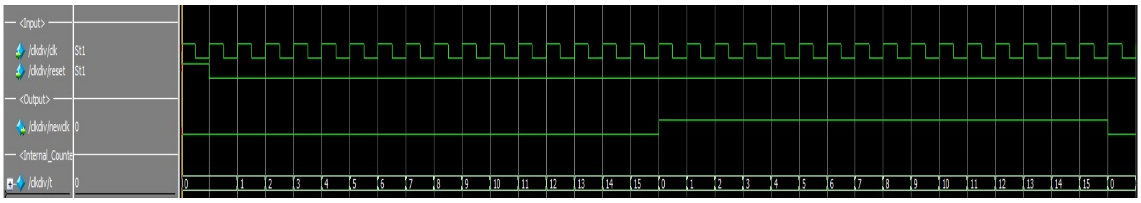


Figure 5: This is an block diagram for the clock divider.

2.2 NES Reader

Inputs: This block takes in a clock signal, 5KHz clock was used in the is project, data from the NES controller, and a active high reset.

Outputs: This block reads the serial data pin from the NES controller and outputs all the buttons separately. This block also uses the input clock to drive the NES clock and latch pin. The NES latch pin tells the controller to load the current state of the buttons in to the controllers shift register, and NES clock shifts the bits out of the shift register through the NES data pin.

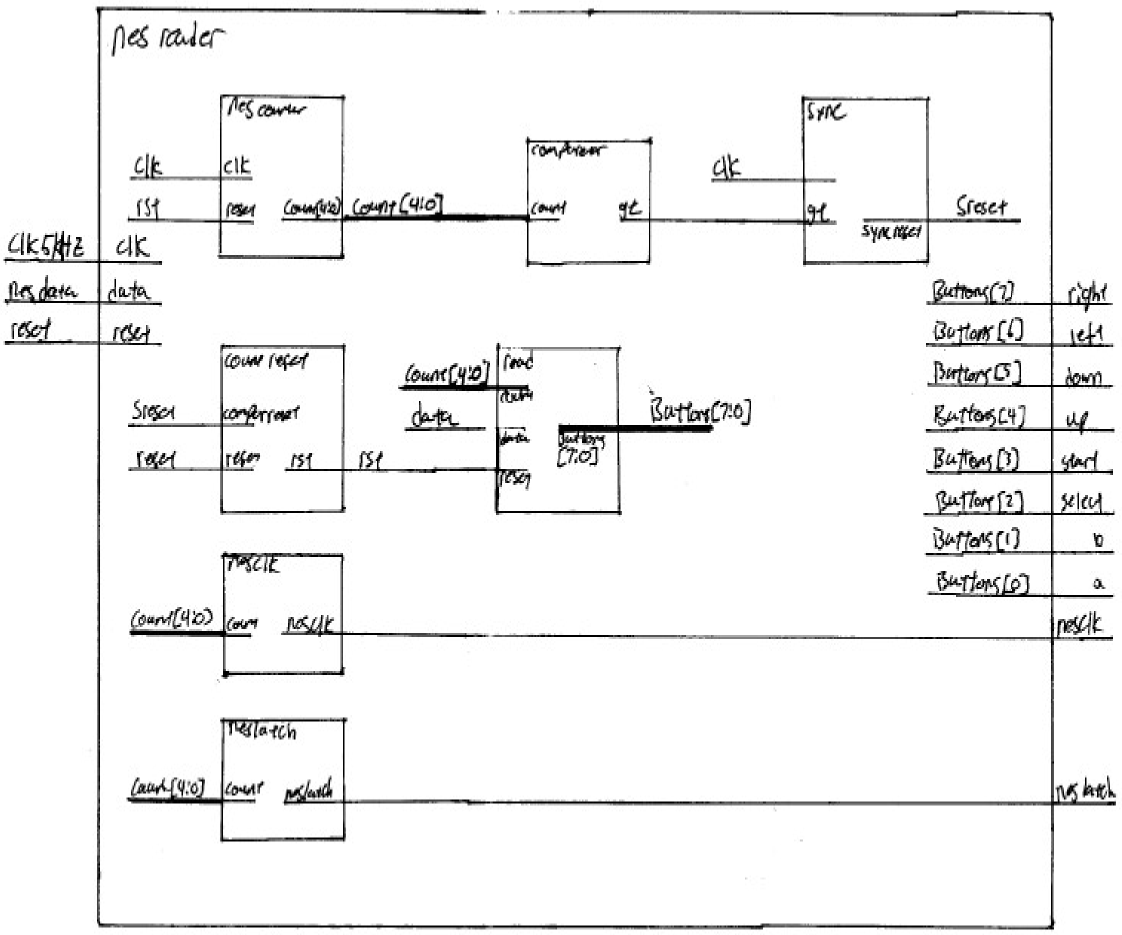


Figure 6: This is an block diagram for the NES reader.

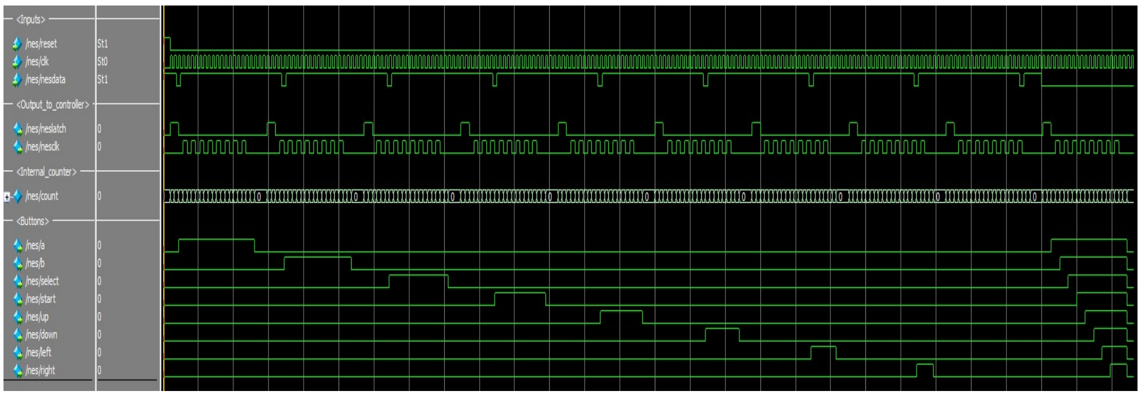


Figure 7: This is the simulation for the NES reader.

2.2.1 NES Counter

The individual block shown in figure 8 uses the input clock, this project a 5KHz clock was used, to create a count that will be used to control the timing of thing with in the NES reader block.

Inputs: This block has two inputs, the 5KHz clock signal used in the NES reader and the active high reset signal used with the NES reader.

Outputs: This block has one output called count, a 5 bit signal that counts up and is used in the timing of the NES reader block.

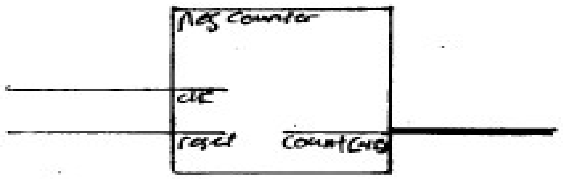


Figure 8: This block is a simple counter that counts up and is used in timing of the NES reader block.

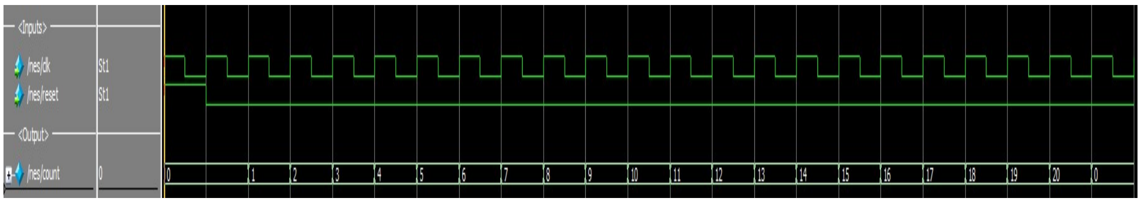


Figure 9: This is the simulation for a simple counter that counts up and is used in timing of the NES reader block.

2.2.2 Comparator

The individual block shown in figure 10 has one input,one output, and two parameters, N the number of bits in count and M the number count is being compared to.

Inputs: The input is the count from the counter in the nescounter block.

Outputs: The output is one if the input is greater then or equal to the second parameter M, and zero if it is not.

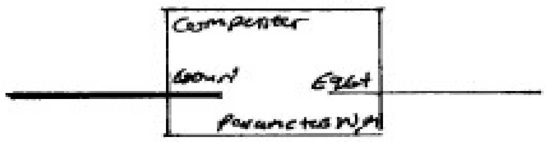


Figure 10: This block is a simple greater then or equal to comparator for the count from the nescounter block.

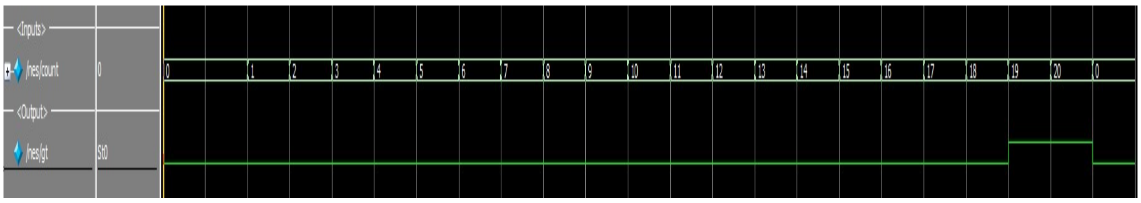


Figure 11: This is the simulation for a simple greater then or equal to comparator for the count from the nescounter block.

2.2.3 Synchronizer

The individual block shown in figure 12 has two input the 5KHz clock and the output from the compartator. The output is a synchronized signal used as a reset for the counter.

Inputs: The inputs are the 5KHz clock and the output from the comparator block used to tell if the counter has reached a certain number.

Outputs: The output a synchronized signal that is used to reset the counter at the desired values that was specified by the M parameter in the comaprator.

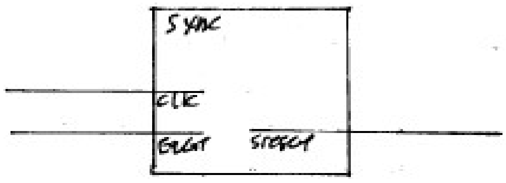


Figure 12: This block is a simple syncronizer used in the reset of the counter.

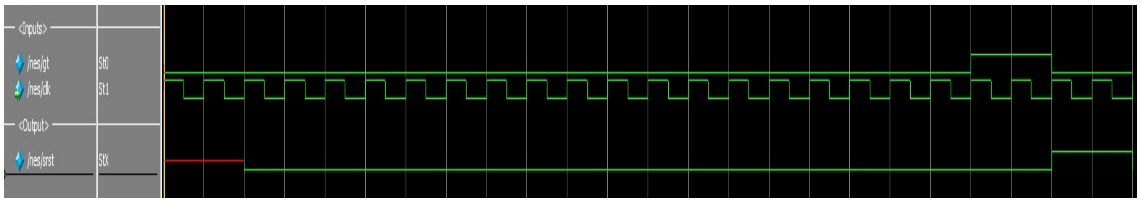


Figure 13: This is the simulation for a simple syncronizer used in the reset of the counter.

2.2.4 Countreset

The individual block shown in figure 14 has two input and the one output, this block is essentially an OR gate.

Inputs: The inputs are the reset signal that is an input for the nesreader block and the output of the syncronizer.

Outputs: The output is a reset signal that is high where either input is high so the counter resets when the overall project reset is high or when it reaches its desired maximum set in the comaprator.

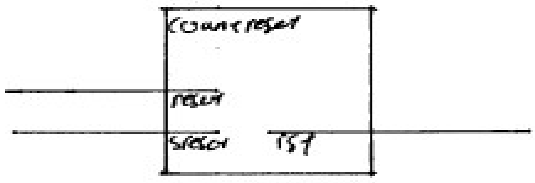


Figure 14: This block is simpily an OR gate used to reset the counter.

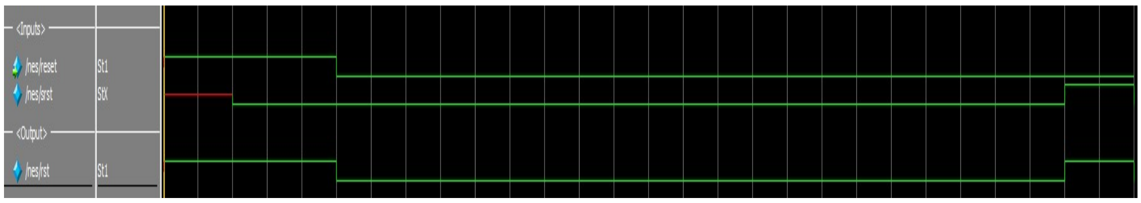


Figure 15: This is the simulation for countreset which is effectively an OR gate used to reset the counter.

2.2.5 Neslatch

The individual block shown in figure 16 has one input, count and one output neslatch. This block controls the latch signal that goes to the NES controller to load the data for the buttons in to the shift register in the controller.

Inputs: The input for this block is the 5 bit count from the nescounter.

Outputs: The output is a single bit latch signal that goes high to load data in to the shift register on the controller.

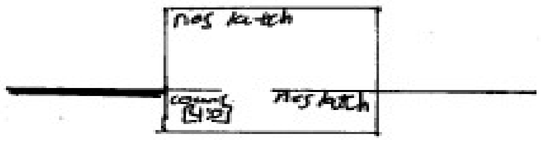


Figure 16: This block uses the counter for timing and outputs a high signal during certain sections of the count to load data in to the shift register of the controller.

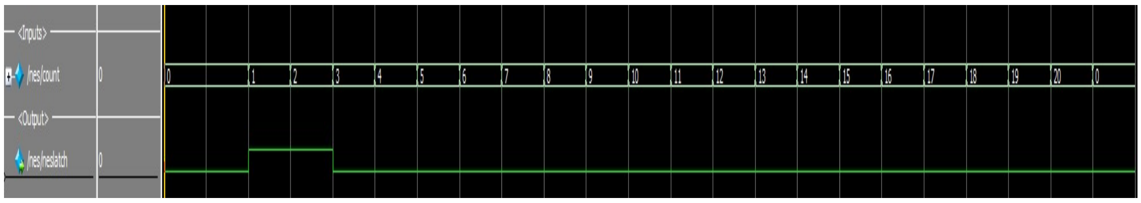


Figure 17: This is the simulation for neslatch module.

2.2.6 Nesclk

The individual block shown in figure 18 has one input, count and one output nesclk. This block controls the clock signal that goes to the NES controller to shift the data for the buttons out of the shift register in the controller.

Inputs: The input for this block is the 5 bit count from the nescounter.

Outputs: The output is a single bit clock signal that goes high to shift data out through the data pin on the controller.

nes clk
count [4:0]
nesclk

Figure 18: This block uses the counter for timing and outputs a high signal during certain sections of the count to shift data out though the the data pin of the controller.

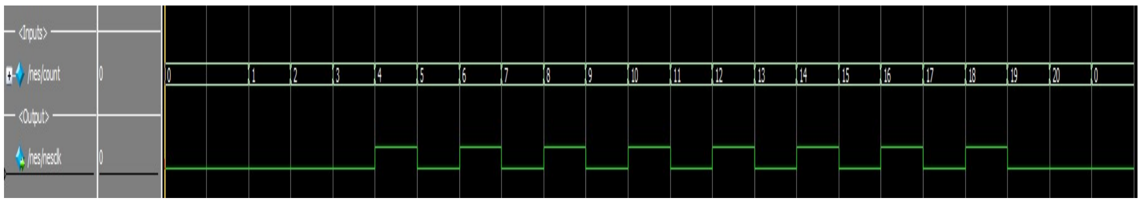


Figure 19: This is the simulation for nesclk module.

2.2.7 Read

The individual block shown in figure 20 has three inputs, data, reset, and count. This block also has one output that is a bus containing all the information for the buttons that gets broken up to the various outputs of the nesreader.

Inputs: The inputs for this block is the 5 bit count from the nescounter, the data from the controller, and the overall project reset.

Outputs: The output is a eight bit signal that has the state of each button in a different bit. The out put is then split in to each of the eight output of nesreader for the different buttons.

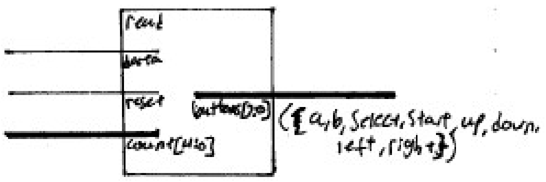


Figure 20: This block uses the counter for timing, and the data from the controller to output the state of each button.

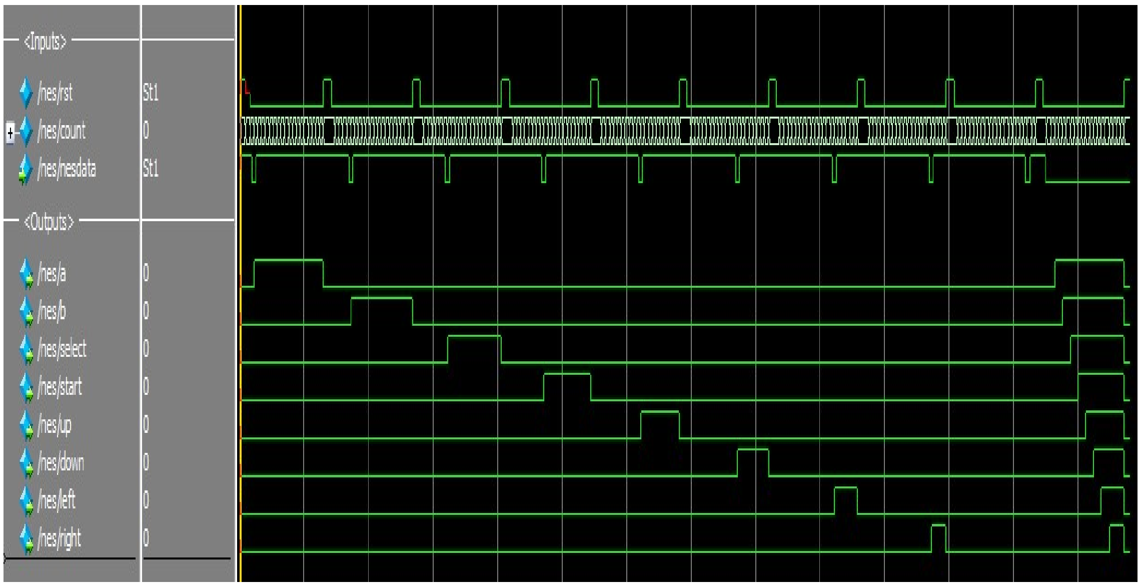


Figure 21: This is the simulation for the read module.

2.3 Grbcounter

Inputs: This block takes in a clock signal, active high reset, up button, down button, and a button for each color red, green, and blue. When the up button and a color button are pressed at the same time that colors value from 0 to 255 is increased, when down and that colors button is pressed the value is decreased.

Outputs: the out put of this block is a 24 bit signal that contains the 0 to 255 value for green in the first eight bits, the value for red in the next eight bits, and the value for blue in the last eight bits.

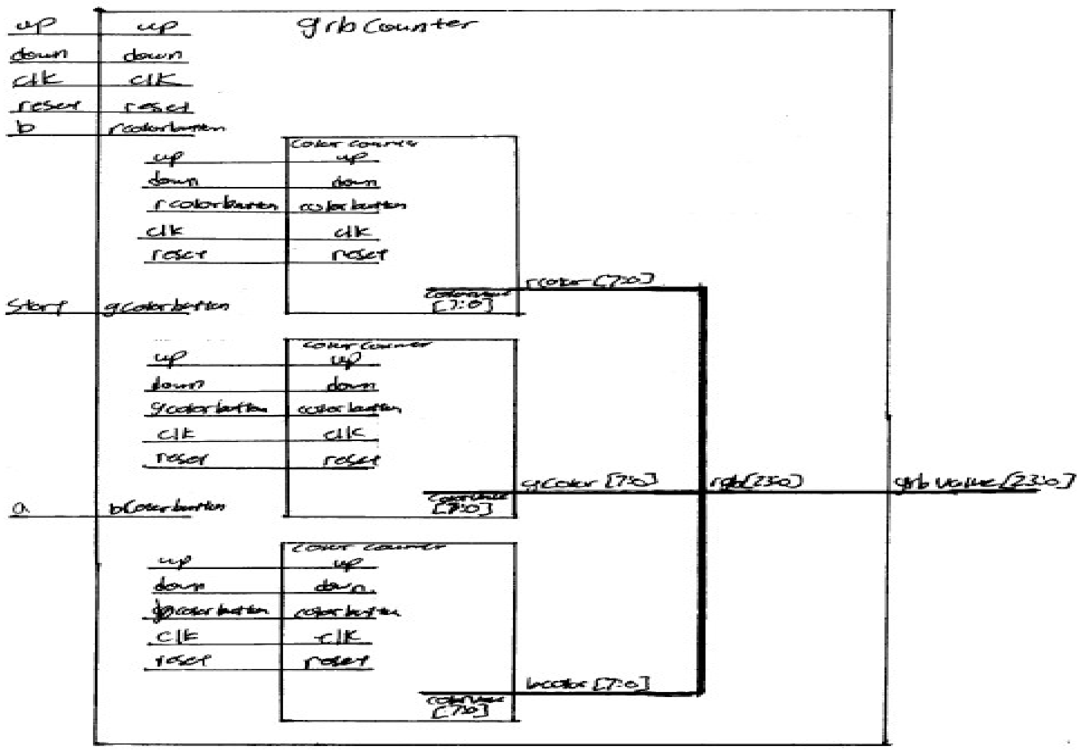


Figure 22: This is an block diagram for the grbcounter block.

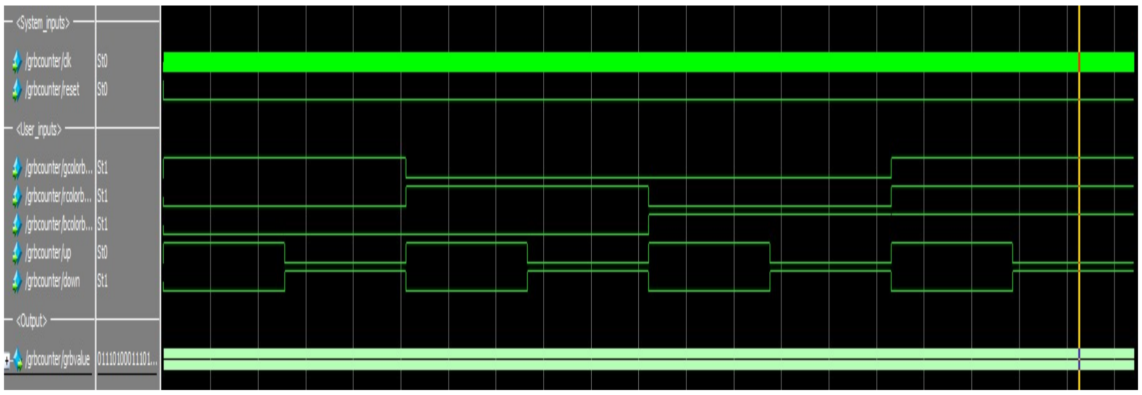

<System_inputs>
/grbcounter/clk St0
/grbcounter/reset St0
<User_inputs>
/grbcounter/gcolorb... St1
/grbcounter/rcolorb... St1
/grbcounter/bcolorb... St1
/grbcounter/up St0
/grbcounter/down St1
<Output>
/grbcounter/grbvalue 01110100011101...

