ECE 271, Final Project, Group 4

Michael Boly, Brenden Lowe, David Headrick

January 15, 2021

Contents

1 Introduction

For our final ECE 271 design project, we decided to use the NES controller and PS/2 Keyboard as inputs and the addressable RGB LEDs, VGA display, and DE10-Lite Seven Segment displays as output. More specifically, we will display different colors on the LEDs and sprites on the VGA that correspond to the buttons pressed on the NES controller. Similarly, we will display a letter or number on the DE10-Lite Seven Segment displays that correspond to the keys pressed on the PS/2 Keyboard. To complete this project, we used Quartus Prime for writing SystemVerilog and ModelSim for testing each module.

1.1 NES Controller

The NES controller was released along side the famous NES (Nintendo Entertainment System) in the year 1985 (US release). The primary purpose of the NES controller was to provide an interface for user inputs to the NES, allowing for users to interact with the video games they were playing. The NES features a directional pad, with buttons for up, down, left, and right, in addition to select, start, a, and b buttons. In this project, the NES controller was chosen as an input interface for both addressable RGBs and VGA outputs.

Figure 1: NES Controller.

1.2 PS/2 Keyboard

The PS/2 Keyboard was introduced in 1987 in order to enable user input to the PS/2 system. Although the PS/2 Keyboard is typically used with a mouse, we will only be using the keyboard key presses as input for our project. The PS/2 Keyboard can be used to interface with computers other than the PS2 System by way of an adapter.

Figure 2: PS/2 Keyboard.

2 Top Level and Hardware Diagrams

2.1 Project Top Level

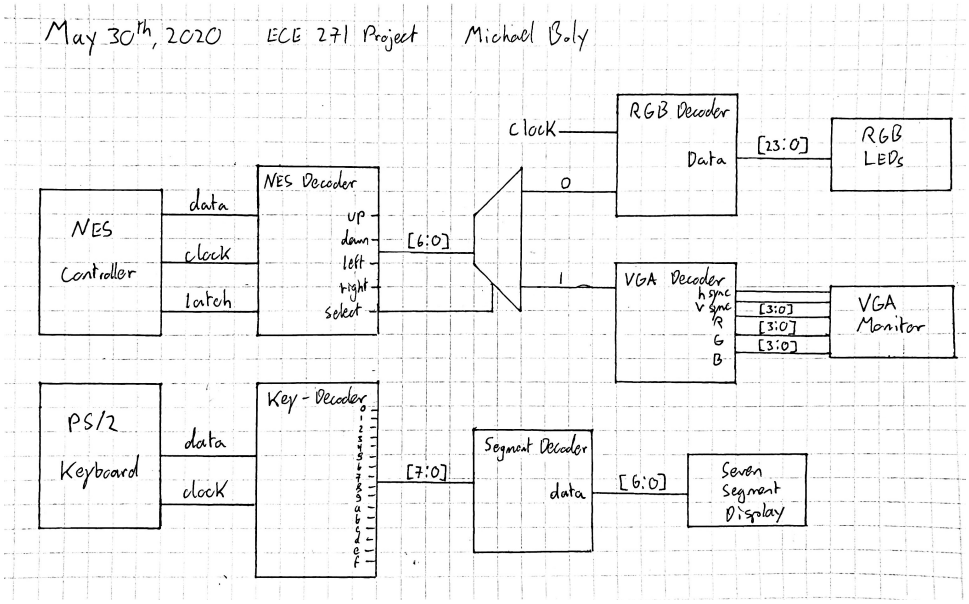

Figure 3: Top level overview of the project design.

2.2 NES Controller Top Level

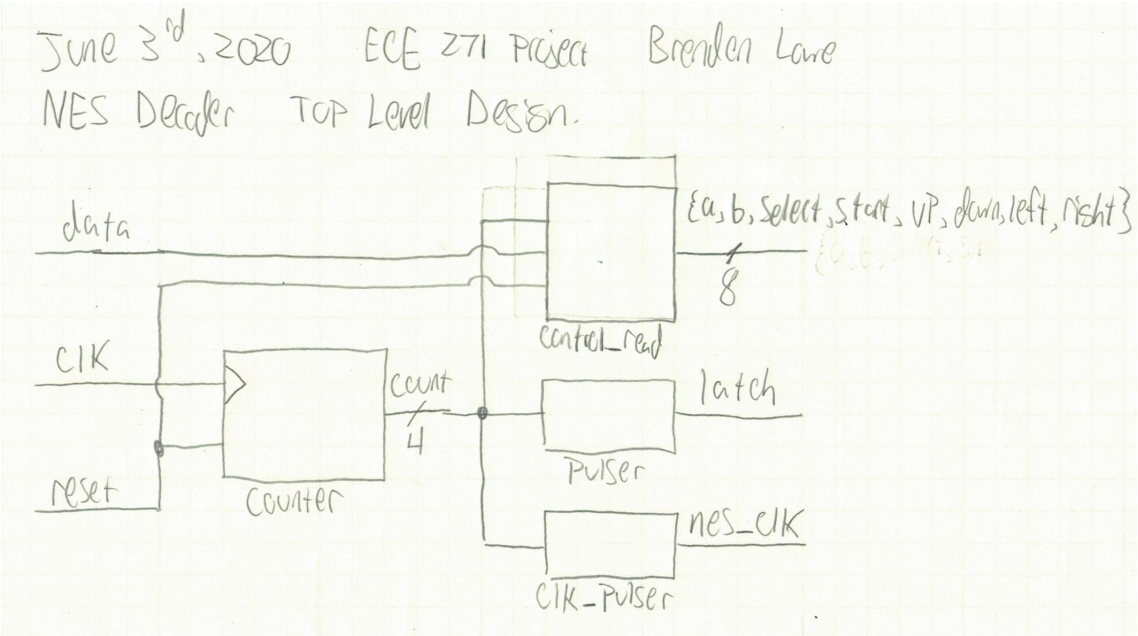

Figure 4: NES Controller Decoder Top Level Diagram.

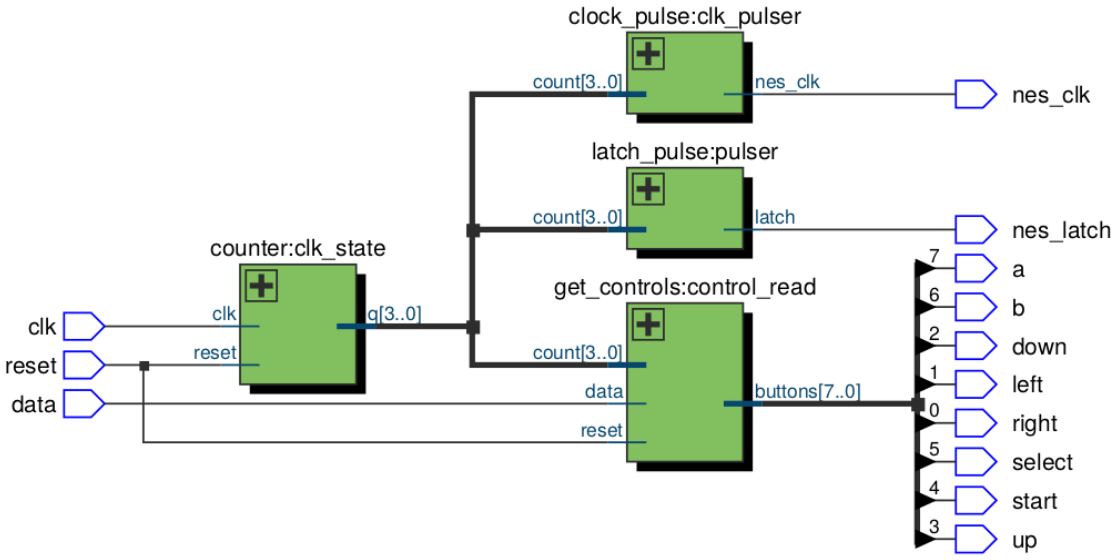

Figure 5: Synthesized NES Decoder Top Level Diagram in Quartus Prime.

The previous two figures illustrate the top level diagram for the NES Controller decoder, which takes in a clock signal from DE10-Lite board, a reset signal from user input, and a data signal from the NES controller itself. The NES decoder has ten outputs. It has eight outputs to represent the buttons being pressed, which will be fed into the VGA and RGB drivers, and two outputs, nes_clk and nes_latch, which connect to the NES controller and signal to assist in the serial data transfer from the controller to the decoder.

2.3 NES Controller Hardware Diagram

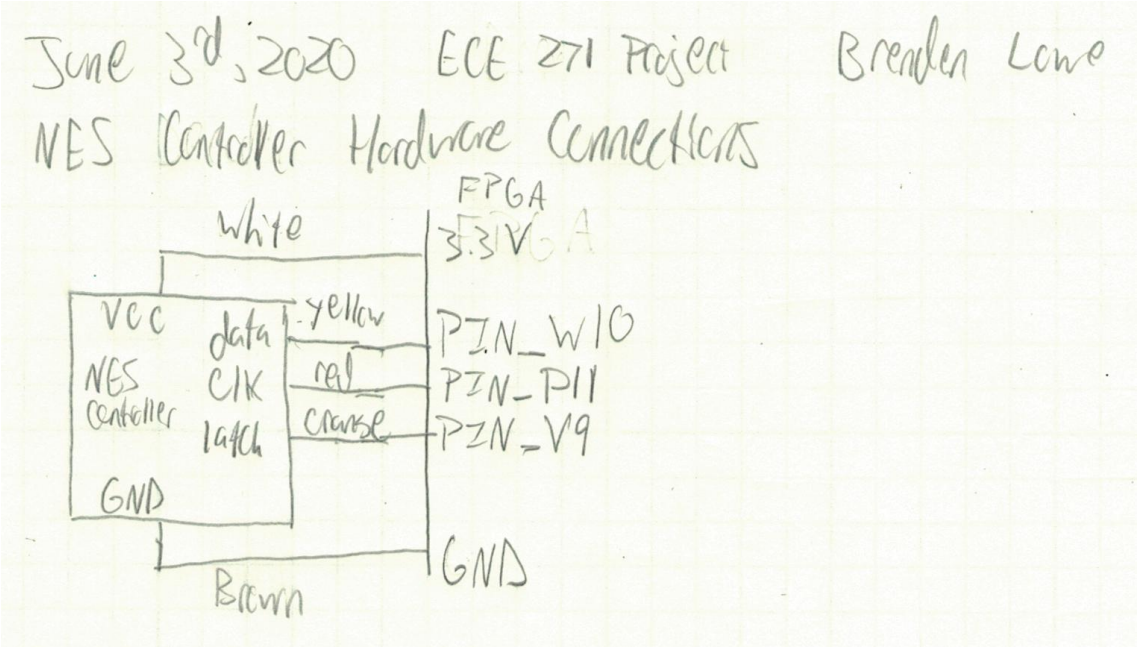

Figure 6: Hardware connections from NES Contoller to DE10-Lite.

2.4 PS/2 Keyboard Top Level

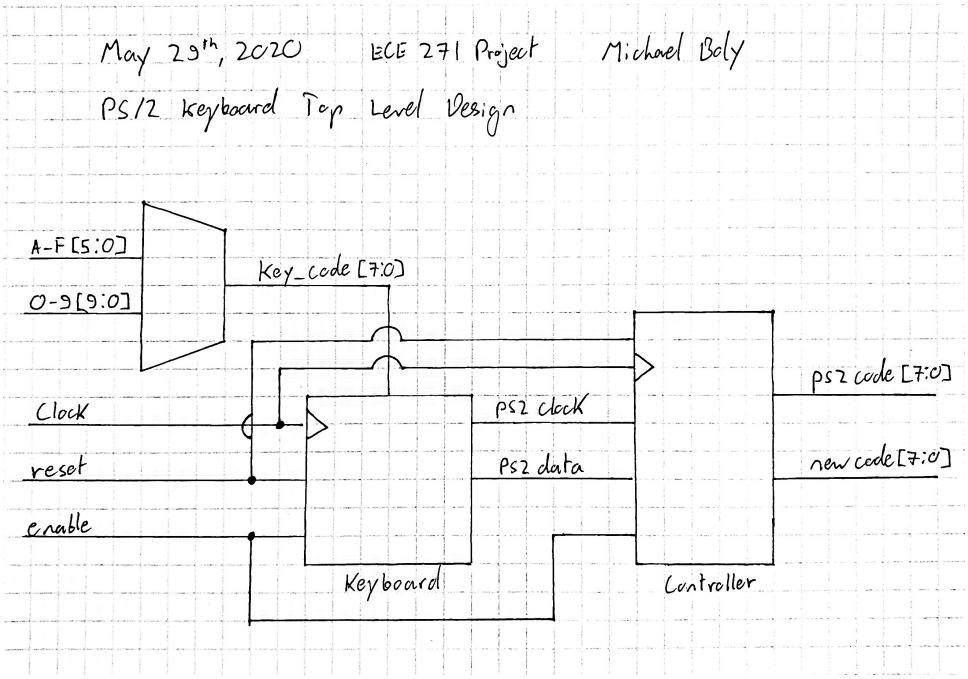

Figure 7: PS/2 Top Level Diagram.

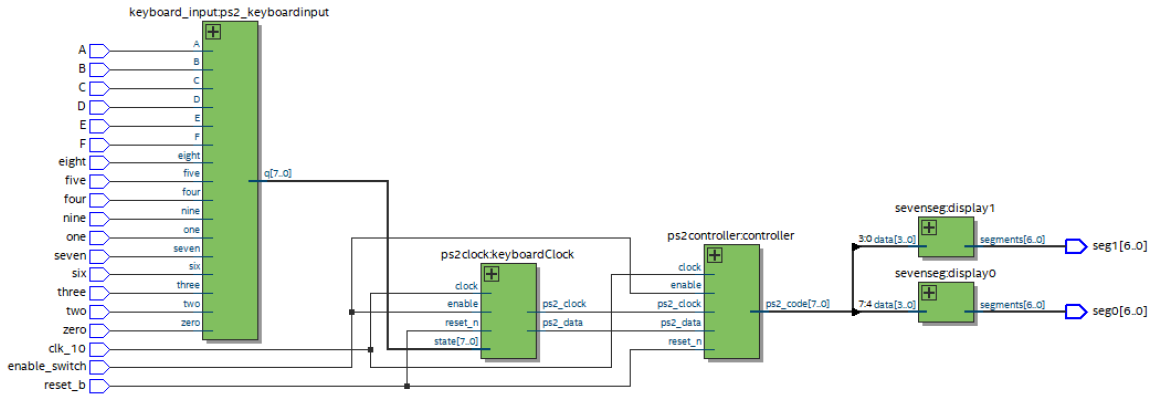

Figure 8: Synthesized PS/2 Top Level Diagram in Quartus Prime.

The figure above illustrates the top level diagram for the PS/2 Keyboard which takes letters A-F and numbers 0-9 as inputs and begins by creating a key code representation to be passed into the ps2clock module. Once received by the clock module, the key code is outputted in 11 bits that are passed into a shift register in the controller module and shifted out one by one. Lastly, the code is sent to the Seven Segment display to represent the key that was pressed.

2.5 PS/2 Keyboard Hardware Diagram

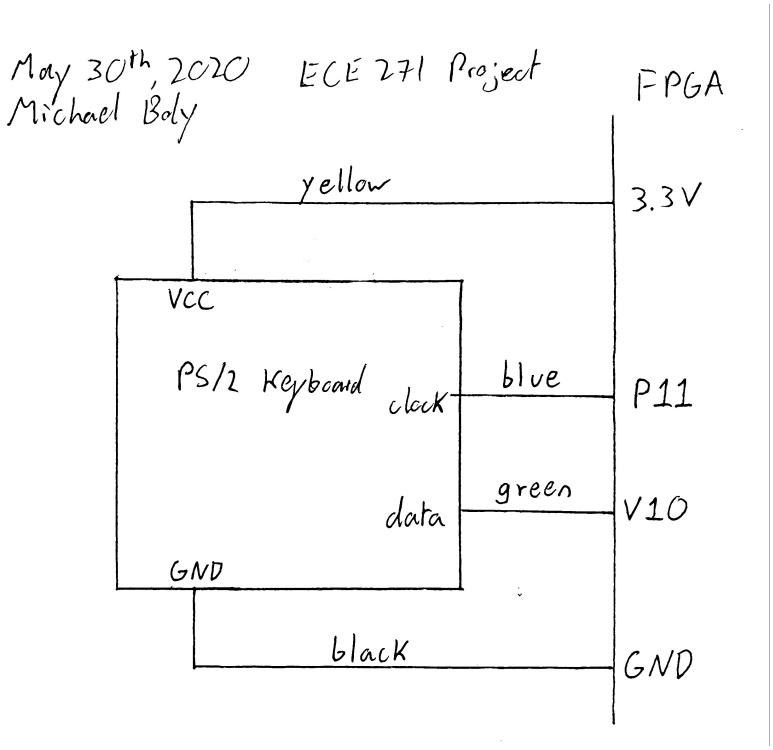

Figure 9: Hardware connections from PS/2 Keyboard to DE10-Lite.

2.6 VGA Driver Top Level

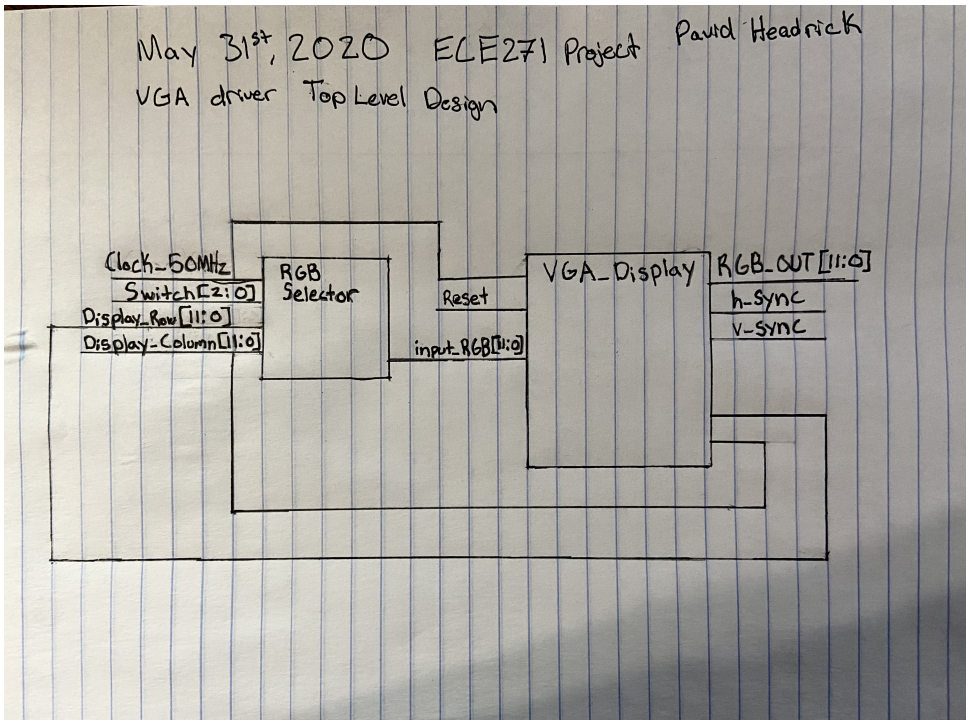

Figure 10: VGA Top Level Diagram.

