ECE 271

Digital Logic Design
Final Project

Nick Olson
Michael Hathaway
Sienna Han

December 6, 2019
Instructor Shuman
Oregon State University

Contents

List of Figures

1 Project Description

The purpose of this project was to design and implement a digital logic design that incorporated the NES controller, the PS/2 Keyboard, and the VCR remote as inputs to a system that produced outputs using a some of the following devices: a seven segment display, VGA output, LEDs, audio waves, or DC motors. Each input option has its own communication protocol that is processed by a driver to produce an output that can be used in one or more of the output devices. An overall description diagram is then shown in **Figure 1** and a hardware diagram is shown in **Figure 2**.

- **Inputs:** NES Controller, PS/2 Keyboard, and VCR remote.
- **Outputs:** Seven Segment Display, VGA output, Square wave audio output, and a L293D DC Motor.

1.1 Project Deliverables

- Supports communication protocols for the NES Controller, PS/2 Keyboard, and VCR Remote.
- Supports output devices: Seven Segment Display, VGA, LM386N-4 Audio IC, and L293D DC Motor.
- Allows the user to control input flow to certain outputs. For example, the seven segment display can use input from both the NES Controller and the VCR Remote.

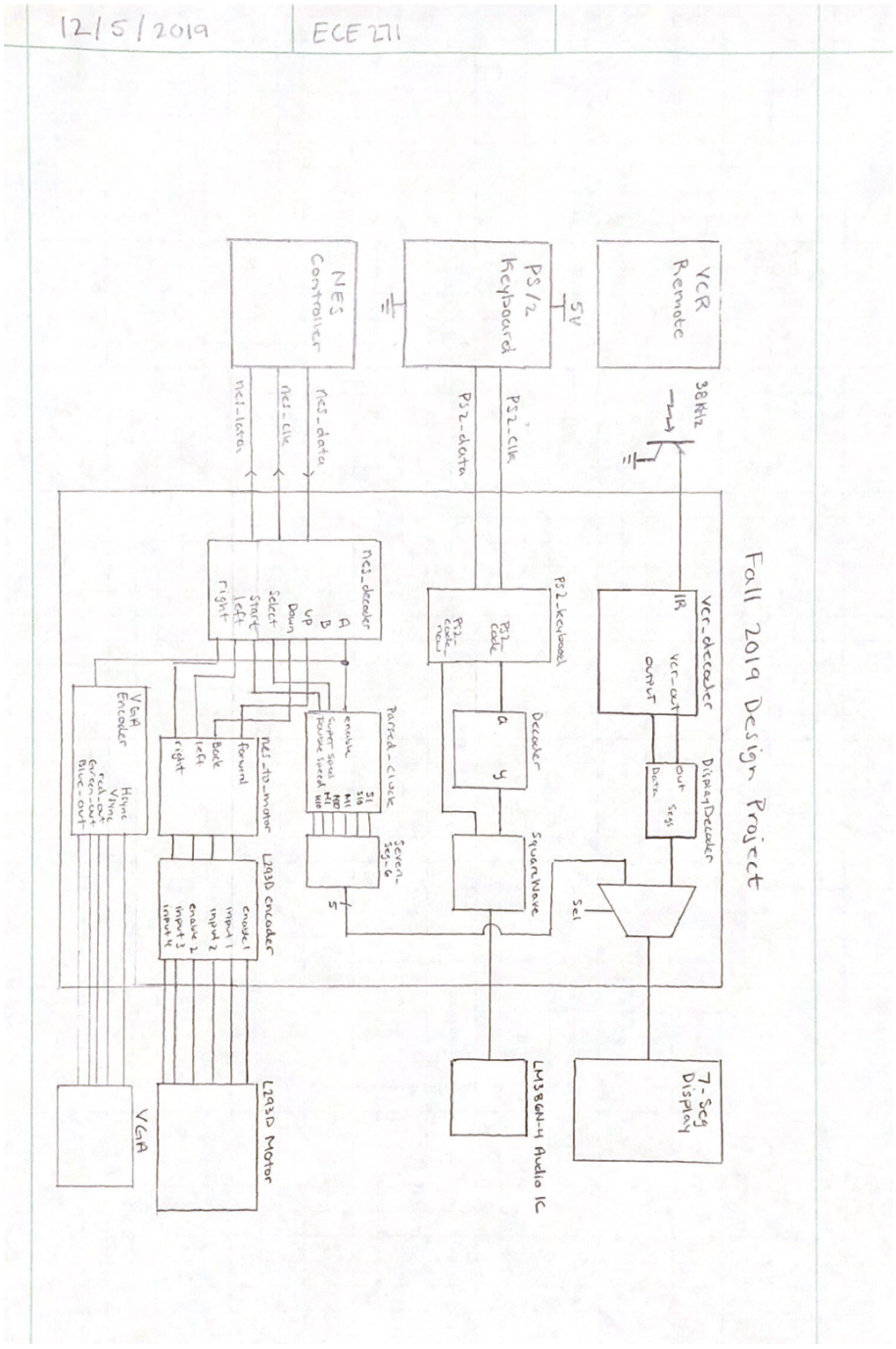


Figure 1: Top Level Diagram for the project. The Diagram shows all the Functional Units used in the project and how they are connected from input to output.

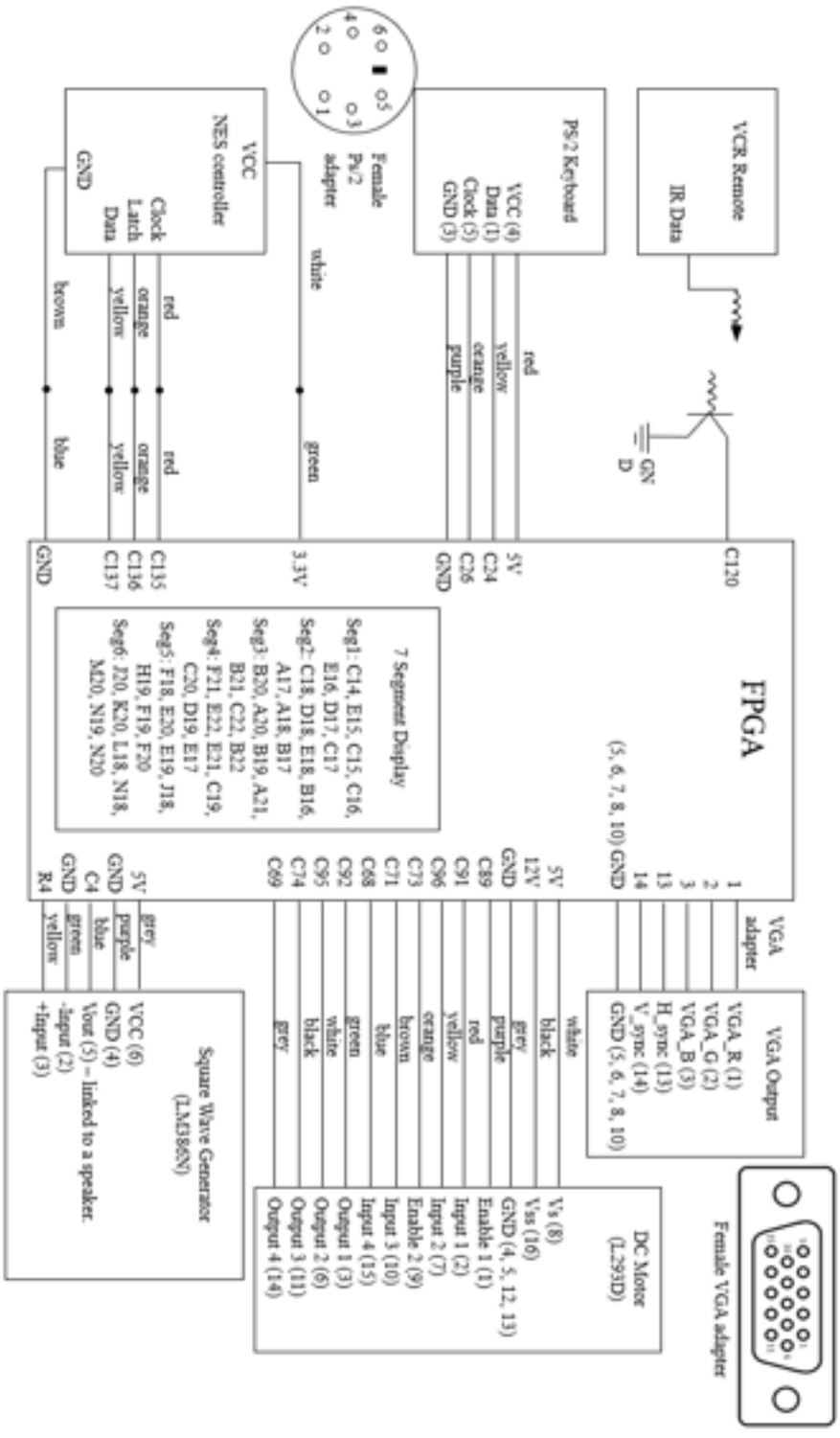


Figure 2: The hardware diagram shows which pins are used on the FPGA, module boards, and relevant supply voltages for the different pieces of hardware used in the system.

2 High Level Description

This section provides a Top level introduction to the project. The input and output specifications are provided below, as well as a toplevel diagram in **Figure 3**, and the simulation results in **Figure 4**.

- **Inputs:** The Top Level of the project has 7 inputs: switch1, switch2, system_reset_button, nes_data, clock_50MHz, ps2_clk, and ps2_data.
  - **switch1:** switch1 is used to enable the square wave generator unit connected to the PS2 Keyboard.
  - **switch2:**
  - **system_reset_button:** system_reset_button is an active high signal used to reset the nes_decoder, vga_encoder, and parsed_clock Functional Units.
  - **nes_data:** the input signal from the NES Controller into the nes_decoder Functional Unit.
  - **clock_50MHz:** the internal clock on the FPGA that is used generate all the other clock signals used in the top level.
  - **ps2_clk:** the clock signal driving the ps_2_keyboard Functional Unit.
  - **ps2_data:** the input signal from the PS2 Keyboard into the ps2_keyboard functional unit.
- **Outputs:** The Top level has outputs going to the NES Controller, the seven segment pins in the FPGA, the L293D DC Controller, the square wave generator, and the VGA pins.