Figure 23: This is the simulation for the grbcounter block.

2.3.1 Colorcounter

The individual block shown in figure **??** has five inputs, a clock signal, a active high reset, an up, a down, and a button signal. This block also has one output that is a eight bit bus containing the count of the counter for that color.

Inputs: The inputs for this block are the 80Hz clock signal provided to the grbcounter block, the active high reset of the overall project, a signal from the up button on the controller, a signal from the down button on the controller, and a signal from the button assigned to represent that color on the controller.

Outputs: The output is a eight bit signal that contains the count of the counter that represents the value of that color on a scale from 0 to 255.

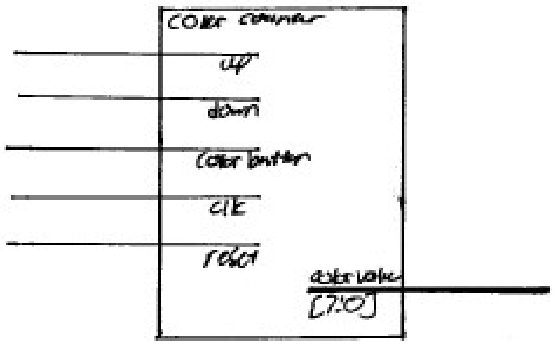


Figure 24: This block takes in a clock and reset to output the count used for timing in the led block.

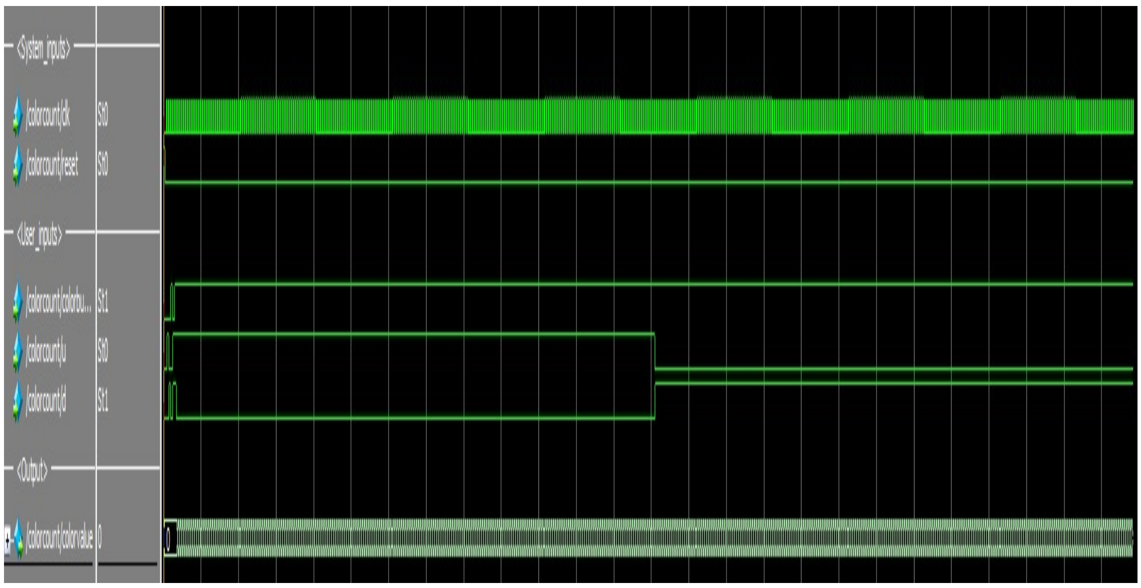


Figure 25: This is the simulation for the counter module.

2.4 Led

Inputs: This block takes in a clock signal, 3.3MHz clock was used in the is project, a 24 bit rgb color values and, a active high reset.

Outputs: This block takes in the 24 bit rgb color value and converts it to as single bit serial output to go to an RGB addressable LED, the LED reads a 0 as high for .3 microseconds followed by low for .9 microseconds and reads a 1 as high for .6 microseconds followed by low for .6 microseconds.

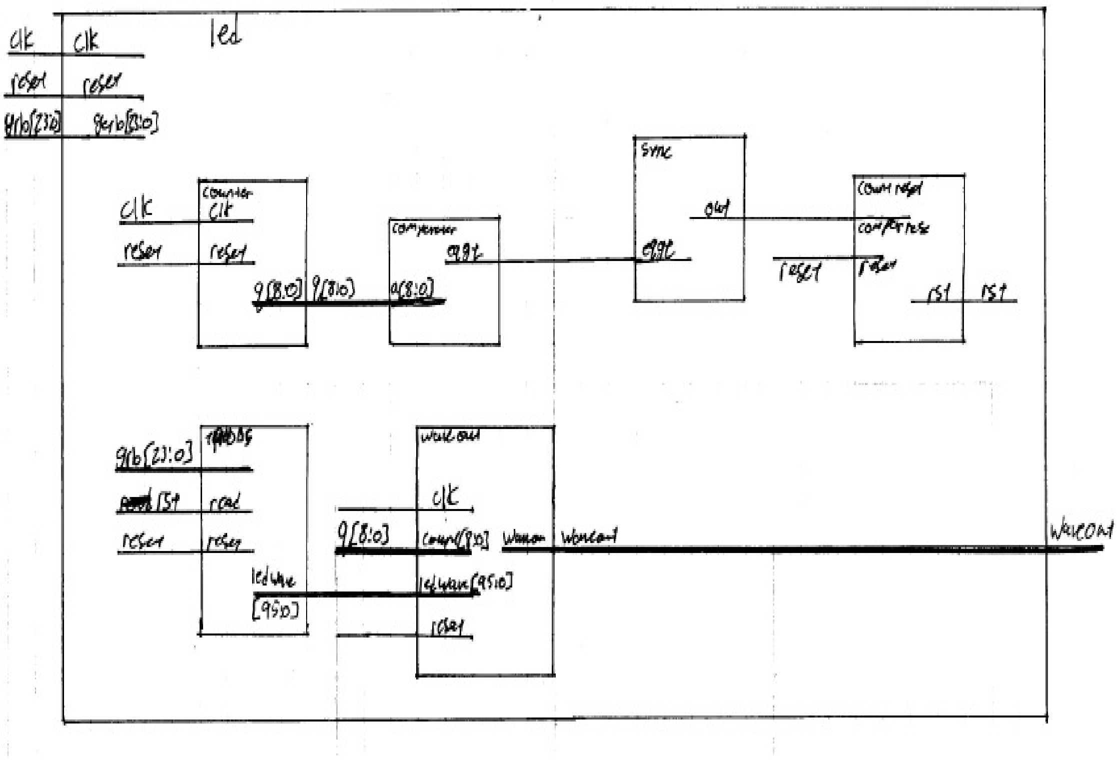


Figure 26: This is an block diagram for the LED block.

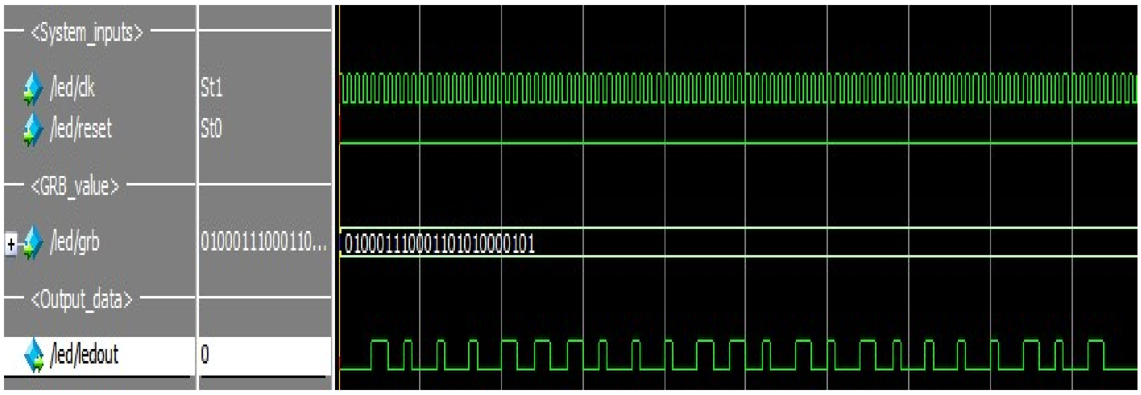


Figure 27: This is the simulation for the led block.

2.4.1 Counter

The individual block shown in figure 28 has two inputs, a clock signal and a active high reset. This block also has one output that is a nine bit bus containing the count of the counter.

Inputs: The inputs for this block are the 3.3MHz clock signal provided to the led block, and the active high reset of the overall project.

Outputs: The output is a nine bit signal that contains the count of the counter that is used for timing in the led block.

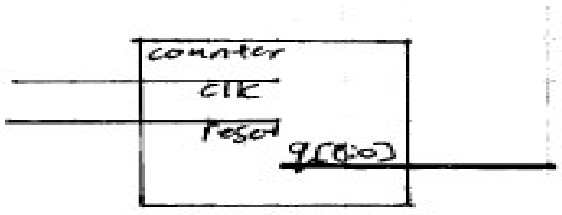


Figure 28: This block takes in a clock and reset to output the count used for timing in the led block.

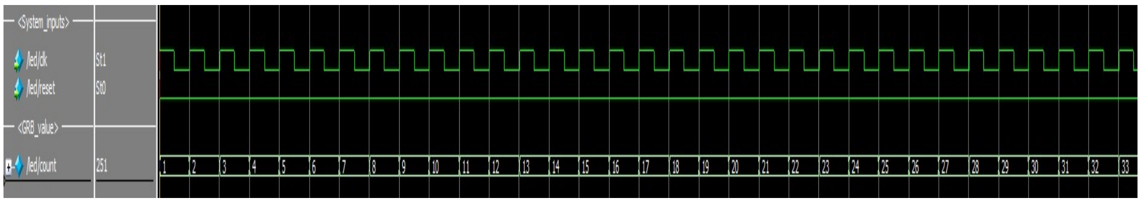


Figure 29: This is the simulation for the counter module.

2.4.2 comparator

This is the same comparator used in the nes reader, please see section 2.2.2.

2.4.3 syncronizer

This is the same syncronizer used in the nes reader, please see section 2.2.3

2.4.4 countreset

This is the same countreset used in the nes reader, please see section 2.2.4

2.4.5 Twenty Four Bit to Ninety Six Bit

The individual block shown in figure 30 has three inputs, grb a 24 bit signal, read, reset. This block also has one output that is a 96 bit bus called ledwave and will be used to output the right serial signal for the addressable LED to know what color is being passed to it.

Inputs: The inputs for this block is the 24 bit values containing the 0 to 255 color values for green in the first eight bits, the 0 to 255 values for red in the next eight bits, and the 0 to 255 value for blue in the last eight bits. read is a signal that tells the module to grab the grb values and convert it to the 96 bit signal, and reset is an active high rest.

Outputs: The output is a 96 bit signal that is used to get the right serial output fed to the led so that it recognizes the string of bit s being passes using the format described in the output section of the led block.

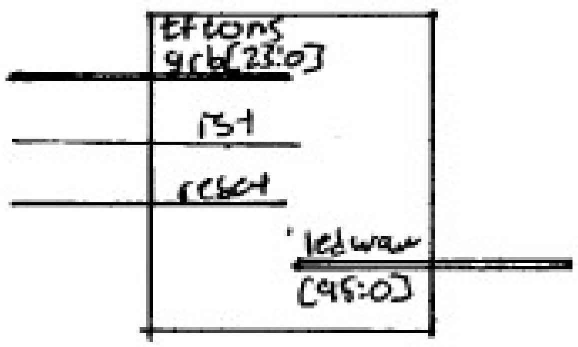


Figure 30: This block converts the 24 bit signal to a 96 bit signal used in the serial output of the rgb color to the addressable led.

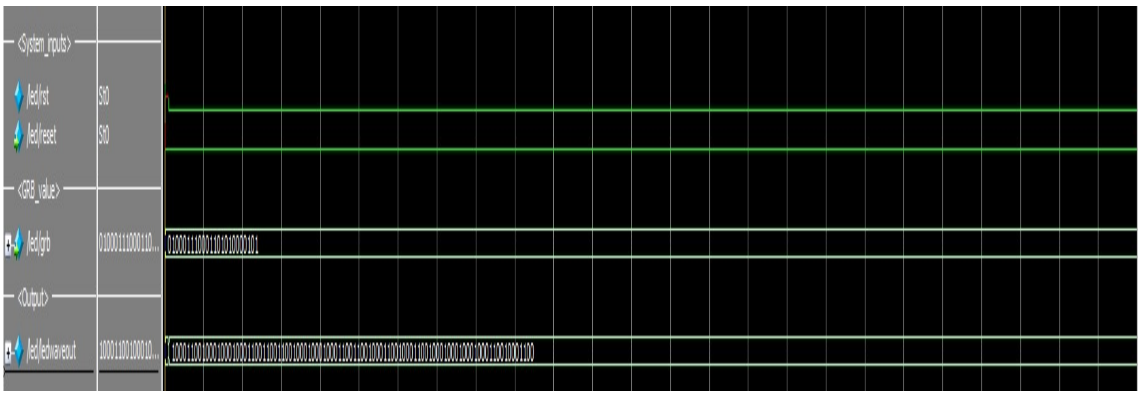


Figure 31: This is the simulation for the tftons module.

2.4.6 waveout

The individual block shown in figure 32 has four inputs, a clock, a reset, count, and ledwave the 96 bit signal. This block also has one output, waveout that is the serial output to the addressable LED.

Inputs: The clock signal is the 3.3MHz clock passed in to the led module. The reset signal is the overall project reset, the count is the count from the counter in the larger led module. Lastly the 96 bit signal is used to output a serial signal that addressable LED will recognize.

Outputs: The output is a single bit serial output called waveout and outputs the rgb color for the addressable LED to show in a format that the addressable LED can recognize.

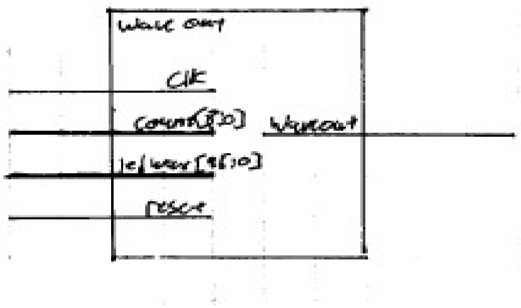


Figure 32: This block uses the counter for timing, and the data from the controller to output the state of each button.

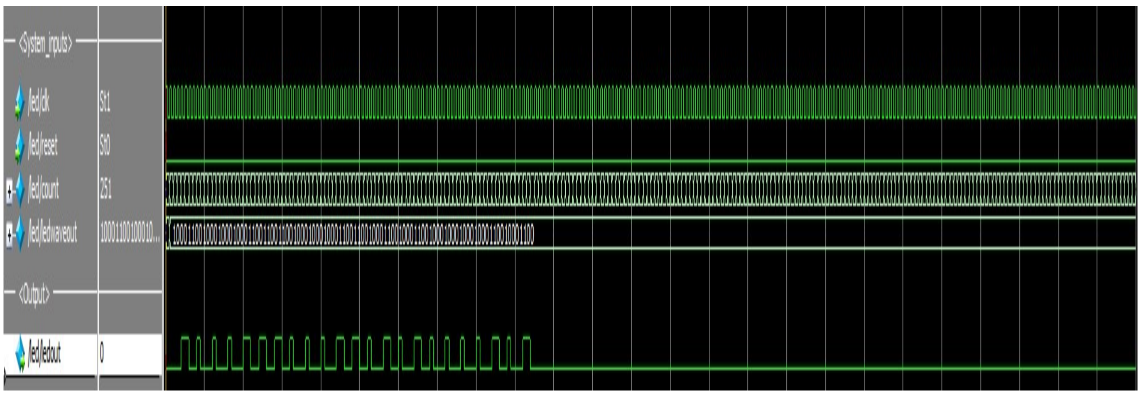
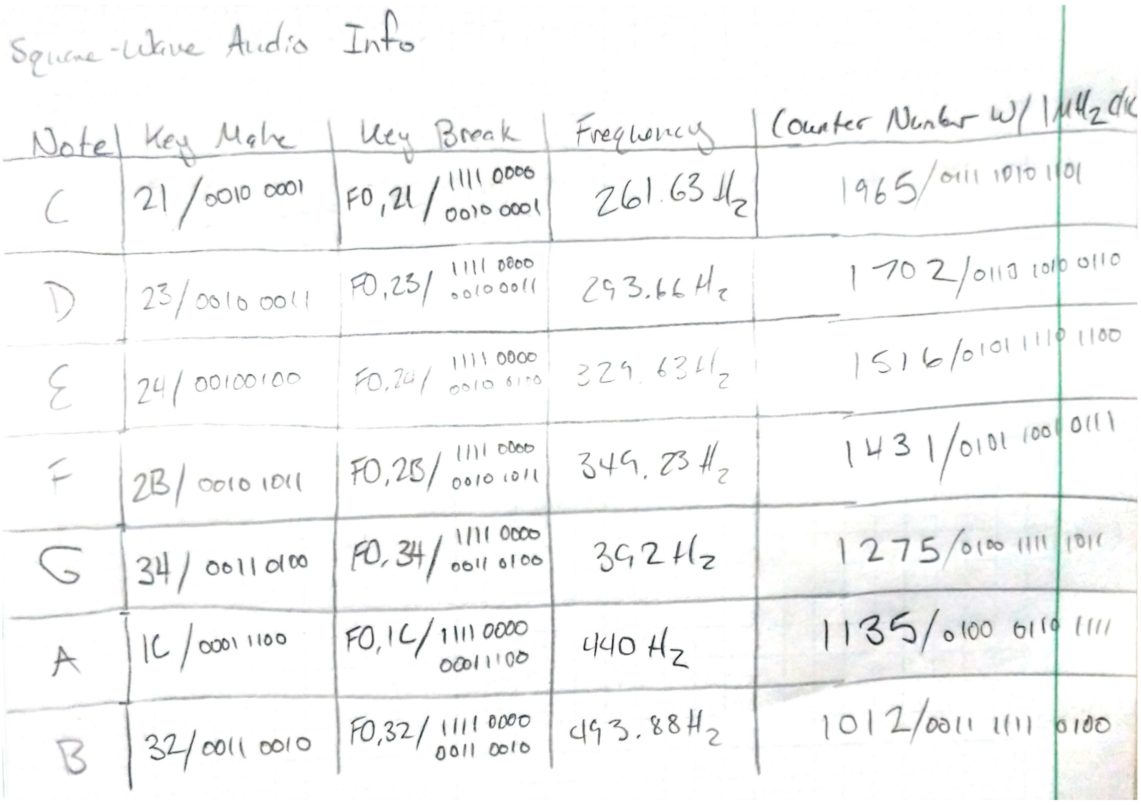


Figure 33: This is the simulation for the read module.

3 PS/2 Keyboard Input - Square Wave Audio Output

Inputs: This reads in a PS/2 Keyboard which inputs it's data with the Data, ClockKeyboard, ResetKeyboard, and Clock50MHz inputs. There is also a resetSwitch input for the SquareWave-Output component that is separate from the ResetKeyboard input.

Outputs: This outputs a Square wave Audio signal that is either a 1 or 0 to create waves as specified frequencies depending on the keyboard button pressed. Figure 34 details the specific frequencies programmed into this design. For example, when the C key is pressed the output will create a sound wave frequency for middle C at 261.23 Hz.



Square-Wave Audio Info

Note	Key Make	Key Break	Frequency	Counter Number W/ 1MHz clk
C	21 / 0010 0001	F0,21 / 1111 0000 0010 0001	261.63 Hz	1965/0111 1010 1101
D	23/ 0010 0011	F0,23/ 1111 0000 0010 0011	293.66 Hz	1702/0110 1010 0110
E	24/ 00100100	F0,24/ 1111 0000 0010 0100	329.63 Hz	1516/0101 1110 1100
F	2B/ 0010 1011	F0,2B/ 1111 0000 0010 1011	349.23 Hz	1431/0101 1001 0111
G	34/ 0011 0100	F0,34/ 1111 0000 0011 0100	392 Hz	1275/0100 1111 1011
A	1C / 0001 1100	F0,1C/ 1111 0000 00011100	440 Hz	1135/0100 0110 1111
B	32/ 0011 0010	F0,32/ 1111 0000 0011 0010	493.88 Hz	1012/0011 1111 0100

Figure 34: The table created to program the different keys on the PS/2 keyboard to match the given frequencies for those notes. If any other key is pressed the program will treat it as no given input and will only recognize these specific keys with their given key make and key break codes.

Design for Full Keyboard to Square Wave Audio Output Circuit :

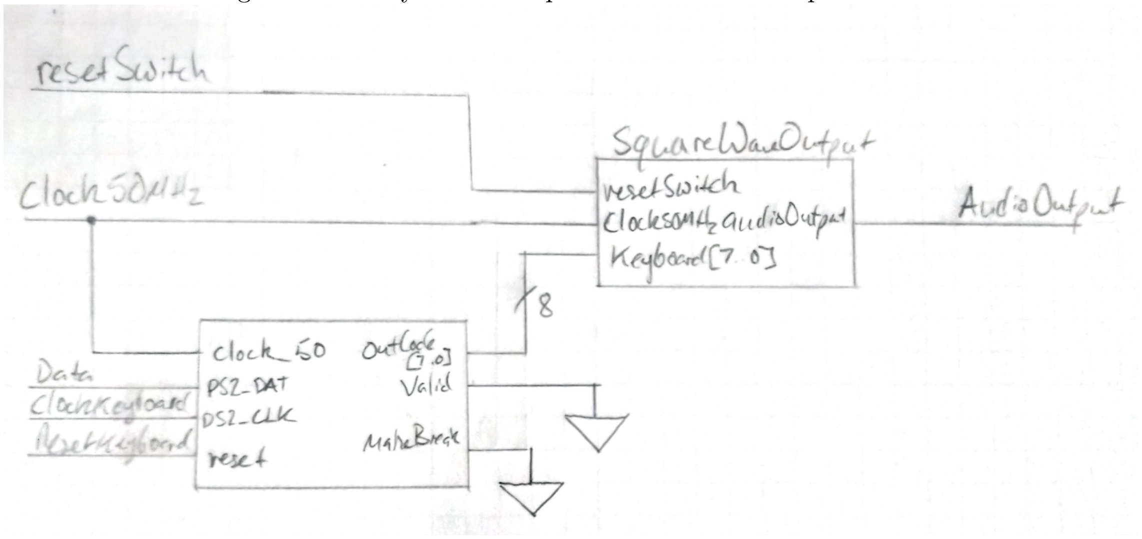
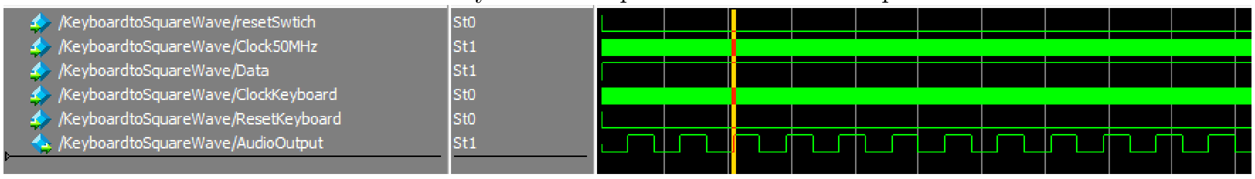


Figure 35: This is the top level circuit design for the PS/2 Keyboard to Square Wave Audio portion of our design. It showcases how the two Functional Units connect to form a bridge between the keyboard input processing and the sound wave forming process.