Figure 10 illustrates the top level diagram for the VGA driver. The RGB generator takes an 3 bit bus to choose which of 3 sprites to display to the screen. The correct RGB outputs are then

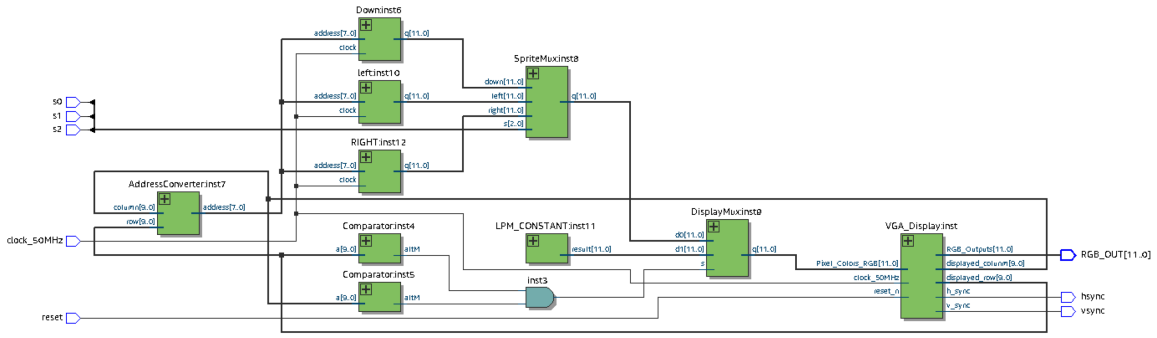

Figure 11: Synthesized VGA Top Level Diagram in Quartus Prime.

output to the VGA display module that synchronizes the RGB values with the VGA display count. The VGA display also generates a hsync and vsync signal for the connected display. Both modules are syncronized using a 50MHz clock.

2.7 VGA Driver Hardware Diagram

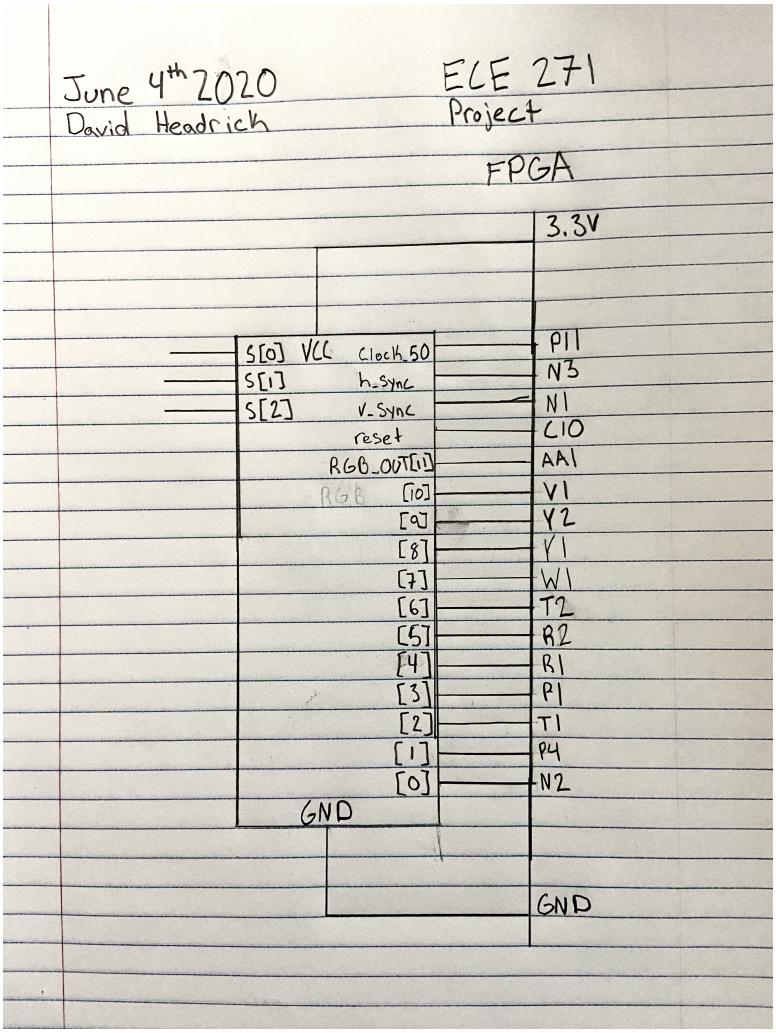

Figure 12: Hardware connections from VGA Driver to DE10-Lite.

2.8 Addressable RGB Driver Top Level

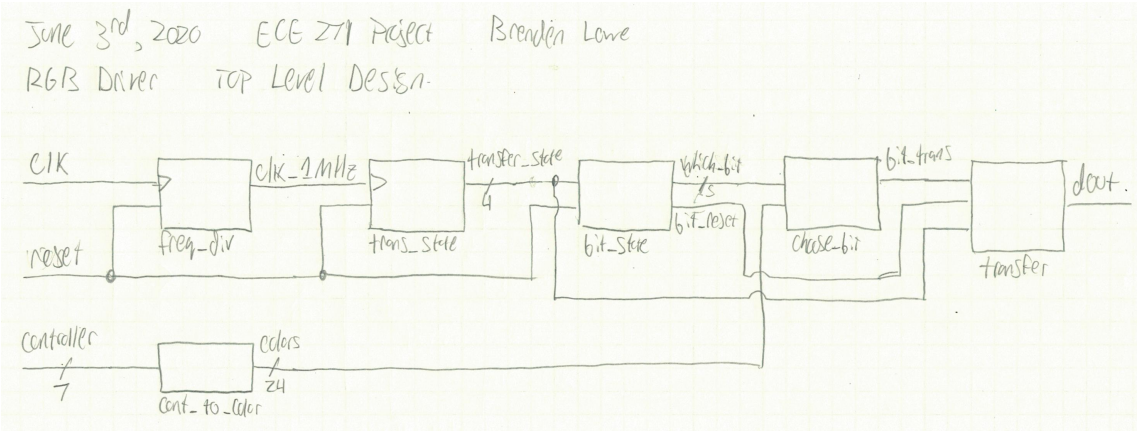

Figure 13: RGB Driver Top Level Diagram.

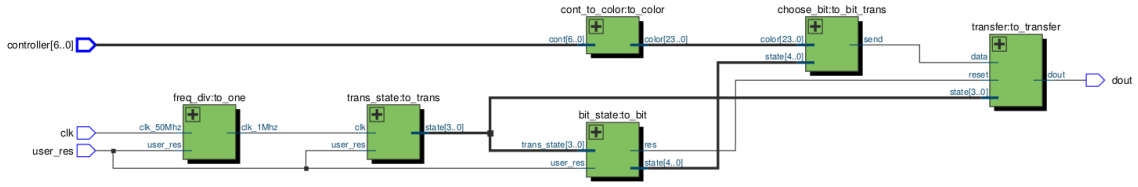

Figure 14: Synthesized RGB Driver Top Level Diagram in Quartus Prime.

The previous two figures illustrate the top level design of the RGB Driver. The RGB driver takes in a 50 MHz clock signal as input, along with a reset signal, and a controller signal. Notice that the controller signal only contains seven bits, as select is reserved for another purpose. The RGB driver has a single output, being dout. The reason for this is due to the fact that the addressable RGBs use a special serial data transfer, being different periods of high and low signals to represent a 1 and 0.

2.9 Addressable RGB Hardware Diagram

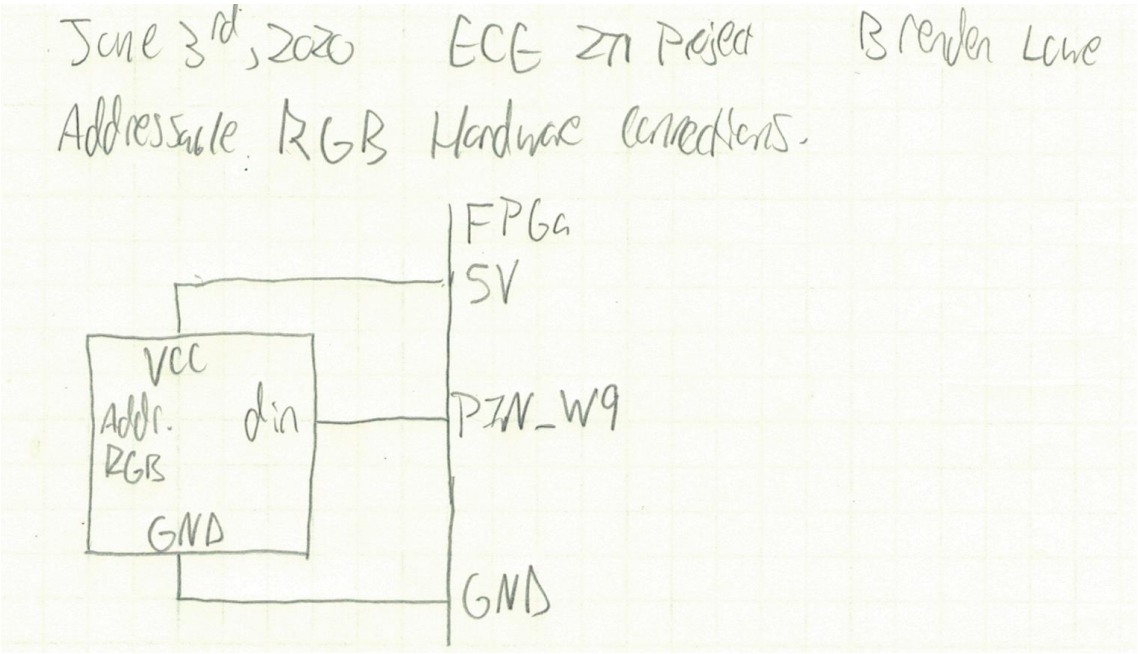

Figure 15: Hardware connections from DE10-Lite to Addressable RGB.

3 FPGA Usage Levels

3.1 NES Controller

The figure below shows the memory being used by the NES_Decoder module on the top level on the MAX10 FPGA after synthesis.

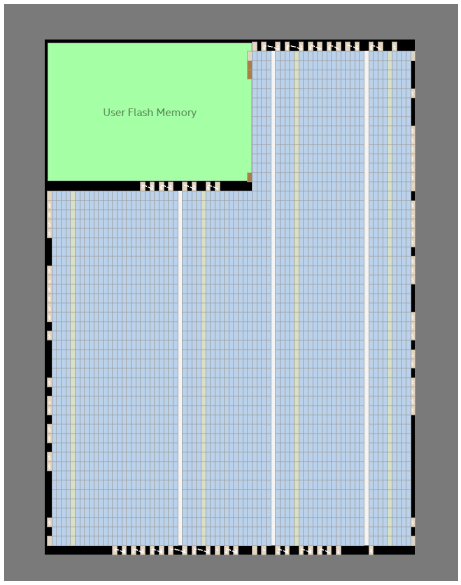

Figure 16: NES Decoder module usage level on FPGA after synthesis.

3.2 PS/2 Keyboard

The figure below represents the memory used by the PS/2 Keyboard top level on the MAX10 FPGA after synthesis. The blue rectangles are the memory used by this module, which is less than 1 percent of its capacity. In fact, the total logic elements used by this module is only 116 out of the 49,760 available.

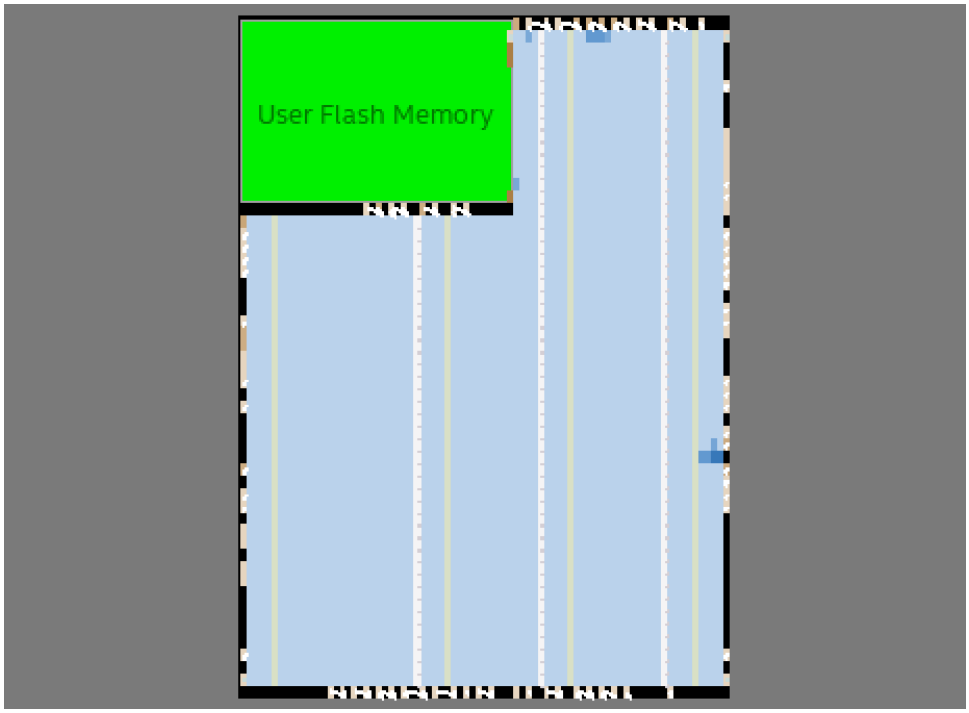

Figure 17: PS/2 Keyboard module usage level on FPGA after synthesis.

3.3 VGA Driver

The VGA Driver module used less than 1% of the available logic units on the MAX 10 chip. In total, 150 logic elements were used to represent the module.

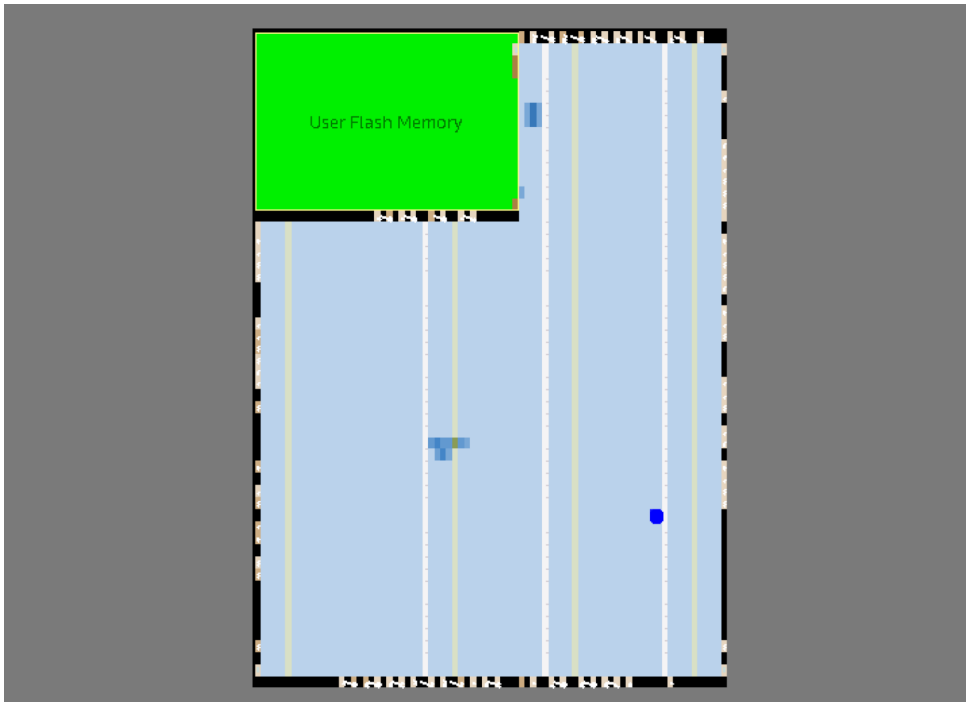

Figure 18: VGA Driver module usage level on FPGA after synthesis. 150 / 49,760 Logic Elements (< 1 %)

3.4 Seven Segemnt Driver

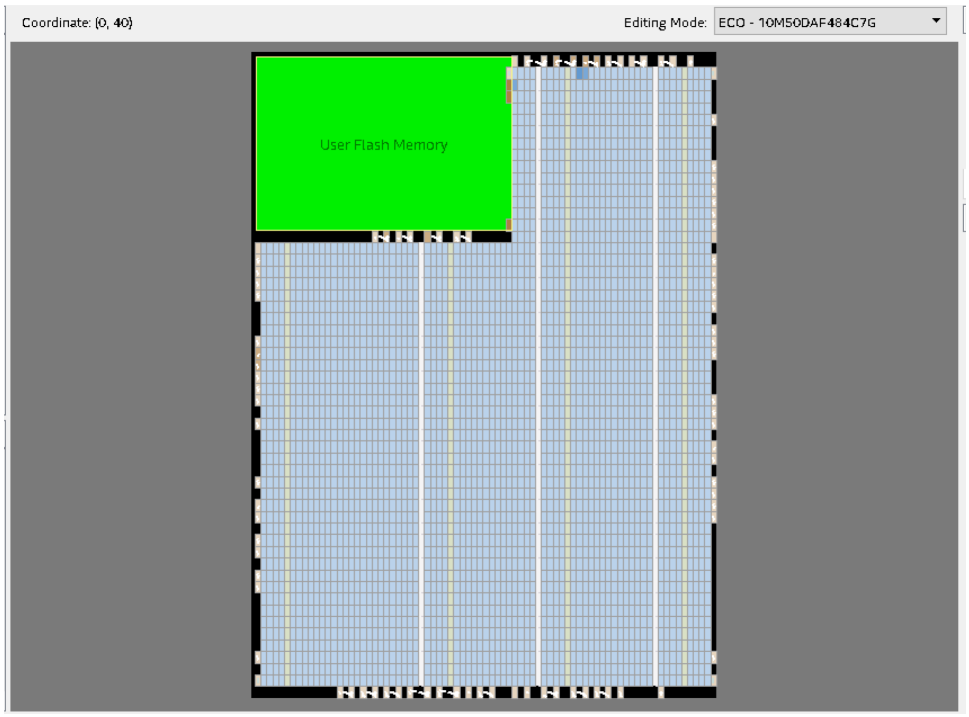

Figure 19: Seven Segment module usage level on FPGA after synthesis. 16 / 49,760 Logic Elements (< 1 %)

3.5 RGB Driver

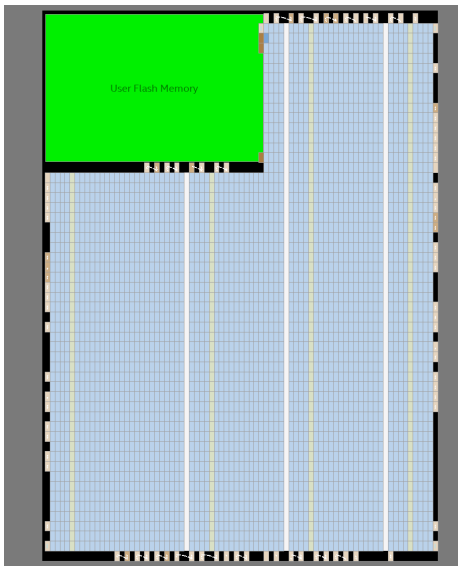

Figure 20: RGB Driver module usage level of FPGA after synthesis.