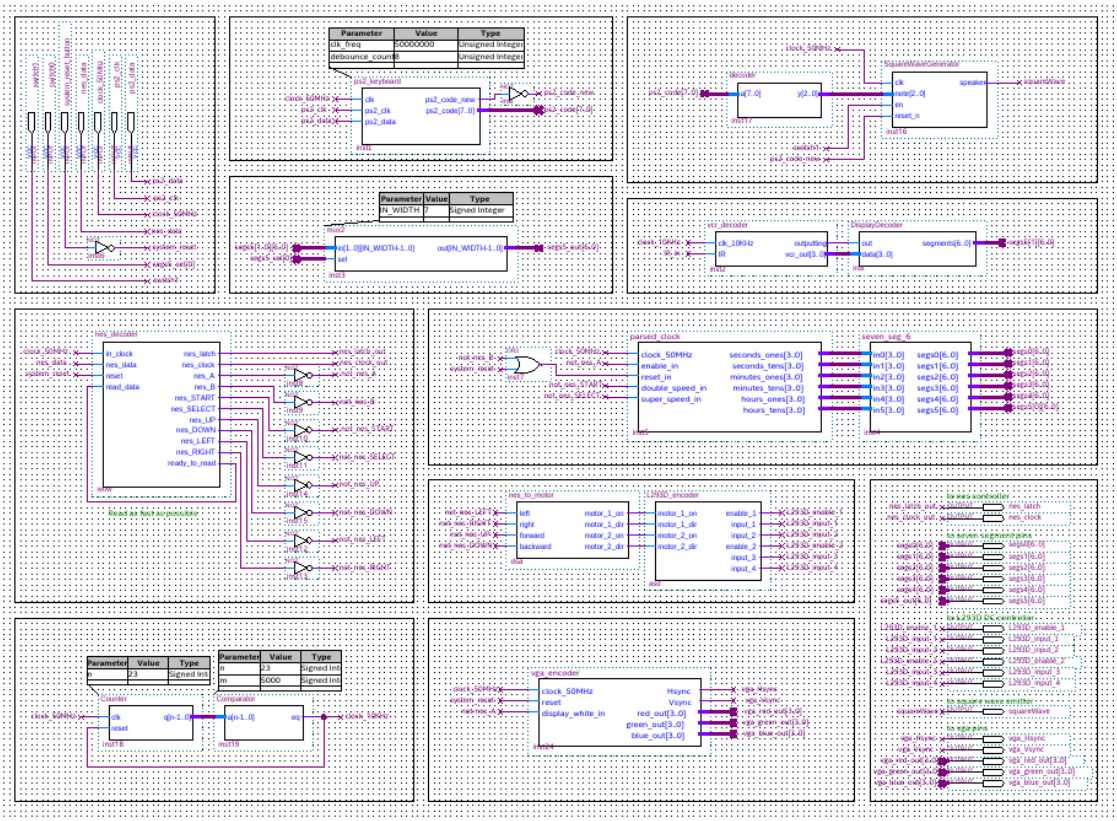
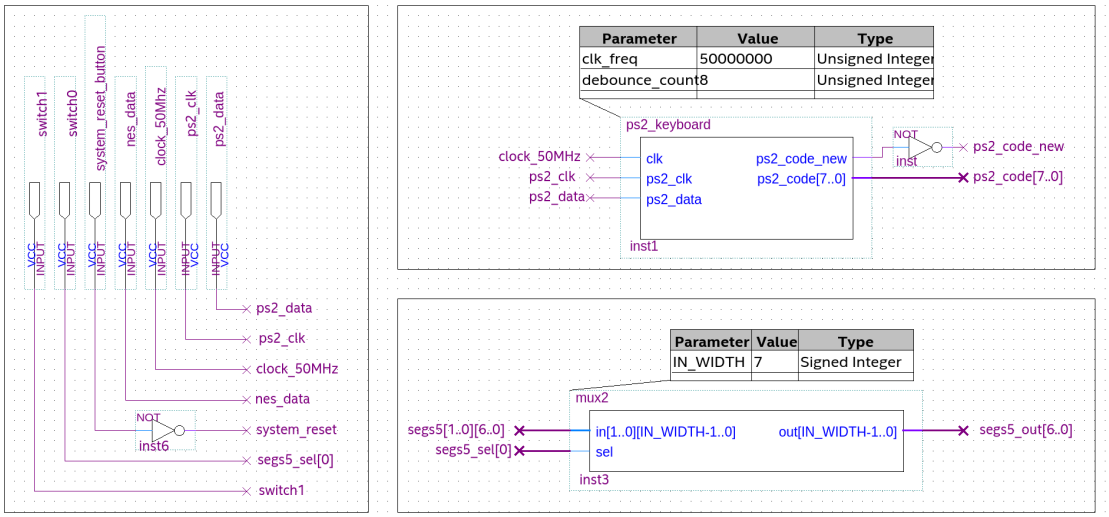
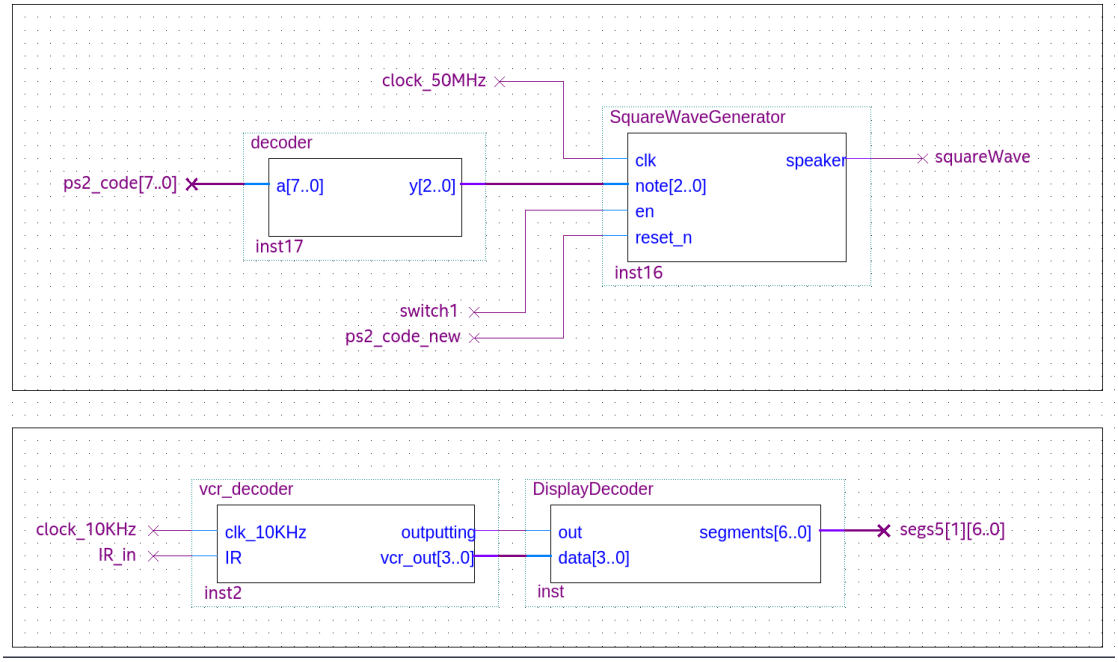


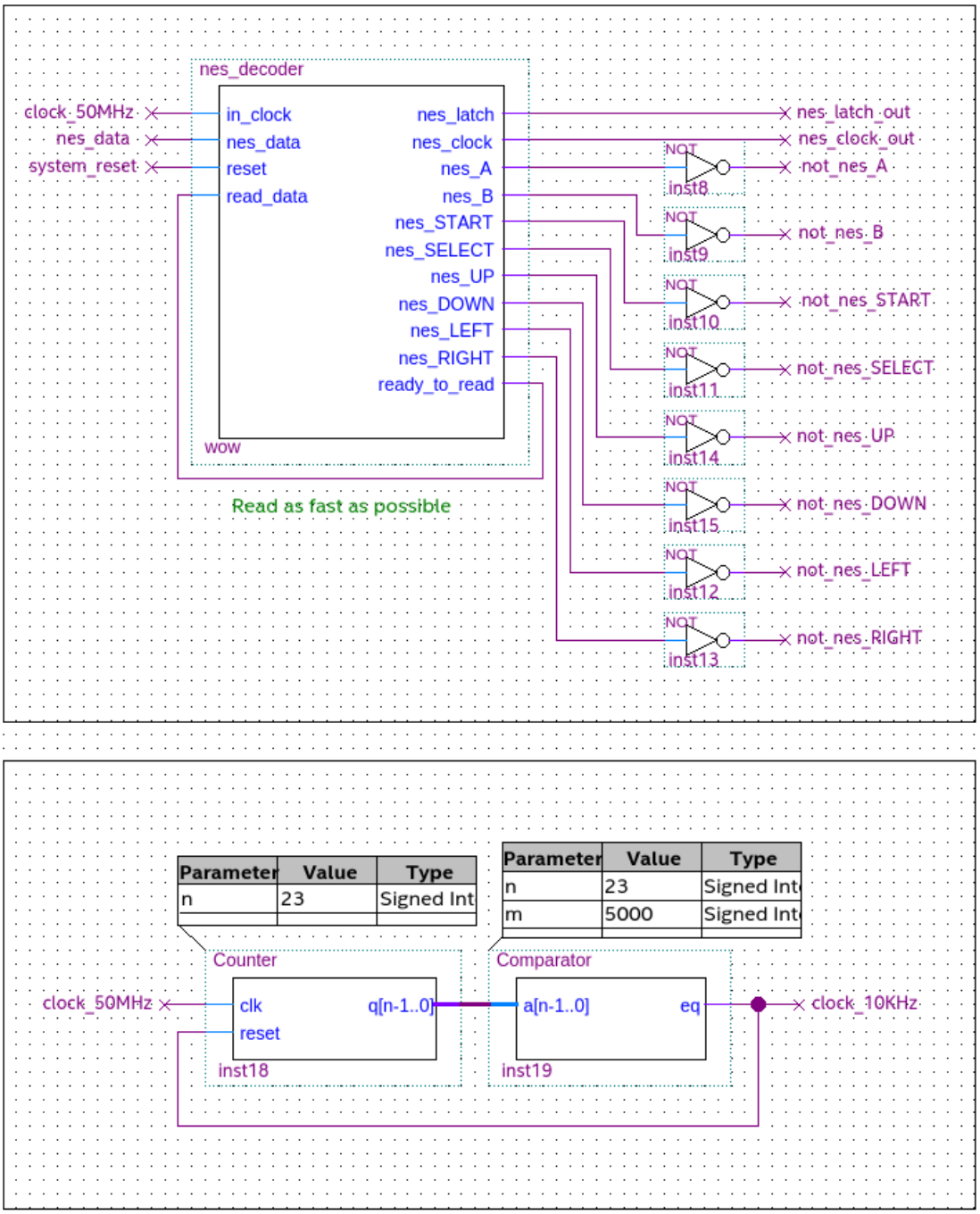
Figure 3: The top level design for the project.