Simulation Results for Full Keyboard to Square Wave Audio Output Circuit:



```
/KeyboardtoSquareWave/resetSwtich      St0
/KeyboardtoSquareWave/Clock50MHz       St1
/KeyboardtoSquareWave/Data             St1
/KeyboardtoSquareWave/ClockKeyboard    St0
/KeyboardtoSquareWave/ResetKeyboard    St0
/KeyboardtoSquareWave/AudioOutput      St1
```

Figure 36: This is the simulation results of the PS/2 Keyboard to Square Wave Audio portion of our design.This showcases the middle C frequency that would be output when the C key is pressed on the keyboard.

3.1 Functional Unit 1 - SquareWaveOutput

Inputs: The inputs to this block are Clock50MHz, resetSwitch, and eight-bit Keyboard inputs. These drive the comparators, clock dividers, counters, D Flip-flops, and synchronizes in this circuit.

Outputs: There is only one output and it a binary signal the acts as a square wave audio wave, alternating on different frequency depending on what eight-bit make code is taken in from a PS/2 keyboard.

Expanded Design for the SquareWaveOutput component Circuit :

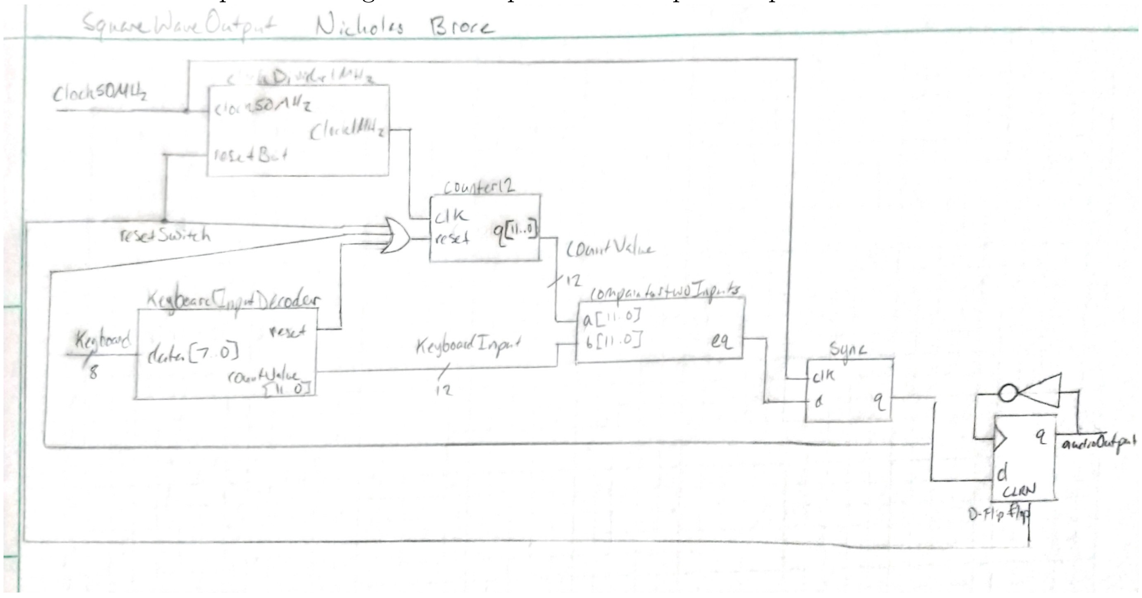
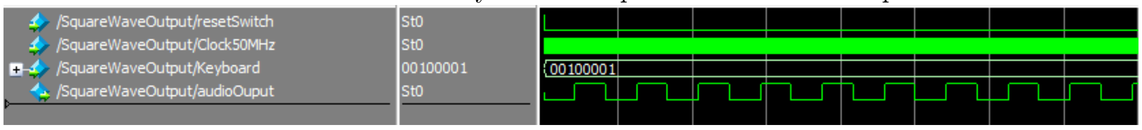


Figure 37: This is the expanded view of the portion shown in the full circuit schematic shown in figure 35 to showcase the processing of the square waves formed by the 8-bit keyboard input.

Simulation Results for Full Keyboard to Square Wave Audio Output Circuit:



```
/SquareWaveOutput/resetSwitch   St0
/SquareWaveOutput/Clock50MHz    St0
/SquareWaveOutput/Keyboard      00100001    00100001
/SquareWaveOutput/audioOuput    St0
```

Figure 38: This figure shows the ModelSim results with the C key keyboard make code given. The audio output is outputting at the correct frequency for the musical note C.

3.1.1 Individual Block 1 - ClockDivider1MHz

Inputs: The inputs for this block are Clock50MHz and a resetBut. Clock50MHz is the FPGA standard 50MHz frequency input and the resetBut is a active high signal to reset the counter and testing blocks that have flip-flops built into their designs as shown later in figure 41 and figure 47 respectively.

Outputs: The output is a clock signal of 1MHz as shown by the named output Clock1MHz. This allows the rest of the circuit to operate with the correct timing.

Expanded Design for the ClockDivider1MHZ Circuit :

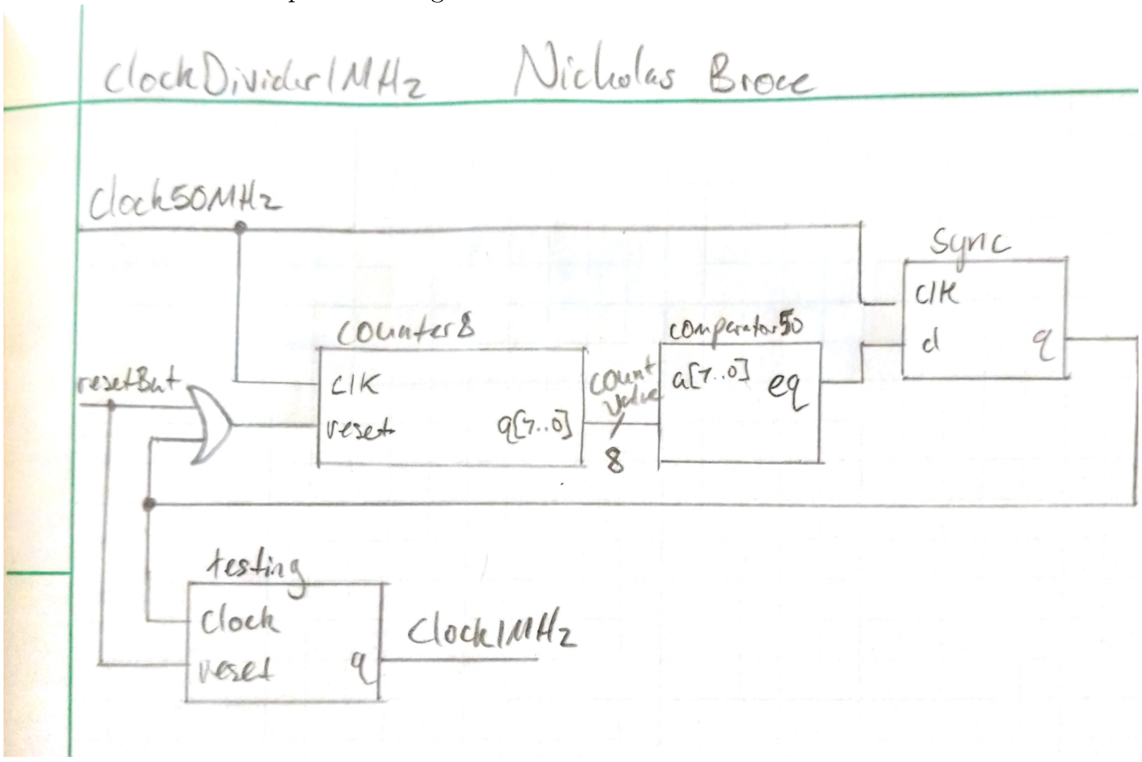


Figure 39: This is the expanded view of a block shown in figure 37. This is a key part of the audio portion of this circuit.

Simulation Results for the ClockDivider1MHZ Circuit :

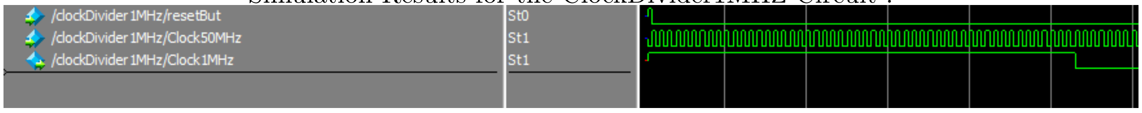

/clockDivider1MHz/resetBut St0
/clockDivider1MHz/Clock50MHz St1
/clockDivider1MHz/Clock1MHz St1

Figure 40: This is a simulation showcasing the slowed down clock cycle outputted compared to the fast clock cycle inputted.

3.1.2 Individual Block 2 - counter8

Inputs: This block takes in a clock cycles called clk, and a binary reset signal to initialize the counter.

Outputs: The output is a eight-bit binary output that increments by one every clock cycle. It begins at 0 until the first positive edge of the clk input.

Block Diagram Design for a basic 8-bit counter

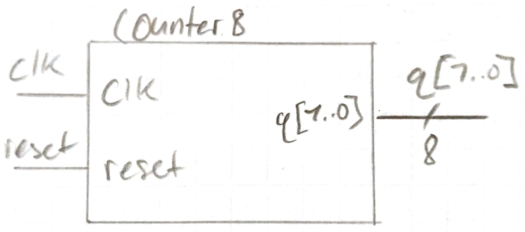


Figure 41: This is a 8-bit counter that takes in a clock cycle to increase the output q by one every clock cycle.

Simulation Results for a basic 8-bit counter

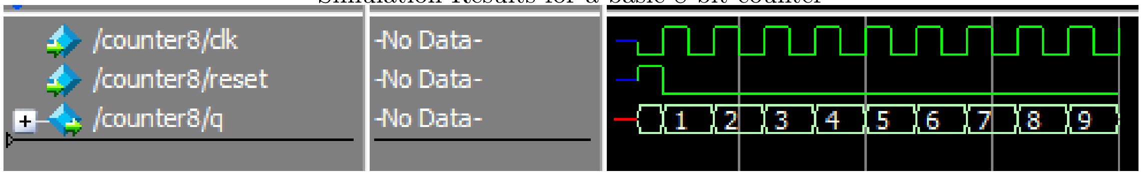


Figure 42: The simulation results showcase the incremental nature of this counter, increasing the output amount by one for every clock cycle of the input clock.

3.1.3 Individual Block 3 - comparator10

Inputs: This block takes in a eight-bit binary value and compares it to a parameter binary value. In this case that parameter is the decimal number 10.

Outputs: The output to this block is a binary signal, 1 if the incoming eight-bit value is equal to the parameter value and 0 if otherwise.

Block Diagram Design for a comparator with the parameter value of 10.

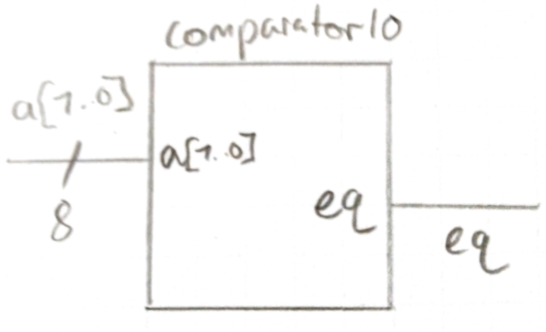


Figure 43: This comparator takes in a value tests it to see if it is equal to a built in parameter to send out a binary signal.

Simulation Results for a 8-bit comparator with the parameter value of 10.

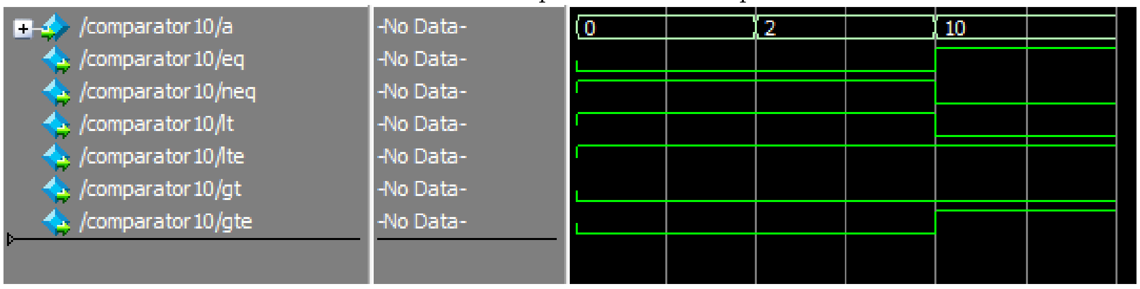


Figure 44: This simulation result shows no only the equal to operator but no equal to, less than, less than and equal to, greater than, and greater than and equal to the parameter value.

3.1.4 Individual Block 4 - sync

Inputs: The sync block takes in a clock signal called clk and a binary data value called d.

Outputs: The output to the sync block is the binary data value labeled q which is equivalent to the inputted data value d but it is only received on the positive edge of the inputted clock signal.

Block Diagram Design for a synchonizer block

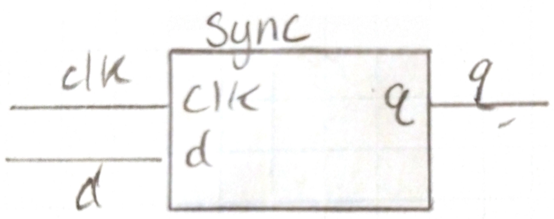


Figure 45: This synchonizer takes in a value d and passes this value onto the output q on the next positive edge of the clk clock cycle input.

Simulation Results for a synchonizer with a single bit data input and output.

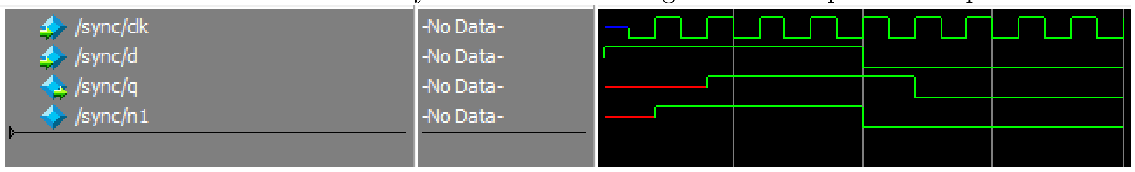


Figure 46: The simulation showcases the input d being changed twice and the delay that comes from the synchonizer. This delay is intentional and allows for the circuit to operate as expected.

3.1.5 Individual Block 5 - alternating

Inputs: The inputs on this block is a clock signal to drive the two D Flip-flops that are inside this block called clock and a reset signal called reset that causes the output q to alternate between 1 and 0.

Outputs: The output to this block is a single binary signal that alternates between 1 and 0 starting at 1 based on the reset input signal.

Block Diagram Design for a block called alternating that acts a reset signal based alternate.

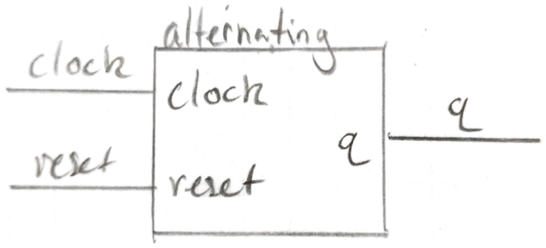


Figure 47: This testing unit acts as a double set of d-flips to allow for the q signal to alternate with the reset signal. This plays an important part in the clock divider circuit in figure39.

Simulation Results for the block called alternating.

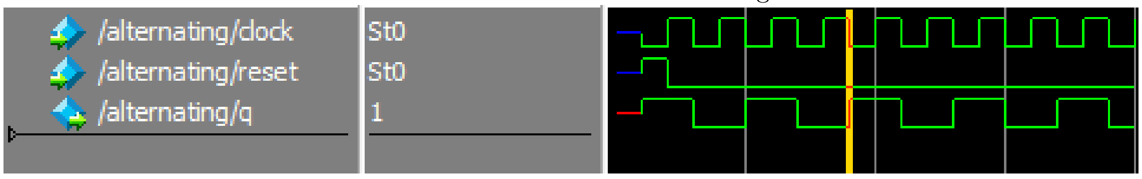


Figure 48: This result from the simulation shows that the testing unit must first be reset by the reset signal upon initialization but form them on the reset signal drives the output to alternate.

3.1.6 Individual Block 6 - keyboardInputDecoder

Inputs: This block takes in a eight-bit input amount called data. These eight-bits make up the data portion of the make codes sent by the PS/2 keyboard when keys are pressed.

Outputs: The outputs of this block are a twelve-bit value that goes into a two input comparator found at figure 53 and a binary reset signal that would go high when a break code is sent. The twelve-bit value is the binary equivalent to the frequency that the outputted sound should be at for each of the implemented keys.

Block Diagram Design for the KeyboardInputDecoder block that takes in a input make or break code from a PS/2 keyboard and turns that into a value to create the correct sound frequency.

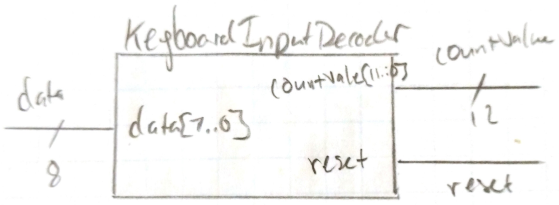
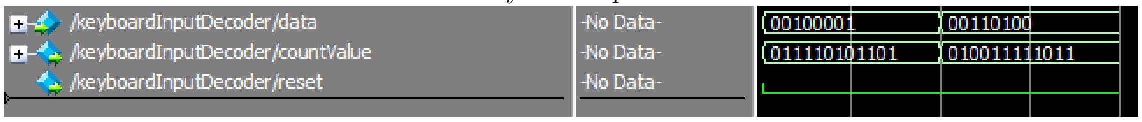


Figure 49: This block takes in inputted keyboard make codes and outputs the specific timing frequency for the implement keys found in 34.

Simulation Results for the KeyboardInputDecoder block.



```
/keyboardInputDecoder/data          -No Data-   00100001       00110100
/keyboardInputDecoder/countValue    -No Data-   011110101101   010011111011
/keyboardInputDecoder/reset         -No Data-
```

Figure 50: This shows the results of two different implemented keyboard make codes to give out the correct frequencies for those musical notes.

3.1.7 Individual Block 7 - counter12

Inputs: This block takes in a clock cycles called clk, and a binary reset signal to initialize the counter.

Outputs: The output is a twelve-bit binary output that increments by one every clock cycle. It begins at 0 until the first positive edge of the clk input.

Block Diagram Design for a basic 12-bit counter

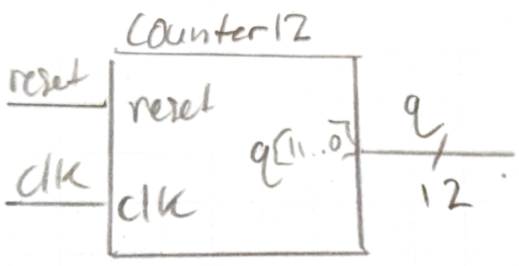


Figure 51: This is a 12-bit counter that takes in a clock cycle to increase the output q by one every clock cycle.

Simulation Results for a basic 12-bit counter

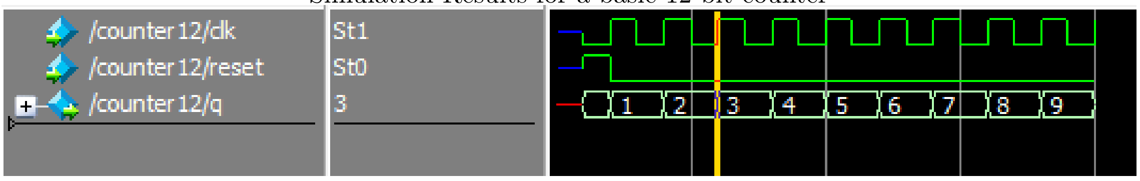


Figure 52: The simulation results showcase the incremental nature of this counter, increasing the output amount by one for every clock cycle of the input clock.

3.1.8 Individual Block 8 - comparatortwoInputs

Inputs: This block takes in two twelve-bit binary values, a and b, and compares them to see if how they equate to one another.

Outputs: The output to this block is a binary signal, 1 if the incoming eight-bit value is equal to the parameter value and 0 if otherwise.

Block Diagram Design for a comparator with two 12-bit inputs

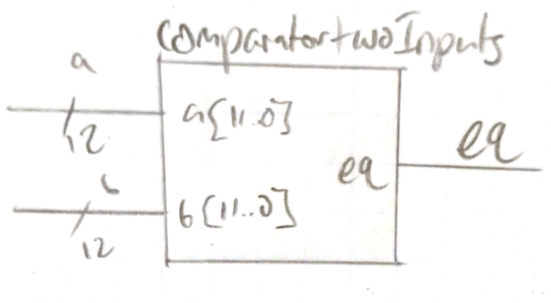


Figure 53: This comparator takes in two values and tests them to see if they are equal to each other and sends out a binary signal of equality.

Simulation Results for a 12-bit comparator

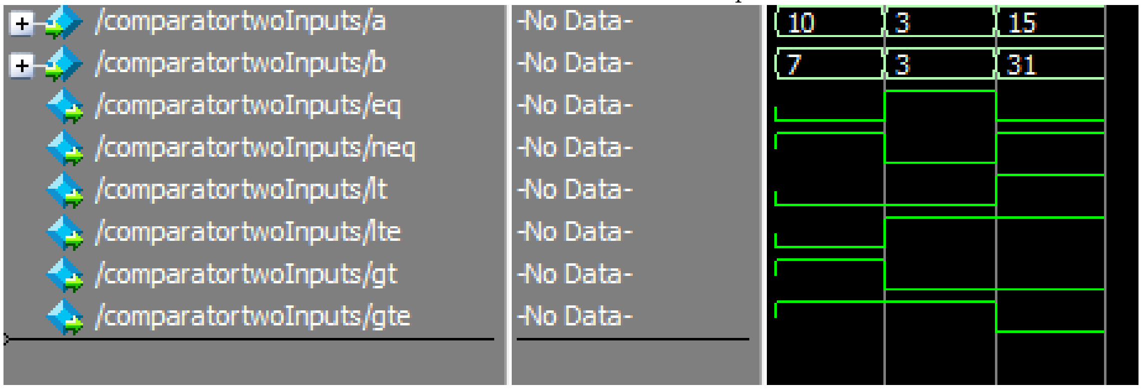


Figure 54: This simulation result shows no only the equal to operator but no equal to, less than, less than and equal to, greater than, and greater than and equal to between the two input values.