4 HDL Modules

4.1 NES Controller

4.1.1 NES_Decoder

Description: The NES_Decoder module is the top level for the NES Decoder circuit The NES_Decoder is able to read in input from an NES controller, and output the appropriate data mapping, as well as outputs to assist in the serial data transfer process from controller to circuit.
Inputs: The NES_Decoder module takes in three inputs. One input is a data signal from the NES controller that represents controller data. The other two inputs are a clock signal, and a user reset signal.
Outputs: The NES_Decoder module has ten outputs. Eight of these outputs represent the current state of the buttons on the NES controller. The other two outputs connect directly to the NES controller. One is the NES latch signal, which signals the NES controller to load in current controller state, and the other being the NES clock signal, which facilitates serial data transfer between the controller and the module.

4.1.2 counter (4-bit)

Description: The counter module is a basic four-bit implementation of a traditional counter module. The counter module is a sequential design that increments its output every clock cycle, and resets the output back to 0 when its reset input is asserted.
Inputs: The counter module takes in two inputs, being a clock signal, and a reset signal.
Outputs: The counter module has a single output, being the current count at a given time. The count signal is 4-bits, and can thus represent any number within that range.

4.1.3 latch_pulse

Description: The latch_pulse module is responsible for pulsing the latch output to the NES controller at the appropriate time. This signals the NES controller to load its inputs into its shift register, and prepare for serial data transfer.
Inputs: The latch_pulse module has one input, being a four bit signal that represents a current controller state during data transmission.
Outputs: The latch_pulse module has a single output, being the latch signal.

4.1.4 clock_pulse

Description: The clock_pulse module is responsible for pulsing and emulating a clock signal to the NES controller. The goal of this signal is to assist in serial data transfer between the NES controller the the NES controller Decoder module.
Inputs: The clock_pulse module takes in a single output, being a four bit signal that represent a current controller state during data transmission.
Outputs: The clock_pulse module has one output, being the NES clock signal.

4.1.5 get_controls

Description: The get_controls module is responsible for reading in the serial data line, and outputting the appropriate controller button mapping. For each transmission on the serial data line, the module is able to determine the corresponding button press, and outputs this result.
Inputs: The get_controls module takes in three inputs. One input is a four bit signal that represents a current controller state during transmission. One input is the data input from the NES controller, that inputs the current end-of-shift-register value. The other input is a user reset signal.
Outputs: The get_controls module has an eight bit output, with each bit representing whether a corresponding button was pressed or not pressed.

4.2 PS/2 Keyboard

4.2.1 ps2keyboard

Description: The ps2keyboard module is the top level for the PS/2 Keyboard that combines all the sub-modules to process a key-press from the keyboard and send it to the seven segment display. This module is in charge of combining the keyboard_input with the ps2clock and ps2controller to form the top level.
Inputs: The ps2keyboard module takes the letters A-F and numbers 0-9 as input from the keyboard. Additionally, a clock, reset (active low), and enable signals are taken as inputs.
Outputs: The ps2keyboard module outputs an 8-bit keycode that can be interpreted and displayed by the seven segment displays.

4.2.2 keyboard_input

Description: The keyboard_input module simply maps every key-press A-F and 0-9 to its corresponding 8-bit keycode. This module is essentially behaving as a virtual keyboard by passing on the 8-bit keycode for each key press.
Inputs: The keyboard_input module takes in a signal for each letter A-F and number 0-9.
Outputs: The keyboard_input module returns an 8-bit hexadecimal code that matches the key press.

4.2.3 keybuffer

Description: The keybuffer module serves as an input buffer of 30 characters so that we can process multiple keys being pressed one at a time. This module saves the project from possible malfunction when multiple keys are pressed simultaneously.
Inputs: The keybuffer module takes in the 8-bit keycode from the keyboard_input module, a reset signal, and a signal that determines when to pass on the output keycode.
Outputs: The keybuffer module outputs the 8-bit keycode for one key press at a time.

4.2.4 ps2clock

Description: The ps2clock module is an emulation of a real PS/2 keyboard that outputs a clock and data signal. On every other clock count, the data signal will represent the next bit in the 12 bit series that begins with a 0 and is followed by each state of the 8-bit keycode, a parity value, and a 1 that represents the end of the series transmission.

Inputs: The ps2clock module takes an active low reset, an enable signal, and the 8-bit keycode from the keyboard_input.
Outputs: The ps2clock module returns a clock and data signal that function like a PS/2 keyboard.

4.2.5 ps2controller

Description: The ps2controller module manages the data output from ps2clock by storing every new bit in a shift register on the negative edge of the ps2clock's clock output. This module outputs the 8-bit PS/2 code as well as a flag called ps2_code_new for every time a new code is processed.
Inputs: The ps2controller module receives the system clock, active-low reset, enable signal, as well as the clock and data from the ps2clock module.
Outputs: The ps2controller module outputs the original 8-bit keycode from the keyboard_input and a flag indicating the complete processing of a single keycode.

4.2.6 flop

Description: The flop module is a simple D flip flop that serves as an N-bit register for outputting the keycode from the shift register in ps2controller. Additionally, the flop is used to pass on the flag signal that a new code has been received by the ps2controller.
Inputs: The flop module takes in a data signal of size N, a active-low reset, enable signal, and the system clock.
Outputs: The flop module outputs a bus that is size N. In this project, we use the flop module to take in 8 bits and 1 bit for the register and flag signal respectively.

4.2.7 shiftRegister

Description: The shiftRegister module is tasked with shifting the data from ps2data to the left on bit at a time and filling the least significant bit with the next bit from ps2data. This procedure takes place on the negative edge of the ps2clock.
Inputs: The shiftRegister module is passed the ps2clock, active-low reset, enable, and ps2data as inputs.
Outputs: The shiftRegister module outputs the contents of the shift register in an N-bit size bus (state) and the most significant bit (q).

4.2.8 idleCounter

Description: The idleCounter module is a very simple counter using the positive edge of the clock.
Inputs: The idleCounter module takes an active low reset and the system clock as inputs.
Outputs: The idleCounter module outputs a 12-bit bus that contains the number of system clock cycles elapsed.

4.2.9 lessThan

Description: The lessThan module is a simple comparator that will return true if the input (d) is less than the parameter (a) and false otherwise.
Inputs: The lessThan module takes a bus of size N as input. In this project, we pass the current number of system clock cycles as the input to see if we have reached 21 cycles.
Outputs: The lessThan module outputs a 1 if the input is less than the parameter value and a 0 otherwise.

4.3 VGA Driver

4.3.1 AddressConverter

Description: The AddressConverter modules converts the row and column number into an address. This address defines a pixel within the ROMs loaded with a bitmap image of a 16x16 down, left, and right arrow.
Inputs: The AddressConverter takes in the currently displayed row and column number as a 10 bit bus.
Outputs: The output is an 8 bit bus that describes the location of a 12 bit word in system ROM.

4.3.2 Comparator

Description: The comparator module compares the input value with a parameter to see whether the input is less than the parameter value.
Inputs: The comparator inputs the displayed row or column as a 10 bit bus. It also uses a parameter for the comparison.
Outputs: The comparator outputs a 1 bit value with the true or false value of the comparison.

4.3.3 SpriteMux

Description: The SpriteMux determines which sprite is displayed to the screen. It uses hot bit encoding to determine which sprite is displayed. In the event that the switch value is not valid, a green color is output.
Inputs: The SpriteMux takes in three 12 bit RGB values for the current pixel on either the right, left, or down sprites.
Outputs: The SpriteMux outputs a 12 bit bus for the correct pixel currently being shown.

4.3.4 DisplayMux

Description: The DisplayMux chooses whether to display the sprite or a constant background color. This creates a solid color background with a sprite in the top left hand corner.
Inputs: The DisplayMux takes in a constant 12 bit constant background RGB color, a sprites 12 bit current pixel color, and a AND'd switch value from two comparators. The comparators determine if the current pixel is in the top 16x16 left hand corner of the screen.
Outputs: The output is a 12 bit bus with the current pixels RGB value.

4.3.5 VGA_Display

Description: The VGA_Display module is a Top Level module for synchronizing the h_sync, v_sync, and RGB output with the current pixel and line count.
Inputs: The VGA_Display takes in a 50MHz clock, a reset value, and a 12 bit bus for the current pixel.
Outputs: The VGA_Display outputs two 10 bit bus values with the current row and column being displayed. It also outputs a 12 bit bus with the current RGB values. Lastly it outputs two different one bit lines for the h_sync and v_sync values.

4.3.6 clockdiv

Description: The clockdiv divides the clock value from 50MHz to 25MHz using 2 as the parameter. The 25MHz clock is the pixel clock used by the syncCounter.
Inputs: The clockdiv takes in a 50MHz clock and a reset value.
Outputs: The clockdiv outputs a 25MHz clock.

4.3.7 syncCounter

Description: The syncCounter module synchronizes when to display the RGB values and when to enable the h_sync and v_sync signals.
Inputs: The syncCounter takes in a 25MHz clock and a reset signal.
Outputs: The syncCounter outputs h_sync and a v_sync signal. It also outputs two signals, hdisplay and vdisplay, which determine whether the row and column are currently being displayed. Lastly, it outputs the current row and column values as a 12 bit bus.

4.4 Seven Segment Driver

4.4.1 SevenSeg

Description: The SevenSeg module converts a hexadecimal 8 bit bus signal into a signal to a seven segment display. The signal to the display is used to display a 0 - f on the display corresponding to the hexadecimal value of the 8 bit bus.
Inputs: The Seven Segment Display takes in an 8 bit bus as the single input.
Outputs: The output is a 7 bit bus. The code in the bus is used to display a value on the seven segment display.

4.5 RGB Driver

4.5.1 RGB_Driver

Description: The RGB_Driver module is the top-level design of the RGB Driver. This module takes in a controller signal, and then outputs a color to an addressable RGB.
Inputs: The RGB_Driver module takes in three inputs. One is a 50 MHz clock signal, and another is a user reset signal. Additionally, the module takes in a seven bit controller signal, representing which buttons were pushed on a NES controller.
Outputs: The RGB_Driver module has a single output, being dout. This signal sends data to an addressable RGB, following the specified protocols to encode 1's and 0's.

4.5.2 counter

Description: The counter module is a traditional take on a standard counter module. This module takes in a parameter to represent how many bits the counter should be. The counter increments its output every clock cycle.
Inputs: The counter module takes in two inputs, being a clock and reset signal.
Outputs: The counter module has a single output, which represents the count at a given time.

4.5.3 comparator

Description: The comparator module is a traditional take on a standard comparator module, only exception being that it only has an equality output. The comparator module determines whether two given numbers are equal. It also takes in a parameter to determine how many bits the inputs will be.
Inputs: The comparator module has two inputs, being the two numbers to be compared.
Outputs: The comparator module has one output, being the equality output, which is TRUE if the given inputs are equal.

4.5.4 sync

Description: The sync module is a traditional take on a standard sync module. The sync module outputs an input in sync with a given clock signal.
Inputs: The sync module has two inputs, being the data to be transmitted, and a clock signal.
Outputs: The sync module has a single output, being the data that was given as input.

4.5.5 freq_div

Description: The freq_div module takes in a 50 MHz clock signal, and uses it to output a 1 MHz clock signal.
Inputs: The freq_div module takes in two inputs, being a 50 MHz clock signal, and a user reset signal.
Outputs: The freq_div module has a single output, being a 1 MHz clock signal.

4.5.6 cont_to_color

Description: The cont_to_color module takes in a seven bit controller signal, and then maps it to a 24 bit color signal to be transmitted to an addressable RGB.
Inputs: The cont_to_color module takes in a single input, being a seven-bit controller signal.
Outputs: The cont_to_color module has a single output, being a 24-bit RGB color signal.

4.5.7 trans_state

Description: The trans_state module keeps track of the current transmission time, in intervals of 12 μs. This is to help keep track of signal encoding in the addressable RGBs.
Inputs: The trans_state module takes in two inputs, being a 1 MHz clock signal, and a user reset signal.
Outputs: The trans_state module has a single four bit output, which represents the current transmission state.

4.5.8 bit_state

Description: The bit_state module keeps track of which bit is to be transferred out of the 24-bit color signal. It also determines when a reset encoding should be sent to the addressable RGBs.
Inputs: The bit_state module takes in two inputs, one being a 4 bit signal representing the current transmission state. The other input is a user reset signal.
Outputs: The bit_state module has two outputs. One is a 5 bit signal that represents the current bit to be transferred. The other signal is a bit-reset signal, which indicates when a reset signal should be encoded to the addressable RGB.

4.5.9 choose_bit

Description: The choose_bit module outputs the desired bit of the color signal to be transferred to the addressable RGB, given a bit state.
Inputs: The choose_bit module has two inputs, being a 24 bit color signal, and a 5 bit signal representing which bit to transfer.
Outputs: The choose_bit module has one output, which is the current bit in the color signal to be transmitted.

4.5.10 transfer

Description: The transfer module transmits a time-specific encoding to the addressable RGBs, based on whether it's trying to encode a 1, 0, or reset signal.
Inputs: The transfer module has 3 inputs. One input is the current bit from the color signal to be encoded. The other is a 4 bit input representing the current transfer state. The last bit is a reset bit that represents when a reset signal should be encoded.
Outputs: The transfer module has a single output, being a data wire connected to the addressable RGB.

5 ModelSim Testing

Following the design and implementation of the individual modules of our project in Quartus Prime, we turned to ModelSim to verify that each module was functioning properly. The waves generated from the following tests in this section were performed in ModelSim and serve to replicate the data that would be received by our design if connected to hardware (the NES Controller or PS/2 Keyboard).

5.1 NES Controller

5.1.1 NES_Decoder

Since the lower level modules were tested and simulated separately, the NES_Decoder module was only tested to see if it accepted inputs and mapped to appropriate outputs at a specific time. Below, you can see the mapping of the a and b buttons to their corresponding outputs from the serial data transfer.

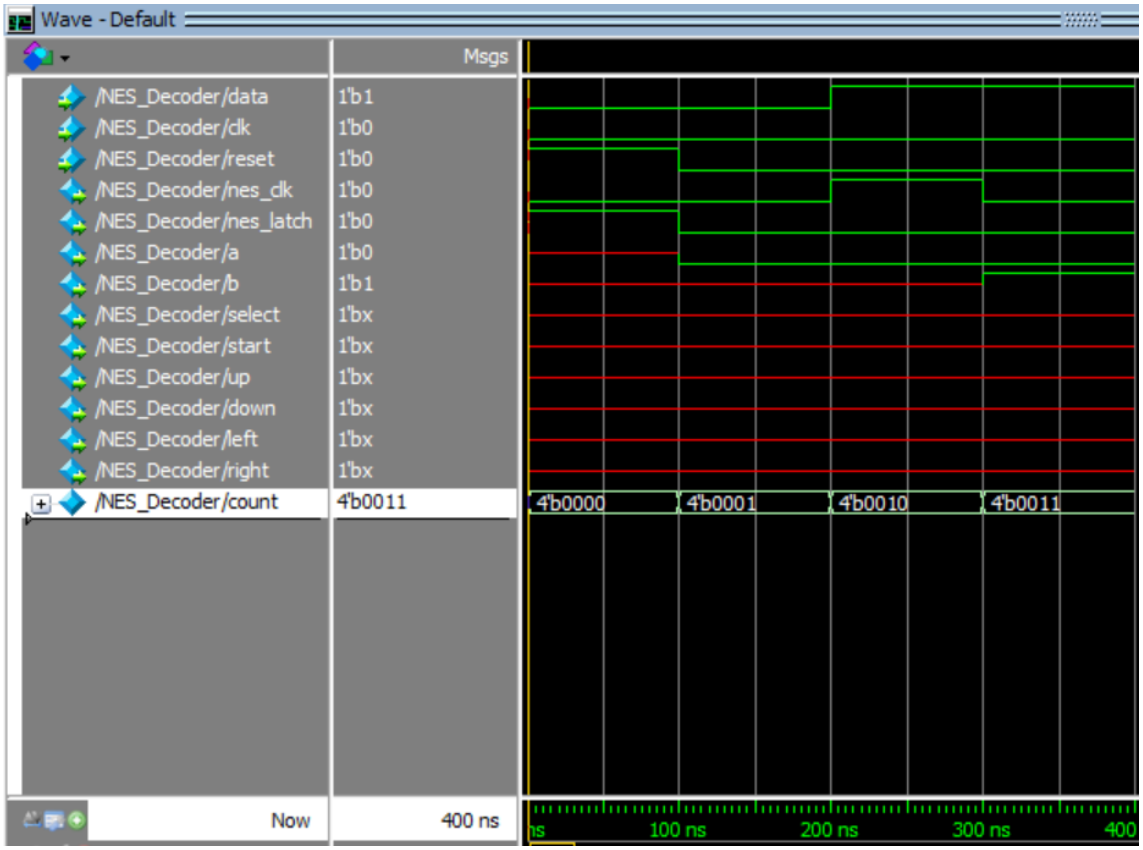

Figure 21: NES Decoder loading in a and b button inputs.