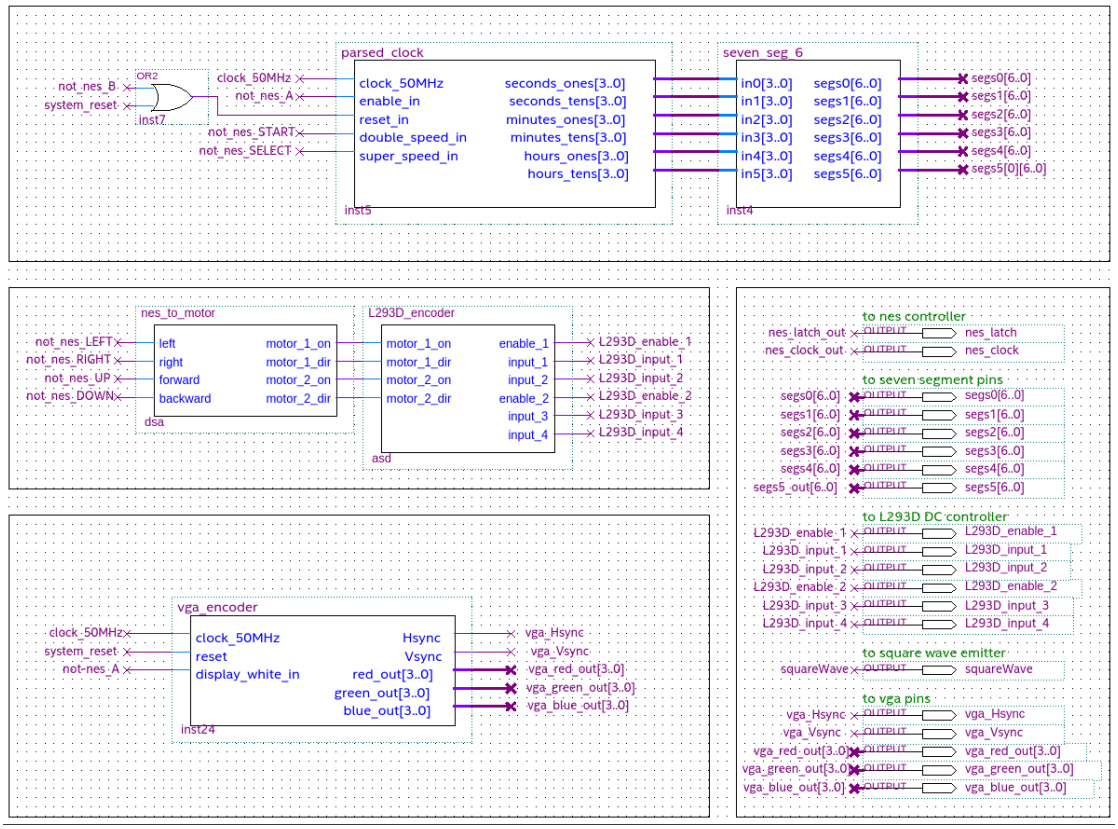
(a) Block 1 of top level. Includes the inputs to the Top Level of the design, the ps2_keyboard Functional unit, and the Multiplexer Functional Unit that controls input to the seven segment display.



(a) Block 2 of top level. Includes the decoder for the ps2_keyboard functional unit, the square wave generator functional unit, the vcr_decoder functional unit, and the DislayDecoder functional unit.



(a) Block 3 of top level. Includes the nes_decoder functional unit and the Counter and Comparator functional units used to generate the clock signal for the vcr_decoder



(a) Block 4 of top level. Includes the parsed_clock, seven_seg_6, nes_to_motor, L293D_Encoder, and vga_endocder functional units. In addition, it includes the outputs for the Top Level of the project.

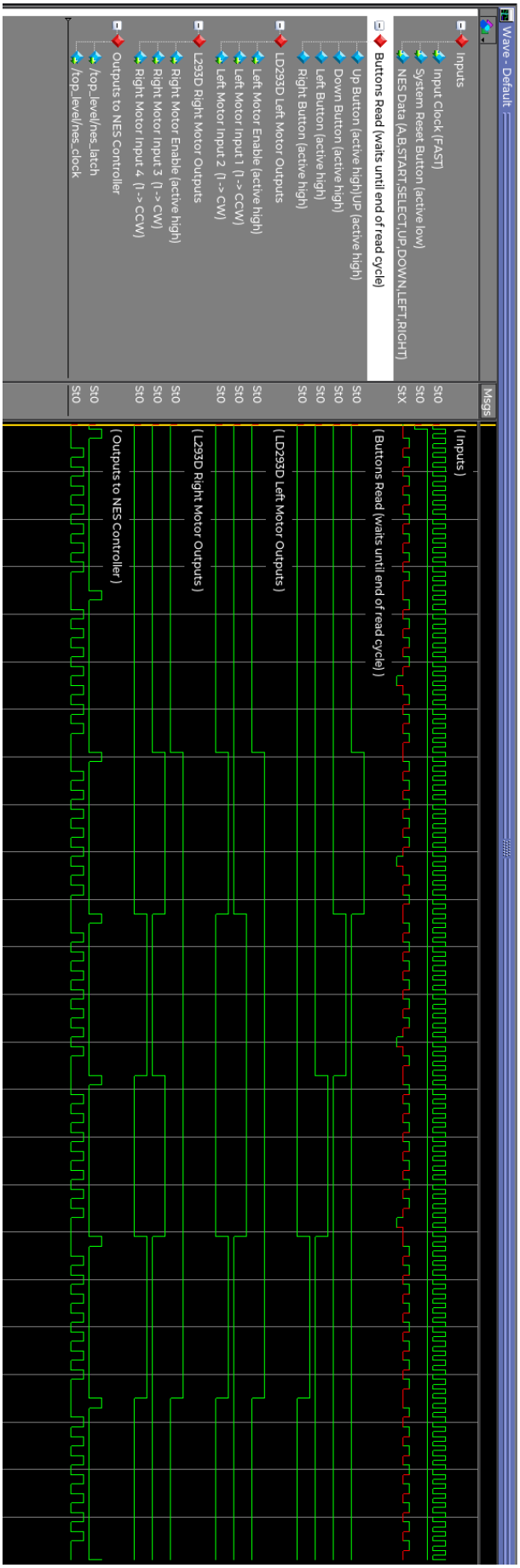


Figure 8: The simulation results for the L293D DC Motor in the Top Level.

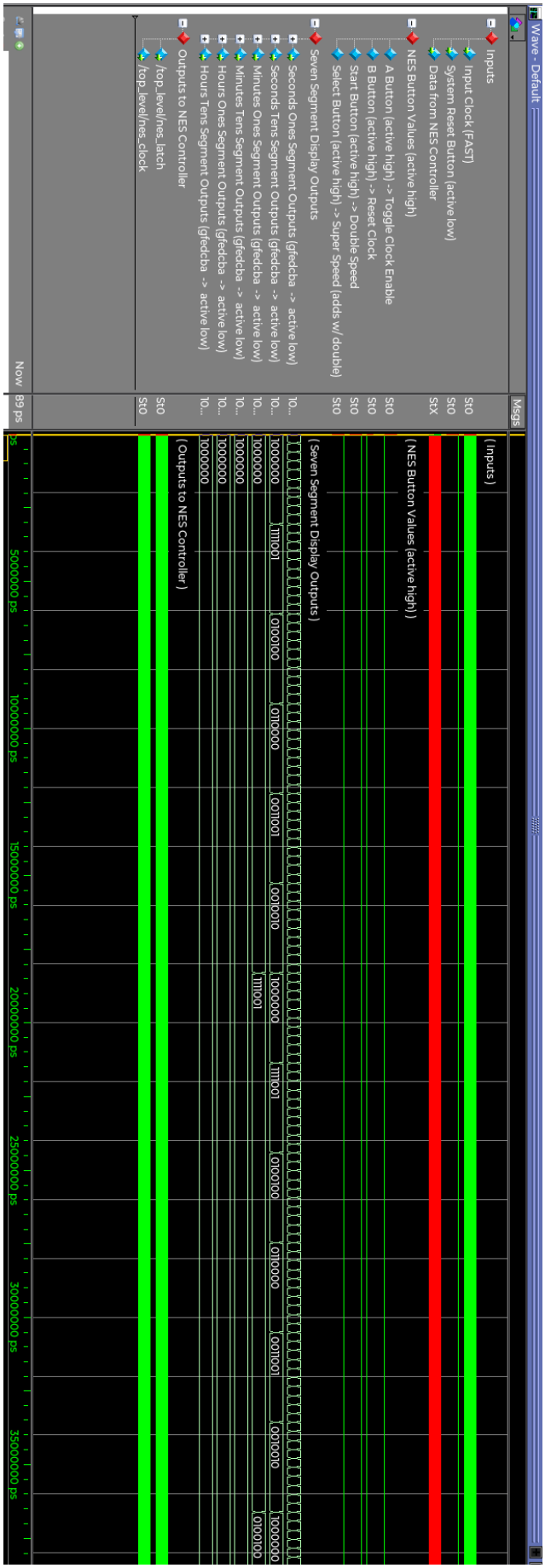


Figure 9: The simulation results for the NES Controller to seven segment display in Top Level.

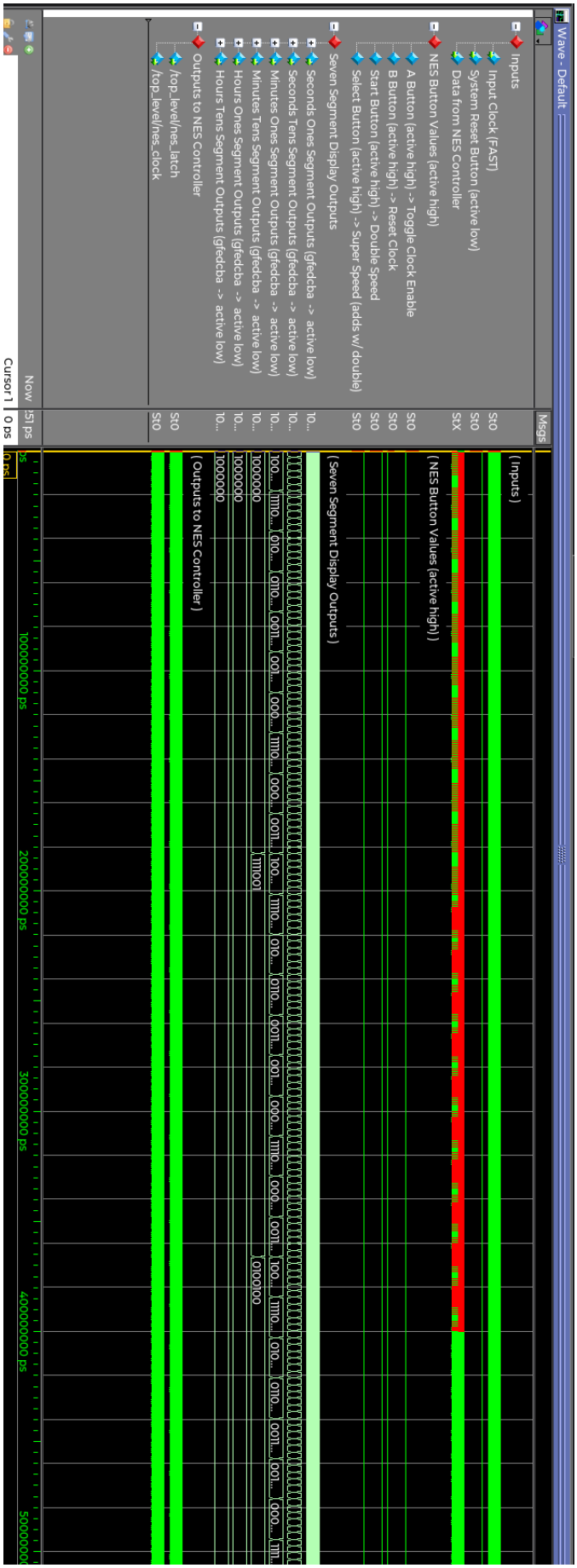


Figure 10: The simulation results for the NES Controller to seven segment display in Top Level.