3.2 Functional Unit 2 - keyboard_press_driver

Inputs: The inputs for this Functional Unit are it's two clock inputs, Clock50 and PS2_CLK, a single bit data stream called PS2_DAT, and a reset signal. The PS2_DAT works by sending a series of binary 1's and 0's to make up an eleven-bit make code, with a starting bit, eight data bits, a parity bit and a ending bit.

Outputs: The outputs of this functional unit are the valid and makeBreak binary signals that go unused in this version of the project and the eight bit OutCode that makes up the data bits of the make code from the keyboard.

Credit and Source: This design [1] of and code [2] and the Fundamental Unit and the following Individual Block come from Professors Scott Hauck and John S. Loomis from the University of Dayton as well as students Kyle Gagner and Jesse Liston.

Block Diagram Design for the outer layer of the PS/2 Keyboard driver

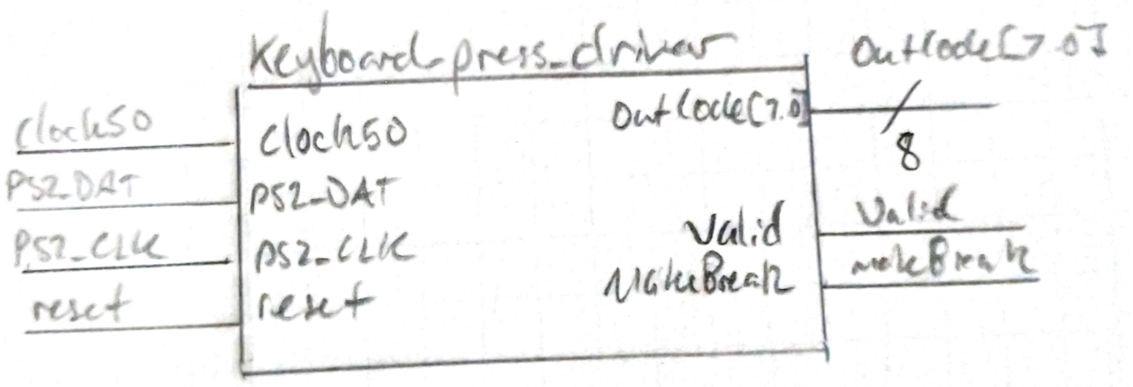
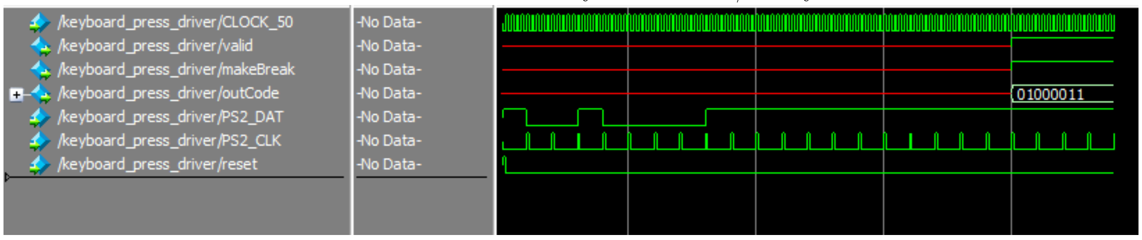


Figure 55: This block diagram shows how the two clocks and a reset signal drive the PS2_DAT to create the needed make code for Fundamental Unit 1 to work.

Simulation Results for the outer layer of the PS/2 Keyboard driver



```
/keyboard_press_driver/CLOCK_50    -No Data-
/keyboard_press_driver/valid       -No Data-
/keyboard_press_driver/makeBreak   -No Data-
/keyboard_press_driver/outCode     -No Data-    01000011
/keyboard_press_driver/PS2_DAT     -No Data-
/keyboard_press_driver/PS2_CLK     -No Data-
/keyboard_press_driver/reset       -No Data-
```

Figure 56: The simulation shows that once enough time has passed on the PS2_CLK input the PS2_DAT creates a proper make code and sends it through the eight-bit OutCode.

3.2.1 Individual Block 9 - keyboard_inner_driver

Inputs: Almost all of the inputs on this inner layer of the keyboard driver are from the outer layer, with the exception of the read input. The rest come from the outer layer and come form the either the PS/2 keyboard or the FPGA.

Outputs: The outputs of this block are the scan_ready binary value and the eight-bit scan_code that goes into the Functional Unit 1 to drive the audio output.

Block Diagram Design for the inner layer of the PS/2 Keyboard driver

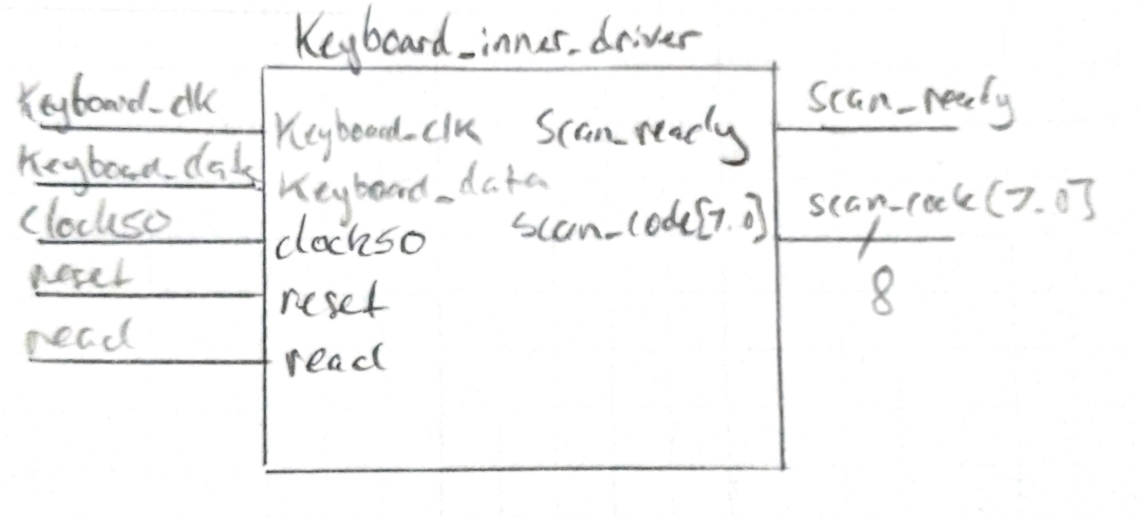
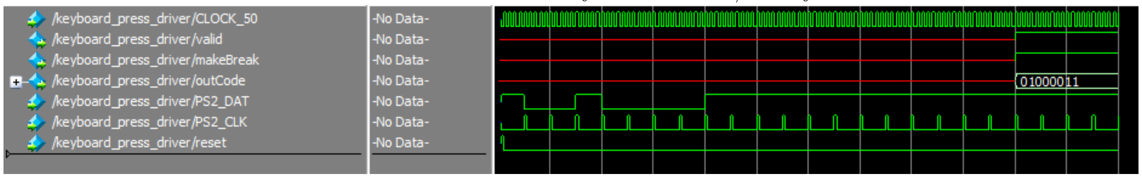


Figure 57: This block deals with the specifics of taking in the proper data from the keyboard_data and turning it into a usable eight-bit scan_code.

Simulation Results for the inner layer of the PS/2 Keyboard driver



```
/keyboard_press_driver/CLOCK_50     -No Data-
/keyboard_press_driver/valid        -No Data-
/keyboard_press_driver/makeBreak    -No Data-
/keyboard_press_driver/outCode      -No Data-   01000011
/keyboard_press_driver/PS2_DAT      -No Data-
/keyboard_press_driver/PS2_CLK      -No Data-
/keyboard_press_driver/reset        -No Data-
```

Figure 58: Once again this simulation shows that once enough time has passed on the keyboard_CLK input the keyboard_DAT creates a proper make code and sends it through the scan_code to create the eight-bit OutCode on the higher level.

4 Infrared Receiver Input - Seven Segment Display Output

The infrared reader module is designed to read infrared code from a remote and interpret them into an address and a command which will be displayed on the seven segment display.

Inputs: This reads in from an infrared receiver module a logic high or low signal. The receiver takes input from infrared waves and does not require coding to provide input to the FPGA. The infrared input comes in the form of waves with small pulses where the gaps between the pulses represent certain signals based on their length. Decoding these gaps is a major part of this module.

Inputs from the built in 50MHz clock and one of the push buttons are also included. The clock was used to keep track of time, while the push button was used to reset the whole system.

Outputs: The infrared reader outputs through the seven segment display. It uses all seven segments of digits zero through three and uses eight specific segments on digits four and five.

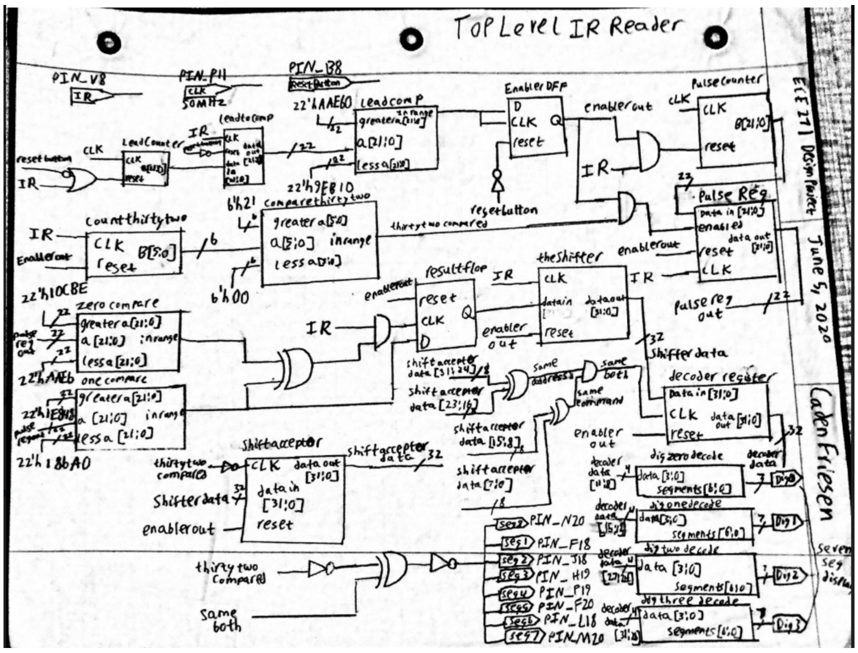


Figure 59: This is the top level design for the infrared reader. It has three inputs in the top left leading to the 35 outputs in the bottom right. If this were to be redone it could definitely benefit from more middle ground modularity. While many modules were made, they were all combined in SystemVerilog in the top file making a somewhat confusing top level diagram.

4.1 Individual Block 1 - Lead Counter

The individual block shown in figure 60 is the first counter in the infrared reader. Its purpose is to count a 50MHz clock signal repeatedly. It resets to zero when either the reset button is pressed, or the infrared input has a rising edge.

Inputs:

clk: The 50MHz onboard clock.

reset: Made of the logical input ((NOT resetbutton) OR IR).

Outputs:

B[21:0]: 22 bit number representing how many clock signals have passed since last reset

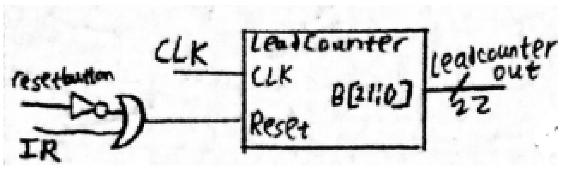


Figure 60: This block is just a simple counter similar to the ones from lab 5. The main difference however is it only resets on the positive edge of the reset, not any time the reset is one.

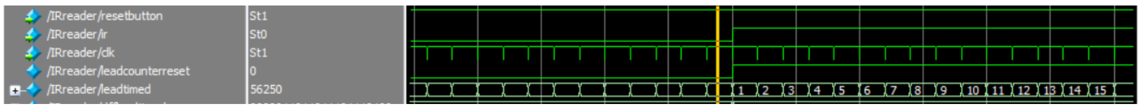

/IRreader/resetbutton St1
/IRreader/ir St0
/IRreader/clk St1
/IRreader/leadcounterreset 0
/IRreader/leadtimed 56250
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Figure 61: This simulation had to be very zoomed in, but it shows that the lead counter counts in time with the 50MHz clock and starts over counting when there is a rise in the infrared wave.

4.2 Individual Block 2 - Lead to Comp

The individual block shown in figure 62 is similar to a D flip flop that leads from the lead counter to the first comparator. It uses the infrared signal's rising edges to trigger, and it is reset to zero when the reset button is pressed.

Inputs:

clk: The infrared signal.

reset: Made of the logical input (NOT resetbutton).

data in[21:0]: The 22 bit number from the lead counter.

Outputs:

data out[21:0]: 22 bit number representing how many clock signals have passed since last reset, now passed through the flip flop.

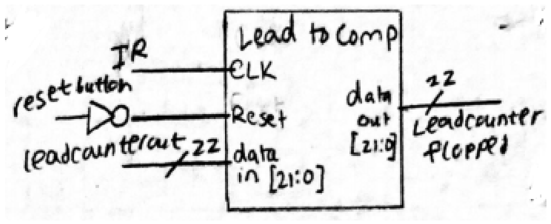


Figure 62: This block is just essentially a d flip flop. Similar to the lead counter its main difference is that it only resets on the rising edge of the reset. This will be a recurring theme for most very simple modules.

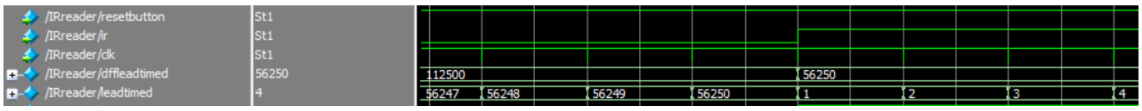


Figure 63: This simulation shows that the flip flop does not prematurely pass along the value of lead counter until there is another rising IR edge at which point it outputs the current lead counter output and continues doing so.

4.3 Individual Block 3 - Lead Comp

The individual block shown in figure 64 is a comparator module that has three inputs and one output. The comparator is used to check if the input denoted a is in between the two other inputs. This is done by checking if it is less than or equal to the input denoted greatera, then checking if it is greater than or equal to the input denoted lessa. If both of these are true then the comparator outputs a one, otherwise it outputs a zero.

The purpose of this module in the overall design is to check if the counter counted the amount of time in a lead bit of an infrared signal (around 13.5ms). It looks for a value between 13ms and 14ms.

Inputs:

greatera[21:0]: The hexadecimal number 22'hAAE60 which is equivalent to the number of 50MHz clock signals in 14 ms.

lessa[21:0]: The hexadecimal number 22'h9EB10 which is equivalent to the number of 50MHz clock signals in 13ms.

a[21:0]: The 22 bit number from the lead counter, passed through the flip flop is used as the number that will be compared to the range expected for a lead signal.

Outputs:

In range: This one bit number signifies whether or not the input a was in between the other two inputs.

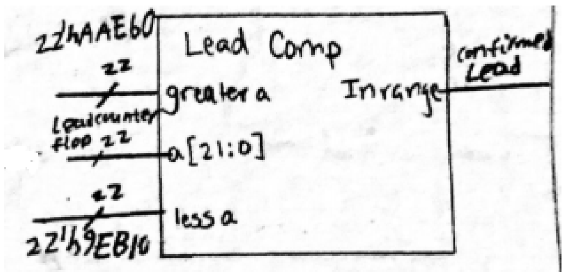


Figure 64: This block is a comparator which outputs a 1 if lessa $\leq$ a $\leq$ greatera.

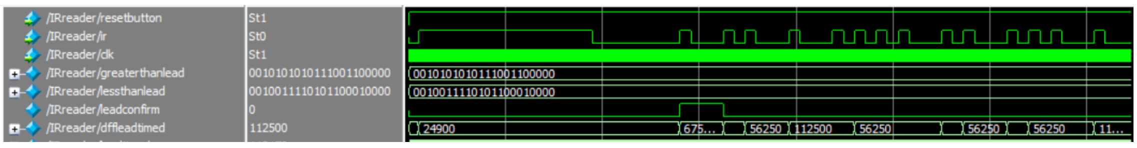

```
/IRreader/resetbutton      St1
/IRreader/ir               St0
/IRreader/clk              St1
/IRreader/greaterthanlead  0010101010111001100000
/IRreader/lessthanlead     0010011110101100010000
/IRreader/leadconfirm      0
/IRreader/dffleadtimed     112500
```

Figure 65: The lead comparator can be shown triggering the confirm for the lead signal once it ends and is fed the count from the lead counter.

4.4 Individual Block 4 - Enabler DFF

The individual block shown in figure 66 is similar to a D flip flop and is used to enable the rest of the system. It uses a clock signal that is the same as its data input. This causes it to output one once the data reaches a one, but does not reset back to zero unless the reset button is pressed once it has been enabled. This block receives its data from the lead comparator and enables the rest of the system once the comparator has confirmed the detection of a lead signal. This is also used as an overall system reset. When this outputs a one for the first time it resets all the future blocks in the system to zero.

Inputs:

clk: Data from lead comparator representing the detection of a lead signal.

reset: Made of the logical input (NOT resetbutton).

D: Data from lead comparator representing the detection of a lead signal.

Outputs:

Q: Used as an enabler signal for the rest of the system as well as the system reset. Is equivalent to D on the last rising edge of the clock input.

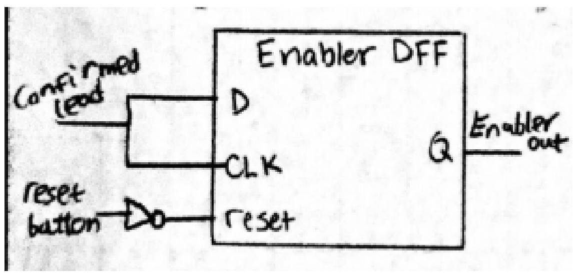


Figure 66: This block is unique from a typical D flip flop for two reasons. The Enabler DFF block uses its data input as its clock signal as well. Similar to previous blocks, it also only resets on the rising edge of the reset, not when the reset changes back to zero.

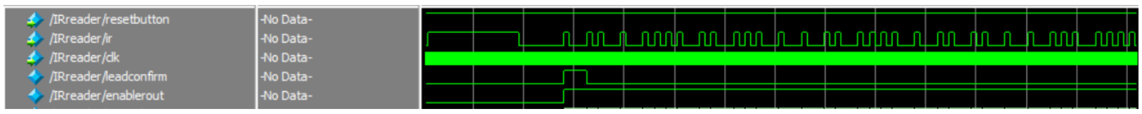


Figure 67: Once the lead signal is detected and lead confirmed is raised, enabler out is raised to one and remains there regardless of changes in lead confirm.

4.5 Individual Block 5 - Pulse Counter

The individual block shown in figure 68 is a counter module used to keep track of how long has passed between rising edges of the infrared pulse signals. It uses the 50MHz clock to increment while it receives its reset from the rising edge of the infrared signal.

Inputs:

clk: The 50MHz clock used to increment the counter.

reset: Made of the logical input (IR AND enablerout). This is done so that the reset happens on the rising edge of IR but only once the enabler has been triggered.

Outputs:

B[21:0]: 22 bit number representing how many clock signals have passed since last reset.

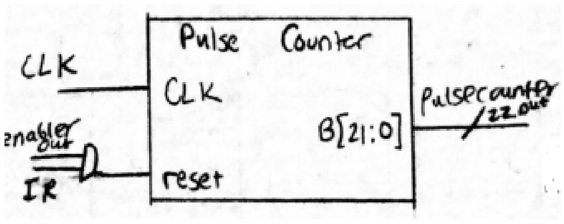


Figure 68: This counter keeps track of the time between rising edges of the infrared pulse. Similar to past blocks it only resets on the rising edge of its reset. This is used as the main input for finding out whether the infrared pulse represents a one or zero.

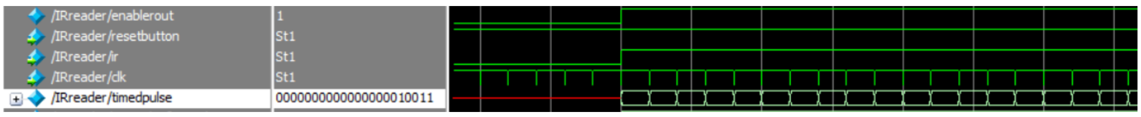


Figure 69: The simulation of the pulse counter must be highly zoomed in. It is shown however that is properly triggers on every 50MHz signal keeping up with the amount of time that has passed.

4.6 Individual Block 6 - Count Thirty Two

The individual block shown in figure 70 is a counter with the purpose of tracking how many infrared signal rises have gone through the system so far. It is made to count up to thirtytwo which is the number of bits expected in an infrared signal. This counters input to the system is used to determine when to do the finishing movements with the fully collected data. Inputs:

clk: The rising edge of the infrared signal is used to increment this clock.

reset: Enablerout resets this counter when it first reaches one.

Outputs:

B[5:0]: This five bit number represents the amount of infrared signal pulses so far.

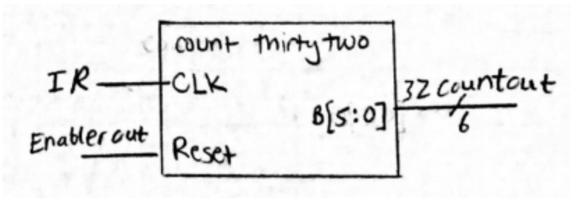


Figure 70: This counter uses a 6 bit output that informs the system on how many infrared pulses have passed so far. It is only reset when enablerout hits one at a rising edge.

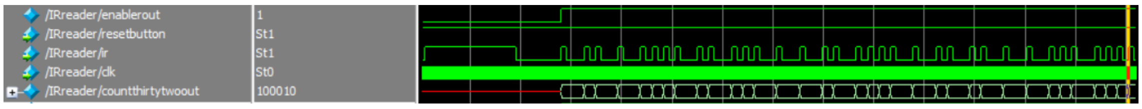


Figure 71: The counter does count each signal at the right time and it does end at 34 which is where it should end due to the lead pulse and ending pulse.

4.7 Individual Block 7 - Compare Thirty Two

The individual block shown in figure 72 is a comparator with the purpose of checking if the Count Thirty Two counter has exceeded thirty two pulses yet. The comparator outputs a one as long as the counted pulses is under thirty two.