5.1.2 counter

The counter was tested by first asserting reset, and then asserting the clock signal to observe the behavior of the output. The goal was to observe the output increment every clock cycle. This behavior can be seen below:

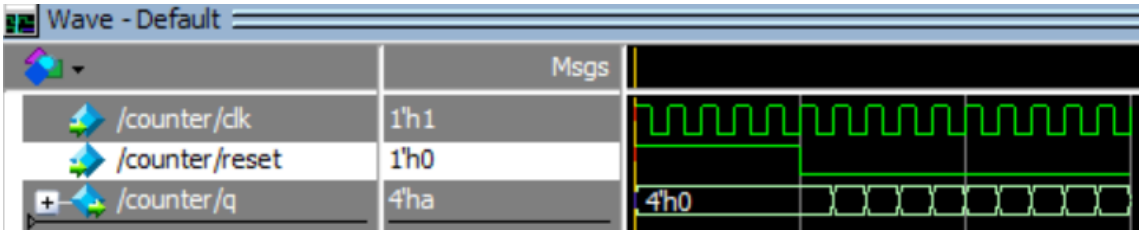

Figure 22: counter increment its output every clock cycle.

5.1.3 latch_pulse

The latch_pulse module was tested for each possible input state. The behavior that was expected was that the latch signal would only be pulsed when the input state was 0, and would

remain 0 if the input state was any other value. This behavior can be observed below:

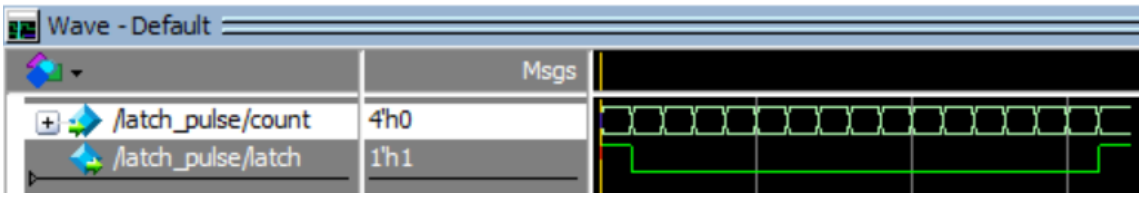

Figure 23: Latch signal pulses only once during state increment.

5.1.4 clock_pulse

The clock_pulse module was tested for each possible input state. The behavior that was expected is that the clock signal would be pulsed on every even input state signal. This behavior can be observed below:

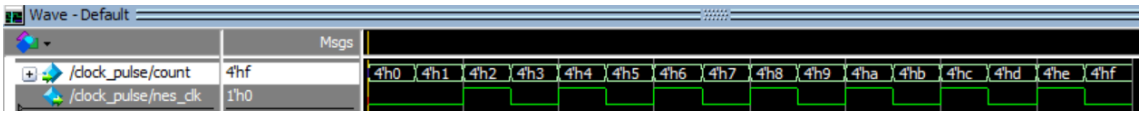

Figure 24: Clock signal pulses on every even input state.

5.1.5 get_controls

The get_controls module was tested for its given input state, data, and reset signals. The behavior that was expected is that when reset was asserted, then the 8-bit output would be all zeros. When reset was not asserted, it was expected that buttons would update an individual bit with the data signal when state is an odd number. This behavior can be observed below:

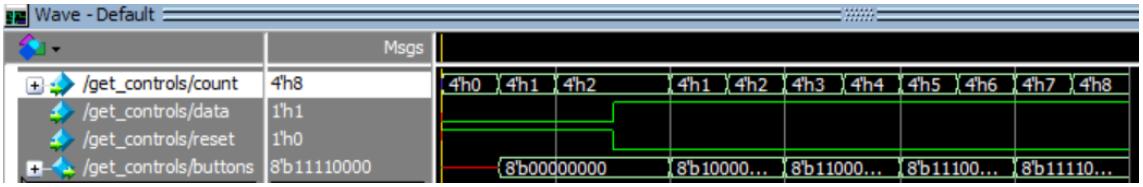

Figure 25: Button outputs updates on every odd state signal.

5.2 PS/2 Keyboard

5.2.1 ps2keyboard Module

We tested the ps2keyboard module first by setting the A key to be pressed and verifying the state code matches the key press.

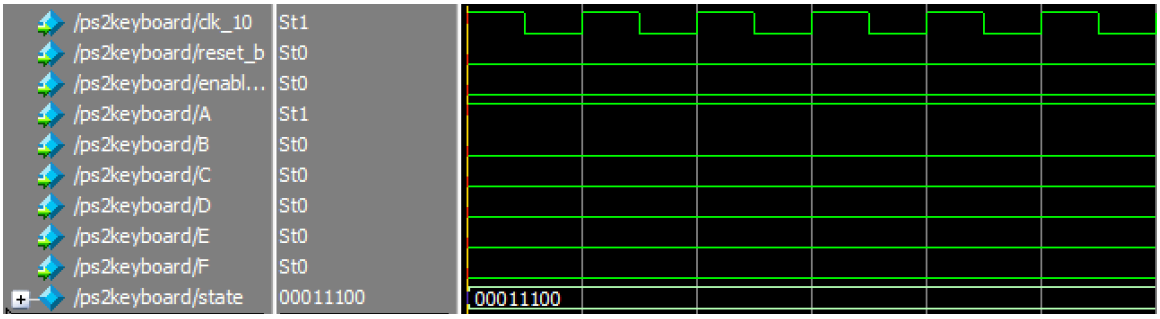

Figure 26: Testing single A key press.

Then we ran the same test for a single key press, but this time we set the B key to be pressed, and all the other keys not pressed.

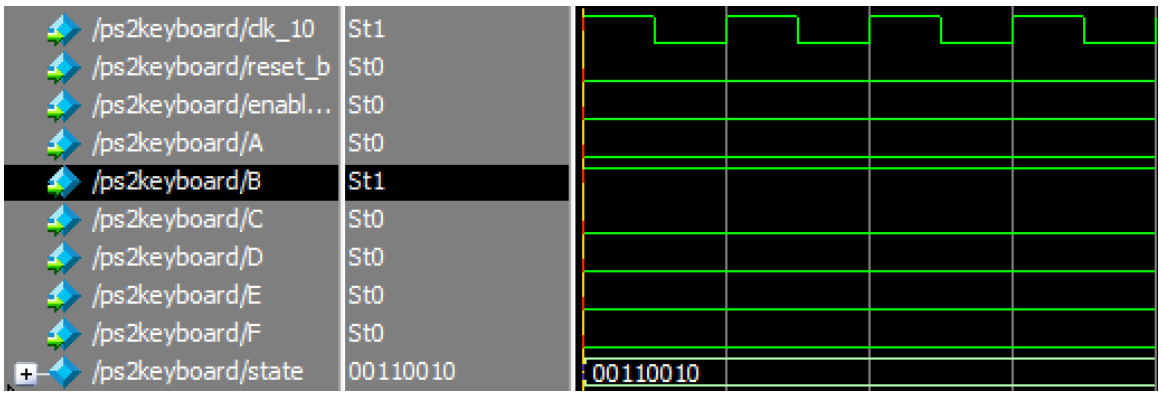

Figure 27: Testing single B key press.

Both state keycodes generated matched what they should be for a single A key press (00011100) and a single B key press (00110010) respectively. The hexidecimal that represents each of these keycodes can be seen in the following figure, thus we know the proper keycodes are generated from the ps2keyboard module.

Key	Make Code (hex)
A	1C
B	32

Figure 28: Hexadecimal key codes for A and B key presses.

We also tested that number key presses were producing the correct keycodes by setting the 0 key to be pressed, then the 1 key, then the 2 key.

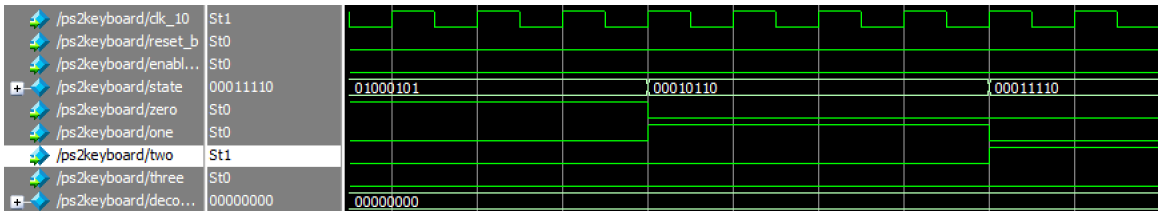

Figure 29: Testing key presses for 0, 1, and 2.

All three key presses resulted in the proper keycode being produced. The following figure shows the corresponding hexadecimal values for a keys 0, 1, and 2 being pressed.

0	45
1	16
2	1E

Figure 30: Expected keycode result in hexadecimal for key presses for 0, 1, and 2.

5.2.2 keyboard_input Module

Next we tested the keyboard_input module to check that the input would be buffered properly if there were multiple key presses simultaneously. The following figure shows how when none of the keys are pressed, the default is the keycode for 0, and when all keys are pressed, the first one enters the buffer begins by processing the first one.

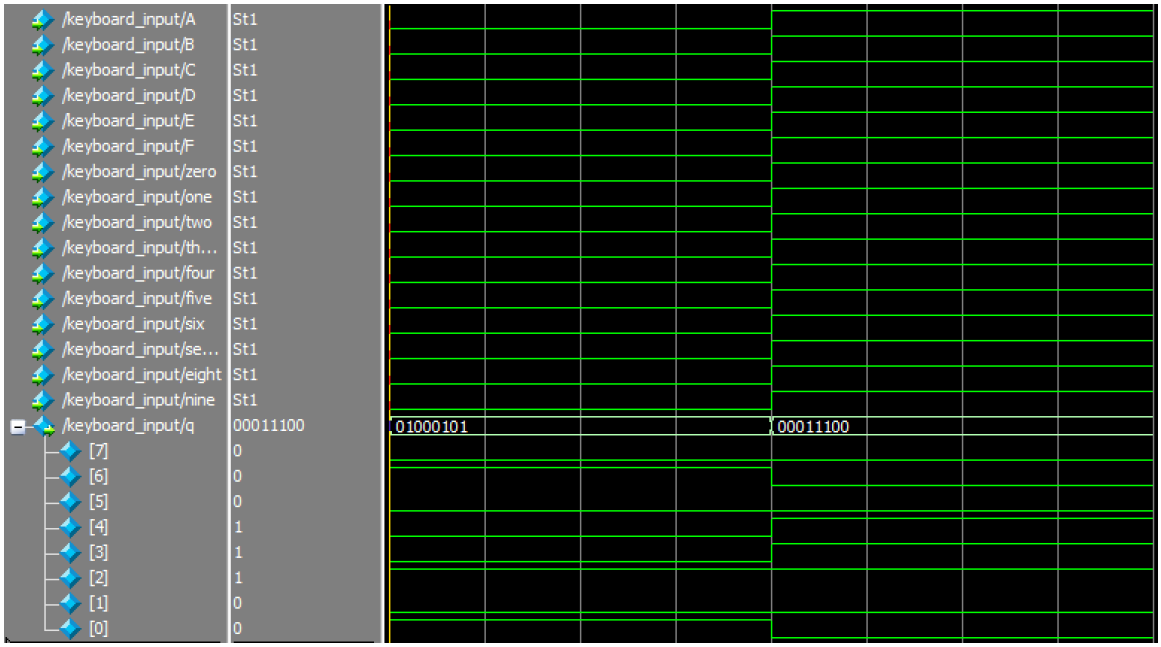

Figure 31: Test buffering for simultaneous key presses.

5.2.3 ps2clock Module

The ps2clock module was tested to see that is keeping time properly based on the system clock. We can also see that the count, lt, and parity are functioning.

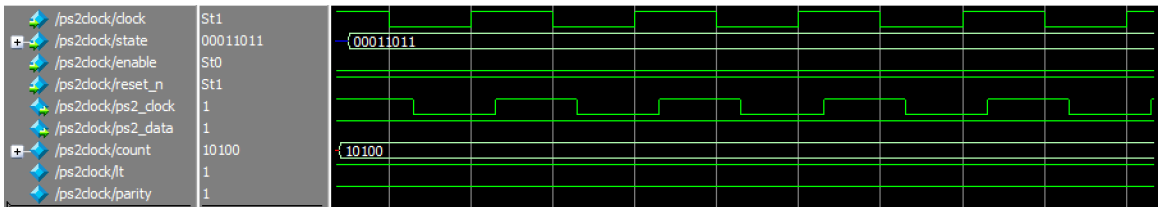

Figure 32: Testing ps2clock for time keeping.

5.2.4 ps2controller Module

The ps2controller module was tested to verify that the shift register was properly storing and pushing through each bit of data into complete 8-bit PS/2 codes. We can also see that the ps2_code_new flag is properly raised every time a new code is processed.

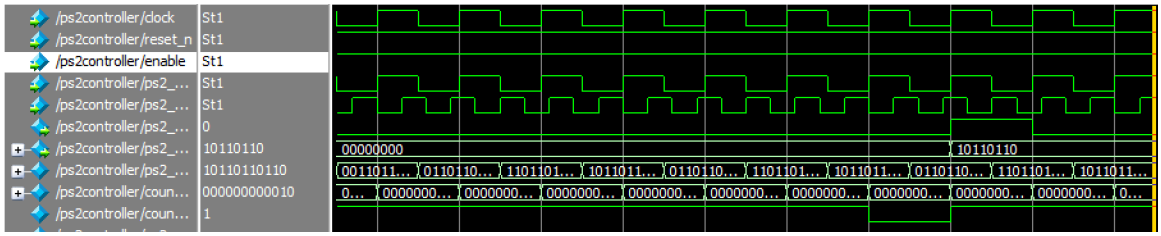

Figure 33: Testing the shift register and processing of new codes.

5.3 VGA Driver

5.3.1 VGA_Driver Module

We tested the VGA driver by forcing the 12 RGB sprite output as 111111111111_2, and the background color as 111100000000_2. We wanted to see if the correct RGB values were displayed at the correct time. Synthesized_Wire_4 is the forced sprite pixel value.

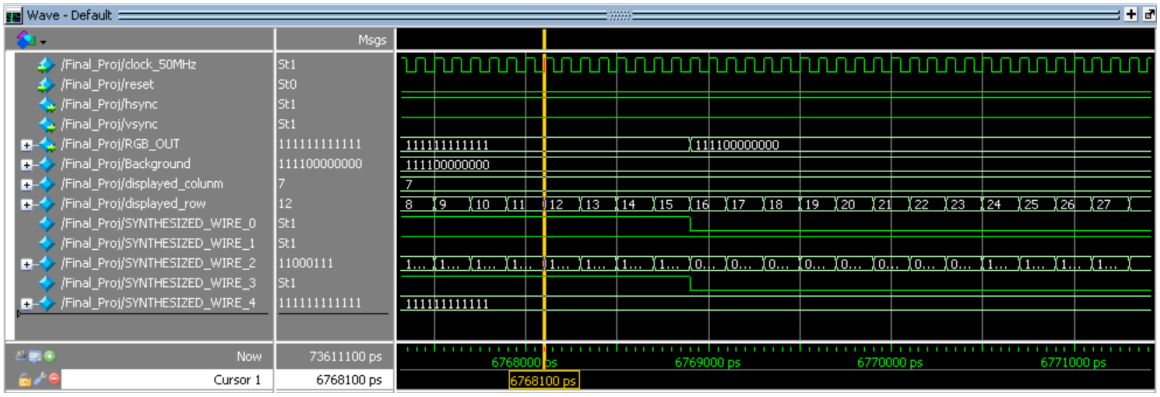

Figure 34: Testing the RGB value switching.

5.3.2 VGA_Display Module

We tested the VGA_Display module by seeing if the RGB outputs were the correct values at the correct times. We also saw if the Hdisplay, Vdisplay, H_Sync and V_Sync signals were correct.

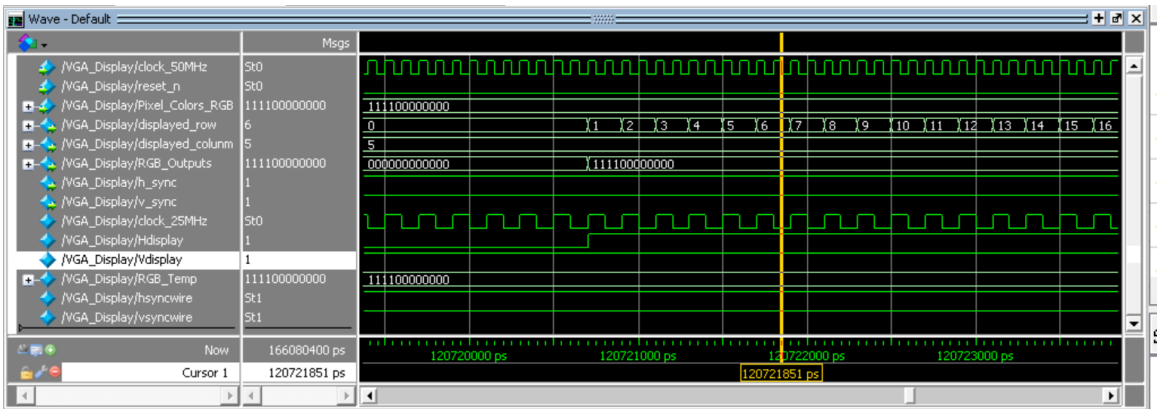

Figure 35: Testing the RGB output and sync signals.

5.3.3 Clock Divider Module

We tested the Clock Divider module by simulating an input clock and observing the output clock. The output clock in the simulations checks out to be half as fast.

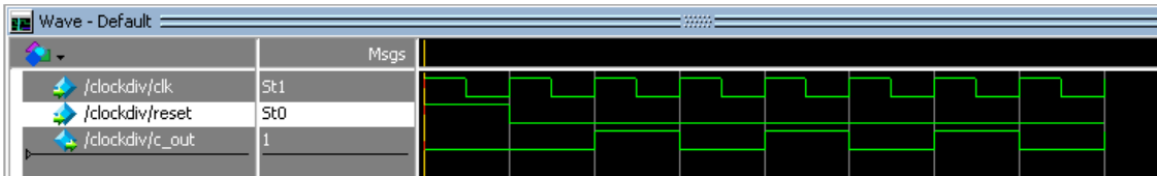

Figure 36: Testing the Clock Divider's timing

5.3.4 Comparator Module

We tested the Comparator by observing the altM wire's output depending on bus a's input. The Comparator tests to see if the input a's value is less than 16. The simulation checks out because altM is low when a is 16.