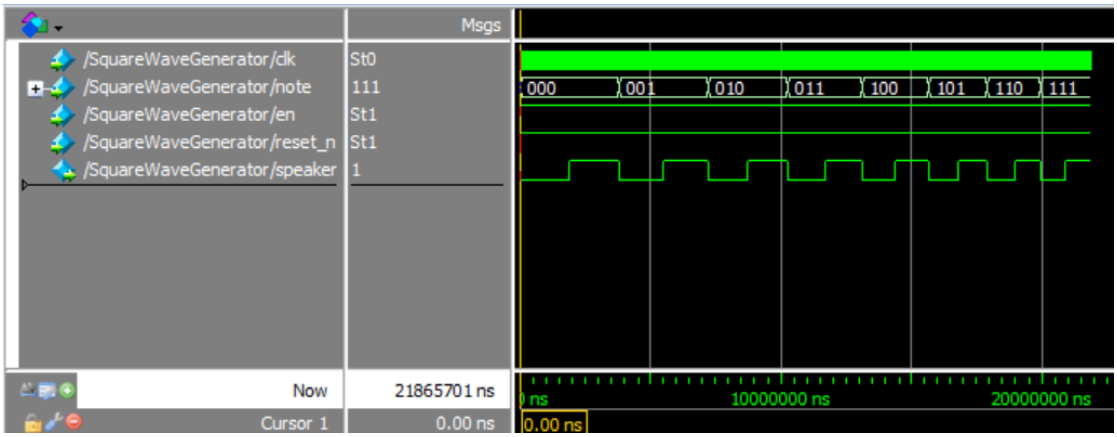


Figure 11: The simulation results for the PS2 Keyboard to the LM386N-4 Audio IC in Top Level.

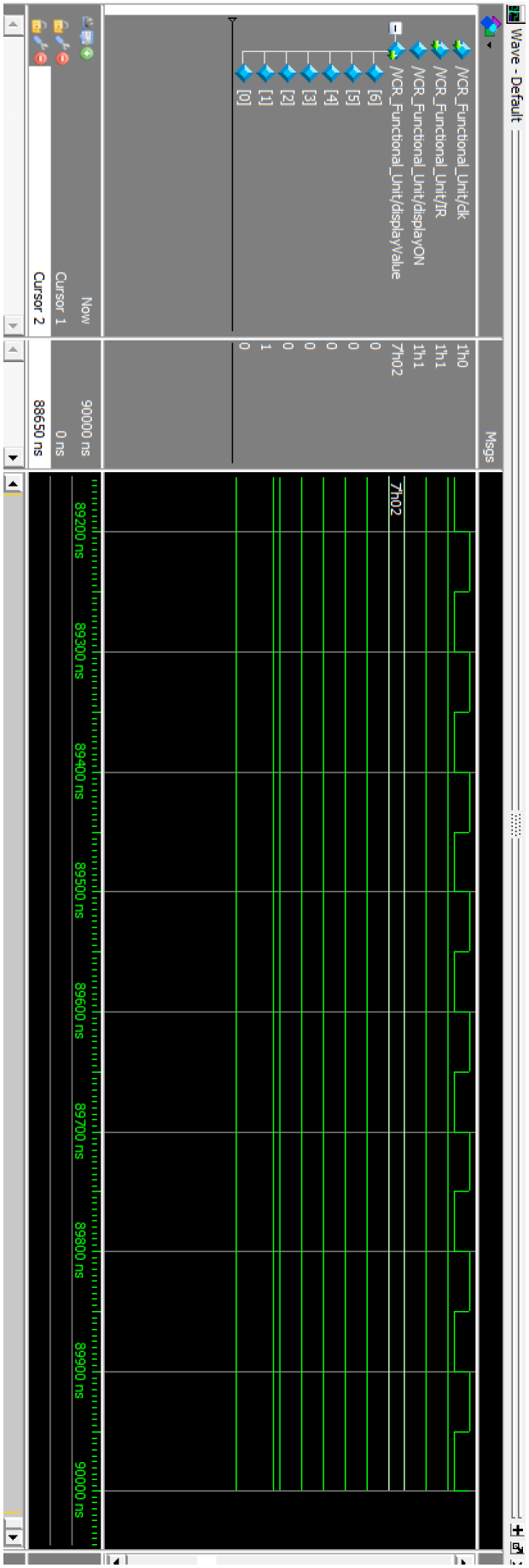


Figure 12: The Simulation results for Top Level VCR output with expected value of 6.

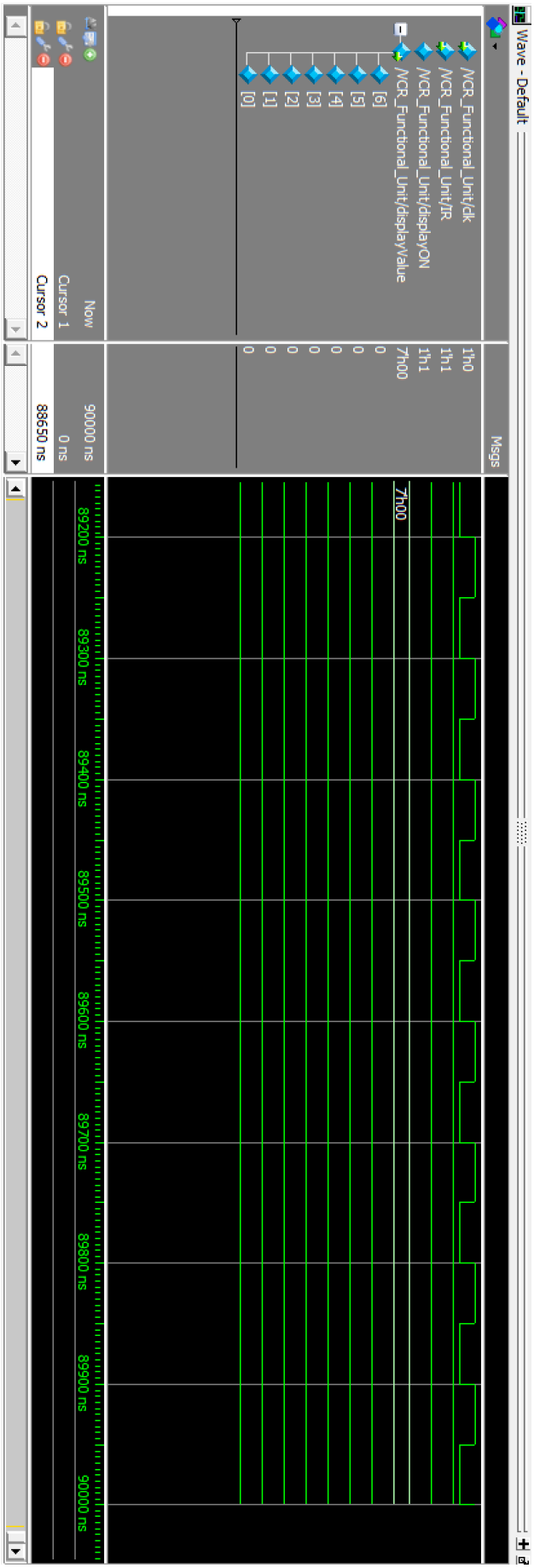


Figure 13: The Simulation results for Top Level VCR output with expected value of 8.

The following subsections will discuss the inputs, outputs, designs, and simulation results of all elements of the design at two levels of scrutiny: functional units and individual blocks of digital logic.

2.1 decoder Functional Unit

The decoder block decodes the data from the PS/2 keyboard to the three-bit data. To match the keys in keyboard from a to t with the piano notes from middle C to the upper octave C, the hexadecimal code of the keyboard is converted to 3-bit data as shown in the simulation. The logic symbol follows in **Figure 15**. The simulation results follow in **Figure 16**.

- **Inputs:** The decoder block has one input, a[7..0]. It indicates the input 8-bit data, which is the make code for keys on the keyboard.
- **Outputs:** The decoder block has one input, y[2..0]. It is converted from 8-bit to 3-bit data and is used to determine the note to generate.

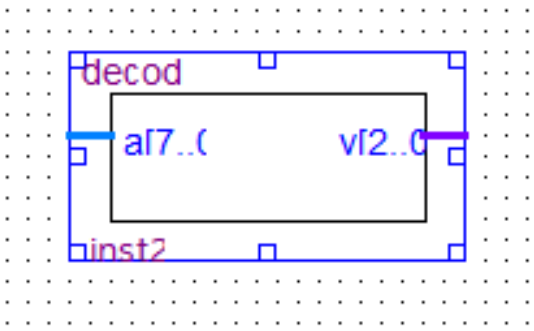


Figure 14: The block symbol of the decoder functional unit used in the final design.

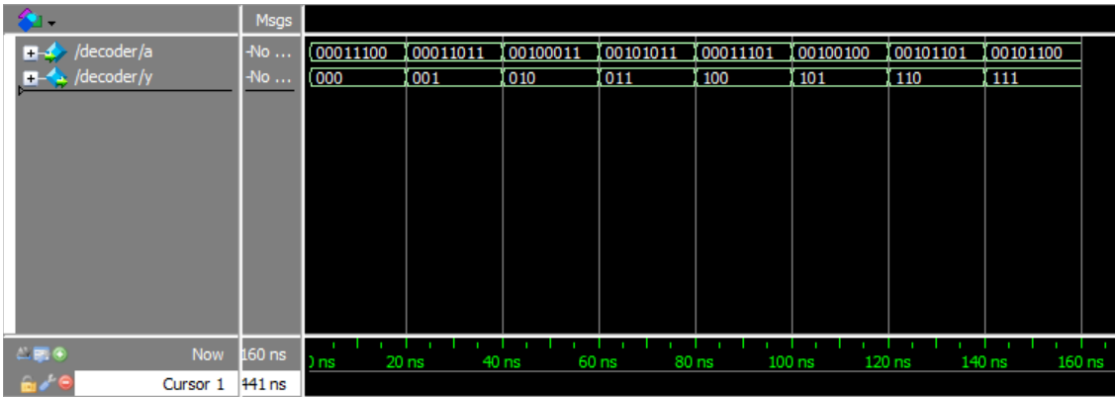


Figure 15: The simulation results for the decoder module.

2.2 seven_seg_6 Functional Unit

The seven_seg_6 module takes in six 4-bit binary values and converts them into the signals required to display each number (which will never exceed 9 in this design) on the FPGA's seven segment display. In this design, these inputs come directly from the parsed_clock module that is controlled by the nes controller, and all but the most significant bit of these seven segment data lines go directly to the FPGA's seven segment display. In this design, either the most significant bit of the parsed_clock OR the output of the vcr_decoder are displayed on the seven segment display's most leftmost digit, depending on the state of the switch0 design input (determined by the mux2 submoduele of the vcr_decoder module). A block diagram follows in **Figure 17**. The logic symbol follows in **Figure 18**. The simulation results follow in **Figure 19**.

- **Inputs:** Six 4-bit binary value inputs from the parsed_clock module that describe the one's and ten's places of the seconds, minutes, and hours of the clock. These values never exceed 59 seconds, 59 minutes, and 23 hours, so each place can only be 0-9 and failure of the sevenseg submodules due to overflow is impossible.
- **Outputs:** Six 7-bit outputs describing how to display each input on a seven segment display. All but the most significant bit of these seven segment data lines go directly to the FPGA's seven segment display. In this design, either the most significant bit of the parsed_clock OR the output of the vcr_decoder are displayed on the seven segment display's most leftmost digit, depending on the state of the switch0 design input (based by the mux2 submoduele of the vcr_decoder module).