Inputs:

a[5:0]: This is the output of the Count Thirty Two counter and can be up to 63 since it is a six bit binary number.

greatera[5:0]: One of the numbers the comparator uses for comparisons. The comparator checks if the input is less than or equal to this input of 6'h21 which is equivalent to 33 since in this system the counter actually reaches 34 once all the processing has been done.

lessa[5:0]: One of the numbers the comparator uses for comparisons. The comparator checks if the input is greater than or equal to this input of 6'h00 which is equivalent to zero. This is mainly a placeholder number for the comparator to use the same module as the rest of the system.

Outputs:

In Range: This one bit signal is a one if 32 pulse signals have not yet passed and is a zero if they have.

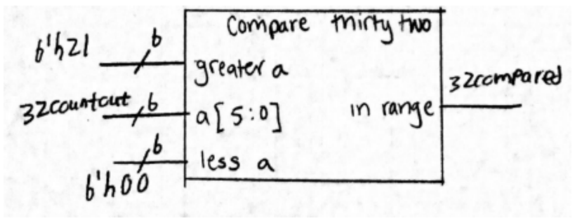
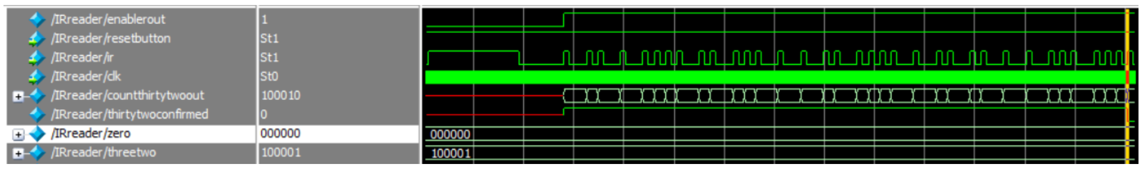


Figure 72: This comparator controls the late phase pieces of the system, telling registers when all the data should be present for decoding.



```
/IRreader/enablerout           1
/IRreader/resetbutton          St1
/IRreader/ir                   St1
/IRreader/clk                  St0
/IRreader/countthirtytwoout    100010
/IRreader/thirtytwoconfirmed   0
/IRreader/zero                 000000
/IRreader/threetwo             100001
```

Figure 73: The simulation properly stays at one until the counted pulses rises above 32 at which point it drops to zero.

4.8 Individual Block 8 - Pulse Reg

The individual block shown in figure 74 is a 22 bit register designed to pass along the data from the pulse counter representing how long the most recent infrared pulse was. It is enabled by the enabler DFF and is also reset to zero when the enabler DFF originally detects lead code. It passes this data along to comparators for decoding into a one or zero.

Inputs:

clk: The clock signal driving this register is the rising edge of the infrared pulse.

data in[21:0]: This is the output of the pulse counter as a 22 bit binary number. It represents the amount of time since the last rising edge of the infrared pulse.

reset: This input resets the pulse reg to zero on the rising edge of the enabler output.

enabled: This input is composed of the logical connection (thirtytwocompared AND IR), and allows the pulse reg to accept more inputs. This is limited to make sure that both the lead code has been seen, and the number of infrared pulses has been less than or equal to 32.

Outputs:

Data Out[21:0]: This 22 bit signal is the amount of time tracked by pulse counter as it is prepared to pass along to the next comparators. It will always represent the last completed infrared pulse.

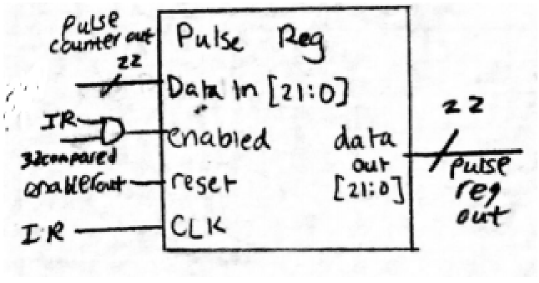
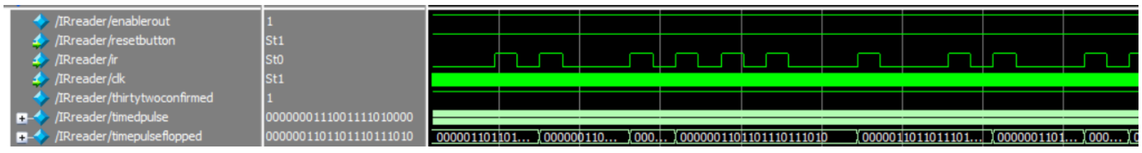


Figure 74: This register is composed of 22 enabled D flip flops and passes along the data from Pulse Counter to a set of comparators.



```
/IRreader/enableout            1
/IRreader/resetbutton          St1
/IRreader/ir                   St0
/IRreader/clk                  St1
/IRreader/thirtytwoconfirmed   1
/IRreader/timedpulse           0000000111001111010000
/IRreader/timepulseflopped     0000001101101110111010
```

Figure 75: The pulse register appears to only output on some infrared clock signals. This is because when a one or a zero is measured multiple times in a row the output value is the same amount of time so the output does not change.

4.9 Individual Block 9 - Zero Compare and One Compare

The individual blocks shown in figure 76 are two comparators that work essentially the same way. They both take in the data output of the pulse register and compare it with their own base inputs. Zero compare tests to see if the measured pulse is a zero, while one compare tests to see if the measured pulse is a one. Both compatators output a one if they detect that the input matches what they are looking for.

These comparators are referring to the expected time for an infrared pulse. For a pulse to be a one, it is expected to be around 2.0 to 2.5 milliseconds. For a pulse to be a zero, it is expected to be around .875 to 1.375 milliseconds.

Inputs for compare zero:

a[21:0]: This is the output of the pulse register which comes from the pulse counter. It measures the last completed infrared signal's time through a 50Mhz clock signal.

greatera[21:0]: One of the numbers the comparator uses for comparisons. The comparator checks if the input is less than or equal to this input of 6'h10CBE which is equivalent to 1.375ms worth of clock cycles.

lessa[5:0]: One of the numbers the comparator uses for comparisons. The comparator checks if the input is greater than or equal to this input of 6'hAAE6 which is equivalent to .875ms worth of clock cycles.

Inputs for compare one:

a[21:0]: This is the output of the pulse register which comes from the pulse counter. It measures the last completed infrared signal's time through a 50Mhz clock signal.

greatera[21:0]: One of the numbers the comparator uses for comparisons. The comparator checks if the input is less than or equal to this input of 6'h1E848 which is equivalent to 2.5ms worth of clock cycles.

lessa[5:0]: One of the numbers the comparator uses for comparisons. The comparator checks if the input is greater than or equal to this input of 6'h186A0 which is equivalent to 2.0ms worth of clock cycles.

Outputs:

In Range: This one bit signal is a one if the comparator involved detects the input is in between its greatera and lessa input, and a zero if it is not.

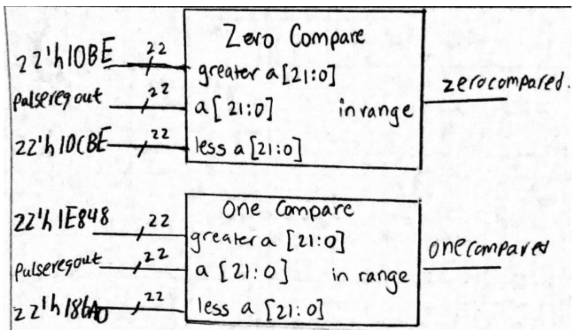


Figure 76: These comparators are designed to decode the time of the last pulse into either a one or a zero.

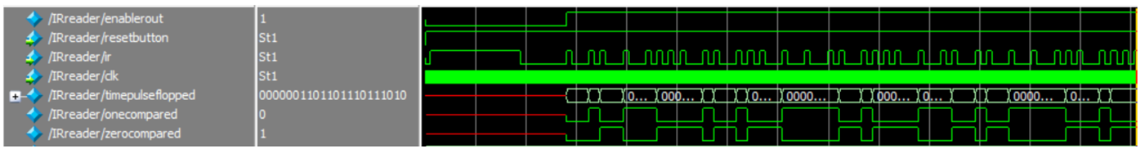


Figure 77: Notably the one and zero compare outputs are always the opposite of each other except at the beginning when there are no inputs. This means values are being properly detected.

4.10 Individual Block 10 - Result Flop

The individual block shown in figure 78 is a D flip flop style module made to pass along the result decided upon by the comparators. It takes in the input from the comparator that looks for ones since this will work either way. If the input is a one the comparator will output a one and if it is a zero it will output a zero. Both comparators are used however in a xor gate to confirm that one of them did get triggered to make sure the pulse isn't neither. This flip flop is made to pass on the result to the shift register for data storage.

Inputs:

clk: The clock signal driving this D flip flop is a made of the logic ((onecompared XOR zerocompared) AND IR). Allowing the flip flop to trigger on the clock signal while only one of the comparators is outputting a one.

D: The data in comes from the output of the One Compare comparator and is a singular bit showing whether the last complete pulse was a one or a zero.

reset: This input resets the result flop to zero on the rising edge of the enabler output.

enabled: This input is composed of the logical connection (thirtytwocompared AND IR), and allows the pulse reg to accept more inputs. This is limited to make sure that both the lead code has been seen, and the number of infrared pulses has been less than or equal to 32.

Outputs:

Q: This output is the same as the input D at the last rising edge of the clock input. It represents the data from the last pulse of the infrared signal.

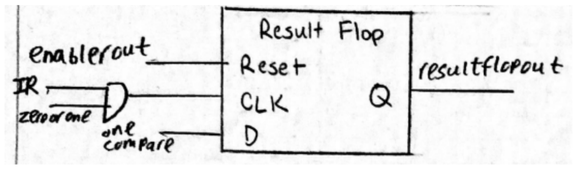


Figure 78: This D flip flop style module is designed to give the decoded infrared pulse information to the shift register.

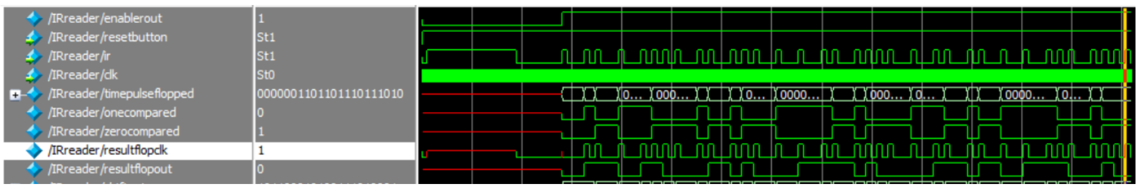

/IRreader/enablerout 1
/IRreader/resetbutton St1
/IRreader/ir St1
/IRreader/clk St0
/IRreader/timepulseflopped 000000110110111011010
/IRreader/onecompared 0
/IRreader/zerocompared 1
/IRreader/resultflopclk 1
/IRreader/resultflopout 0

Figure 79: In this simulation it is shown sometimes the result flop maintains the same values. This is because sometimes multiple ones or zeros come in a row so it doesn't have to change.

4.11 Individual Block 11 - The Shifter

The individual block shown in figure 80 is a 32 bit shift register designed to store the 32 bits of the infrared input. It takes in one bit at a time and shifts them down a chain of 32 bits through an internal set of D flip flops. It is reset fully to zero when the enabler output is first triggered, and it is triggered to take in a new input each time the infrared pulse begins.

Inputs:

clk: The clock signal driving this shift register is the rising edge of the infrared signal.

Data in: The data in comes from the output of the result flop and is a singular bit showing whether the last complete pulse was a one or a zero.

reset: This input resets the entire shift register to zero on the rising edge of the enabler output.

Outputs:

Data Out[31:0]: This is the output of the 32 bits of the infrared signal. It starts off as 32 zeros with each progressive input being put in the least significant bit and pushing the previous inputs one bit more significant.

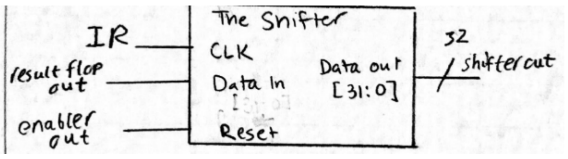
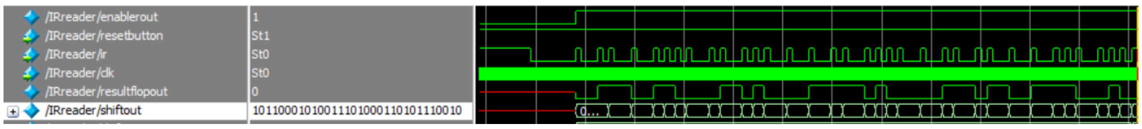


Figure 80: This shift register combined of 32 d flip flops chained together is designed to shift the infrared signal from serial logic to paralell logic.



```
/IRreader/enablerout      1
/IRreader/resetbutton     St1
/IRreader/ir              St0
/IRreader/clk             St0
/IRreader/resultflopout   0
/IRreader/shiftout        10110001010011101000110101110010
```

Figure 81: While it is hard to show in one picture, the shift register does slowly shift its input down the line. The final output is shown on the left and is equal to the proper infrared signal.

4.12 Individual Block 12 - Shift Acceptor

The individual block shown in figure 82 is a 32 bit register designed to take the information from the shift register and test it for errors. The shift acceptor takes in the data from the shift register when the 32 counter reaches 32 completed infrared pulses. This is done on the negative edge of the Compare Thirty Two comparator.

Inputs:

clk: The clock signal driving this shift register is the sinking edge of the Thirty Two Comparator.

Data in[31:0]: The data in comes from the output of the shift register and is all 32 parallel bits of the infrared signal.

reset: This input resets the entire shift register acceptor to zero on the rising edge of the enabler output.

Outputs:

Data Out[31:0]: This is the output of the 32 bits of the infrared signal. Here it is output to further logic to verify if it is a real infrared signal.

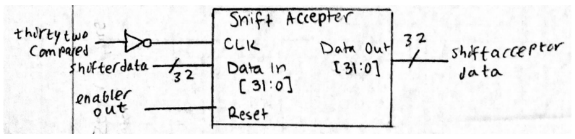
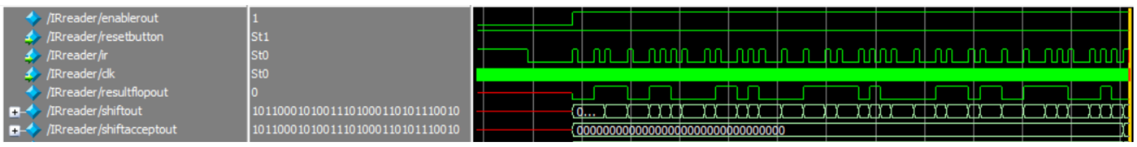


Figure 82: This shift acceptor is a simple 32 bit register composed of D flip flops and is designed to pass along the shift register info once all 32 bits have been collected.



```
/IRreader/enablerout         1
/IRreader/resetbutton        St1
/IRreader/ir                 St0
/IRreader/clk                St0
/IRreader/resultflopout      0
/IRreader/shiftout           10110001010011101000110101110010
/IRreader/shiftacceptout     10110001010011101000110101110010
```

Figure 83: Shift acceptor doesn't start putting out the shift register's data until 32 infrared pulses have passed.

4.13 Individual Block 13 - Same command, Same Address, and Same Both

The individual block shown in figure 84 is a set of logical connections that result in the logic output labelled "same both". The goal of this block is to check if the 0-7 and 8-15 bits as well as the 16-23 and 24-31 bits are truly logical inverses of each other as they should be. This is done by putting each corresponding bit into a XOR gate and then seeing if all the XOR gates output a one as they should. If they do same both is declared one. If they do not same both is a zero.

Inputs:

Shift Acceptor[31:0]: The data in comes from the output of the shift acceptor and represents the 32 parallel bits of the infrared signal.

Outputs:

Same Both: This output is a one if the command and command inverse are truly inverse as well as if the address and address inverse are truly inverses. Otherwise it is a zero output.

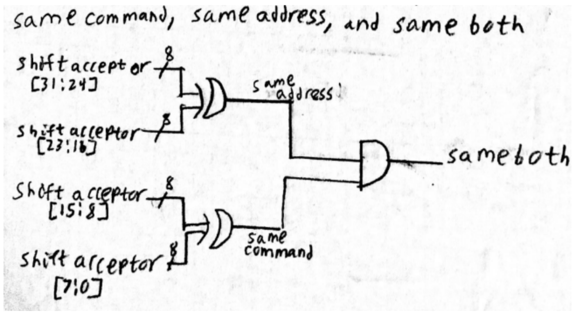


Figure 84: This set of logic works as built in error handling for the system and will stop the program later if it fails.

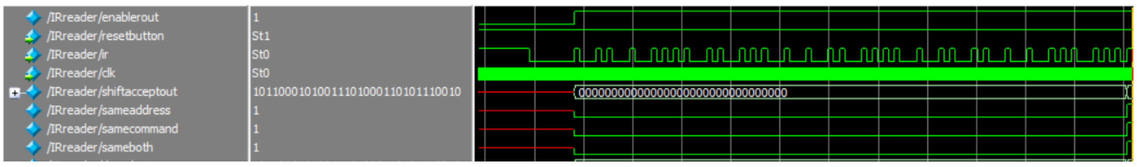

```
/IRreader/enablerout      1
/IRreader/resetbutton     St1
/IRreader/ir              St0
/IRreader/clk             St0
/IRreader/shiftacceptout  10110001010011101000110101110010
/IRreader/sameaddress     1
/IRreader/samecommand     1
/IRreader/sameboth        1
```

Figure 85: Same both only switches to one once both same command and same address have been met. Looking at the data inputs it can be confirmed that the address and inverse address as well as the command and inverse command inputs are truly inverse proving this functionality.

4.14 Individual Block 14 - Decoder Register

The individual block shown in figure 86 is another 32 bit register designed to take data from the shift register. This is done later on once the data has been confirmed to be a real infrared signal. This register is triggered to take in data on the rising edge of Same Both. This allows it to take in the data as soon as it has been declared valid. It then outputs this data to the decoders for output preparation.

Inputs:

Shifter Data[31:0]: The data in comes from the output of the shift register and represents the 32 parallel bits of the infrared signal.

clk: The clock for this block is triggered by the rising edge of the Same Both variable from the previous logic section.

reset: This input resets the entire decoder register to zero on the rising edge of the enabler output.

Outputs:

Data Out[31:0]: This output is the data from the shift register being sent to the decoders now that it is confirmed to be an infrared signal.

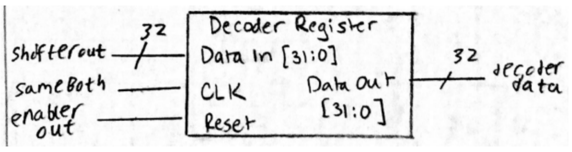


Figure 86: This register is the final block between the decoders outputting the infrared code and the shift register.

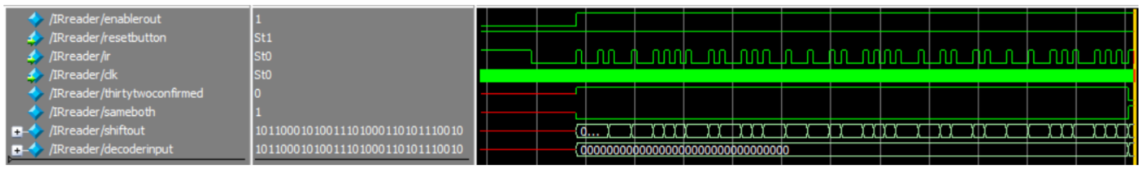

```
/IRreader/enablerout           1
/IRreader/resetbutton          St1
/IRreader/ir                   St0
/IRreader/clk                  St0
/IRreader/thirtytwoconfirmed   0
/IRreader/sameboth             1
/IRreader/shiftout             10110001010011101000110101110010
/IRreader/decoderinput         10110001010011101000110101110010
```

Figure 87: This register is triggered as soon as same both switches to one allowing the shift register data to pass through.

4.15 Individual Block 15 - Digzero decode through Digthree encode

The four individual blocks shown in figure 88 are all decoders with the same purpose. Each one decodes four bits of data and then outputs signals to the seven segments of a digit on the FPGA. They are each connected to the digit associated with their number. The internals are built with a seven segment decoder designed to display the four digit binary input in hexadecimal. Digits three and two display the eight bit address signal while digits one and zero display the eight bit command signal.

Inputs:

Decoder Data[11:8] -> Digzero

Decoder Data[15:12] -> Digone

Decoder Data[27:24] -> Digtwo

Decoder Data[31:28] -> Digthree

These data inputs are the four bit binary signals that will be transformed to hexadecimal.

Outputs:

All four of these output a 7 bit logic output to the seven segments of their designated digit. A zero causes the segment to light up while a one causes the segment to stay off.

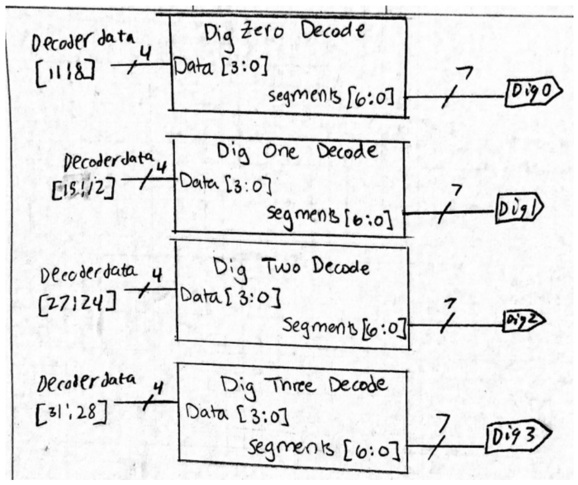
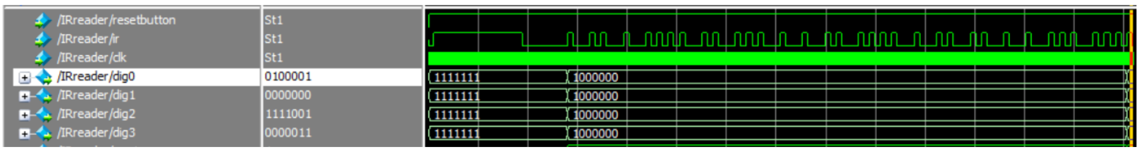


Figure 88: The four decoders are largely unchanged from the labs earlier in Digital Logic Design. It is important to note if the decoder register is reset all of these digits will turn off.



```
/IRreader/resetbutton   St1
/IRreader/ir            St1
/IRreader/clk           St1
/IRreader/dig0          0100001     1111111   1000000
/IRreader/dig1          0000000     1111111   1000000
/IRreader/dig2          1111001     1111111   1000000
/IRreader/dig3          0000011     1111111   1000000
```

Figure 89: The digits stay as all ones until they are enabled. After this they shift to all zeros before finishing displaying the address and command of the infrared code.

4.16 Individual Block 16 - Not Equal Outputs

The set of logic shown in figure 90 is an error handling section of the infrared reader. This section is based off of the Thirty Two Compared and the Same Both inputs. Using these the logic determines whether an error occurred during the process of the decoding. If it did, eight segments in digits four and five of the display will light up to display the letters NE for not equal.