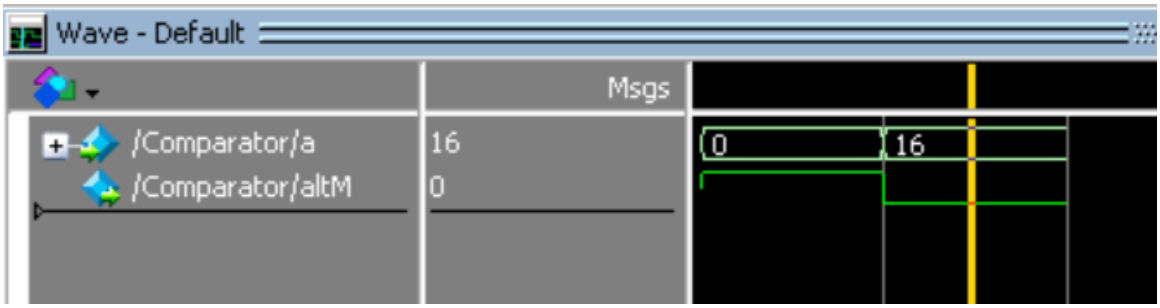

Figure 37: Testing the Comparator's output value

5.3.5 DisplayMux Module

We tested the DisplayMux in modelSim by forcing values for both d0 and d1. These are the RGB value inputs from either a sprite or a background color. The mux successfully switches the output RGB value upon toggling of the s value.

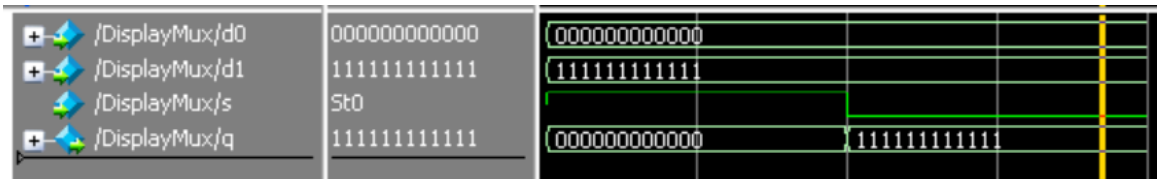

Figure 38: Testing the RGB output of the DisplayMux

5.3.6 SpriteMux Module

We tested the SpriteMux Module by forcing three different values for each sprite. Then we forced each possible hot-bit value, and a default value. The SpriteMux successfully toggles between each sprite input RGB value, and uses the background value when the hot bit input is invalid.

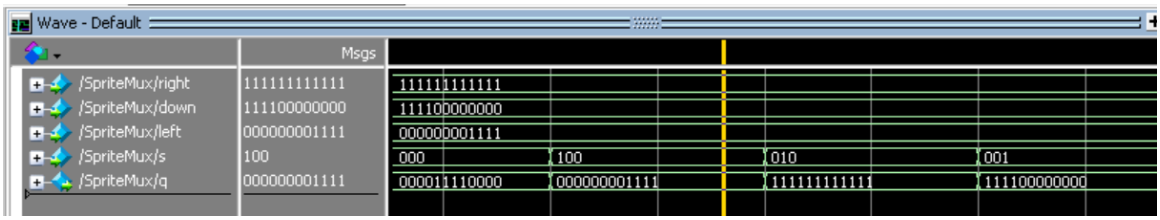

Figure 39: Testing the Sprite Mux module RGB output value.

5.3.7 Address Converter Module

We tested the Address Converter module by forcing values in the row and column input and observing the address output

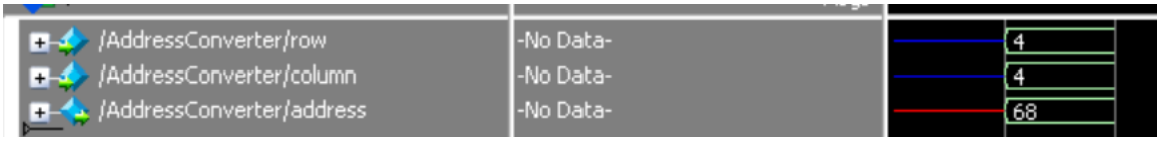

Figure 40: Testing the Address Converter Module address value

5.3.8 Synchronous Counter Module

We tested the Synchronous Counter Module by observing the pixel count, line count, h_sync, and v_sync values to make sure they were within the correct values and timings needed to display to a VGA monitor.

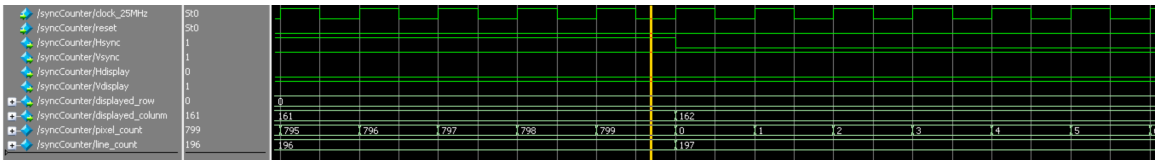

Figure 41: Testing the SyncCounter timings.

5.4 SevenSeg Module

We tested the SevenSeg module by forcing every possible hexadecimal value 0-f. This was done using a modelsim DO script. The outputs for each hexadecimal value was observed.

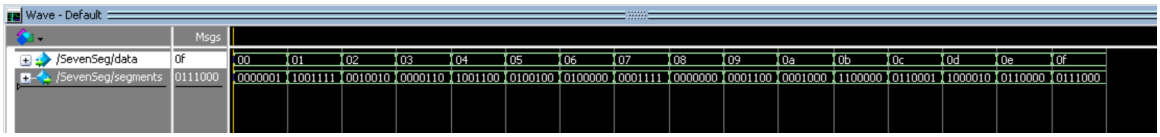

Figure 42: Modelsim waves while testing output values for Seven Segment controller using do script.

5.5 RGB Driver

5.5.1 RGB_Driver

Because all of the lower level modules were separately tested and simulated. the top level was simulated to test whether correct encoding and timing were sent to the addressable RGB. The first figure below shows the correct encoding for sending a 0 bit, and the second figure shows the correct encoding for sending a 1 bit.

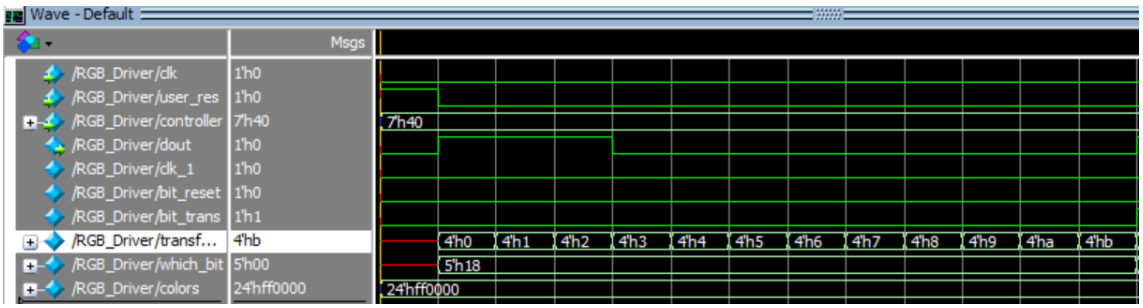

Figure 43: Correct timing and encoding for sending a 0 bit.

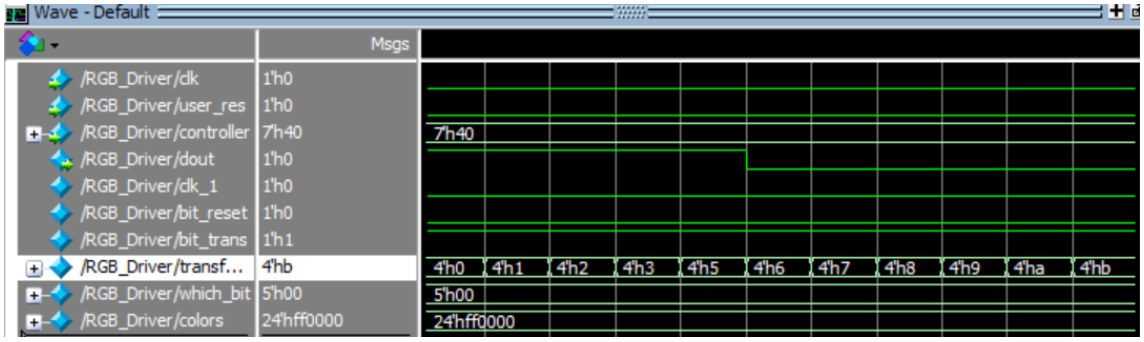

Figure 44: Correct timing and encoding for sending a 1 bit.

5.5.2 counter

The counter module was tested and simulated to observe whether the output incremented every clock cycle, and returns to 0 when reset is asserted. This behavior can be observed below:

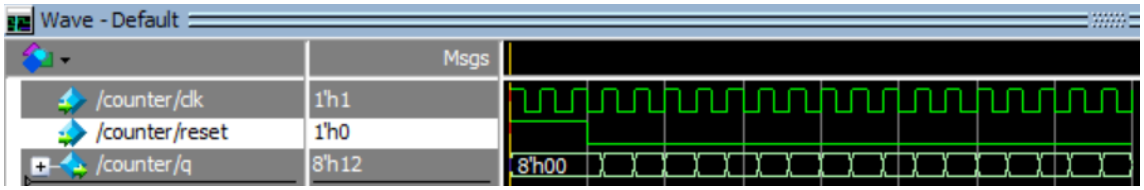

Figure 45: Output increments every clock cycle.

5.5.3 comparator

The comparator module was tested and simulated to observe whether the output signal was pulsed if and only if the two inputs were equal. This behavior can be observed below:

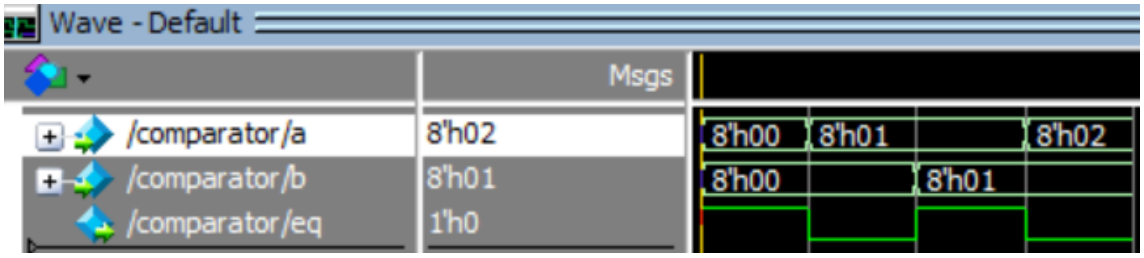

Figure 46: Output only when both inputs are equal.

5.5.4 sync

The sync module was tested and simulated to observer that an input is synced to a given clock signal, and sent in sync with the clock cycle. This behavior can be observed below:

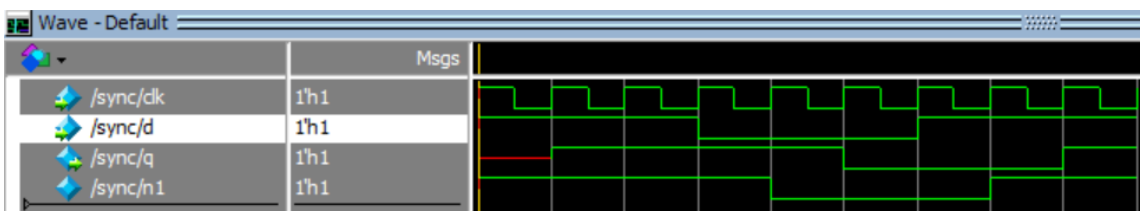

Figure 47: Input is passed to output in sync with clock.

5.5.5 freq_div

The freq_div module was tested and simulated to observe that if given an input clock cycle, whether it pulses it's clock output when it's internal counter reaches 50. This behavior can be observed below:

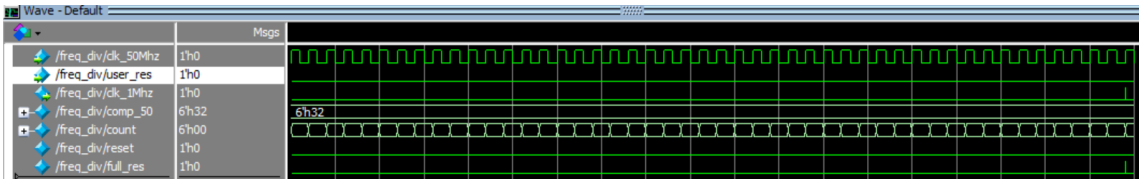

Figure 48: Output line is only pulsed when internal counter reaches 50.

5.5.6 cont_to_color

The cont_to_color module was tested for every possible controller input case. The expected behavior was that each controller input would have a unique corresponding color output. This behavior can be observed below:

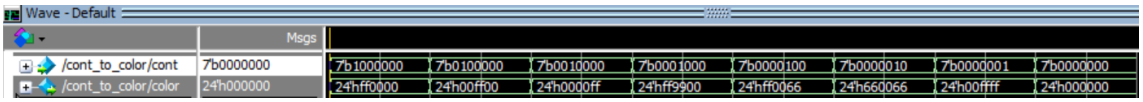

Figure 49: Each controller input has a unique color output.

5.5.7 trans_state

The trans_state module was tested to assert that the state was incremented every clock cycle, and that it reset when it reached 12. This behavior can be seen below:

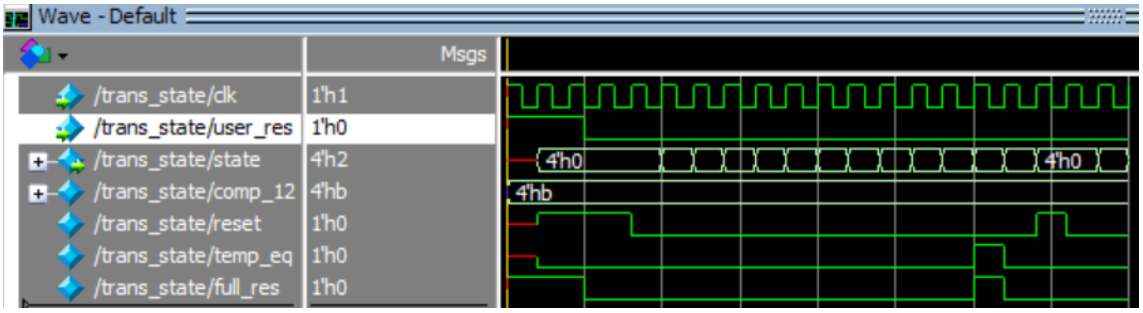

Figure 50: Output is continuously incremented every clock cycle until it reaches 12.

5.5.8 bit_state

The bit_state module was tested to assert that when the input state reaches 12, a inc signal is pulsed, which then increments the current bit state output. This behavior can be seen below:

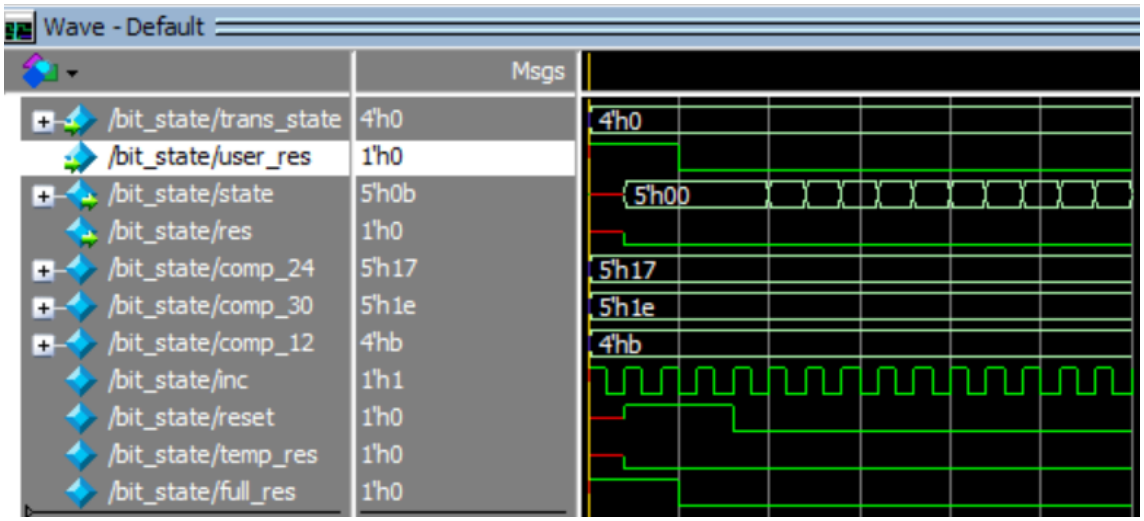

Figure 51: Bit state increments every time the input state reaches 12 and pulses the inc signal.

5.5.9 choose_bit

The choose_bit module was tested and simulated to assert that the appropriate bit from a color signal is sent, given a bit state. The behavior can be seen below:

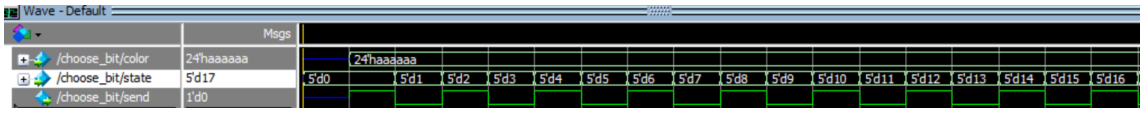

Figure 52: Appropriate bit from color signal is chosen based off of bit state signal to be sent to output.

5.5.10 transfer

The transfer module was tested and simulated to assert that given a data signal, it sent the correct signal encoding and timing to the output (which connects to the addressable RGB LEDs). The expected behavior for a data of 1 and a data of 0 can be seen below:

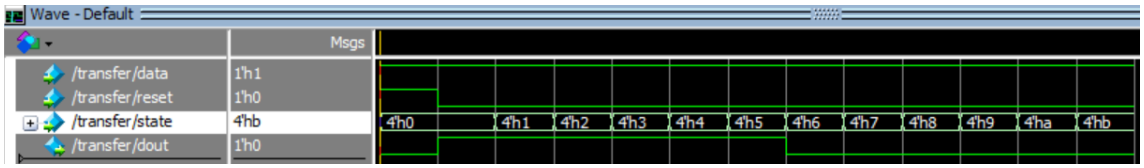

Figure 53: Expected signal encoding and timing given a data of 1.