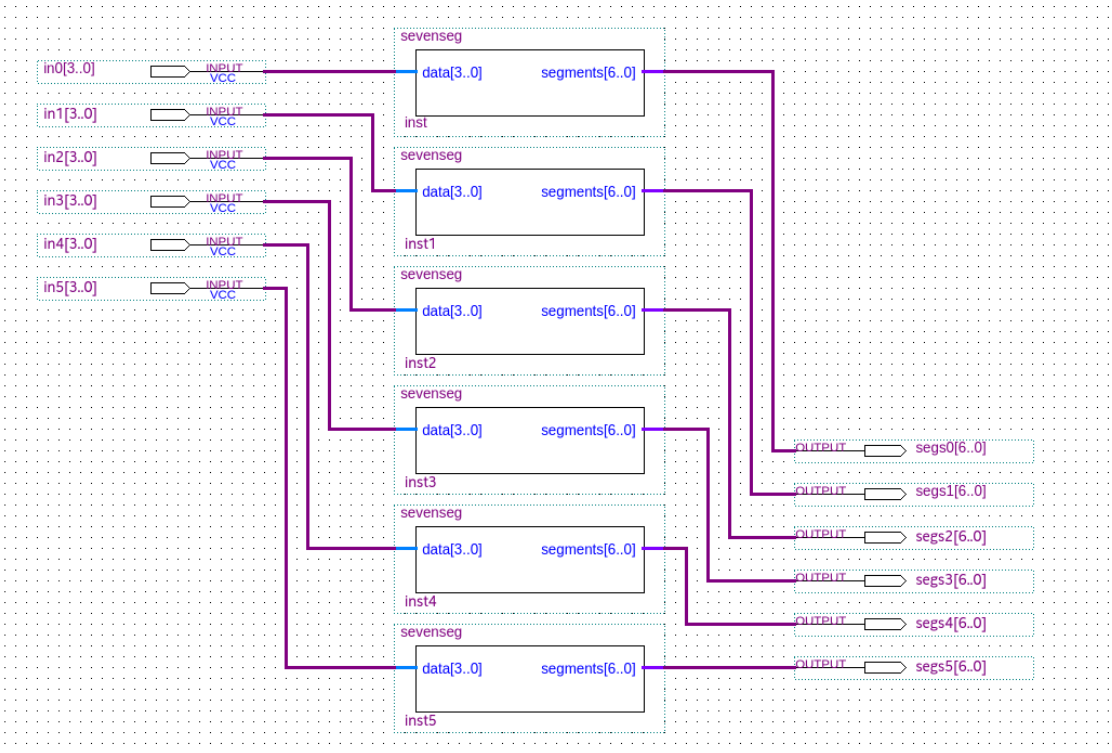


Figure 16: The logic design of the seven_seg_6 functional unit used in the final design.

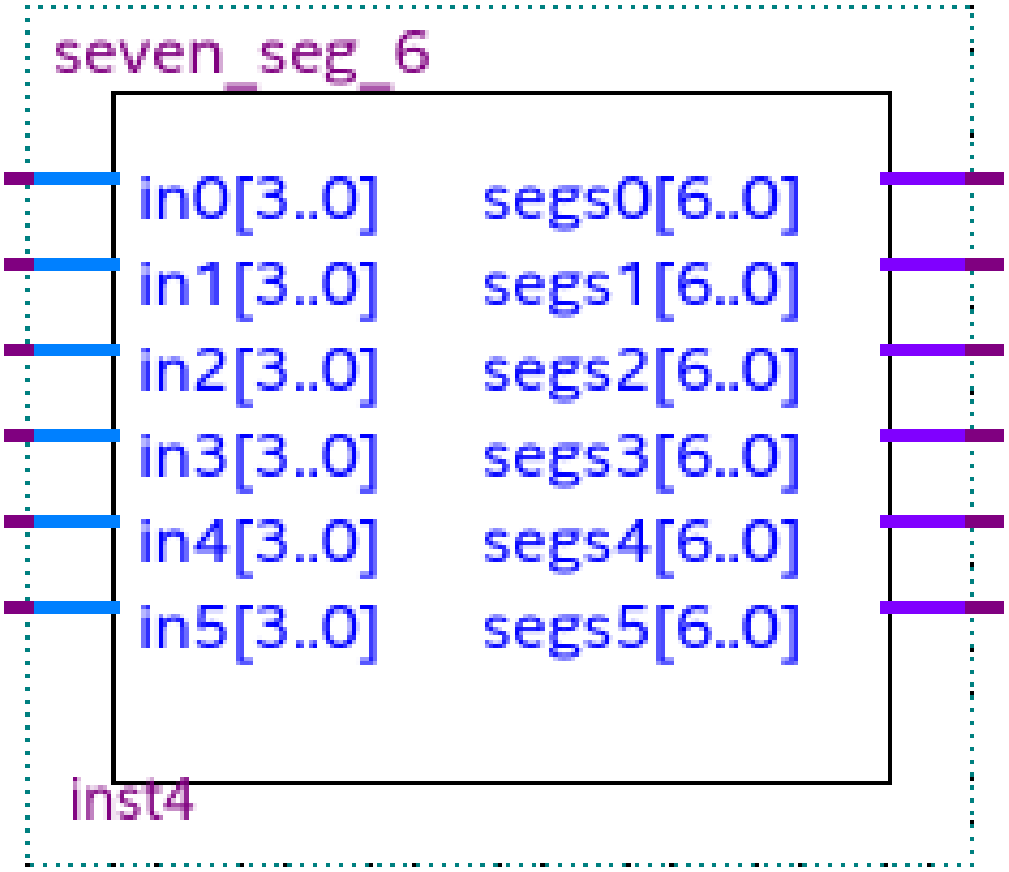


Figure 17: The block symbol of the seven_seg_6 functional unit used in the final design.

Figure 18: The simulation results for the seven_seg_6 module.

2.2.1 sevenseg Component Block

The sevenseg submodule converts a single 4-bit value between 0 and 9 into a 7-bit output describing how to display the input on a seven segment display. In this design, six of these sevenseg blocks are used in the seven_seg_6 module. The logic symbol follows in **Figure 20**. The simulation results follow in **Figure 21**.

- **Inputs:** A single 4-bit value between 0 and 9 to be converted in to a seven segment representation (data[3:0]).
- **Outputs:** A 7-bit output describing how to display the input value on a seven segment display (segments[6:0]).

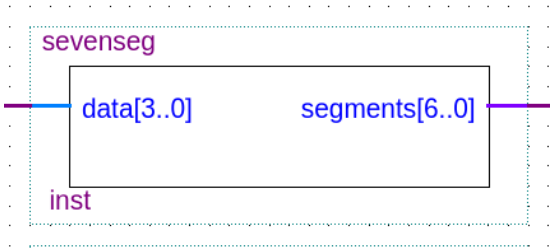


Figure 19: The block symbol of the sevenseg module used in the seven_seg_6 functional unit.

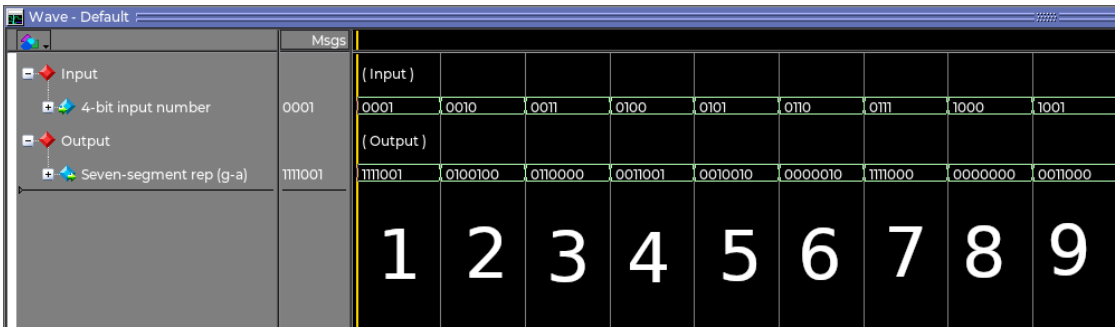


Figure 20: The simulation results for the sevenseg module used in the seven_seg_6 functional unit.

2.3 L293D_encoder Functional Unit

The L293D_encoder functional unit converts internal system values describing how two motors should be driven to data in the format of the L293D DC motor controller. It is connected to these outputs, and takes its inputs from the nes_to_motor module. The logic symbol follows in **Figure 22**. The simulation results follow in **Figure 23**.

- **Inputs:** Whether motors 1 (left) and 2 (right) should be spinning (motor_1_on and motor_2_on); and which direction each motor should spin if also enabled, where logic high correlates to clockwise motion (motor_1_dir and motor_2_dir for each motor, respectively).
- **Outputs:** The signals that the L293D DC motor controller takes in as inputs: enable_1 and enable_2 that describe whether motor 1 (left) and motor 2 (right) should be run; input_1 and input_2 which are always opposite signals describing the direction of rotation of the first (left) motor (where input_1 high and input_2 low correlates with clockwise rotation, for example); and input_3 and input_4 which are always opposite signals describing the direction of rotation of the second (right) motor (where input_1 high and input_2 low correlates with clockwise rotation, for example).

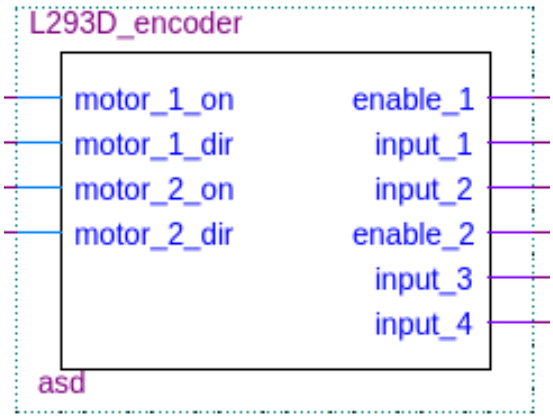


Figure 21: The block symbol of the L293D_encoder functional unit used in the final design.

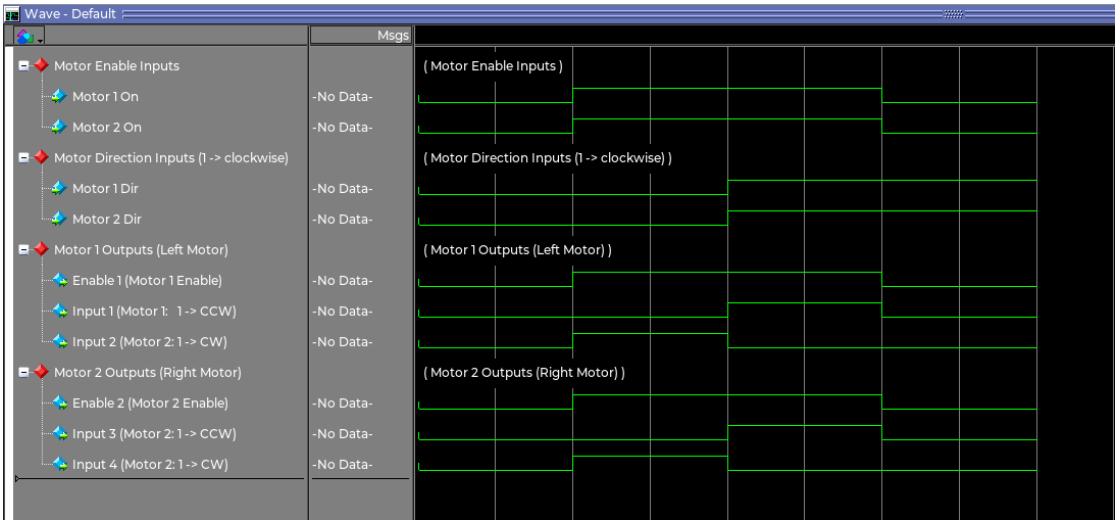


Figure 22: The simulation results for the L293D_encoder module.