Inputs:

Thirty Two Compared: This input is inverted and gives out a one if all thirty two bits of infrared data have been collected.

Same Both: This input is a zero if the command and address were not truly the inverse of their inverse data.

Outputs:

Seg1 through Seg8: These outputs represent the letters NE on the fourth and fifth seven segment display digits. They are turned on if given a zero as the output.

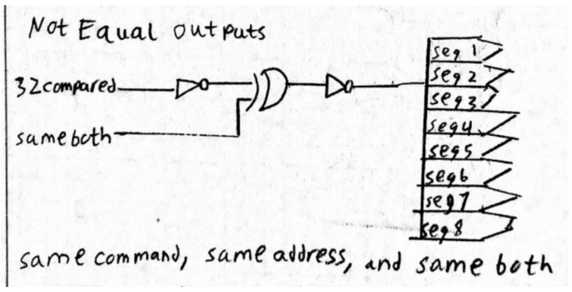


Figure 90: This logic checks to make sure that either same both is incorrect (or not given data yet) and there has not been thirty two inputs, or same both is correct and there has been thirty two inputs. If the truth is not one of these the logic gates input zeros to the segments from the seven segment display turning them on and displaying NE.

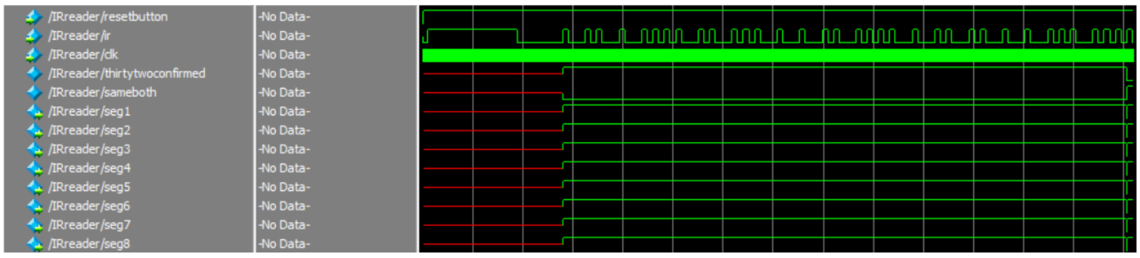


Figure 91: In the simulation these segments stay as ones since Thirty Two Confirmed and Same Both always remain opposite of each other. These don't activate at all until the lead code is detected.

A SystemVerilog Files

```
// Copyright (C) 2018  Intel Corporation. All rights reserved.
// Your use of Intel Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Intel Program License
// Subscription Agreement, the Intel Quartus Prime License Agreement,
// the Intel FPGA IP License Agreement, or other applicable license
// agreement, including, without limitation, that your use is for
// the sole purpose of programming logic devices manufactured by
// Intel and sold by Intel or its authorized distributors.  Please
// refer to the applicable agreement for further details.

// PROGRAM              "Quartus Prime"
// VERSION              "Version 18.0.0 Build 614 04/24/2018 SJ Lite Edition"
// CREATED              "Fri Jun 05 21:05:30 2020"

module designprojectfinal(
        clk50mhz,
        reset,
        nesdata,
        keyboarddata,
        clockkeyboard,
        resetkeyboard,
        irdata,
        nesclk,
        neslatch,
        motorcw,
        motorccw,
        leddata,
        audioout,
        seg1,
        seg2,
        seg3,
        seg4,
        seg5,
        seg6,
        seg7,
        seg8,
        dig0,
        dig1,
        dig2,
        dig3
);

```

```

input wire       clk50mhz;
input wire       reset;
input wire       nesdata;
input wire       keyboarddata;
input wire       clockkeyboard;
input wire       resetkeyboard;
input wire       irdata;
output wire      nesclk;
output wire      neslatch;
output wire      motorcw;
output wire      motorccw;
output wire      leddata;
output wire      audioout;
output wire      seg1;
output wire      seg2;
output wire      seg3;
output wire      seg4;
output wire      seg5;
output wire      seg6;
output wire      seg7;
output wire      seg8;
output wire      [6:0] dig0;
output wire      [6:0] dig1;
output wire      [6:0] dig2;
output wire      [6:0] dig3;

wire    clk50;
wire    rst;





KeyboardtoSquareWave    b2v_Broce1(
        .resetSwtich(rst),
        .Clock50MHz(clk50),
        .KeyboardData(keyboarddata),
        .ClockKeyboard(clockkeyboard),
        .ResetKeyboard(resetkeyboard),
        .AudioOutput(audioout));


IRreader        b2v_Friesen1(
        .resetbutton(rst),
        .ir(irdata),
        .clk(clk50),
        .seg1(seg1),
        .seg2(seg2),
        .seg3(seg3),
        .seg4(seg4),
        .seg5(seg5),
        .seg6(seg6),
        .seg7(seg7),
        .seg8(seg8),
        .dig0(dig0),
        .dig1(dig1),
        .dig2(dig2),
        .dig3(dig3));


horinepartDP    b2v_Horine1(
        .rst(rst),
        .nesdata(nesdata),
        .clk50mhz(clk50),
        .nesclk(nesclk),
        .neslatch(neslatch),
        .outcw(motorcw),
        .outccw(motorccw),
        .leddata(leddata));

assign  rst = reset;
assign  clk50 = clk50mhz;

endmodule
```

A.1 NES Controller Input - Motor and Addressable LED Output

```
// Copyright (C) 2019  Intel Corporation. All rights reserved.
// Your use of Intel Corporation's design tools, logic functions
// and other software and tools, and any partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Intel Program License
// Subscription Agreement, the Intel Quartus Prime License Agreement,
// the Intel FPGA IP License Agreement, or other applicable license
// agreement, including, without limitation, that your use is for
// the sole purpose of programming logic devices manufactured by
// Intel and sold by Intel or its authorized distributors.  Please
// refer to the applicable agreement for further details, at
// https://fpgasoftware.intel.com/eula.

// PROGRAM              "Quartus Prime"
// VERSION              "Version 19.1.0 Build 670 09/22/2019 SJ Lite Edition"
// CREATED              "Fri Jun 05 12:05:06 2020"
//Trevor Horine
//This is the generated verilog for my toplevel file that was built in schmatic view.

module horinepartDP(
        clk50mhz,
        data,
        rst,
        up,
        down,
        left,
        right,
        start,
        select,
        a,
        b,
        nesl,
        datap,
        nesclk,
        neslatch,
        sysclk,
        outcw,
        outccw,
        leddata,
        D1,
        D2,
        D3,
        D4,
        D5,
        D6
);


input  wire         clk50mhz;
input  wire         data;
input  wire         rst;
output  wire        up;
output  wire        down;
output  wire        left;
output  wire        right;
output  wire        start;
output  wire        select;
output  wire        a;
output  wire        b;
output  wire        nesl;
output  wire        datap;
output  wire        nesclk;
output  wire        neslatch;
output  wire        sysclk;
output  wire        outcw;
output  wire        outccw;
output  wire        leddata;
output  wire        [6:0]  D1;
output  wire        [6:0]  D2;
output  wire        [6:0]  D3;
output  wire        [6:0]  D4;
output  wire        [6:0]  D5;
output  wire        [6:0]  D6;

wire      a_ALTERA_SYNTHESIZED;
wire      b_ALTERA_SYNTHESIZED;
wire      clk16mhz;
wire      clktispoint3us;
wire      countclk;
wire      datapin;
wire      down_ALTERA_SYNTHESIZED;
wire      [23:0]  grbvalue;
wire      left_ALTERA_SYNTHESIZED;
wire      neslat;
wire      right_ALTERA_SYNTHESIZED;
wire      start_ALTERA_SYNTHESIZED;
wire      up_ALTERA_SYNTHESIZED;




clkdiv    b2v_horine1(
        .clk(clk50mhz),
        .reset(rst),
        .newclk(clk16mhz));
        defparam          b2v_horine1.N = 100000;


sevseg    b2v_horine10(
        .data(grbvalue[15:12]),
        .seg(D4));


sevseg    b2v_horine11(
        .data(grbvalue[19:16]),
        .seg(D5));


sevseg    b2v_horine12(
        .data(grbvalue[23:20]),
        .seg(D6));


clkdiv    b2v_horine2(
        .clk(clk50mhz),
        .reset(rst),
        .newclk(clktispoint3us));
        defparam          b2v_horine2.N = 15;


clkdiv    b2v_horine3(
        .clk(clk50mhz),
        .reset(rst),
        .newclk(countclk));
        defparam          b2v_horine3.N = 625000;


nes       b2v_horine4(
        .clk(clk16mhz),
        .reset(rst),
        .nesdata(datapin),
        .nesclk(nesclk),
        .neslatch(neslat),
        .up(up_ALTERA_SYNTHESIZED),
        .down(down_ALTERA_SYNTHESIZED),
        .left(left_ALTERA_SYNTHESIZED),
        .right(right_ALTERA_SYNTHESIZED),
        .start(start_ALTERA_SYNTHESIZED),
        .select(select),
        .a(a_ALTERA_SYNTHESIZED),
        .b(b_ALTERA_SYNTHESIZED));


grbcounter        b2v_horine5(
        .clk(countclk),
        .reset(rst),
        .gcolorbutton(start_ALTERA_SYNTHESIZED),
        .rcolorbutton(b_ALTERA_SYNTHESIZED),
        .bcolorbutton(a_ALTERA_SYNTHESIZED),
        .up(up_ALTERA_SYNTHESIZED),
        .down(down_ALTERA_SYNTHESIZED),
        .grbvalue(grbvalue));


led     b2v_horine6(
        .reset(rst),
        .clk(clktispoint3us),
        .grb(grbvalue),
        .ledout(leddata));


sevseg  b2v_horine7(
        .data(grbvalue[3:0]),
        .seg(D1));


sevseg  b2v_horine8(
        .data(grbvalue[7:4]),
        .seg(D2));


sevseg  b2v_horine9(
        .data(grbvalue[11:8]),
        .seg(D3));

assign  up = up_ALTERA_SYNTHESIZED;
assign  datapin = data;
assign  down = down_ALTERA_SYNTHESIZED;
assign  left = left_ALTERA_SYNTHESIZED;
assign  right = right_ALTERA_SYNTHESIZED;
assign  start = start_ALTERA_SYNTHESIZED;
assign  a = a_ALTERA_SYNTHESIZED;
assign  b = b_ALTERA_SYNTHESIZED;
assign  nesl = neslat;
assign  datap = datapin;
assign  neslatch = neslat;
assign  sysclk = clk16mhz;
assign  outcw = right_ALTERA_SYNTHESIZED;
assign  outccw = left_ALTERA_SYNTHESIZED;

endmodule
```

```
//Trevor Horine
//This is a clock divider that toggels at integer values of the input clock.
module clkdiv #(parameter N = 15) //100000000
                (input logic clk,
                input logic reset,
                output logic newclk);

                logic [25:0] t;

                always_ff@(posedge clk, posedge reset) begin
                if (reset) begin
                t <= 0;
                newclk <= 0;
                end
                else if (t == N)begin
                        t <= 0;
                        newclk <= !newclk;
                end
                else t <= t + 1;
                end
endmodule
```

```
//Trevor Horine
//This module counts color values between 0 and 255
module colorcount #(parameter N = 8)
                (input logic clk,
                input logic colorbutton,
                input logic reset,
                input logic u,
                input logic d,
                output logic[N-1:0] colorvalue);

        always_ff@(posedge clk, posedge reset)
                if (reset)
                             colorvalue <= 0;
                else if (clk & colorbutton & u & ~d)            colorvalue <= colorvalue + 1;
                else if (clk & colorbutton & d & ~u)            colorvalue <= colorvalue - 1;
                else
                                     colorvalue <= colorvalue;
endmodule
```

```
//Trevor Horine
//This is a greater than or equal to comparator
module comparator #(parameter N = 4, M = 10)
                (input logic [N-1:0] a,
                output logic gt);

        assign gt = (a >= M);
endmodule
```

```
//Trevor Horine
//This is a counter
module counter #(parameter N = 4)
                (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
```

```
//Trevor Horine
//This modules is practicly an OR gate used for resets
module countreset (input logic reset,
                                                input logic comparrst,
                                                output logic rst);

        assign rst = comparrst | reset;
endmodule
```

```
//Trevor Horine
//This module counts color values for red green and blue and outputs rgb values
module grbcounter(input logic clk,
                        input logic reset,
                        input logic gcolorbutton,
                        input logic rcolorbutton,
                        input logic bcolorbutton,
                        input logic up,
                        input logic down,
                        output logic [23:0] grbvalue);

        colorcount horineg(
                .clk                                    (clk),
                .reset                          (reset),
                .colorbutton            (rcolorbutton),
                .u                                              (up),
                .d                                              (down),
                .colorvalue                     (grbvalue[23:16])
                );

        colorcount horiner(
                .clk                                    (clk),
                .reset                          (reset),
                .colorbutton            (gcolorbutton),
                .u                                              (up),
                .d                                              (down),
                .colorvalue                     (grbvalue[15:8])
        );

        colorcount horineb(
                .clk                                    (clk),
                .reset                          (reset),
                .colorbutton            (bcolorbutton),
                .u                                              (up),
                .d                                              (down),
                .colorvalue                     (grbvalue[7:0])
        );
endmodule
```

```
//Trevor Horine
//This module drives an addressable LED
module led (input logic [23:0] grb,
                                input logic reset,
                                input logic clk,
                                output logic ledout);

                wire [8:0] count;
                wire gt;
                wire srst;
                wire rst;
                wire [95:0] ledwaveout;

counter#(.N(9)) horine0(
.clk            (clk),
.reset  (rst),
.q                      (count)
);
comparator #(.N(9), .M(363)) horine1(
.a                                                      (count),
.gt                                             (gt)
);

sync horine2(
.clk                            (clk),
.d                                      (gt),
.q                                      (srst)
);

countreset horine3(
.reset                  (reset),
.comparrst              (srst),
.rst                            (rst)
);

test horine4(
.grb                            (grb),
.reset                  (reset),
.read                           (rst),
.ledwaveout             (ledwaveout)
);

waveout horine5(
        .count                                  (count),
        .clk                                            (clk),
        .ledwave                                        (ledwaveout),
        .reset                                  (rst),
        .waveout                                        (ledout)
        );

endmodule
```

```
//Trevor Horine
//This module reads inputs from an NES controller
module nes (
        input logic clk,
        input logic reset,
        input logic nesdata,
        output logic nesclk,
        output logic neslatch,
        output logic up,
        output logic down,
        output logic left,
        output logic right,
        output logic start,
        output logic select,
        output logic a,
        output logic b);

        wire [4:0] count;
        wire gt;
        wire srst;
        wire rst;

nescounter horine1(
.clk            (clk),
.reset  (rst),
.count  (count)
);

comparator #(.N(5), .M(19)) horine2(
.a              (count),
.gt     (gt)
);

sync horine3(
.clk                            (clk),
.d                                      (gt),
.q                                      (srst)
);

countreset horine4(
.reset                  (reset),
.comparrst              (srst),
.rst                            (rst)
);

neslatchmod horine5(
.count          (count),
.neslatch       (neslatch)
);

nesclkmod horine6(
.count          (count),
.nesclk         (nesclk)
);

read horine7(
.data                   (nesdata),
.reset          (rst),
.count          (count),
.buttons                ({right, left, down, up, start, select, b, a})
);
endmodule


module nescounter(
  input logic clk, reset,
  output logic [4:0] count);

  always_ff @ (posedge clk, posedge reset)
    if(reset) count <= 7'b0;
    else count <= count + 1;
endmodule

module neslatchmod(
        input logic [4:0] count,
        output logic neslatch);

        always_comb
                case(count)
                        5'd1: neslatch = 1'b1;
                        5'd2: neslatch = 1'b1;
                        default: neslatch = 1'b0;
                endcase
endmodule

module nesclkmod (
        input logic [4:0] count,
        output logic nesclk);

        always_comb
                case(count)
                        5'd4: nesclk = 1'b1;
                        5'd6: nesclk = 1'b1;
                        5'd8: nesclk = 1'b1;
                        5'd10: nesclk = 1'b1;
                        5'd12: nesclk = 1'b1;
                        5'd14: nesclk = 1'b1;
                        5'd16: nesclk = 1'b1;
                        5'd18: nesclk = 1'b1;
                        default: nesclk = 1'b0;
                endcase
endmodule

module read (
        input logic data,
        input logic reset,
        input logic [4:0]count,
        output logic [7:0] buttons);

        always_ff @ (posedge count[0], posedge reset)
                if(reset) buttons <= 8'b0;
                else case(count)
                        5'd3: buttons[0] <= ! data; //a
                        5'd5: buttons[1] <= ! data; //b
                        5'd7: buttons[2] <= ! data; //select
                        5'd9: buttons[3] <= ! data; //start
                        5'd11: buttons[4] <= ! data; //up
                        5'd13: buttons[5] <= ! data; //down
                        5'd15: buttons[6] <= ! data; //left
                        5'd17: buttons[7] <= ! data; //right
                        default: buttons <= buttons;
                endcase
endmodule
```

```
//Trevor Horine
//This is a syncronizer
module sync(input logic clk,
                input logic d,
                output logic q);

        logic n1;

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

```
//Trevor Horine
//This module puts all of the converted 4 bit strings together.
module test (input logic [23:0] grb,
                              input logic reset,
                              input logic read,
                              output logic [95:0] ledwaveout);

               wire [95:0] ledw;

oneorzero horine1(
.colorbit       (grb[23]),
.ledlogicvalue  (ledw[95:92])
);

oneorzero horine2(
.colorbit       (grb[22]),
.ledlogicvalue  (ledw[91:88])
);

oneorzero horine3(
.colorbit       (grb[21]),
.ledlogicvalue  (ledw[87:84])
);

oneorzero horine4(
.colorbit       (grb[20]),
.ledlogicvalue  (ledw[83:80])
);

oneorzero horine5(
.colorbit       (grb[19]),
.ledlogicvalue  (ledw[79:76])
);

oneorzero horine6(
.colorbit       (grb[18]),
.ledlogicvalue  (ledw[75:72])
);

oneorzero horine7(
.colorbit       (grb[17]),
.ledlogicvalue  (ledw[71:68])
);

oneorzero horine8(
.colorbit       (grb[16]),
.ledlogicvalue  (ledw[67:64])
);

oneorzero horine9(
.colorbit       (grb[15]),
.ledlogicvalue  (ledw[63:60])
);

oneorzero horine10(
.colorbit       (grb[14]),
.ledlogicvalue  (ledw[59:56])
);

oneorzero horine11(
.colorbit       (grb[13]),
.ledlogicvalue  (ledw[55:52])
);

oneorzero horine12(
.colorbit       (grb[12]),
.ledlogicvalue  (ledw[51:48])
);

oneorzero horine13(
.colorbit       (grb[11]),
.ledlogicvalue  (ledw[47:44])
);

oneorzero horine14(
.colorbit       (grb[10]),
.ledlogicvalue  (ledw[43:40])
);

oneorzero horine15(
.colorbit       (grb[9]),
.ledlogicvalue  (ledw[39:36])
);

oneorzero horine16(
.colorbit       (grb[8]),
.ledlogicvalue  (ledw[35:32])
);

oneorzero horine17(
.colorbit       (grb[7]),
.ledlogicvalue  (ledw[31:28])
);

oneorzero horine18(
.colorbit       (grb[6]),
.ledlogicvalue  (ledw[27:24])
);

oneorzero horine19(
.colorbit       (grb[5]),
.ledlogicvalue  (ledw[23:20])
);

oneorzero horine20(
.colorbit       (grb[4]),
.ledlogicvalue  (ledw[19:16])
);

oneorzero horine21(
.colorbit       (grb[3]),
.ledlogicvalue  (ledw[15:12])
);

oneorzero horine22(
.colorbit       (grb[2]),
.ledlogicvalue  (ledw[11:8])
);

oneorzero horine23(
.colorbit       (grb[1]),
.ledlogicvalue  (ledw[7:4])
);

oneorzero horine24(
.colorbit       (grb[0]),
.ledlogicvalue  (ledw[3:0])
);

getdata horine25(
.read                           (read),
.ledw                           (ledw),
.reset                  (reset),
.ledwaveout             (ledwaveout)
);

endmodule

module oneorzero (input logic colorbit,
                                                output logic [3:0] ledlogicvalue);

        always_comb
                case (colorbit)
                        1'b1: ledlogicvalue = 1100;
                        1'b0: ledlogicvalue = 1000;
                        default: ledlogicvalue = 0000;
                endcase
endmodule

module getdata (input logic read,
                                        input logic [95:0] ledw,
                                        input logic reset,
                                        output logic [95:0] ledwaveout);

        always_ff @(negedge read, posedge reset)
                if(reset) ledwaveout <= 96'b0;
                else case(read)
                        9'b0: ledwaveout <= ledw;
                        default: ledwaveout <= 0;
                endcase
endmodule
```

```
//Trevor Horine
//This module takes the 96 bit signal to a serial output
module waveout (input logic [8:0]count,
                                        input logic clk,
                                        input logic [95:0] ledwave,
                                        input logic reset,
                                        output logic waveout);

        always_ff @ (posedge clk, posedge reset)
    if(reset) waveout <= 1'b0;
    else case(count[8:0])
                9'd0: waveout <= ledwave[95];
9'd1: waveout <= ledwave[94];
9'd2: waveout <= ledwave[93];
9'd3: waveout <= ledwave[92];
9'd4: waveout <= ledwave[91];
9'd5: waveout <= ledwave[90];
9'd6: waveout <= ledwave[89];
9'd7: waveout <= ledwave[88];
9'd8: waveout <= ledwave[87];
9'd9: waveout <= ledwave[86];
9'd10: waveout <= ledwave[85];
9'd11: waveout <= ledwave[84];
9'd12: waveout <= ledwave[83];
9'd13: waveout <= ledwave[82];
9'd14: waveout <= ledwave[81];
9'd15: waveout <= ledwave[80];
9'd16: waveout <= ledwave[79];
9'd17: waveout <= ledwave[78];
9'd18: waveout <= ledwave[77];
9'd19: waveout <= ledwave[76];
9'd20: waveout <= ledwave[75];
9'd21: waveout <= ledwave[74];
9'd22: waveout <= ledwave[73];
9'd23: waveout <= ledwave[72];
9'd24: waveout <= ledwave[71];
9'd25: waveout <= ledwave[70];
9'd26: waveout <= ledwave[69];
9'd27: waveout <= ledwave[68];
9'd28: waveout <= ledwave[67];
9'd29: waveout <= ledwave[66];
9'd30: waveout <= ledwave[65];
9'd31: waveout <= ledwave[64];
9'd32: waveout <= ledwave[63];
9'd33: waveout <= ledwave[62];
9'd34: waveout <= ledwave[61];
9'd35: waveout <= ledwave[60];
9'd36: waveout <= ledwave[59];
9'd37: waveout <= ledwave[58];
9'd38: waveout <= ledwave[57];
9'd39: waveout <= ledwave[56];
9'd40: waveout <= ledwave[55];
9'd41: waveout <= ledwave[54];
9'd42: waveout <= ledwave[53];
9'd43: waveout <= ledwave[52];
9'd44: waveout <= ledwave[51];
9'd45: waveout <= ledwave[50];
9'd46: waveout <= ledwave[49];
9'd47: waveout <= ledwave[48];
9'd48: waveout <= ledwave[47];
9'd49: waveout <= ledwave[46];
9'd50: waveout <= ledwave[45];
9'd51: waveout <= ledwave[44];
9'd52: waveout <= ledwave[43];
9'd53: waveout <= ledwave[42];
9'd54: waveout <= ledwave[41];
9'd55: waveout <= ledwave[40];
9'd56: waveout <= ledwave[39];
9'd57: waveout <= ledwave[38];
```

```
9'd58: waveout <= ledwave[37];
9'd59: waveout <= ledwave[36];
9'd60: waveout <= ledwave[35];
9'd61: waveout <= ledwave[34];
9'd62: waveout <= ledwave[33];
9'd63: waveout <= ledwave[32];
9'd64: waveout <= ledwave[31];
9'd65: waveout <= ledwave[30];
9'd66: waveout <= ledwave[29];
9'd67: waveout <= ledwave[28];
9'd68: waveout <= ledwave[27];
9'd69: waveout <= ledwave[26];
9'd70: waveout <= ledwave[25];
9'd71: waveout <= ledwave[24];
9'd72: waveout <= ledwave[23];
9'd73: waveout <= ledwave[22];
9'd74: waveout <= ledwave[21];
9'd75: waveout <= ledwave[20];
9'd76: waveout <= ledwave[19];
9'd77: waveout <= ledwave[18];
9'd78: waveout <= ledwave[17];
9'd79: waveout <= ledwave[16];
9'd80: waveout <= ledwave[15];
9'd81: waveout <= ledwave[14];
9'd82: waveout <= ledwave[13];
9'd83: waveout <= ledwave[12];
9'd84: waveout <= ledwave[11];
9'd85: waveout <= ledwave[10];
9'd86: waveout <= ledwave[9];
9'd87: waveout <= ledwave[8];
9'd88: waveout <= ledwave[7];
9'd89: waveout <= ledwave[6];
9'd90: waveout <= ledwave[5];
9'd91: waveout <= ledwave[4];
9'd92: waveout <= ledwave[3];
9'd93: waveout <= ledwave[2];
9'd94: waveout <= ledwave[1];
9'd95: waveout <= ledwave[0];
default: waveout  = 0;
        endcase
endmodule
```

A.2 PS/2 Keyboard Input - Square Wave Audio Output

```
// Copyright (C) 2018  Intel Corporation. All rights reserved.
// Your use of Intel Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Intel Program License
// Subscription Agreement, the Intel Quartus Prime License Agreement,
// the Intel FPGA IP License Agreement, or other applicable license
// agreement, including, without limitation, that your use is for
// the sole purpose of programming logic devices manufactured by
// Intel and sold by Intel or its authorized distributors.  Please
// refer to the applicable agreement for further details.