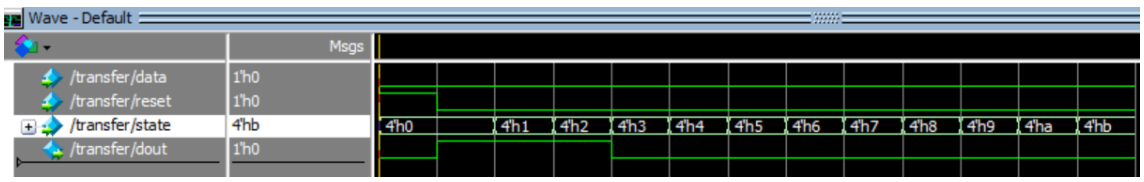

Figure 54: Expected signal encoding and timing given a data of 0.

6 Hardware Testing

6.0.1 VGA_Driver Module

The VGA Driver was tested on a VGA monitor. Durring this test, the input from the NES Controller module was emulated with three switches on the FPGA. A video demonstrating the VGA Driver working on hardware is avalible at: https://youtu.be/LUShy7jyn_Q .

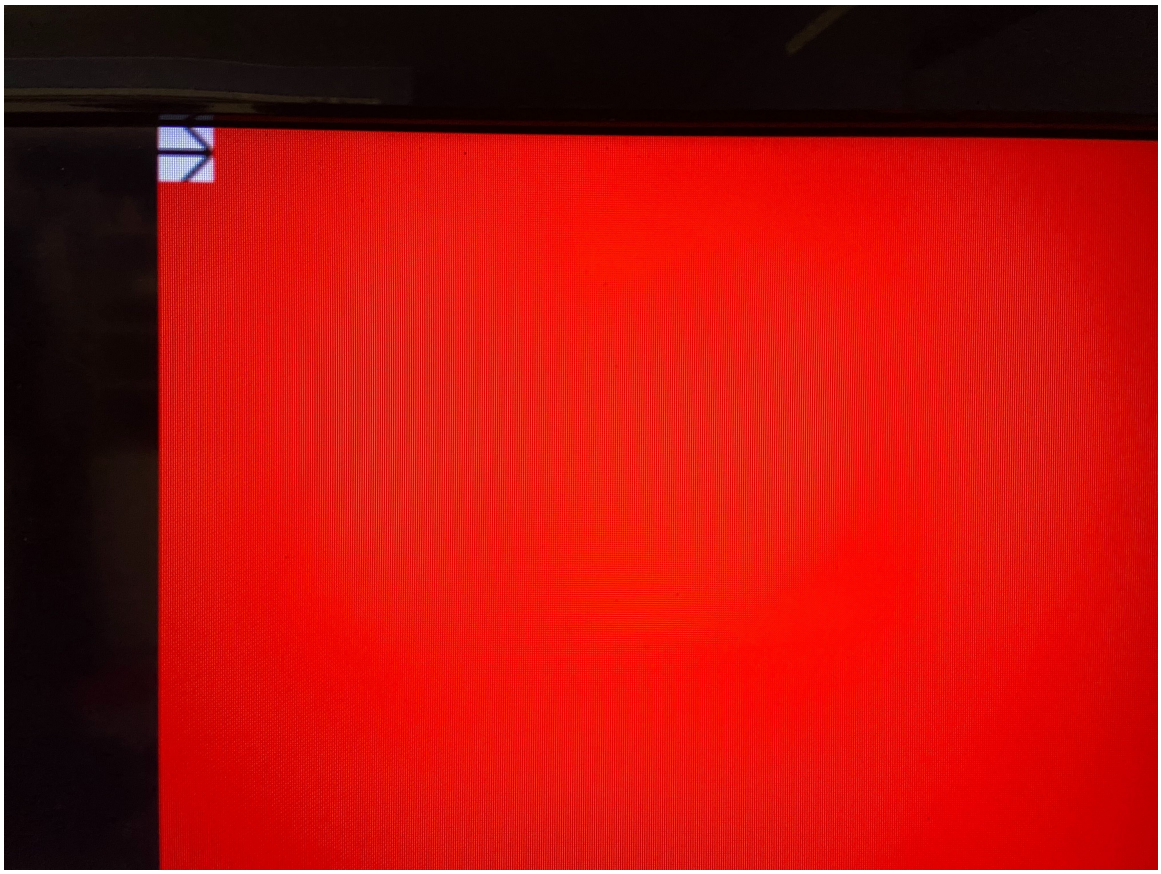

Figure 55: Right arrow sprite being shown during hardware testing.

A SystemVerilog Files

A.1 NES Decoder

A.1.1 NES_Decoder

```
1  //////////////////////////////////////////////////////////////////////////////////
2  // Company:          Oregon State University
3  // Engineer:         Brenden Lowe
4  //
5  // Create Date:      06/01/20
6  // Design Name:      NES Decoder
7  // Module Name:      NES_Decoder
8  // Project Name:     ECE 271 Design Project
9  // Target Devices:   Max 10 FPGA
10 // Description:      Top level module for NES Decoder. Identifies current controller
11 //                   state.
12 //
13 // Dependencies:
14 //
15 // Revision:
16 // Revision 0.01 - File Created
17 // additional Comments:
18 //
19 //////////////////////////////////////////////////////////////////////////////////
20
21 module NES_Decoder(input logic data, clk, reset,
22                    output logic nes_clk, nes_latch, a, b, select, start, up, down, left, right);
23         // Define a counter to assist in keeping track of current clock state
24         logic [3:0] count;
25
26         counter clk_state(clk, reset, count);
27         // Feed count into latch pulser
28         latch_pulse pulser(count, nes_latch);
29         // Feed count into clock pulser
```

```
30          clock_pulse clk_pulser(count, nes_clk);
31          // Feed count into contoller reader
32          get_controls control_read(count, data, reset, {a, b, select, start, up, down, left, right});
33
34  endmodule
```

A.1.2 counter

```
1   //////////////////////////////////////////////////////////////////////////////////
2   // Company:          Oregon State University
3   // Engineer:         Brenden Lowe
4   //
5   // Create Date:      06/01/20
6   // Design Name:      NES Decoder
7   // Module Name:      counter
8   // Project Name:     ECE 271 Design Project
9   // Target Devices:   Max 10 FPGA
10  // Description:      Increments output every clock cycle.
11  //
12  // Dependencies:
13  //
14  // Revision:
15  // Revision 0.01 - File Created
16  // additional Comments:
17  //
18  //////////////////////////////////////////////////////////////////////////////////
19
20  module counter(input logic clk, reset,
21                 output logic [3:0] q);
22
23          always_ff@(posedge clk)
24                  if(reset) q <= 0;
25                  else q <= q+1;
26
27  endmodule
```

A.1.3 latch_pulse

```
1   //////////////////////////////////////////////////////////////////////////////////
2   // Company:          Oregon State University
3   // Engineer:         Brenden Lowe
4   //
5   // Create Date:      06/01/20
6   // Design Name:      NES Decoder
7   // Module Name:      latch_pulse
8   // Project Name:     ECE 271 Design Project
9   // Target Devices:   Max 10 FPGA
10  // Description:      Pulses the NES latch line to signal inputs to be loaded.
11  //
12  // Dependencies:
13  //
14  // Revision:
15  // Revision 0.01 - File Created
16  // additional Comments:
17  //
18  //////////////////////////////////////////////////////////////////////////////////
19
20  module latch_pulse(input logic [3:0] count,
21                     output logic latch);
22
23              always_comb
24                      if(count == 0) latch = 1;
25                      else latch = 0;
26
27  endmodule
```

A.1.4 clock_pulse

```
1   //////////////////////////////////////////////////////////////////////////////////
2   // Company:          Oregon State University
3   // Engineer:         Brenden Lowe
4   //
5   // Create Date:      06/01/20
6   // Design Name:      NES Decoder
7   // Module Name:      clock_pulse
8   // Project Name:     ECE 271 Design Project
9   // Target Devices:   Max 10 FPGA
10  // Description:      Pulses the NES clock line to signal inputs to be transferred.
11  //
12  // Dependencies:
13  //
14  // Revision:
15  // Revision 0.01 - File Created
16  // additional Comments:
17  //
18  //////////////////////////////////////////////////////////////////////////////////
19
20  // Define module to pulse clock signals to NES controller
21  module clock_pulse(input logic [3:0] count,
22                                                  output logic nes_clk);
23
24              always_comb
25                      case(count)
26                                      2: nes_clk = 1;
27                                      4: nes_clk = 1;
28                                      6: nes_clk = 1;
29                                      8: nes_clk = 1;
30                                      10: nes_clk = 1;
31                                      12: nes_clk = 1;
32                                      14: nes_clk = 1;
33                                      default: nes_clk = 0;
34                      endcase
35
36  endmodule
```

A.1.5 get_controls

```
////////////////////////////////////////////////////////////////////////////////
// Company:         Oregon State University
// Engineer:        Brenden Lowe
//
// Create Date:     06/01/20
// Design Name:     NES Decoder
// Module Name:     get_controls
// Project Name:    ECE 271 Design Project
// Target Devices:  Max 10 FPGA
// Description:     Receives info from NES data line, and maps to corresponding
//                  output.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module get_controls(input logic [3:0] count,
                                                input logic data, reset,
                                                output logic [7:0] buttons);

        always_ff@(posedge count[0])
                if(reset) buttons <= 0;
                else case(count)
                        1: buttons[7] <= data;
                        3: buttons[6] <= data;
                        5: buttons[5] <= data;
                        7: buttons[4] <= data;
                        9: buttons[3] <= data;
                        11: buttons[2] <= data;
                        13: buttons[1] <= data;
                        15: buttons[0] <= data;
                        default: buttons = buttons;
                endcase

endmodule
```

A.2 PS/2 Driver

A.2.1 ps2keyboard

```
////////////////////////////////////////////////////////////////////////
//Engineers : Michael Boly
//
//Create Date: 05/28/2020
//Design Name: PS2Keyboard
//Module Name: ps2keyboard
//Project Name: ECE 271 Final Project
//Target Devices : Max10 FPGA
//Description : This module simulates the function of an PS2 Keyboard.
//
//Dependencies :
//Revision : Revision 0.01 File Created
//additional Comments:
//
////////////////////////////////////////////////////////////////////////////
module ps2keyboard (input logic clk_10, reset_b, enable_switch,
                                                input logic A, B, C, D, E, F, zero , one , two , three
                                                        , four , five , six , seven , eight , nine,
                                                output logic [6:0] seg0);

        logic [7:0] decode_bus;
        logic [7:0] state;
        logic ps2_new_wire;
        logic ps2_data_wire;
        logic ps2_clock_wire ;

        /*******************************
        takes keyboard inputs A to Z, 0 to 9 and
        outputs an 8 digit hexadecimal key
        *******************************/
        keyboard_input ps2_keyboardinput (
                .A(A),
                .B(B),
                .C(C),
                .D(D),
                .E(E),
                .F(F),
                .zero (zero),
                .one (one),
                .two (two),
                .three (three),
                .four (four),
                .five (five),
                .six (six),
                .seven (seven),
                .eight (eight),
                .nine (nine),
                .q (state)
        );

        /*******************************
        takes outputs of keyboardClock and
        determines the code that was
        initially created by ps2_keyboardinput
        *******************************/
        ps2controller controller (
                .clock (clk_10),
                .reset_n (reset_b),
                .enable (enable_switch),
                .ps2_clock (ps2_clock_wire),
                .ps2_data (ps2_data_wire),
                .ps2_code_new (ps2_new_wire),
                .ps2_code (decode_bus)
        );

        /*******************************
        takes the code outputted by
        ps2_keyboardinput and sends the
        coded message to a host
```

```
69          *******************************/
70          ps2clock keyboardClock (
71                  .clock (clk_10),
72                  .state (state),
73                  .enable (enable_switch),
74                  .reset_n (reset_b),
75                  .ps2_clock (ps2_clock_wire),
76                  .ps2_data (ps2_data_wire)
77          );
78
79          /*******************************
80          Displays the code created by
81          ps2_keyboardinput
82          *******************************/
83
84          sevenseg display0 (
85                  .data (decode_bus [7:0]),
86                  .segments (seg0)
87          );
88
89
90  endmodule
```

A.2.2 ps2clock

```
1   //////////////////////////////////////////////////////////////////////////////////
2   //Engineers : Michael Boly
3   //
4   //Create Date: 05/28/2020
5   //Design Name: PS2Keyboard
6   //Module Name: ps2clock
7   //Project Name: ECE 271 Final Project
8   //Target Devices : Max10 FPGA
9   //Description : This module outputs the cooresponding keycode for a key press.
10  //
11  //Dependencies :
12  //Revision : Revision 0.01 File Created
13  //additional Comments:
14  //
15  //////////////////////////////////////////////////////////////////////////////////
16  module ps2clock ( input logic clock,
17                                          input logic [7:0] state,
18                                               input logic enable,
19                                               input logic reset_n,
20                                               output logic ps2_clock,
21                                               output logic ps2_data);
22
23          logic [4:0] count;
24          logic lt;
25          logic parity;
26
27          assign ps2_clock = count [0];
28          assign parity = !(( state [7] + state [6] + state [5] + state [4] + state [3] + state [2] +
                state [1] + state [0]) % 2);
29
30          always_ff@(posedge clock, negedge reset_n)
31          begin
32                  if (!reset_n) count <= 0;
33                  else if (enable) count <= count;
34
35                  case (count)
36                          0: ps2_data = 0;
37                          2: ps2_data = state [7];
38                          4: ps2_data = state [6];
39                          6: ps2_data = state [5];
40                          8: ps2_data = state [4];
41                          10: ps2_data = state [3];
42                          12: ps2_data = state [2];
43                          14: ps2_data = state [1];
44                          16: ps2_data = state [0];
45                          18: ps2_data = parity;
46                          20: ps2_data = 1;
47                          21: ps2_data = 1;
48                          default ps2_data = ps2_data;
49                  endcase
50
51                  if (~count == 21) //Using ~ to prevent user heirarchy error
52                          count <= 0;
53                  else
54                          count <= count + 5'b1;
55          end
56  endmodule
```

A.2.3 ps2controller

```
1   //////////////////////////////////////////////////////////////////////////////////
2   //Engineers : Michael Boly
3   //
4   //Create Date: 05/28/2020
5   //Design Name: PS2Keyboard
6   //Module Name: ps2controller
7   //Project Name: ECE 271 Final Project
8   //Target Devices : Max10 FPGA
9   //Description : This module stores the outputs of ps2clock in a shift register.
10  //
11  //Dependencies :
12  //Revision : Revision 0.01 File Created
13  //additional Comments:
14  //
15  //////////////////////////////////////////////////////////////////////////////////
16  module ps2controller (input logic clock,
17                                                          input logic reset_n,
18                                                          input logic enable,
19                                                          input logic ps2_clock,
20                                                          input logic ps2_data,
21                                                          output logic ps2_code_new,
22                                                          output logic [7:0] ps2_code);
23
24          logic [10:0] ps2_register;
25          logic ps2_clock_int;
26          logic ps2_data_int;
27          logic [11:0] count_idle;
```

```
28          logic count_check;
29
30          shiftRegister #(.N(11)) shiftReg(
31                  .clk (ps2_clock),
32                  .reset (reset_n),
33                  .d (ps2_data),
34                  .state (ps2_register) //shiftRegister "register" input not assigned but "state" output
                        is
35          );
36
37          flop ps2code(
38                  .clk (clock),
39                  .reset (reset_n),
40                  .enable (count_check),
41                  .d (ps2_register [8:1]),
42                  .q (ps2_code)
43          );
44
45          flop #(.N(1)) newps2Code(
46                  .clk (clock),
47                  .reset (reset_n),
48                  .enable (0) ,
49                  .d (!count_check),
50                  .q (ps2_code_new)
51          );
52
53          idleCounter idleCount (
54                  .clk (clock),
55                  .rst (count_check),
56                  .count (count_idle)
57          );
58
59          lessThan #(.N(12), .a(21)) count21(
60                  .d (count_idle),
61                  .q (count_check)
62          );
63
64  endmodule
```

A.2.4 keyboard_input

```
1   //////////////////////////////////////////////////////////////////////////////////
2   //Engineers : Michael Boly
3   //
4   //Create Date: 05/28/2020
5   //Design Name: PS2Keyboard
6   //Module Name: keyboard_input
7   //Project Name: ECE 271 Final Project
8   //Target Devices : Max10 FPGA
9   //Description : This module defines the output 8-bit hex for any key press.
10  //
11  //Dependencies :
12  //Revision : Revision 0.01 File Created
13  //additional Comments:
14  //
15  //////////////////////////////////////////////////////////////////////////////////
16  module keyboard_input(input logic A, B, C, D, E, F, zero, one, two, three, four, five, six, seven, eight
        , nine,
17                                                                  output logic [7:0] q);
18
19          always_comb
20          begin
21                  if (A) q = 8'h1C;
22                  else if (B) q = 8'h32;
23                  else if (C) q = 8'h21;
24                  else if (D) q = 8'h23;
25                  else if (E) q = 8'h24;
26                  else if (F) q = 8'h2B;
27                  else if (zero) q = 8'h45;
28                  else if (one) q = 8'h16;
29                  else if (two) q = 8'h1E;
30                  else if (three) q = 8'h26;
31                  else if (four) q = 8'h25;
32                  else if (five) q = 8'h2E;
33                  else if (six) q = 8'h36;
34                  else if (seven) q = 8'h3D;
35                  else if (eight) q = 8'h3E;
36                  else if (nine) q = 8'h46;
37                  else q = 8'h45;
38          end
39  endmodule
```

A.2.5 keybuffer

```
1   //////////////////////////////////////////////////////////////////////////////////
2   //Engineers : Michael Boly
3   //
4   //Create Date: 05/28/2020
5   //Design Name: PS2Keyboard
6   //Module Name: keybuffer
7   //Project Name: ECE 271 Final Project
8   //Target Devices : Max10 FPGA
9   //Description : This module is an input buffer to handle simultaneous key presses.
10  //
11  //Dependencies :
12  //Revision : Revision 0.01 File Created
13  //additional Comments:
14  //
15  //////////////////////////////////////////////////////////////////////////////////
16  module keybuffer(input logic [7:0] input_keycode,
17                                     input logic pass_input,
18                                     input logic reset,
19                                     output logic [7:0] output_keycode
20  );
21
22          //Register to store up to 30 characters in the buffer.
23          logic [239:0] keystore = 0;
24          //Count that counts how many characters are in the buffer
25          logic [4:0] charCounter = 0;
26
27          always_comb
28          begin
```

```
29
30                  //Reset logic low
31                  if(reset == 0)
32                  begin
33                          charCounter = 0;
34                          keystore = 0;
35                  end
36                  else
37                  //The buffer cannot get input and send input at the same time
38                  if(input_keycode != 0 && charCounter < 5'd30)
39                  begin
40                          keystore[(5'd8 * charCounter) +: 5'd8] = input_keycode;
41                          charCounter = charCounter + 5'b00001;
42                  end
43
44
45                  if(pass_input && charCounter != 0)
46                  begin
47                          output_keycode = keystore[7:0];
48                          keystore[7:0] = 0;
49                          keystore = keystore >>> 8;
50                          charCounter = charCounter - 5'b00001;
51                  end
52          end
53
54  endmodule
```