2.4 Counter Functional Unit

The Counter Functional Unit increments a counter on every rising edge of the input clock signal. This is used to generate a 10 KHz clock signal to drive the vcr_decoder Functional Unit. The logic symbol follows in **Figure 24**. The simulation results follow in **Figure 25**.

- **Inputs:** The Counter Functional Unit has 2 inputs: clk and reset. clk is the 50 MHz FPGA clock that is used to drive the counter. reset is an input signal that resets the counter to 0.
- **Outputs:** The Counter Functional Unit has 1 output: q. q is a 23-bit value with the current value of the counter.

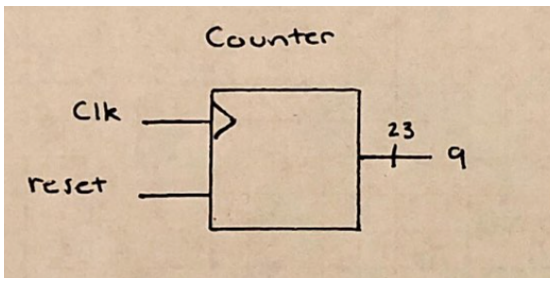


Figure 23: The block symbol of the Counter functional unit used in the final design.

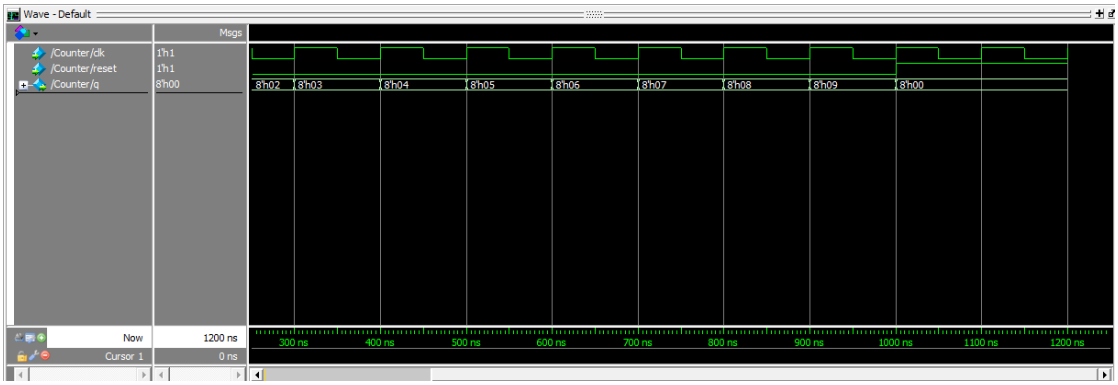


Figure 24: The simulation results for the Counter module.

2.5 vga_encoder Functional Unit

VGA is a standard interface for controlling analog monitors. A vga_encoder is a display shows a white screen when a nes_A button is pushed. It generally has hsync and vsync values and their timing specification is for determining whether it is okay to draw on the screen. The screen in this project are designed to generate only a white screen, values of RGB are set to 1. When nes_A has a value of logic 1, the screen shows a white screen. Otherwise, RGB values are 0 and it shows a black screen. A block diagram follows in **Figure 26**. The logic symbol follows in **Figure 27**.

- **Inputs:** A vga_encoder has three inputs, clock_50MHz, reset, and display_white_in. A clock_50MHz is the clock with a frequency of 50MHz, a reset is a switch to reset the h_sync and v_sync values, and display_white_in indicates a nes_A button to change the screen into a white or black screen.
- **Outputs:** A vga_encoder has four outputs, hsync, vsync, red_out[3..0], green[3..0], and blue[3..0]. A value of hsync and vsync is compared to the time specification and indicates whether the user is able to draw on the screen. Other output signals become a 4-bit data for the colors.

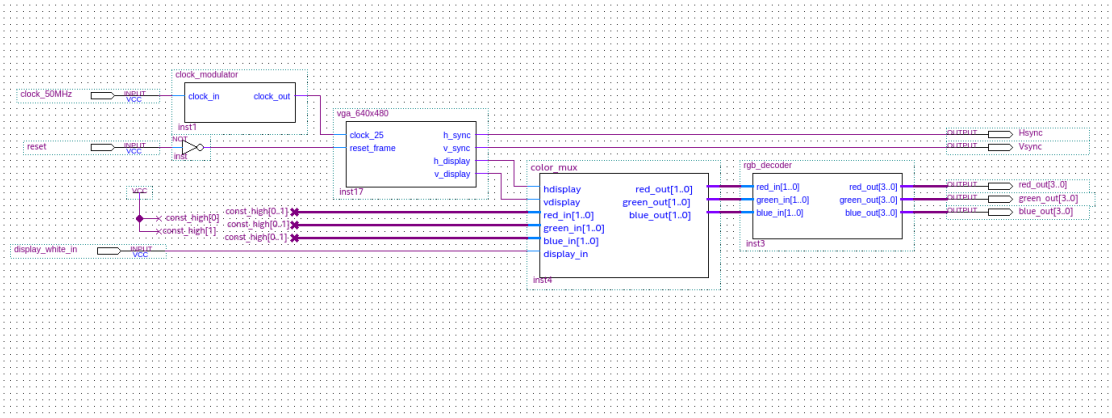


Figure 25: The logic design of the vga_encoder functional unit used in the final design.

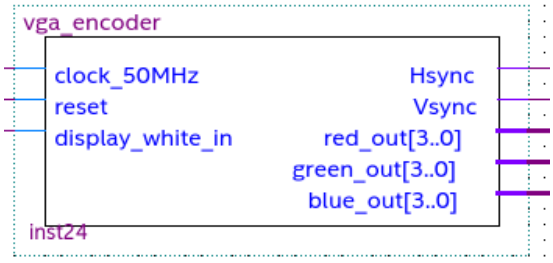


Figure 26: The block symbol of the vga_encoder functional unit used in the final design.

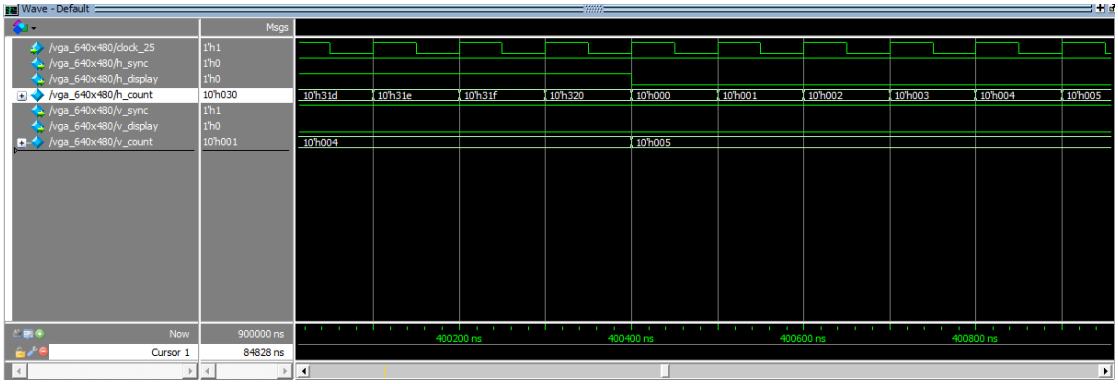


Figure 27: The Simulation results of the vga_encoder functional unit used in the final design.

2.5.1 color_mux Component Block

A color_mux block generates 2-bit data, 00 or 11, depending on the nes_A button. If nes_A button is on, all the data for RGB becomes 11. Otherwise, they become 00.

- **Inputs:** Input: A color_mux block has 6 inputs, hdisplay, vdisplay, red_in[1..0], green_in[1..0], blue_in[1..0], and display_white_in. A value of hdisplay and vdisplay indicates whether it is on the screen or not, and display_white_in indicates the nes_A button. Other data is for RGB.
- **Outputs:** A color_mux block has 3 output, red_out[1..0], green_out[1..0], and blue_out[1..0]. As shown in the simulation below, they become 00 when the display_white_in is off even though h_display and v_display are 1. If display_white_in, h_display, and v_display are all 1, the data for RGB becomes 11.

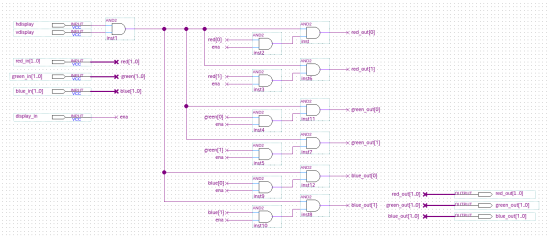


Figure 28: The block symbol of the color_mux module used in the vga_encoder functional unit.

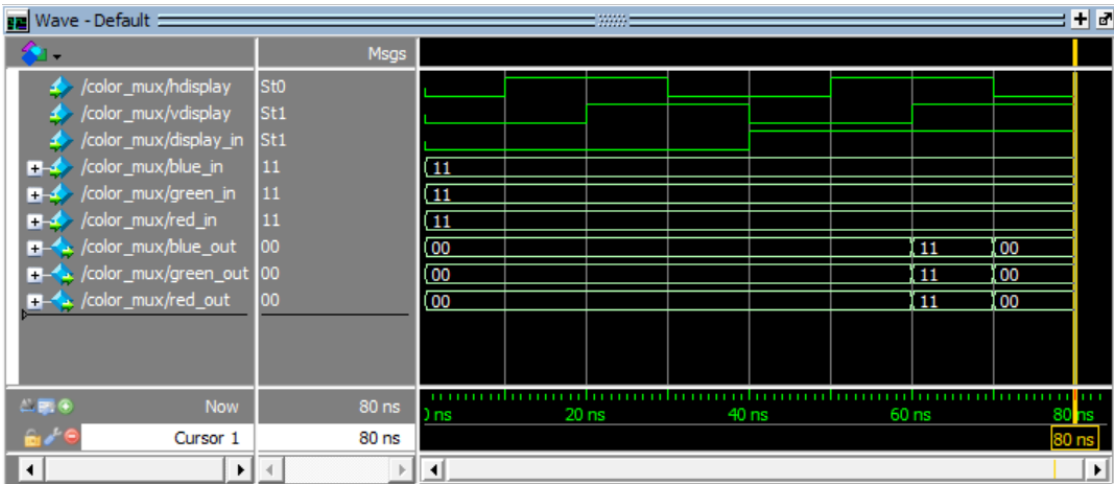


Figure 29: The simulation results of the color_mux individual block used in the vga_encoder functional unit.

2.5.2 clock_modulator Component Block

Because a VGA operates with a clock with a frequency of 25MHz, a clock_modulator slows down the 50MHz-clock to have a frequency of 25MHz. As shown in the simulation, the clock frequency has decreased by half.

- **Inputs:** A clock_modulator has one input, clock_in. The clock_in is the clock with a frequency of 50Mhz.
- **Outputs:** A clock_modulator has one output, clock_out. The clock_out is the clock that is slowed down by half.

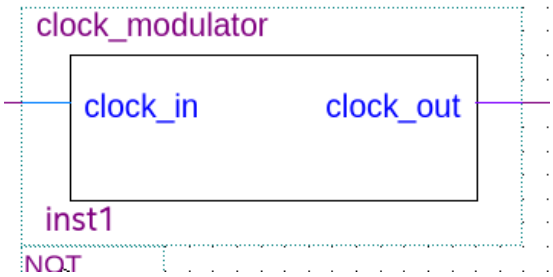


Figure 30: The block symbol of the clock_modulator module used in the vga_encoder functional unit.