// PROGRAM              "Quartus Prime"
// VERSION              "Version 18.0.0 Build 614 04/24/2018 SJ Lite Edition"
// CREATED              "Tue Jun 02 16:32:32 2020"

module KeyboardtoSquareWave(
        resetSwtich,
        Clock50MHz,
        Data,
        ClockKeyboard,
        ResetKeyboard,
        AudioOutput
);


input wire      resetSwtich;
input wire      Clock50MHz;
input wire      Data;
input wire      ClockKeyboard;
input wire      ResetKeyboard;
output wire     AudioOutput;

wire    [7:0] SYNTHESIZED_WIRE_0;





SquareWaveOutput        b2v_inst(
        .Clock50MHz(Clock50MHz),
        .resetSwitch(resetSwtich),
        .Keyboard(SYNTHESIZED_WIRE_0),
        .audioOuput(AudioOutput));


keyboard_press_driver   b2v_inst2(
        .CLOCK_50(Clock50MHz),
        .PS2_DAT(Data),
        .PS2_CLK(ClockKeyboard),
        .reset(ResetKeyboard),


        .outCode(SYNTHESIZED_WIRE_0));
        defparam        b2v_inst2.FIRST = 1'b0;
        defparam        b2v_inst2.NULL = 8'b00000000;
        defparam        b2v_inst2.SEENF0 = 1'b1;


endmodule
```

```
// Copyright (C) 2018  Intel Corporation. All rights reserved.
// Your use of Intel Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Intel Program License
// Subscription Agreement, the Intel Quartus Prime License Agreement,
// the Intel FPGA IP License Agreement, or other applicable license
// agreement, including, without limitation, that your use is for
// the sole purpose of programming logic devices manufactured by
// Intel and sold by Intel or its authorized distributors.  Please
// refer to the applicable agreement for further details.

// PROGRAM              "Quartus Prime"
// VERSION              "Version 18.0.0 Build 614 04/24/2018 SJ Lite Edition"
// CREATED              "Mon Jun 01 17:54:08 2020"

module SquareWaveOutput(
        resetSwitch,
        Clock50MHz,
        Keyboard,
        audioOuput
);


input wire      resetSwitch;
input wire      Clock50MHz;
input wire      [7:0] Keyboard;
output wire     audioOuput;

wire    clockInWire;
wire    resetWire;
wire    SYNTHESIZED_WIRE_0;
wire    [11:0] SYNTHESIZED_WIRE_1;
wire    [11:0] SYNTHESIZED_WIRE_2;
wire    SYNTHESIZED_WIRE_3;
wire    SYNTHESIZED_WIRE_8;
wire    SYNTHESIZED_WIRE_5;
wire    SYNTHESIZED_WIRE_7;
reg     DFF_inst7;

assign  audioOuput = DFF_inst7;




sync    b2v_inst(
        .clk(Clock50MHz),
        .d(SYNTHESIZED_WIRE_0),
        .q(SYNTHESIZED_WIRE_8));


clockDivider1MHz        b2v_inst2(
        .Clock50MHz(Clock50MHz),
        .resetBut(resetSwitch),
        .Clock1MHz(clockInWire));


comparatortwoInputs     b2v_inst3(
        .a(SYNTHESIZED_WIRE_1),
        .b(SYNTHESIZED_WIRE_2),
        .eq(SYNTHESIZED_WIRE_0)




        );


counter12       b2v_inst4(
        .clk(clockInWire),
        .reset(resetWire),
        .q(SYNTHESIZED_WIRE_1));

assign  resetWire = SYNTHESIZED_WIRE_3 | SYNTHESIZED_WIRE_8 | resetSwitch;


keyboardInputDecoder    b2v_inst6(
        .data(Keyboard),
        .reset(SYNTHESIZED_WIRE_3),
        .countValue(SYNTHESIZED_WIRE_2));


always@(posedge SYNTHESIZED_WIRE_8 or negedge SYNTHESIZED_WIRE_5)
begin
if (!SYNTHESIZED_WIRE_5)
        begin
        DFF_inst7 <= 0;
        end
else
        begin
        DFF_inst7 <= SYNTHESIZED_WIRE_7;
        end
end

assign  SYNTHESIZED_WIRE_7 =  ~DFF_inst7;

assign  SYNTHESIZED_WIRE_5 =  ~resetSwitch;


endmodule
```

```
// Copyright (C) 2018  Intel Corporation. All rights reserved.
// Your use of Intel Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Intel Program License
// Subscription Agreement, the Intel Quartus Prime License Agreement,
// the Intel FPGA IP License Agreement, or other applicable license
// agreement, including, without limitation, that your use is for
// the sole purpose of programming logic devices manufactured by
// Intel and sold by Intel or its authorized distributors.  Please
// refer to the applicable agreement for further details.

// PROGRAM              "Quartus Prime"
// VERSION              "Version 18.0.0 Build 614 04/24/2018 SJ Lite Edition"
// CREATED              "Mon Jun 01 17:53:29 2020"

```

```
module clockDivider1MHz(
        resetBut,
        Clock50MHz,
        Clock1MHz
);


input wire      resetBut;
input wire      Clock50MHz;
output wire     Clock1MHz;

wire    syncEnd;
wire    [7:0] SYNTHESIZED_WIRE_0;
wire    SYNTHESIZED_WIRE_1;
wire    SYNTHESIZED_WIRE_2;





testing b2v_inst(
        .clock(syncEnd),
        .reset(resetBut),
        .q(Clock1MHz));


comparator10    b2v_inst2(
        .a(SYNTHESIZED_WIRE_0),
        .eq(SYNTHESIZED_WIRE_1)




        );
        defparam        b2v_inst2.b = 50;


sync    b2v_inst3(
        .clk(Clock50MHz),
        .d(SYNTHESIZED_WIRE_1),
        .q(syncEnd));


counter8        b2v_inst4(
        .clk(Clock50MHz),
        .reset(SYNTHESIZED_WIRE_2),
        .q(SYNTHESIZED_WIRE_0));

assign  SYNTHESIZED_WIRE_2 = syncEnd | resetBut;


endmodule
```

```
module counter8 (
        input logic clk,
        input logic reset,
        output logic [7:0] q
        );

always_ff @(posedge clk, posedge reset)

if (reset) q <= 0;

else q <= q + 1;

endmodule
```

```
module comparator10#(parameter b = 10)(
        input logic [7:0] a,
        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 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
```

```
module alternating(
        input logic clock,
        input logic reset,
        output logic q
        );

        logic d;

        always_ff@(posedge clock, posedge reset)
                if(reset)
                        begin
                                d <= 0;
                        end
                else
                        begin
```

```
                                d <= q;
                        end

        always_ff@(negedge clock, posedge reset)
                if(reset)
                        begin
                                q <= 1;
                        end
                else
                        begin
                                q <= ~d;
                        end
endmodule
```

```
module keyboardInputDecoder (
        input logic [7:0] data,
        output logic [11:0] countValue,
        output logic reset
        );

        always_comb
                case(data)

                /*C*/8'b0010_0001: begin countValue = 12'b0111_1010_1101; reset = 0; end
                /*D*/8'b0010_0011: begin countValue = 12'b0110_1010_0110; reset = 0; end
                /*E*/8'b0010_0100: begin countValue = 12'b0101_1110_1100; reset = 0; end
                /*F*/8'b0010_1011: begin countValue = 12'b0101_1001_0111; reset = 0; end
                /*G*/8'b0011_0100: begin countValue = 12'b0100_1111_1011; reset = 0; end
                /*A*/8'b0001_1100: begin countValue = 12'b0100_0110_1111; reset = 0; end
                /*B*/8'b0011_0010: begin countValue = 12'b0011_1111_0100; reset = 0; end
                /*Key_Break*/8'b1111_0000: begin countValue = '1; reset = 1; end
                default: begin countValue = '1; reset = 0; end

                endcase
        endmodule
```

```
module counter12 (
        input logic clk,
        input logic reset,
        output logic [11:0] q
        );

always_ff @(posedge clk, posedge reset)

if (reset) q <= 0;

else q <= q + 1;

endmodule
```

```
module comparatortwoInputs(
        input logic [11:0] a,
        input logic [11:0] 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 keyboard_press_driver(
  input  CLOCK_50,
  output reg valid, makeBreak,
  output reg [7:0] outCode,
  input    PS2_DAT, // PS2 data line
  input    PS2_CLK, // PS2 clock line
  input reset
);

parameter FIRST = 1'b0, SEENF0 = 1'b1;
reg state;
reg [1:0] count;
wire [7:0] scan_code;
reg [7:0] filter_scan;
wire scan_ready;
reg read;
parameter NULL = 8'h00;

initial
begin
        state = FIRST;
        filter_scan = NULL;
        read = 1'b0;
        count = 2'b00;
end


// inner driver that handles the PS2 keyboard protocol
// outputs a scan_ready signal accompanied with a new scan_code
keyboard_inner_driver kbd(
  .keyboard_clk(PS2_CLK),
  .keyboard_data(PS2_DAT),
  .clock50(CLOCK_50),
  .reset(reset),
  .read(read),
  .scan_ready(scan_ready),
  .scan_code(scan_code)
);

always @(posedge CLOCK_50)
        case(count)
                2'b00:
                        if(scan_ready)
                                count <= 2'b01;
                2'b01:
                        if(scan_ready)
                                count <= 2'b10;
                2'b10:
```

```
            begin
                read <= 1'b1;
                count <= 2'b11;
                valid <= 0;
                outCode <= scan_code;
                case(state)
                    FIRST:
                        case(scan_code)
                            8'hF0:
                                begin
                                    state <= SEENF0;
                                end
                            8'hE0:
                                begin
                                    state <= FIRST;
                                end
                            default:
                                begin
                                    filter_scan <= scan_code;
                                    if(filter_scan != scan_code)
                                        begin
                                            valid <= 1'b1;
                                            makeBreak <= 1'b1;
                                        end
                                end
                        endcase
                    SEENF0:
                        begin
                            state <= FIRST;
                            if(filter_scan == scan_code)
                                begin
                                    filter_scan <= NULL;
                                end
                            valid <= 1'b1;
                            makeBreak <= 1'b0;
                        end
                endcase
            end
        2'b11:
            begin
                read <= 1'b0;
                count <= 2'b00;
                valid <= 0;
            end
    endcase
endmodule
```

```
module keyboard_inner_driver(keyboard_clk, keyboard_data, clock50, reset, read, scan_ready, scan_code);
input keyboard_clk;
input keyboard_data;
input clock50; // 50 Mhz system clock
input reset;
input read;
output scan_ready;
output [7:0] scan_code;
reg ready_set;
reg [7:0] scan_code;
reg scan_ready;
reg read_char;
reg clock; // 25 Mhz internal clock

reg [3:0] incnt;
reg [8:0] shiftin;

reg [7:0] filter;
reg keyboard_clk_filtered;

// scan_ready is set to 1 when scan_code is available.
// user should set read to 1 and then to 0 to clear scan_ready

always @ (posedge ready_set or posedge read)
if (read == 1) scan_ready <= 0;
else scan_ready <= 1;

// divide-by-two 50MHz to 25MHz
always @(posedge clock50)
    clock <= ~clock;



// This process filters the raw clock signal coming from the keyboard
// using an eight-bit shift register and two AND gates

always @(posedge clock)
begin
   filter <= {keyboard_clk, filter[7:1]};
   if (filter==8'b1111_1111) keyboard_clk_filtered <= 1;
   else if (filter==8'b0000_0000) keyboard_clk_filtered <= 0;
end


// This process reads in serial data coming from the terminal

always @(posedge keyboard_clk_filtered)
begin
   if (reset==1)
   begin
      incnt <= 4'b0000;
      read_char <= 0;
   end
   else if (keyboard_data==0 && read_char==0)
   begin
    read_char <= 1;
    ready_set <= 0;
   end
   else
   begin
       // shift in next 8 data bits to assemble a scan code
       if (read_char == 1)
           begin
              if (incnt < 9)
              begin
                incnt <= incnt + 1'b1;
                shiftin = { keyboard_data, shiftin[8:1]};
                ready_set <= 0;
            end
        else
            begin
                incnt <= 0;
                scan_code <= shiftin[7:0];
                read_char <= 0;
                ready_set <= 1;
            end
        end
    end
end

endmodule
```

A.3 Infrared Receiver Input - Seven Segment Display Output

```
//IR Reader
//By: Caden Friesen


module IRreader(
                input logic resetbutton,
                input logic ir,
                input logic clk,
                output logic [6:0] dig0, dig1, dig2, dig3,
                output logic seg1, seg2, seg3, seg4, seg5, seg6, seg7, seg8);

logic leadcounterreset;
logic [21:0] leadtimed;
logic [21:0] dffleadtimed;
logic [21:0] greaterthanlead;
logic [21:0] lessthanlead;
logic leadconfirm;
logic enablerout;
logic [5:0] countthirtytwoout;
logic [5:0] zero;
logic [5:0] threetwo;
logic combinedenabler;
logic thirtytwoconfirmed;
logic [21:0] timedpulse;
logic [21:0] timepulseflopped;
logic onecompared;
logic zerocompared;
logic [21:0] greaterone;
logic [21:0] greaterzero;
logic[21:0] lesszero;
logic [21:0] lessone;
logic resultflopclk;
logic resultflopout;
logic [31:0] shiftout;
logic [31:0] shiftacceptout;
logic [31:0] decoderinput;
logic [6:0] digzero;
logic [6:0] digone;
logic [6:0] digtwo;
logic [6:0] digthree;
logic sameaddress;
logic samecommand;
logic sameboth;
logic neenabler;
logic oneorzero;
logic resetterout;
logic pulsecounterreset;
logic enablerreset;



//equal to 14ms in clock periods which are 20ns (700,000)
assign greaterthanlead = 22'b0010101010111001100000;

//equal to 13 ms in clock periods which are 20ns (650,000)
assign lessthanlead    = 22'b0010011110101100010000;

//assigns the range for the count to thirty two
assign threetwo = 6'b100001;
assign zero =     6'b000000;


assign leadcounterreset = ~resetbutton || ir;

assign combinedenabler = thirtytwoconfirmed && enablerout;

assign pulsecounterreset = enablerout && ir;

//125000 clock cycles 2.5 ms
assign greaterone =  22'b0000011110100001001000;
//100000 clock cycles 2.0 ms
assign lessone =     22'b0000011000011010100000;

//68750 clock cycles 1.375 ms
assign greaterzero = 22'b0000010000110010001110;
//43750 clock cycles .875 ms
assign lesszero =    22'b0000001010101011100110;

assign oneorzero = onecompared ^ zerocompared;
assign resultflopclk = oneorzero && ir;


assign enablerreset = ~resetbutton;

//check for the inverse address and commands received
assign sameaddress = (shiftacceptout[31] ^ shiftacceptout[23]) && (shiftacceptout[30] ^ shiftacceptout
     [22]) && (shiftacceptout[29] ^ shiftacceptout[21]) && (shiftacceptout[28] ^ shiftacceptout[20]) &&
     (shiftacceptout[27] ^ shiftacceptout[19]) && (shiftacceptout[26] ^ shiftacceptout[18]) && (
     shiftacceptout[25] ^ shiftacceptout[17]) && (shiftacceptout[24] ^ shiftacceptout[16]); //
     shiftacceptout[31:24] ^ shiftacceptout[23:16];
assign samecommand = (shiftacceptout[15] ^ shiftacceptout[7]) && (shiftacceptout[14] ^ shiftacceptout
     [6]) && (shiftacceptout[13] ^ shiftacceptout[5]) && (shiftacceptout[12] ^ shiftacceptout[4]) && (
     shiftacceptout[11] ^ shiftacceptout[3]) && (shiftacceptout[10] ^ shiftacceptout[2]) && (
     shiftacceptout[9] ^ shiftacceptout[1]) && (shiftacceptout[8] ^ shiftacceptout[0]);
          //shiftacceptout[15:8] ^ shiftacceptout[7:0];
assign sameboth = samecommand && sameaddress;

```

```
//lights up NE if not inverses
assign neenabler = (~thirtytwoconfirmed) ^ sameboth;
assign seg1 = ~neenabler;
assign seg2 = ~neenabler;
assign seg3 = ~neenabler;
assign seg4 = ~neenabler;
assign seg5 = ~neenabler;
assign seg6 = ~neenabler;
assign seg7 = ~neenabler;
assign seg8 = ~neenabler;


//Search for lead code section
counterir #(22) leadcounter(clk, leadcounterreset, leadtimed);
twotwobitreg leadtocomp (ir, ~resetbutton, leadtimed, dffleadtimed);
comparatorir leadcomp(dffleadtimed, greaterthanlead, lessthanlead, leadconfirm);
myDFF enablerdff(leadconfirm, leadconfirm, enablerreset, enablerout);
myDFF enablerresetter(enablerout, enablerout, resetterout, resetterout);


//counts to thirty two to see when code should be done
counterir #(6) countthirtytwo(ir, enablerout, countthirtytwoout);
comparatorir #(6) comparethirtytwo(countthirtytwoout, threetwo, zero, thirtytwoconfirmed);

//Counts length of ir pulses
counterir #(22) pulsecounter(clk, pulsecounterreset, timedpulse);
twotwobitenabledreg pulsereg(ir, enablerout, combinedenabler, timedpulse, timepulseflopped);

//checks if ir pulse is zero or one
comparatorir zerocompare (timepulseflopped, greaterzero, lesszero, zerocompared);
comparatorir onecompare (timepulseflopped, greaterone, lessone, onecompared);


//shift register
myDFF resultflop(onecompared , resultflopclk, enablerout, resultflopout);
shiftregister theshifter(ir, enablerout, resultflopout, shiftout);
threetwobitreg shiftaccepter(~thirtytwoconfirmed, enablerout, shiftout, shiftacceptout);



threetwobitreg decoderregister(sameboth, enablerout, shiftout, decoderinput);

//seven segment decoders
sevenseg digzerodecode(decoderinput [11:8], dig0);
sevenseg digonedecode(decoderinput [15:12], dig1);
sevenseg digtwodecode(decoderinput [27:24], dig2);
sevenseg digthreedecode(decoderinput [31:28], dig3);

endmodule
```

```
//Counter
//By: Caden Friesen

module counterir #(parameter N=22)
        (input logic clk,
        input logic reset,
        output logic [N-1:0] B);



always_ff@(posedge clk)
                        B<=B+1;

always_ff@(posedge reset)
                        B<=0;
endmodule
```

```
//22 bit enabled register
////By: Caden Friesen

module twotwobitenabledreg(input logic clk, reset, enabled,
                input logic [21:0] datain,
                output logic [21:0] dataout);

        enabledDFF s1 (datain[0], clk, reset, enabled, dataout[0]);
        enabledDFF s2 (datain[1], clk, reset, enabled, dataout[1]);
        enabledDFF s3 (datain[2], clk, reset, enabled, dataout[2]);
        enabledDFF s4 (datain[3], clk, reset, enabled, dataout[3]);
        enabledDFF s5 (datain[4], clk, reset, enabled, dataout[4]);
        enabledDFF s6 (datain[5], clk, reset, enabled, dataout[5]);
        enabledDFF s7 (datain[6], clk, reset, enabled, dataout[6]);
        enabledDFF s8 (datain[7], clk, reset, enabled, dataout[7]);
        enabledDFF s9 (datain[8], clk, reset, enabled, dataout[8]);
        enabledDFF s10 (datain[9], clk, reset, enabled, dataout[9]);
        enabledDFF s11 (datain[10], clk, reset, enabled, dataout[10]);
        enabledDFF s12 (datain[11], clk, reset, enabled, dataout[11]);
        enabledDFF s13 (datain[12], clk, reset, enabled, dataout[12]);
        enabledDFF s14 (datain[13], clk, reset, enabled, dataout[13]);
        enabledDFF s15 (datain[14], clk, reset, enabled, dataout[14]);
        enabledDFF s16 (datain[15], clk, reset, enabled, dataout[15]);
        enabledDFF s17 (datain[16], clk, reset, enabled, dataout[16]);
        enabledDFF s18 (datain[17], clk, reset, enabled, dataout[17]);
        enabledDFF s19 (datain[18], clk, reset, enabled, dataout[18]);
        enabledDFF s20 (datain[19], clk, reset, enabled, dataout[19]);
        enabledDFF s21 (datain[20], clk, reset, enabled, dataout[20]);
        enabledDFF s22 (datain[21], clk, reset, enabled, dataout[21]);
endmodule
```