A.2.6 shiftRegister

```
1   //////////////////////////////////////////////////////////////////////////////////
2   //Engineers : Michael Boly
3   //
4   //Create Date: 05/28/2020
5   //Design Name: PS2Keyboard
6   //Module Name: shiftRegister
7   //Project Name: ECE 271 Final Project
8   //Target Devices : Max10 FPGA
9   //Description : This module shifts the data coming from ps2clock module.
10  //
11  //Dependencies :
12  //Revision : Revision 0.01 File Created
13  //additional Comments:
14  //
15  //////////////////////////////////////////////////////////////////////////////////
16  module shiftRegister #(parameter N=7)(input logic clk, enable, input logic reset, input logic d, output
        logic q, output logic [N-1:0] state);
17
18          logic [N-1:0] count;
19
20          always_ff@(negedge clk, negedge reset)
21          begin
22                  if (!reset) count <= 0;
23                  else count <= count << 1;
24                  count [0] <= d;
25          end
26
27          assign q = count [N-1];
28          assign state = count;
29  endmodule
```

A.2.7 lessThan

```
1   //////////////////////////////////////////////////////////////////////////////////
2   //Engineers : Michael Boly
3   //
4   //Create Date: 05/28/2020
5   //Design Name: PS2Keyboard
6   //Module Name: lessThan
7   //Project Name: ECE 271 Final Project
8   //Target Devices : Max10 FPGA
9   //Description : This module is a compartor that performs the less than comparison.
10  //
11  //Dependencies :
12  //Revision : Revision 0.01 File Created
13  //additional Comments:
14  //
15  //////////////////////////////////////////////////////////////////////////////////
16  module lessThan #(parameter a=10, parameter N=5) ( input logic [N-1:0] d,
17
18
19          always_comb
20          begin
21                  if(d < a) q = 1;
22                  else q = 0;
23          end
24
25  endmodule
```

A.2.8 idleCounter

```
1   //////////////////////////////////////////////////////////////////////////////////
2   //Engineers : Michael Boly
3   //
4   //Create Date: 05/28/2020
5   //Design Name: PS2Keyboard
6   //Module Name: idleCounter
7   //Project Name: ECE 271 Final Project
8   //Target Devices : Max10 FPGA
9   //Description : This module is a simple counter with an active low reset.
10  //
11  //Dependencies :
12  //Revision : Revision 0.01 File Created
```

```
//additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module idleCounter (input logic clk,
                                                input logic rst,
                                                output logic [11:0] count);

        always_ff@(posedge clk)
        begin
                if (!rst) count <= 0;
                else count <= count + 1;
        end

endmodule
```

A.2.9 flop

```
//////////////////////////////////////////////////////////////////////////////////
//Engineers : Michael Boly
//
//Create Date: 05/28/2020
//Design Name: PS2Keyboard
//Module Name: flop
//Project Name: ECE 271 Final Project
//Target Devices : Max10 FPGA
//Description : This module is a register that stores N bits.
//
//Dependencies :
//Revision : Revision 0.01 File Created
//additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module flop #(parameter N=8)(input logic clk,
                                                                        input logic reset,
                                                                        input logic enable,
                                                                        input logic [N-1:0]d,
                                                                        output logic [N-1:0]q);

always_ff@(posedge clk, negedge reset)
begin
        if(!reset) q <= 8'b0;
        else if (enable) q <= q;
        else q <= d;
end

endmodule
```

A.2.10 sevenseg

```
//////////////////////////////////////////////////////////////////////////////////
//Engineers : Michael Boly
//
//Create Date: 05/28/2020
//Design Name: PS2Keyboard
//Module Name: sevenseg
//Project Name: ECE 271 Final Project
//Target Devices : Max10 FPGA
//Description : This module will display the state on the FPGA display.
//
//Dependencies :
//Revision : Revision 0.01 File Created
//additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module sevenseg(input logic [7:0] data,
                                        output logic [6:0] segments);
 always_comb
        case(data)
                                                                                // abc_defg
        8'b01000101: segments = 7'b100_0000; //0
        8'b00010110: segments = 7'b111_1001; //1
        8'b00011110: segments = 7'b010_0100; //2
        8'b00100110: segments = 7'b011_0000; //3
        8'b00100101: segments = 7'b001_1001; //4
        8'b00101110: segments = 7'b001_0010; //5
        8'b00110110: segments = 7'b000_0010; //6
        8'b00111101: segments = 7'b111_1000; //7
        8'b00111110: segments = 7'b000_0000; //8
        8'b01000110: segments = 7'b001_1000; //9
        8'b00011100: segments = 7'b000_1000; //A
        8'b00110010: segments = 7'b110_0000; //B
        8'b00100001: segments = 7'b011_0001; //C
        8'b00100011: segments = 7'b100_0010; //D
        8'b00100100: segments = 7'b011_0000; //E
        8'b00101011: segments = 7'b011_1000; //F

        default: segments = 7'b111_1111;
 endcase
endmodule
```

A.3 VGA_Driver

A.3.1 Comparator

```
//////////////////////////////////////////////////////////////////////////////////
// Engineer:        David Headrick
//
// Create Date:     05/29/2020
// Design Name:     VGA_Driver
// Module Name:     Comparator
// Project Name:
// Target Devices: DE10-Lite
// Description:     Compares to see if the current pixel is less than the parameter
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
```

```
// additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module Comparator  #(parameter M = 10)
                   (input  logic[9:0] a,
                    output logic altM);


    always_comb
           altM = a < M;


  endmodule
```

A.3.2 SpriteMux

```
//////////////////////////////////////////////////////////////////////////////////
// Engineer:        David Headrick
//
// Create Date:     05/29/2020
// Design Name:     VGA_Driver
// Module Name:     VGA_Display
// Project Name:
// Target Devices: DE10-Lite
// Description:     Chooses which sprite to display based off of the 3 bit switch value.
//                  Uses hot bit encoding.
//
// Dependencies: clockdiv.sv, syncCounter.sv
//
// Revision:
// Revision 0.01 - File Created
// additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module SpriteMux  (input  logic[11:0] right,
                   input  logic[11:0] down,
                   input  logic[11:0] left,
                   input  logic[2:0] s,
                   output logic[11:0] q);

  always_comb
  begin
          case(s)
            3'b100 : q = left;
                 3'b010 : q = right;
                 3'b001 : q = down;
                 default : q = 12'b000011110000;
          endcase
  end


  endmodule
```

A.3.3 DisplayMux

```
//////////////////////////////////////////////////////////////////////////////////
// Engineer:        David Headrick
//
// Create Date:     05/29/2020
// Design Name:     VGA_Driver
// Module Name:     DisplayMux
// Project Name:
// Target Devices: DE10-Lite
// Description:     Chooses to display the background or the sprite
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module DisplayMux  (input  logic[11:0] d0,
                                          input  logic[11:0] d1,
                                               input  logic s,
                    output logic[11:0] q);


    assign q = s ? d0 : d1;

  endmodule
```

A.3.4 AddressConverter

```
//////////////////////////////////////////////////////////////////////////////////
// Engineer:        David Headrick
//
// Create Date:     05/29/2020
// Design Name:     VGA_Driver
// Module Name:     AddressConverter
// Project Name:
// Target Devices: DE10-Lite
// Description:     Converts the current pixel location to a address for sprite in ROM
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module AddressConverter  #(parameter ROWVALUE = 16)
                  (input  logic [9:0] row,
                                          input  logic [9:0] column,
                   output  logic [7:0] address);

  assign address = row * ROWVALUE + column;

endmodule
```

A.3.5 VGA_Display

```
//////////////////////////////////////////////////////////////////////////////////
// Engineer:        David Headrick
//
// Create Date:     05/29/2020
// Design Name:     VGA_Driver
// Module Name:     VGA_Display
// Project Name:
// Target Devices: DE10-Lite
// Description:     Toplevel for synchronizing VGA output
//
// Dependencies: clockdiv.sv, syncCounter.sv
//
// Revision:
// Revision 0.01 - File Created
// additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module VGA_Display        (input  logic clock_50MHz,
                                       input  logic reset_n,
                                                    input  logic[11:0] Pixel_Colors_RGB,
                                       output logic[9:0] displayed_row,
                                                    output logic[9:0] displayed_colunm,
                                                    output logic[11:0] RGB_Outputs,
                                                    output logic h_sync,
                                                    output logic v_sync);
   wire clock_25MHz;
   logic Hdisplay;
   logic Vdisplay;
   logic[11:0] RGB_Temp = 12'b111100000000;
   wire hsyncwire;
   wire vsyncwire;


      clockdiv #(2) clockmod (clock_50MHz, reset_n, clock_25MHz);
      syncCounter sc (clock_25MHz, reset_n, hsyncwire, vsyncwire, Hdisplay, Vdisplay, displayed_row,
            displayed_colunm);

            assign h_sync = hsyncwire;
            assign v_sync = vsyncwire;

            assign RGB_Outputs = (Hdisplay == 1 && Vdisplay == 1) ? Pixel_Colors_RGB : 0;


endmodule
```

A.3.6 syncCounter

```
//////////////////////////////////////////////////////////////////////////////////
// Engineer:        David Headrick
//
// Create Date:     05/29/2020
// Design Name:     VGA_Driver
// Module Name:     syncCounter
// Project Name:
// Target Devices: DE10-Lite
// Description:     Generates VGA signals from 25MHz clock
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module syncCounter        (input    wire clock_25MHz,
                                                    input    logic reset,
                                       output  logic Hsync,
                                                    output   logic Vsync,
                                                    output   logic Hdisplay,
                                                    output   logic Vdisplay,
                                                    output   logic[9:0] displayed_row,
                                                    output   logic[9:0] displayed_colunm);

   reg[9:0] pixel_count;  //max count 800
   reg[9:0] line_count;  //max count 525

   always_ff @(posedge clock_25MHz, posedge reset)
     begin

                if(reset)
                begin
                  pixel_count <= 0;
                  line_count  <= 0;
                end
                else if(pixel_count >= 799)
                begin
                  pixel_count <= 0;
              line_count  <= line_count + 1;
                end
                else
                begin
                  pixel_count  <= pixel_count + 1;
                  if(line_count > 525)
                    line_count <= 0;
            end

     end

          always_comb
          begin

        Hsync = ~((pixel_count >= 0) && (pixel_count <= 96));

        Vsync = ~(line_count > 0 && line_count <= 2);

        if((pixel_count > 144) && (pixel_count <= 784))
                  begin
                    Hdisplay = 1;
                          displayed_row = pixel_count - 144;
                  end
                else
```

```
65                      begin
66                        Hdisplay = 0;
67                              displayed_row = 0;
68                      end
69
70          if((line_count > 35) && (line_count <= 515))
71                      begin
72                        Vdisplay = 1;
73                              displayed_colunm = line_count - 35;
74                      end
75                    else
76                      begin
77                        Vdisplay = 0;
78                              displayed_colunm = 0;
79                      end
80
81            end
82
83  endmodule
```

A.3.7 clockdiv

```
1   //////////////////////////////////////////////////////////////////////////////////
2   // Engineer:        David Headrick
3   //
4   // Create Date:     05/29/2020
5   // Design Name:     VGA_Driver
6   // Module Name:     clockdiv
7   // Project Name:
8   // Target Devices: DE10-Lite
9   // Description:     Divides the clock by the parameter value
10  //
11  // Dependencies:
12  //
13  // Revision:
14  // Revision 0.01 - File Created
15  // additional Comments:
16  //
17  //////////////////////////////////////////////////////////////////////////////////
18  module clockdiv  #(parameter D = 2)
19                   (input  logic clk,
20                                                input  logic reset,
21                    output  logic c_out);
22
23    reg[28:0] count;
24
25    always @(posedge clk, posedge reset)
26      begin
27              if (reset) count <= 1;
28         else if (count >= D) count <= 1;
29                 else count <= count + 1;
30      end
31
32    assign c_out = (count > (D/2))?1:0;
33  endmodule
```

A.3.8 VGA_Driver

```
1   //////////////////////////////////////////////////////////////////////////////////
2   // Engineer:        David Headrick
3   //
4   // Create Date:     05/29/2020
5   // Design Name:     VGA_Display
6   // Module Name:     VGA_Display
7   // Project Name:
8   // Target Devices: DE10-Lite
9   // Description:     Toplevel for driving a VGA signal
10  //
11  // Dependencies: AddressConverter.sv, SpriteMux.sv, Comparator.sv, DisplayMux.sv, VGA_Display.sv
12  //
13  // Revision:
14  // Revision 0.01 - File Created
15  // additional Comments:
16  //
17  //////////////////////////////////////////////////////////////////////////////////
18  // Copyright (C) 2019  Intel Corporation. All rights reserved.
19  // Your use of Intel Corporation's design tools, logic functions
20  // and other software and tools, and any partner logic
21  // functions, and any output files from any of the foregoing
22  // (including device programming or simulation files), and any
23  // associated documentation or information are expressly subject
24  // to the terms and conditions of the Intel Program License
25  // Subscription Agreement, the Intel Quartus Prime License Agreement,
26  // the Intel FPGA IP License Agreement, or other applicable license
27  // agreement, including, without limitation, that your use is for
28  // the sole purpose of programming logic devices manufactured by
29  // Intel and sold by Intel or its authorized distributors.  Please
30  // refer to the applicable agreement for further details, at
31  // https://fpgasoftware.intel.com/eula.
32
33  // PROGRAM              "Quartus Prime"
34  // VERSION              "Version 19.1.0 Build 670 09/22/2019 SJ Lite Edition"
35  // CREATED              "Sun May 31 18:25:23 2020"
36
37  module Final_Proj(
38           clock_50MHz,
39           reset,
40           s2,
41           s1,
42           s0,
43           hsync,
44           vsync,
45           RGB_OUT
46  );
47
48
49  input wire       clock_50MHz;
50  input wire       reset;
51  input wire       s2;
52  input wire       s1;
53  input wire       s0;
54  output wire      hsync;
```

```
55  output wire      vsync;
56  output wire      [11:0] RGB_OUT;
57
58  wire     [11:0] Background;
59  wire     [9:0] displayed_colunm;
60  wire     [9:0] displayed_row;
61  wire     [11:0] input_color;
62  wire     [7:0] SYNTHESIZED_WIRE_10;
63  wire     SYNTHESIZED_WIRE_2;
64  wire     SYNTHESIZED_WIRE_3;
65  wire     [11:0] SYNTHESIZED_WIRE_5;
66  wire     [11:0] SYNTHESIZED_WIRE_6;
67  wire     [11:0] SYNTHESIZED_WIRE_7;
68  wire     SYNTHESIZED_WIRE_8;
69  wire     [11:0] SYNTHESIZED_WIRE_9;
70
71  wire     [2:0] GDFX_TEMP_SIGNAL_0;
72
73
74  assign   GDFX_TEMP_SIGNAL_0 = {s2,s1,s0};
75
76
77  VGA_Display      b2v_inst(
78          .clock_50MHz(clock_50MHz),
79          .reset_n(reset),
80          .Pixel_Colors_RGB(input_color),
81          .h_sync(hsync),
82          .v_sync(vsync),
83          .displayed_colunm(displayed_colunm),
84          .displayed_row(displayed_row),
85          .RGB_Outputs(RGB_OUT));
86
87
88  left     b2v_inst10(
89          .clock(clock_50MHz),
90          .address(SYNTHESIZED_WIRE_10),
91          .q(SYNTHESIZED_WIRE_6));
92
93
94  lpm_constant_0   b2v_inst11(
95          .result(Background));
96
97
98  RIGHT    b2v_inst12(
99          .clock(clock_50MHz),
100         .address(SYNTHESIZED_WIRE_10),
101         .q(SYNTHESIZED_WIRE_7));
102
103 assign   SYNTHESIZED_WIRE_8 = SYNTHESIZED_WIRE_2 & SYNTHESIZED_WIRE_3;
104
105
106 Comparator       b2v_inst4(
107         .a(displayed_row),
108         .altM(SYNTHESIZED_WIRE_2));
109         defparam         b2v_inst4.M = 16;
110
111
112 Comparator       b2v_inst5(
113         .a(displayed_colunm),
114         .altM(SYNTHESIZED_WIRE_3));
115         defparam         b2v_inst5.M = 16;
116
117
118 Down     b2v_inst6(
119         .clock(clock_50MHz),
120         .address(SYNTHESIZED_WIRE_10),
121         .q(SYNTHESIZED_WIRE_5));
122
123
124 AddressConverter         b2v_inst7(
125         .column(displayed_colunm),
126         .row(displayed_row),
127         .address(SYNTHESIZED_WIRE_10));
128         defparam         b2v_inst7.ROWVALUE = 16;
129
130
131 SpriteMux        b2v_inst8(
132         .down(SYNTHESIZED_WIRE_5),
133         .left(SYNTHESIZED_WIRE_6),
134         .right(SYNTHESIZED_WIRE_7),
135         .s(GDFX_TEMP_SIGNAL_0),
136         .q(SYNTHESIZED_WIRE_9));
137
138
139 DisplayMux       b2v_inst9(
140         .s(SYNTHESIZED_WIRE_8),
141         .d0(SYNTHESIZED_WIRE_9),
142         .d1(Background),
143         .q(input_color));
144
145
146 endmodule
147
148 module lpm_constant_0(result);
149 /* synthesis black_box */
150
151 output [11:0] result;
152
153 endmodule
```