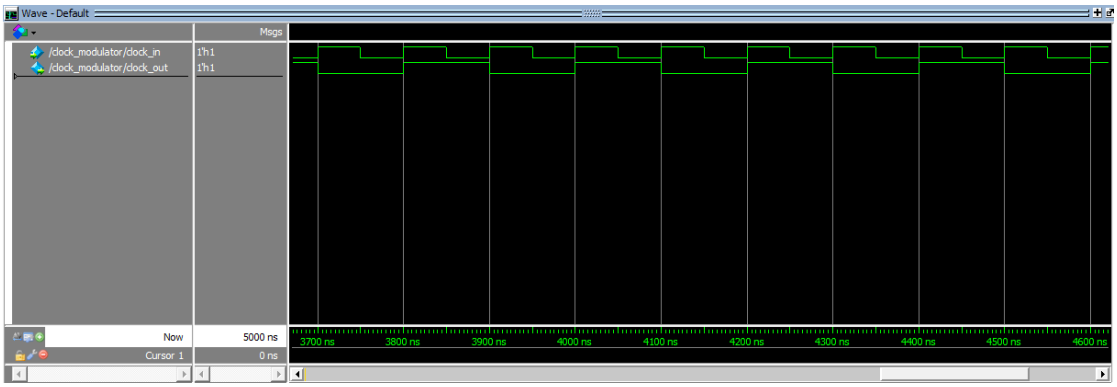


Figure 31: The simulation results of the clock_modulator individual block used in the vga_encoder functional unit.

2.5.3 vga_640x480 Component Block

A vga_640x480 generates the h_sync and v_sync with the horizontal and vertical timing specification. If the internal counters v_count and h_count reach specific zone, h_sync and v_sync becomes 1 or 0 when the counter is smaller than the front porch. In the first simulation, it focuses on h_sync with a simulation during (16+96+48+640+16+96+48)*40ns = 960*40ns = 38400ns. The h_sync is 1 before it passes 800*40ns = 32000ns and becomes 0 after it passes. On the other hand, the second simulation focuses on v_sync and it is 1 before it reaches 32000*525*40ns = 16800000ns and becomes 0 after it reaches.

- **Inputs:** A vga_640x480 block has two inputs, clock_25, and reset_frame. A clock_25 is a clock with a frequency of 25MHz, and reset_frame is the switch that resets the internal counters, v_count and h_count.
- **Outputs:** A vga_640x480 has four outputs, h_sync, v_sync, h_display, and v_display. The values of h_sync and v_sync depends on whether the clock cycle or lines are in the period of front porch or not. If they are, the values become 0. The values of h_display and v_display are the signals indicating whether it is able to draw on the screen or not.

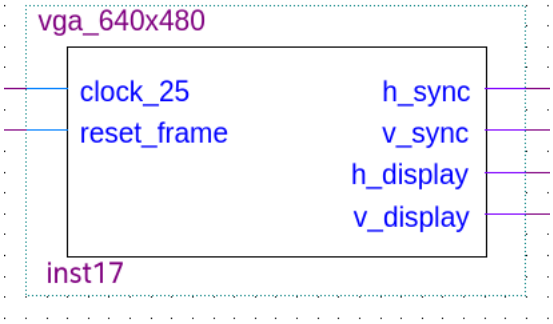


Figure 32: The block symbol of the vga_640x480 module used in the vga_encoder functional unit.

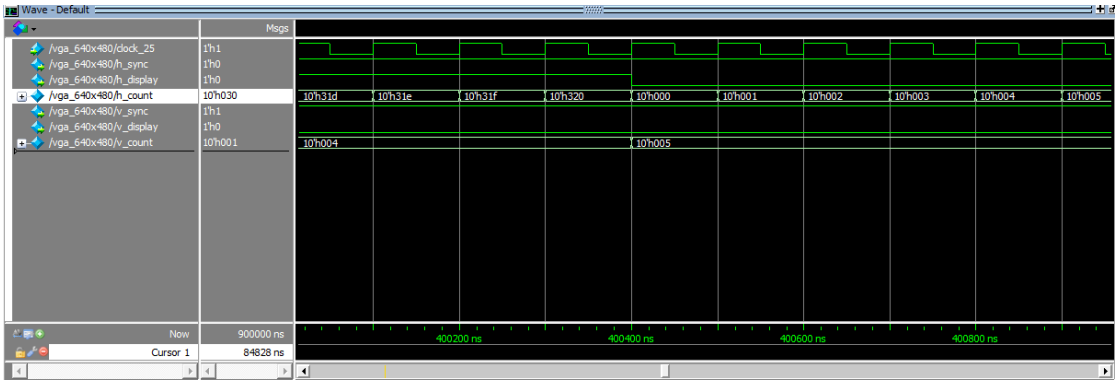


Figure 33: The simulation results of the vga_640x480 individual block used in the vga_encoder functional unit.

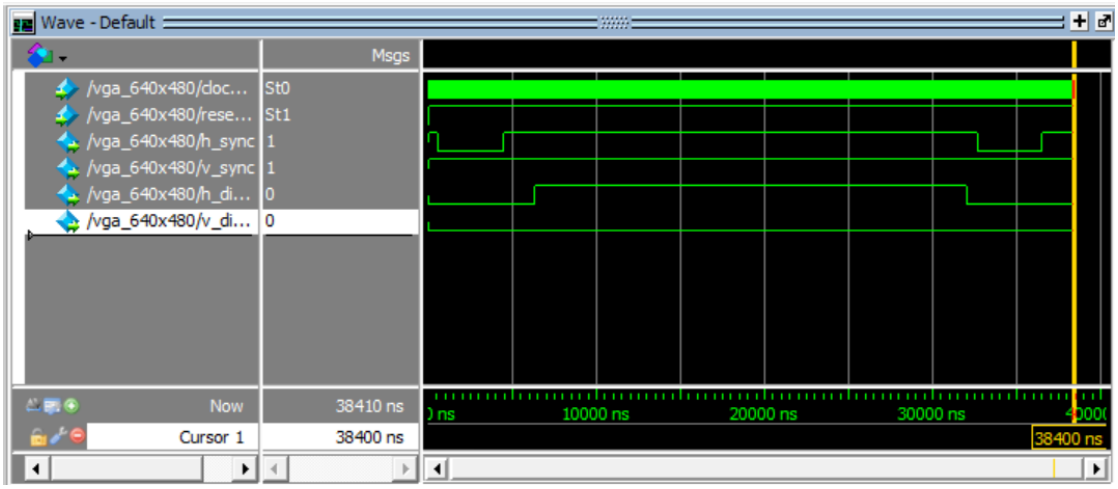


Figure 34: The simulation results of the vga_640x480 individual block used in the vga_encoder functional unit.

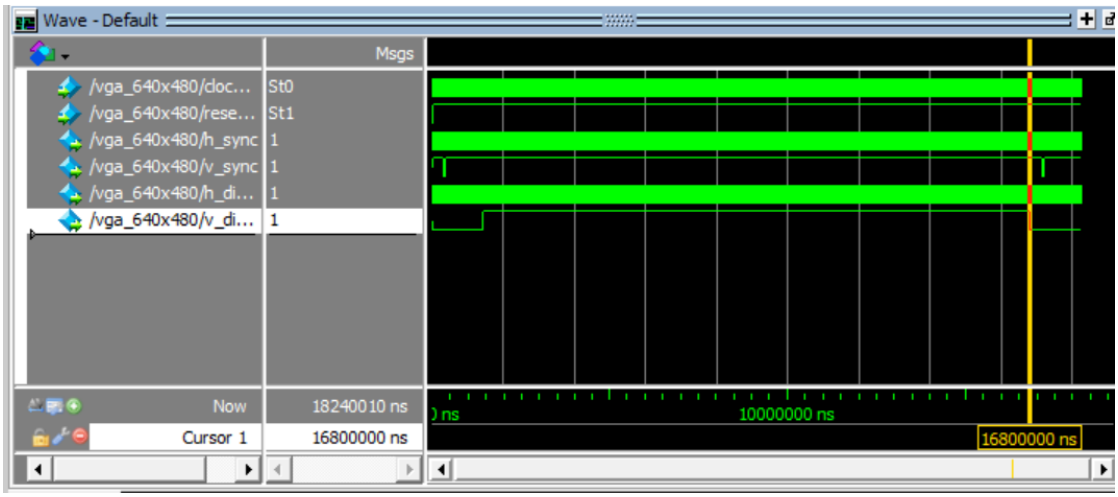


Figure 35: The simulation results of the vga_640x480 individual block used in the vga_encoder functional unit.

2.5.4 rgb_decoder Component Block

A rgb_decoder decodes 2-bit inputs into 4-bit outputs by duplicating the data. For example, 00 becomes 0000, 01 becomes 0101, 10 becomes 1010, 11 becomes 1111. However, in this project, only 00 and 11 is used for the input because only the white and black color is used.

- **Inputs:** A rgb_decoder has three inputs, red_in[1..0], green_in[1..0], and blue_in[1..0]. They are only 00 or 11 in this project to generate only white and black color.
- **Outputs:** A rgb_decoder has three outputs, red_out[3..0], green_out[3..0], and blue_out[3..0]. They are decoded data from the inputs, and they are only 0000 and 1111 in this project to generate only white and black color.

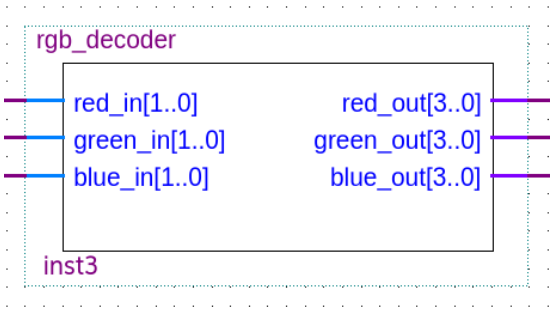


Figure 36: The block symbol of the rgb_decoder module used in the vga_encoder functional unit.

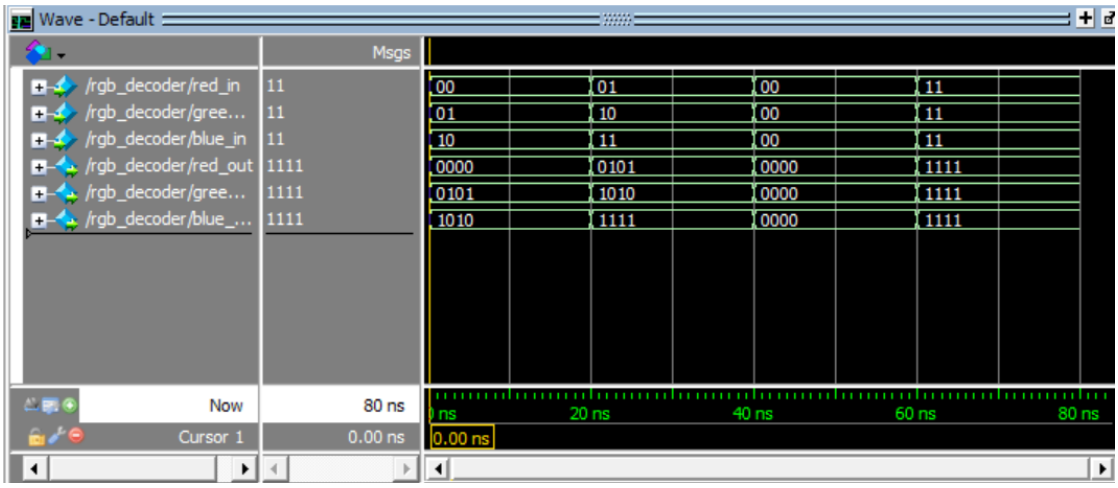


Figure 37: The simulation results of the rgb_decoder individual block used in the vga_encoder functional unit.