```
//22 Bit register
//By: Caden Friesen

module twotwobitreg(input logic clk, reset,
                input logic [21:0] datain,
                output logic [21:0] dataout);

        myDFF s1 (datain[0], clk, reset, dataout[0]);
        myDFF s2 (datain[1], clk, reset, dataout[1]);
        myDFF s3 (datain[2], clk, reset, dataout[2]);
        myDFF s4 (datain[3], clk, reset, dataout[3]);
        myDFF s5 (datain[4], clk, reset, dataout[4]);
        myDFF s6 (datain[5], clk, reset, dataout[5]);
        myDFF s7 (datain[6], clk, reset, dataout[6]);
        myDFF s8 (datain[7], clk, reset, dataout[7]);
        myDFF s9 (datain[8], clk, reset, dataout[8]);
        myDFF s10 (datain[9], clk, reset, dataout[9]);
        myDFF s11 (datain[10], clk, reset, dataout[10]);
        myDFF s12 (datain[11], clk, reset, dataout[11]);
        myDFF s13 (datain[12], clk, reset, dataout[12]);
        myDFF s14 (datain[13], clk, reset, dataout[13]);
        myDFF s15 (datain[14], clk, reset, dataout[14]);
        myDFF s16 (datain[15], clk, reset, dataout[15]);
        myDFF s17 (datain[16], clk, reset, dataout[16]);
        myDFF s18 (datain[17], clk, reset, dataout[17]);
        myDFF s19 (datain[18], clk, reset, dataout[18]);
        myDFF s20 (datain[19], clk, reset, dataout[19]);
        myDFF s21 (datain[20], clk, reset, dataout[20]);
        myDFF s22 (datain[21], clk, reset, dataout[21]);
endmodule
```

```
//32 Bit Register
//By: Caden Friesen

module threetwobitreg(input logic clk, reset,
                input logic [31:0] datain,
                output logic [31:0] dataout);

        myDFF s1 (datain[0], clk, reset, dataout[0]);
        myDFF s2 (datain[1], clk, reset, dataout[1]);
        myDFF s3 (datain[2], clk, reset, dataout[2]);
        myDFF s4 (datain[3], clk, reset, dataout[3]);
        myDFF s5 (datain[4], clk, reset, dataout[4]);
        myDFF s6 (datain[5], clk, reset, dataout[5]);
        myDFF s7 (datain[6], clk, reset, dataout[6]);
        myDFF s8 (datain[7], clk, reset, dataout[7]);
        myDFF s9 (datain[8], clk, reset, dataout[8]);
        myDFF s10 (datain[9], clk, reset, dataout[9]);
        myDFF s11 (datain[10], clk, reset, dataout[10]);
        myDFF s12 (datain[11], clk, reset, dataout[11]);
        myDFF s13 (datain[12], clk, reset, dataout[12]);
        myDFF s14 (datain[13], clk, reset, dataout[13]);
        myDFF s15 (datain[14], clk, reset, dataout[14]);
        myDFF s16 (datain[15], clk, reset, dataout[15]);
        myDFF s17 (datain[16], clk, reset, dataout[16]);
        myDFF s18 (datain[17], clk, reset, dataout[17]);
        myDFF s19 (datain[18], clk, reset, dataout[18]);
        myDFF s20 (datain[19], clk, reset, dataout[19]);
        myDFF s21 (datain[20], clk, reset, dataout[20]);
        myDFF s22 (datain[21], clk, reset, dataout[21]);
        myDFF s23 (datain[22], clk, reset, dataout[22]);
        myDFF s24 (datain[23], clk, reset, dataout[23]);
        myDFF s25 (datain[24], clk, reset, dataout[24]);
        myDFF s26 (datain[25], clk, reset, dataout[25]);
        myDFF s27 (datain[26], clk, reset, dataout[26]);
        myDFF s28 (datain[27], clk, reset, dataout[27]);
        myDFF s29 (datain[28], clk, reset, dataout[28]);
        myDFF s30 (datain[29], clk, reset, dataout[29]);
        myDFF s31 (datain[30], clk, reset, dataout[30]);
        myDFF s32 (datain[31], clk, reset, dataout[31]);
endmodule
```

```
//Comparator
//By: Caden Friesen

module comparatorir#(parameter N= 22)
                (input logic [N-1:0] a, greatera, lessa,
                output logic inrange);
        logic lte;
        logic gte;
        assign lte = (a <= greatera);
        assign gte = (a >= lessa);
        assign inrange = lte && gte;
endmodule
```

```
//DFF
////By: Caden Friesen

module myDFF(input logic D,
                input logic clk,
                input logic reset,
                output logic Q);

        always@(posedge clk)
                        Q = D;

        always@(posedge reset)
                Q = 1'b0;
endmodule
```

```
//Seven Segment Decoder
//By: Caden Friesen

module sevenseg(input logic [3:0] data,
                output logic [6:0] segments);
        always_comb
                case(data)
                // gfe_dcba
                //Assigns an output bus depending on the input decimal number
                0: segments = 7'b100_0000;
                1: segments = 7'b111_1001;
                2: segments = 7'b0100_100;
                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;
                10: segments = 7'b000_1000;
                11: segments = 7'b000_0011;
                12: segments = 7'b100_0110;
                13: segments = 7'b010_0001;
                14: segments = 7'b000_0110;
                15: segments = 7'b000_1110;
                //default shows nothing
                default: segments = 7'b111_1111;
                endcase
endmodule
```

```
//Enabled DFF
//By: Caden Friesen

module enabledDFF(input logic D,
                input logic clk,
                input logic reset,
                input logic enabled,
                output logic Q);

        always@(posedge clk)
                        if(enabled)
                                Q<= D;

        always@(posedge reset)
                        Q <= 0;
endmodule
```

```
//Shift Register
//By: Caden Friesen

module shiftregister(input logic clk, reset,
                     input logic datain,
                     output logic [31:0] dataout);


        myDFF s0 (datain, clk, reset, dataout[0]);
        myDFF s1 (dataout[0], clk, reset, dataout[1]);
        myDFF s2 (dataout[1], clk, reset, dataout[2]);
        myDFF s3 (dataout[2], clk, reset, dataout[3]);
        myDFF s4 (dataout[3], clk, reset, dataout[4]);
        myDFF s5 (dataout[4], clk, reset, dataout[5]);
        myDFF s6 (dataout[5], clk, reset, dataout[6]);
        myDFF s7 (dataout[6], clk, reset, dataout[7]);
        myDFF s8 (dataout[7], clk, reset, dataout[8]);
        myDFF s9 (dataout[8], clk, reset, dataout[9]);
        myDFF s10 (dataout[9], clk, reset, dataout[10]);
        myDFF s11 (dataout[10], clk, reset, dataout[11]);
        myDFF s12 (dataout[11], clk, reset, dataout[12]);
        myDFF s13 (dataout[12], clk, reset, dataout[13]);
        myDFF s14 (dataout[13], clk, reset, dataout[14]);
        myDFF s15 (dataout[14], clk, reset, dataout[15]);
        myDFF s16 (dataout[15], clk, reset, dataout[16]);
        myDFF s17 (dataout[16], clk, reset, dataout[17]);
        myDFF s18 (dataout[17], clk, reset, dataout[18]);
        myDFF s19 (dataout[18], clk, reset, dataout[19]);
        myDFF s20 (dataout[19], clk, reset, dataout[20]);
        myDFF s21 (dataout[20], clk, reset, dataout[21]);
        myDFF s22 (dataout[21], clk, reset, dataout[22]);
        myDFF s23 (dataout[22], clk, reset, dataout[23]);
        myDFF s24 (dataout[23], clk, reset, dataout[24]);
        myDFF s25 (dataout[24], clk, reset, dataout[25]);
        myDFF s26 (dataout[25], clk, reset, dataout[26]);
        myDFF s27 (dataout[26], clk, reset, dataout[27]);
        myDFF s28 (dataout[27], clk, reset, dataout[28]);
        myDFF s29 (dataout[28], clk, reset, dataout[29]);
        myDFF s30 (dataout[29], clk, reset, dataout[30]);
        myDFF s31 (dataout[30], clk, reset, dataout[31]);
endmodule
```

B Simulation Files (Do scripts)

B.1 NES Controller Input - Motor and Addressable LED Output

```
restart -nowave -force
add wave -divider -height 30 <Inputs>
add wave rst
add wave clk50mhz
add wave data
add wave -divider -height 30 <Output_to_controller>
add wave neslatch
add wave nesclk
add wave -divider -height 30 <Outputs>
add wave outccw
add wave outcw
add wave leddata
add wave -divider -height 30 <Useful_Informations(Buttons)>
add wave a
add wave b
add wave select
add wave start
add wave up
add wave down
add wave left
add wave right

force rst 1
force clk50mhz 0
force data 1
run 10
force rst 0
force clk50mhz 1 0, 0 {1 ps} -r 2
force data 0 @ 19, 1 @ 25, 0 @ 169, 1 @ 175, 0 @ 319, 1 @ 325, 0 @ 469, 1 @ 475, 0 @ 619, 1 @ 625, 0 @
      769, 1 @ 775, 0 @ 919, 1 @ 925, 0 @ 1069, 1 @ 1075, 0 @ 1219, 1 @ 1225, 0 @ 1250
force data  0 @ 2500, 1 @ 2515, 0 @ 2638, 1 @ 2653, 0 @ 2776, 1 @ 2791, 0 @ 2914, 1 @ 2929, 0 @ 3052, 1
      @ 3067, 0 @ 3190, 1 @ 3205, 0 @ 3328, 1 @ 3343, 0 @ 3466, 1 @ 3481, 0 @ 3604, 1 @ 3619, 0 @ 3742, 1
       @ 3757, 0 @ 3880, 1 @ 3895, 0 @ 4018, 1 @ 4033, 0 @ 4156, 1 @ 4171, 0 @ 4294, 1 @ 4309, 0 @ 4432,
      1 @ 4447, 0 @ 4570, 1 @ 4585, 0 @ 4708, 1 @ 4723, 0 @ 4846, 1 @ 4861, 0 @ 4984, 1 @ 4999, 0 @ 5122,
       1 @ 5137
force data  0 @ 2551, 1 @ 2557, 0 @ 2689, 1 @ 2695, 0 @ 2827, 1 @ 2833, 0 @ 2965, 1 @ 2971, 0 @ 3103, 1
      @ 3109, 0 @ 3241, 1 @ 3247, 0 @ 3379, 1 @ 3385, 0 @ 3517, 1 @ 3523, 0 @ 3655, 1 @ 3661, 0 @ 3793, 1
       @ 3799, 0 @ 3931, 1 @ 3937, 0 @ 4069, 1 @ 4075, 0 @ 4207, 1 @ 4213, 0 @ 4345, 1 @ 4351, 0 @ 4483,
      1 @ 4489, 0 @ 4621, 1 @ 4627, 0 @ 4759, 1 @ 4765, 0 @ 4897, 1 @ 4903, 0 @ 5035, 1 @ 5041, 0 @ 5173,
       1 @ 5179
run 6000
```

```
restart -force -nowave
add wave -divider -height 30 <Input>
add wave clk
add wave reset
add wave -divider -height 30 <Output>
add wave newclk
add wave -divider -height 30 <Internal_Counter>
add wave t
radix signal t unsigned

force reset 1
force clk 1 0, 0 {1ps} -r 2
run 2
force reset 0
run 66
```

```
restart -force -nowave
add wave -divider -height 30 <Inputs>
add wave count
radix signal count unsigned
add wave -divider -height 30 <Output>
add wave gt

force reset 1
force clk 1 0, 0 {1ps} -r 2
run 2
force reset 0
run 44
```

```
restart -force -nowave
add wave -divider -height 30 <Inputs>
add wave reset
add wave srst
add wave -divider -height 30 <Output>
add wave rst

force reset 1
force clk 1 0, 0 {1ps} -r 2
run 10
force reset 0
run 46
```

```
restart -force -nowave
add wave -divider -height 30 <Inputs>
add wave count
radix signal count unsigned
add wave -divider -height 30 <Output>
add wave nesclk

force reset 1
force clk 1 0, 0 {1ps} -r 2
run 2
force reset 0
run 44
```

```
restart -force -nowave
add wave -divider -height 30 <Inputs>
add wave clk
add wave reset
add wave -divider -height 30 <Output>
add wave count
radix signal count unsigned

force reset 1
force clk 1 0, 0 {1ps} -r 2
run 2
force reset 0
run 44
```

```
restart -force -nowave
add wave -divider -height 30 <Inputs>
add wave count
radix signal count unsigned
add wave -divider -height 30 <Output>
add wave neslatch

force reset 1
force clk 1 0, 0 {1ps} -r 2
run 2
force reset 0
run 44
```

```
restart -nowave -force
add wave -divider -height 30 <Inputs>
add wave rst
add wave count
radix signal count unsigned
add wave nesdata
add wave -divider -height 30 <Outputs>
add wave a
add wave b
add wave select
add wave start
add wave up
add wave down
add wave left
add wave right

force reset 1
force clk 0
force nesdata 1
run 10
force reset 0
force clk 1 0, 0 {3 ps} -r 6
force nesdata 0 @ 19, 1 @ 25, 0 @ 169, 1 @ 175, 0 @ 319, 1 @ 325, 0 @ 469, 1 @ 475, 0 @ 619, 1 @ 625, 0
    @ 769, 1 @ 775, 0 @ 919, 1 @ 925, 0 @ 1069, 1 @ 1075, 0 @ 1219, 1 @ 1225, 0 @ 1250
run 1372
```

```
restart -force -nowave
add wave -divider -height 30 <Inputs>
add wave gt
add wave clk
add wave -divider -height 30 <Output>
add wave srst

force reset 1
force clk 1 0, 0 {1ps} -r 2
run 2
force reset 0
run 46
```

```
restart -nowave -force
add wave -divider -height 30 <System_inputs>
add wave clk
add wave reset
add wave -divider -height 30 <User_inputs>
add wave colorbutton
add wave u
add wave d
add wave -divider -height 30 <Output>
add wave colorvalue
radix signal colorvalue unsigned

force reset 1
force clk 0
force colorbutton 0
force u 0
force d 0
run 10
force reset 0
force clk 0 @ 10, 1 @ 15, 0 @ 20, 1 @ 25 -r 20
run 10
force u 1
run 10
force u 0
force d 1
run 10
force d 0
force colorbutton 1
run 10
force colorbutton 0
force u 1
force d 1
run 10
force colorbutton 1
run 10
force d 0
run 2550
force u 0
force d 1
run 2550
```

```
restart -force -nowave
add wave -divider -height 30 <System_inputs>
add wave clk
add wave reset
add wave -divider -height 30 <GRB_value>
add wave count
radix signal count unsigned

force reset 1
force clk 0
force grb 0
run 10
force reset 0
force clk 1 0, 0 {1000 ps} -r 2ns
force grb 010001110001101010000101
run 500ns
```

```
restart -nowave -force
add wave -divider -height 30 <System_inputs>
add wave clk
add wave reset
add wave -divider -height 30 <User_inputs>
add wave gcolorbutton
add wave rcolorbutton
add wave bcolorbutton
add wave up
add wave down
add wave -divider -height 30 <Output>
add wave grbvalue

force reset 1
force clk 0
force gcolorbutton 0
force rcolorbutton 0
force bcolorbutton 0
force up 0
force down 0
run 10
force reset 0
force clk 0 @ 10, 1 @ 15, 0 @ 20, 1 @ 25 -r 20
force gcolorbutton 1
force up 1
run 2550
force up 0
force down 1
run 2550
force down 0
force gcolorbutton 0
force rcolorbutton 1
force up 1
run 2550
force up 0
force down 1
run 2550
force down 0
force rcolorbutton 0
force bcolorbutton 1
force up 1
run 2550
```

```
force up 0
force down 1
run 2550
force down 0
force bcolorbutton 0
force gcolorbutton 1
force rcolorbutton 1
force bcolorbutton 1
force up 1
run 2550
force up 0
force down 1
run 2550
force down 0
force gcolorbutton 0
force rcolorbutton 0
force bcolorbutton 0
```

```
restart -force -nowave
add wave -divider -height 30 <System_inputs>
add wave clk
add wave reset
add wave -divider -height 30 <GRB_value>
add wave grb
add wave -divider -height 30 <Output_data>
add wave ledout

force reset 1
force clk 0
force grb 0
run 10
force reset 0
force clk 1 0, 0 {1000 ps} -r 2ns
force grb 010001110001101010000101
run 500ns
```

```
restart -force -nowave
add wave -divider -height 30 <System_inputs>
add wave rst
add wave reset
add wave -divider -height 30 <GRB_value>
add wave grb
add wave -divider -height 30 <Output>
add wave ledwaveout

force reset 1
force clk 0
force grb 0
run 10
force reset 0
force clk 1 0, 0 {1000 ps} -r 2ns
force grb 010001110001101010000101
run 500ns
```

```
restart -force -nowave
add wave -divider -height 30 <System_inputs>
add wave clk
add wave reset
add wave count
add wave ledwaveout
radix signal count unsigned
add wave -divider -height 30 <Output>
add wave ledout

force reset 1
force clk 0
force grb 0
run 10
force reset 0
force clk 1 0, 0 {1000 ps} -r 2ns
force grb 010001110001101010000101
run 500ns
```

B.2 PS/2 Keyboard Input - Square Wave Audio Output

```
vsim.work.KeyboardtoSquareWave

add wave *
force resetSwtich 1 @ 1, 0 @ 2
force ResetKeyboard 1 @ 1, 0 @ 2
force Clock50MHz 0 @ 1, 1 @ 2 -r 2
force ClockKeyboard 0 @ 1, 1 @ 10 -r 10
force Data 1 @ 1
force Data 0 @ 10
force Data 0 @ 20
force Data 1 @ 30
force Data 0 @ 40
force Data 0 @ 50
force Data 0 @ 60
force Data 0 @ 70
force Data 1 @ 80
force Data 1 @ 90
run 20 us
```

```
vsim.work.SquareWaveOutput

add wave *
force Clock50MHz 0 @ 1, 1 @ 2 -r 2
force resetSwitch 1 @ 1, 0 @ 2
run 20
force Keyboard 00100001 @ 25
run 20
```

```
vsim.work.clockDivder1MHz

add wave *
force Clock50MHz 0 @ 1, 1 @ 2 -r 2
force resetBut 1 @ 1, 0 @ 2
run 20 us
```

```
vsim.work.counter8

add wave *
force clk 0 @ 1, 1 @ 2 -r 2
force reset 1 @ 1, 0 @ 2
run 20
```

```
vsim.work.comparator10

add wave *
force a 000000
run 10
force a 000010
run 10
force a 001010
run 10
```

```
vsim.work.sync

add wave *
force clk 0 @ 1, 1 @ 2 -r 2
force d 1 @ 0, 0 @ 10
run 20
```

```
vsim.work.alternating

add wave *
force clock 0 @ 1, 1 @ 2 -r 2
force reset 1 @ 1, 0 @ 2
force reset 1 @ 5, 0 @ 10 -r 5
run 20
```

```
vsim.work.keyboardInputDecoder

add wave *
force data 00100001
run 20
force data 00110100
run 20
```

```
vsim.work.counter12

add wave *
force clk 0 @ 1, 1 @ 2 -r 2
force reset 1 @ 1, 0 @ 2
run 20
```

```
vsim.work.comparatortwoInputs

add wave *
force a 001010
force b 000111
run 20
force a 000011
force b 000011
run 20
force a 01111
force b 11111
run 20
```

```
vsim.work.keyboard_press_driver

add wave *
force CLOCK_50 0 @ 1, 1 @ 2 -r 2
force reset 1 @ 1, 0 @ 2
force PS2_CLK 0 @ 1, 1 @ 10 -r 10
force PS2_DAT 1 @ 1
force PS2_DAT 0 @ 10
force PS2_DAT 0 @ 20
force PS2_DAT 1 @ 30
force PS2_DAT 0 @ 40
force PS2_DAT 0 @ 50
force PS2_DAT 0 @ 60
force PS2_DAT 0 @ 70
force PS2_DAT 1 @ 80
force PS2_DAT 1 @ 90
run 100
```

```
vsim.work.keyboard_inner_driver

add wave *
force clock50 0 @ 1, 1 @ 2 -r 2
force reset 1 @ 1, 0 @ 2
force read 1 @ 1, 0 @ 2
force keyboard_clk 0 @ 1, 1 @ 10 -r 10
force keyboard_data 1 @ 1
force keyboard_data 0 @ 10
force keyboard_data 0 @ 20
force keyboard_data 1 @ 30
force keyboard_data 0 @ 40
force keyboard_data 0 @ 50
force keyboard_data 0 @ 60
force keyboard_data 0 @ 70
force keyboard_data 1 @ 80
force keyboard_data 1 @ 90
run 100
```

B.3 Infrared Receiver Input - Seven Segment Display Output

```
force clk 0 0, 1 1 -r 20000
force resetbutton 0 0, 1 1000000, 0 2000000, 1 3000000
force ir 0 0, 1 500000000, 0 9500000000, 1 14000000000, 0 14562000000.5, 1 16250000000, 0 16812000000.5,
      1 17375000000, 0 17937000000.5, 1 19625000000, 0 20187000000.5, 1 21875000000, 0 22437000000.5, 1
     23000000000, 0 23562000000.5, 1 24125000000, 0 24687000000.5, 1 25250000000, 0 25812000000.5, 1
     27500000000, 0 28062000000.5, 1 28625000000, 0 29187000000.5, 1 30875000000, 0 31437000000.5, 1
     32000000000, 0 32562000000.5, 1 33125000000, 0 33687000000.5, 1 35375000000, 0 35937000000.5, 1
     37625000000, 0 38187000000.5, 1 39875000000, 0 40437000000.5, 1 41000000000, 0 41562000000.5, 1
     43250000000, 0 43812000000.5, 1 44375000000, 0 44937000000.5, 1 45500000000, 0 46062000000.5, 1
     46625000000, 0 47187000000.5, 1 48875000000, 0 49437000000.5, 1 51125000000, 0 51687000000.5, 1
     52250000000, 0 52812000000.5, 1 54500000000, 0 55062000000.5, 1 55625000000, 0 56187000000.5, 1
     57875000000, 0 58437000000.5, 1 60125000000, 0 60687000000.5, 1 62375000000, 0 62937000000.5, 1
     63500000000, 0 64062000000.5, 1 64625000000, 0 65187000000.5, 1 66875000000, 0 67437000000.5, 1
     68000000000, 0 68562000000.5, 1 69125000000, 0 69687000000.5, 1 70250000000, 0 70812000000.5


run 71000000000
```

References

[1] D. S. Hauck, “De1-soc interfaces and peripherals.” https://class.ece.uw.edu/271/hauck2/de1/index.html.

[2] D. S. Hauck, “Ps/2 keyboard tutorial.” https://class.ece.uw.edu/271/hauck2/de1/keyboard/PS2Keyboard.pdf.