A.4 Seven Segment Display Driver

A.4.1 SevenSeg

```
1  //////////////////////////////////////////////////////////////////////////////////
2  // Engineer:        David Headrick
3  //
4  // Create Date:     05/29/2020
5  // Design Name:     SevenSeg
6  // Module Name:     SevenSeg
7  // Project Name:
8  // Target Devices: DE10-Lite
9  // Description:     Drives a seven segment display based off of an 8 bit bus input
```

```
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// additional Comments: Derived from Matthew Shuman's seven segment driver design
//
//////////////////////////////////////////////////////////////////////////////////
module SevenSeg   (input   logic[7:0] data,
                                                output   logic[0:6] segments);
  always_comb
    case(data)
          8'b00000000:          segments = 7'b000_0001; //0
          8'b00000001:          segments = 7'b100_1111; //1
          8'b00000010:          segments = 7'b001_0010; //2
          8'b00000011:          segments = 7'b000_0110; //3
          8'b00000100:          segments = 7'b100_1100; //4
          8'b00000101:          segments = 7'b010_0100; //5
          8'b00000110:          segments = 7'b010_0000; //6
          8'b00000111:          segments = 7'b000_1111; //7
          8'b00001000:          segments = 7'b000_0000; //8
          8'b00001001:          segments = 7'b000_1100; //9
          8'b00001010:          segments = 7'b000_1000; //a
          8'b00001011:          segments = 7'b110_0000; //b
          8'b00001100:          segments = 7'b011_0001; //c
          8'b00001101:          segments = 7'b100_0010; //d
          8'b00001110:          segments = 7'b011_0000; //e
          8'b00001111:          segments = 7'b011_1000; //f
          default:              segments = 7'b111_1111;
  endcase
endmodule
```

A.5 RGB Driver

A.5.1 RGB_Driver

```
//////////////////////////////////////////////////////////////////////////////////
// Company:         Oregon State University
// Engineer:        Brenden Lowe
//
// Create Date:     06/01/20
// Design Name:     RGB Driver
// Module Name:     RGB_Driver
// Project Name:    ECE 271 Design Project
// Target Devices: Max 10 FPGA
// Description:     Top level of RGB Driver. Converts controller input to color
//                  output.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module RGB_Driver (input logic clk, user_res,
                                                input logic[6:0] controller,
                                                output logic dout);

        // Define intermediate nodes
        logic clk_1, bit_reset, bit_trans;
        logic [3:0] tranfer_state;
        logic [4:0] which_bit;
        logic [23:0] colors;

        // Feed clk into frequency divider
        freq_div to_one (
                .clk_50Mhz (clk),
                .user_res (user_res),
                .clk_1Mhz (clk_1)
        );

        // Feed controller input into colors
        cont_to_color to_color (
                .cont (controller),
                .color (colors)
        );

        // Feed clk_1 into transfer state module
        trans_state to_trans (
                .clk (clk_1),
                .user_res (user_res),
                .state (transfer_state)
        );

        // Feed transfer state to get bit state
        bit_state to_bit (
                .trans_state (transfer_state),
                .user_res (user_res),
                .state (which_bit),
                .res (bit_reset)
        );

        // Feed bit state to get appropriate bit to transfer
        choose_bit to_bit_trans (
                .color (colors),
                .state (which_bit),
                .send (bit_trans)
        );

        // Feed transfer bit into transfer module
        transfer to_transfer (
                .data (bit_trans),
                .reset (bit_reset),
                .state (transfer_state),
                .dout (dout)
        );
endmodule
```

A.5.2 counter

```
1  //////////////////////////////////////////////////////////////////////////////////
2  // Company:          Oregon State University
3  // Engineer:         Brenden Lowe
4  //
5  // Create Date:      06/01/20
6  // Design Name:      RGB Driver
7  // Module Name:      counter
8  // Project Name:     ECE 271 Design Project
9  // Target Devices:   Max 10 FPGA
10 // Description:      Increments output every clock cycle.
11 //
12 // Dependencies:
13 //
14 // Revision:
15 // Revision 0.01 - File Created
16 // additional Comments:
17 //
18 //////////////////////////////////////////////////////////////////////////////////
19
20 module counter #(parameter N = 8) (input logic clk, reset,
21                                                                    output logic [
                                                                        N-1:0] q)
                                                                        ;
22
23         always_ff@(posedge clk, posedge reset)
24                 if (reset) q <= 0;
25                 else q <= q + 1;
26 endmodule
```

A.5.3 comparator

```
1  //////////////////////////////////////////////////////////////////////////////////
2  // Company:          Oregon State University
3  // Engineer:         Brenden Lowe
4  //
5  // Create Date:      06/01/20
6  // Design Name:      RGB Driver
7  // Module Name:      comparator
8  // Project Name:     ECE 271 Design Project
9  // Target Devices:   Max 10 FPGA
10 // Description:      Compares outputs and determines if they are equal.
11 //
12 // Dependencies:
13 //
14 // Revision:
15 // Revision 0.01 - File Created
16 // additional Comments:
17 //
18 //////////////////////////////////////////////////////////////////////////////////
19
20 module counter #(parameter N = 8) (input logic clk, reset,
21                                                                    output logic [
                                                                        N-1:0] q)
                                                                        ;
22
23         always_ff@(posedge clk, posedge reset)
24                 if (reset) q <= 0;
25                 else q <= q + 1;
26 endmodule
```

A.5.4 sync

```
1  //////////////////////////////////////////////////////////////////////////////////
2  // Company:          Oregon State University
3  // Engineer:         Brenden Lowe
4  //
5  // Create Date:      06/01/20
6  // Design Name:      RGB Driver
7  // Module Name:      sync
8  // Project Name:     ECE 271 Design Project
9  // Target Devices:   Max 10 FPGA
10 // Description:      Synchronizes signal with clock cycle.
11 //
12 // Dependencies:
13 //
14 // Revision:
15 // Revision 0.01 - File Created
16 // additional Comments:
17 //
18 //////////////////////////////////////////////////////////////////////////////////
19
20 module sync (input logic clk, d,
21              output logic q);
22
23      logic n1;
24
25      always_ff@(posedge clk)
26              begin
27                      n1 <= d;
28                      q <= n1;
29              end
30 endmodule
```

A.5.5 freq_div

```
1  //////////////////////////////////////////////////////////////////////////////////
2  // Company:          Oregon State University
3  // Engineer:         Brenden Lowe
4  //
5  // Create Date:      06/01/20
6  // Design Name:      RGB Driver
7  // Module Name:      freq_div
8  // Project Name:     ECE 271 Design Project
9  // Target Devices:   Max 10 FPGA
10 // Description:      Divides 50 MHz signal into 1 MHz signal.
11 //
12 // Dependencies:
13 //
14 // Revision:
```

```
// Revision 0.01 - File Created
// additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module freq_div (input logic clk_50Mhz, user_res,
                                        output logic clk_1Mhz);

        logic [5:0] comp_50, count;
        logic reset, full_res;

        // Give comp_50 value
        assign comp_50 = 50;

        // Feed 50Mhz into counter
        counter #(6) to_one(
                .clk (clk_50Mhz),
                .reset (full_res),
                .q (count)
        );

        // Feed count into comparator
        comparator #(6) is_50 (
                .a (count),
                .b (comp_50),
                .eq (clk_1Mhz)
        );

        assign full_res = (clk_1Mhz | user_res);

        // Feed clk_1Mhz into sync to reset counter
        sync send_res (
                .clk (clk_50Mhz),
                .d (full_res),
                .q (reset)
        );
endmodule
```

A.5.6 cont_to_color

```
//////////////////////////////////////////////////////////////////////////////////
// Company:         Oregon State University
// Engineer:        Brenden Lowe
//
// Create Date:     06/01/20
// Design Name:     RGB Driver
// Module Name:     cont_to_color
// Project Name:    ECE 271 Design Project
// Target Devices: Max 10 FPGA
// Description:     Converts controller input into color-code output.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

// module that converts controller input into color output
module cont_to_color (input logic [6:0] cont,
                                                output logic [23:0] color);

        always_comb
                case(cont)
                        7'b100_0000: color = 24'hFF0000; // a maps to red
                        7'b010_0000: color = 24'h00FF00; // b maps to green
                        7'b001_0000: color = 24'h0000FF; // start maps to blue
                        7'b000_1000: color = 24'hFF9900; // up maps to orange
                        7'b000_0100: color = 24'hFF0066; // down maps to pink
                        7'b000_0010: color = 24'h660066; // left maps to purple
                        7'b000_0001: color = 24'h00FFFF; // right maps to cyan
                        default: color = 0;
                endcase
endmodule
```

A.5.7 trans_state

```
//////////////////////////////////////////////////////////////////////////////////
// Company:         Oregon State University
// Engineer:        Brenden Lowe
//
// Create Date:     06/01/20
// Design Name:     RGB Driver
// Module Name:     trans_state
// Project Name:    ECE 271 Design Project
// Target Devices: Max 10 FPGA
// Description:     Keeps track of current transmission state.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module trans_state (input logic clk, user_res,
                                                output logic [3:0] state);

        logic [3:0] comp_12;
        logic reset, temp_eq, full_res;

        // Give comp_12 value
        assign comp_12 = 11; // 11, because we are counting zero as a 1 unit of time

        // Feed clk into counter
        counter #(4) cur_state(
                .clk (clk),
                .reset (reset),
                .q (state)
```

```
34          );
35
36          // Feed count into comparator
37          comparator #(4) is_12 (
38                  .a (state),
39                  .b (comp_12),
40                  .eq (temp_eq)
41          );
42
43          assign full_res = (temp_eq | user_res);
44
45          // Feed into sync to reset counter
46          sync res_send (
47                  .clk (clk),
48                  .d (full_res),
49                  .q (reset)
50          );
51  endmodule
```

A.5.8 bit_state

```
1   //////////////////////////////////////////////////////////////////////////////////
2   // Company:         Oregon State University
3   // Engineer:        Brenden Lowe
4   //
5   // Create Date:     06/01/20
6   // Design Name:     RGB Driver
7   // Module Name:     bit_state
8   // Project Name:    ECE 271 Design Project
9   // Target Devices:  Max 10 FPGA
10  // Description:     Keeps track of which bit to transmit.
11  //
12  // Dependencies:
13  //
14  // Revision:
15  // Revision 0.01 - File Created
16  // additional Comments:
17  //
18  //////////////////////////////////////////////////////////////////////////////////
19
20  module bit_state (input logic [3:0] trans_state,
21                                                  input logic user_res,
22                                                  output logic [4:0] state,
23                                                  output logic res);
24
25          // Define intermediate nodes
26          logic [4:0] comp_24, comp_30;
27          logic [3:0] comp_12;
28          logic inc, reset, temp_res, full_res;
29
30          // Give comp_12 a value
31          assign comp_12 = 11;
32
33          // Give comp_24 a value
34          assign comp_24 = 23;
35
36          // Give comp_30 a value
37          assign comp_30 = 30;
38
39          // Feed trans_state into comparator
40          comparator #(4) is_inc (
41                  .a (trans_state),
42                  .b (comp_12),
43                  .eq (inc)
44          );
45
46          // Feed inc into counter
47          counter #(5) count_bit (
48                  .clk (inc),
49                  .reset (reset),
50                  .q (state)
51          );
52
53          // Feed state into a comparator
54          comparator #(5) is_24 (
55                  .a (state),
56                  .b (comp_30),
57                  .eq (temp_res)
58          );
59
60          assign full_res = (temp_res | user_res);
61
62          // Feed into sync to reset counter
63          sync res_send (
64                  .clk (inc),
65                  .d (full_res),
66                  .q (reset)
67          );
68
69          assign res = (state > comp_24);
70  endmodule
```

A.5.9 choose_bit

```
1   //////////////////////////////////////////////////////////////////////////////////
2   // Company:         Oregon State University
3   // Engineer:        Brenden Lowe
4   //
5   // Create Date:     06/01/20
6   // Design Name:     RGB Driver
7   // Module Name:     choose_bit
8   // Project Name:    ECE 271 Design Project
9   // Target Devices:  Max 10 FPGA
10  // Description:     Outputs the current bit to be transmitted.
11  //
12  // Dependencies:
13  //
14  // Revision:
15  // Revision 0.01 - File Created
16  // additional Comments:
17  //
```

```
////////////////////////////////////////////////////////////////////////////////

module choose_bit (input logic [23:0] color,
                                                input logic [4:0] state,
                                                output logic send);

        always_comb
                case(state)
                        0: send = color[23];
                        1: send = color[22];
                        2: send = color[21];
                        3: send = color[20];
                        4: send = color[19];
                        5: send = color[18];
                        6: send = color[17];
                        7: send = color[16];
                        8: send = color[15];
                        9: send = color[14];
                        10: send = color[13];
                        11: send  = color[12];
                        12: send = color[11];
                        13: send = color[10];
                        14: send = color[9];
                        15: send = color[8];
                        16: send = color[7];
                        17: send = color[6];
                        18: send = color[5];
                        19: send = color[4];
                        20: send = color[3];
                        21: send = color[2];
                        22: send = color[1];
                        23: send = color[0];
                        default: send = 0;
                endcase
endmodule
```

A.5.10 transfer

```
////////////////////////////////////////////////////////////////////////////////
// Company:         Oregon State University
// Engineer:        Brenden Lowe
//
// Create Date:     06/01/20
// Design Name:     RGB Driver
// Module Name:     transfer
// Project Name:    ECE 271 Design Project
// Target Devices: Max 10 FPGA
// Description:     Outputs the correct encoding based on given signal.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module transfer (input logic data, reset,
                                        input logic [3:0] state,
                                        output logic dout);

        always_comb
                if (reset)
                begin
                        dout = 0;
                end
                else
                begin
                        if (data)
                        begin
                                case(state)
                                        0: dout = 1;
                                        1: dout = 1;
                                        2: dout = 1;
                                        3: dout = 1;
                                        4: dout = 1;
                                        5: dout = 1;
                                        6: dout = 0;
                                        7: dout = 0;
                                        8: dout = 0;
                                        9: dout = 0;
                                        10: dout = 0;
                                        11: dout = 0;
                                        default: dout = 0;
                                endcase
                        end
                        else
                        begin
                                case(state)
                                        0: dout = 1;
                                        1: dout = 1;
                                        2: dout = 1;
                                        3: dout = 0;
                                        4: dout = 0;
                                        5: dout = 0;
                                        6: dout = 0;
                                        7: dout = 0;
                                        8: dout = 0;
                                        9: dout = 0;
                                        10: dout = 0;
                                        11: dout = 0;
                                        default: dout = 0;
                                endcase
                        end
                end
endmodule
```

B Simulation Files (Do scripts)

B.1 NES Decoder

```
vsim work.NES_Decoder

add wave *
force -freeze sim:/NES_Decoder/data 1'h0 0
force -freeze sim:/NES_Decoder/clk 1'h0 0
force -freeze sim:/NES_Decoder/reset 1'h1 0
force -freeze sim:/NES_Decoder/count 4'b0000 0
run 100 ns
force -freeze sim:/NES_Decoder/reset 1'h0 0
force -freeze sim:/NES_Decoder/count 4'b0001 0
force -freeze sim:/NES_Decoder/reset 1'h0 0
run 100 ns
force -freeze sim:/NES_Decoder/count 4'b0010 0
force -freeze sim:/NES_Decoder/data 1'h1 0
run 100 ns
force -freeze sim:/NES_Decoder/count 4'h3 0
run 100 ns
```

B.2 PS/2 Keyboard

```
vsim ps2keyboard
add wave *
force -freeze clk_10 1 0, 0 {50 ps} -r 100
force reset_b 1
force enable_switch 1
force A 0
force B 0
force C 0
force D 0
force E 0
force F 0
force zero 0
force one 1
force two 0
force three 0
force four 0
force five 0
force six 0
force seven 0
force eight 0
force nine 0
run 200
```

B.3 Seven Segment Driver

```
vsim work.SevenSeg

add wave data
add wave segments


force data 16#0 0
force data 16#1 10
force data 16#2 20
force data 16#3 30
force data 16#4 40
force data 16#5 50
force data 16#6 60
force data 16#7 70
force data 16#8 80
force data 16#9 90
force data 16#a 100
force data 16#b 110
force data 16#c 120
force data 16#d 130
force data 16#e 140
force data 16#f 150

run 160 ps
```

B.4 RGB Driver

```
vsim work.RGB_Decoder

add wave *
force -freeze sim:/RGB_Driver/clk 1'h0 0
force -freeze sim:/RGB_Driver/user_res 1'h1 0
force -freeze sim:/RGB_Driver/controller 7'b100_0000 0
force -freeze sim:/RGB_Driver/clk_1 1'h0 0
force -freeze sim:/RGB_Driver/bit_reset 1'h0 0
run 50 ns
force -freeze sim:/RGB_Driver/user_res 1'h0 0
force -freeze sim:/RGB_Driver/transfer_state 4'h0 0
force -freeze sim:/RGB_Driver/which_bit 5'd24 0
run 50 ns
force -freeze sim:/RGB_Driver/transfer_state 4'h1 0
run 50 ns
force -freeze sim:/RGB_Driver/transfer_state 4'h2 0
run 50 ns
force -freeze sim:/RGB_Driver/transfer_state 4'h3 0
run 50 ns
force -freeze sim:/RGB_Driver/transfer_state 4'h4 0
run 50 ns
force -freeze sim:/RGB_Driver/transfer_state 4'h5 0
run 50 ns
force -freeze sim:/RGB_Driver/transfer_state 4'h6 0
run 50 ns
force -freeze sim:/RGB_Driver/transfer_state 4'h7 0
run 50 ns
force -freeze sim:/RGB_Driver/transfer_state 4'h8 0
run 50 ns
force -freeze sim:/RGB_Driver/transfer_state 4'h9 0
run 50 ns
force -freeze sim:/RGB_Driver/transfer_state 4'ha 0
run 50 ns
force -freeze sim:/RGB_Driver/transfer_state 4'hb 0
run 50 ns
force -freeze sim:/RGB_Driver/transfer_state 4'h0 0
force -freeze sim:/RGB_Driver/which_bit 5'h00 0
```

```
38  run 50 ns
39  force -freeze sim:/RGB_Driver/transfer_state 4'h1 0
40  run 50 ns
41  force -freeze sim:/RGB_Driver/transfer_state 4'h2 0
42  run 50 ns
43  force -freeze sim:/RGB_Driver/transfer_state 4'h3 0
44  run 50 ns
45  force -freeze sim:/RGB_Driver/transfer_state 4'h5 0
46  run 50 ns
47  force -freeze sim:/RGB_Driver/transfer_state 4'h6 0
48  run 50 ns
49  force -freeze sim:/RGB_Driver/transfer_state 4'h7 0
50  run 50 ns
51  force -freeze sim:/RGB_Driver/transfer_state 4'h8 0
52  run 50 ns
53  force -freeze sim:/RGB_Driver/transfer_state 4'h9 0
54  run 50 ns
55  force -freeze sim:/RGB_Driver/transfer_state 4'ha 0
56  run 50 ns
57  force -freeze sim:/RGB_Driver/transfer_state 4'hb 0
58  run 50 ns
```