2.6 vcr_decoder Functional Unit

The vcr_decoder module converts an IR signal sent from a VCR remote into a decimal value between 0 and 9. The logic symbol follows in **Figure 28**.

- **Inputs:** The vcr_decoder module has two inputs, clk and IR. clk is a 10 KHz clock signal that is used to drive the module. IR is the Infrared signal coming from the VCR remote that will be translated by the module.
- **Outputs:** The vcr_decoder module has a single output, displayValue, which is the 0-9 representing the IR signal that was received by the module as input.

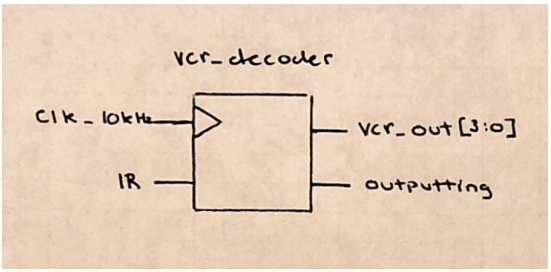


Figure 38: The block symbol of the vcr_decoder functional unit used in the final design.

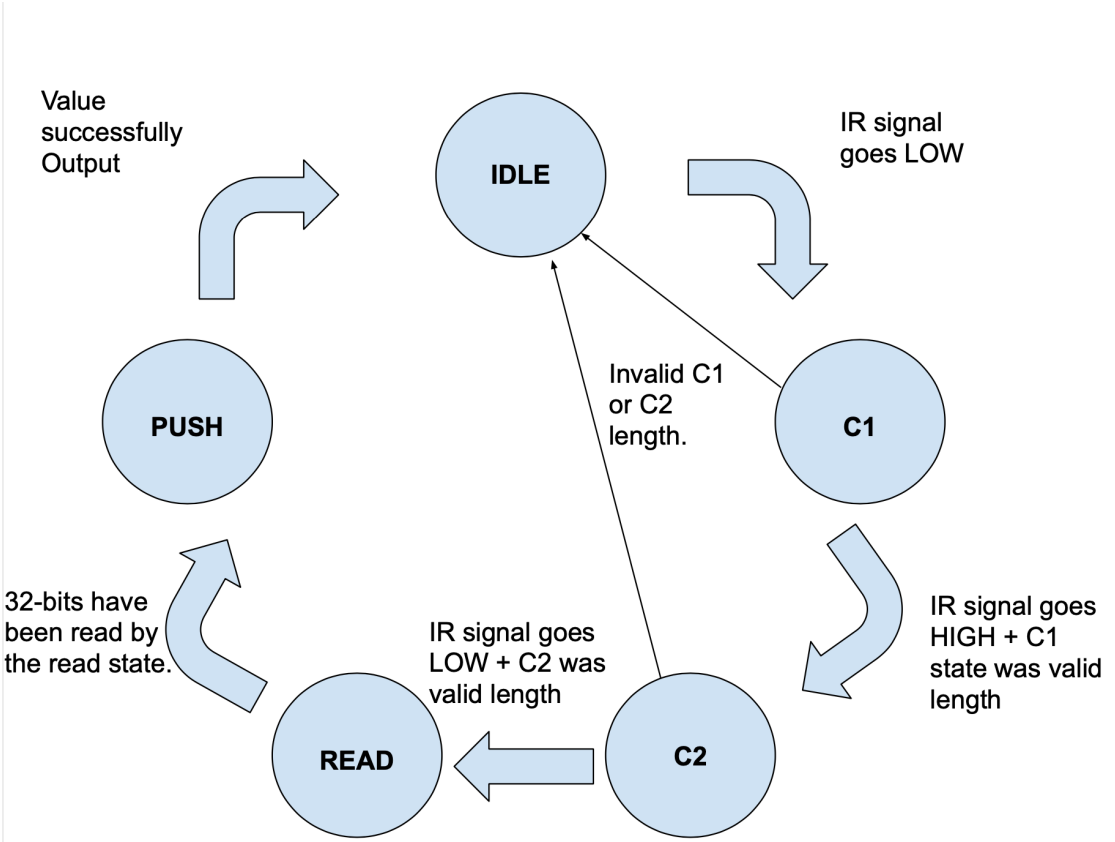


Figure 39: State Diagram for the vcr_decoder Functional Unit. The state machine had five states: IDLE, C1, C2, READ, PUSH. The incoming IR signal from the VCR remote is a logic HIGH by default. If the vcr_decoder is receiving a continuous logic HIGH from the remote, it will stay in the IDLE state indefinitely. When the signal goes LOW for the first time the state machine switches to the C1 state, which is a control signal state. It will stay in this state until the signal goes HIGH again at which point it will switch to the C2 control state. Both the control state have unique pulse lengths that are checked before moving to the next state. When in the C2 state, if the IR signal goes low, the state machine will begin the READ state and read the 32-bit value encoded for by the IR signal. Once the 32-bits have been read, the state machine will switch to the push state and output the value encoded for by the signal.

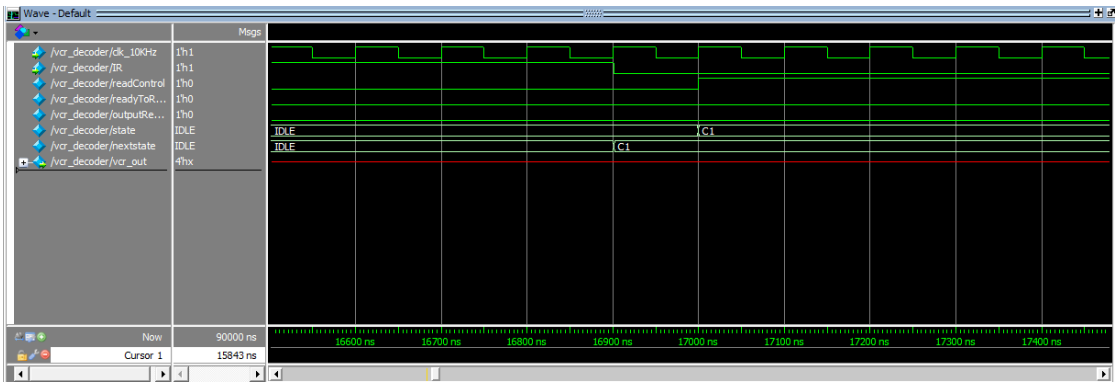


Figure 40: Simulation results of the vcr_decoder Functional Unit showing the state transition from the IDLE state to the C1 control state following the first time the IR signal goes to a logic LOW.

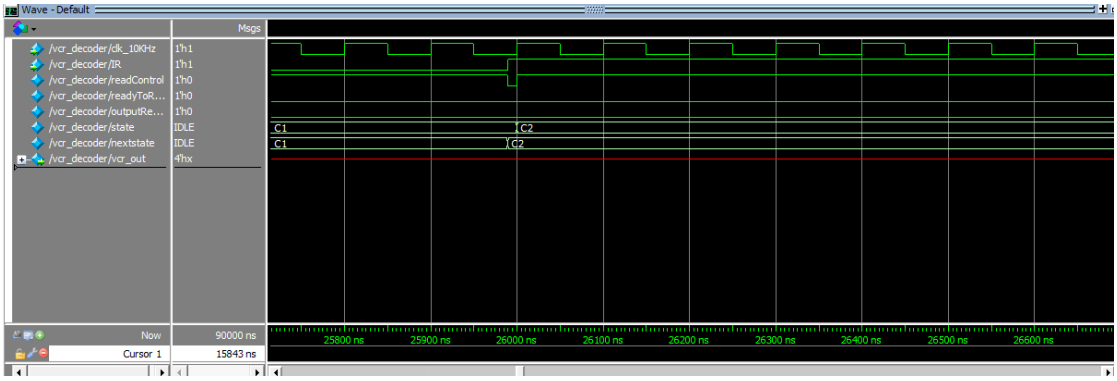


Figure 41: Simulation results of the vcr_decoder Functional Unit showing the state transition from the C1 control state to the C2 control state.

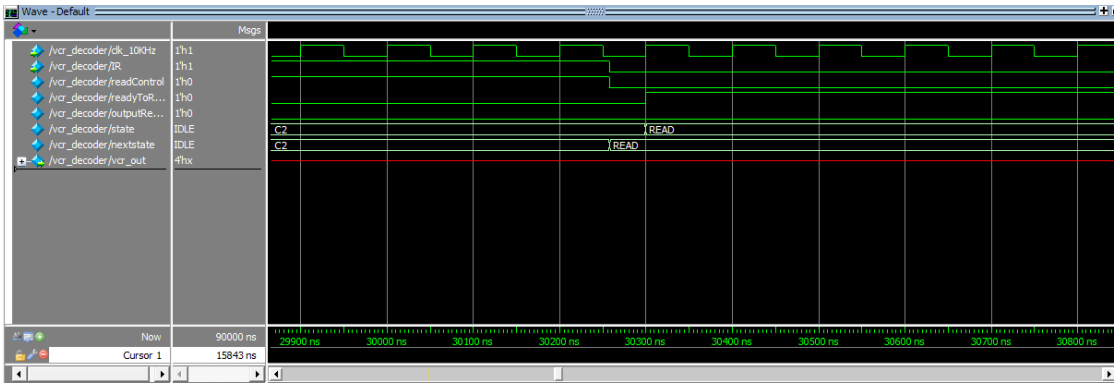


Figure 42: Simulation results of the vcr_decoder Functional Unit showing the transition from the C2 control state to the READ state, indicicating that it is currently reading a 32-bit IR signal corresponding the button on the VCR remote that was pressed.

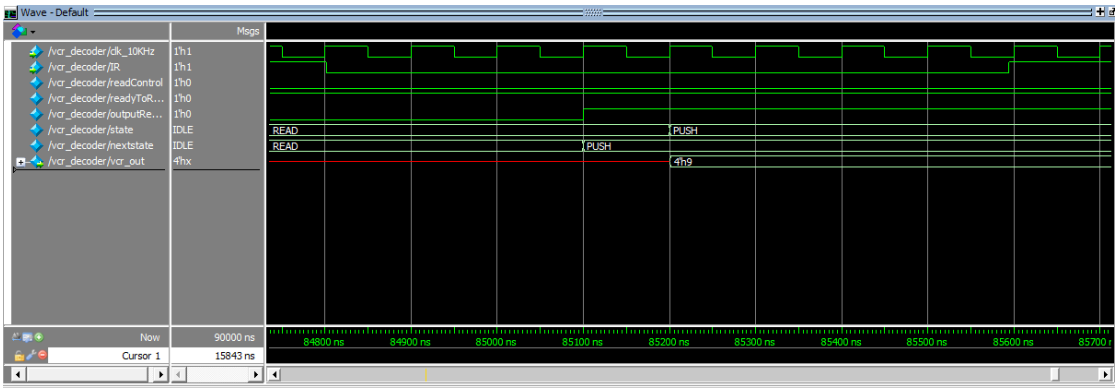


Figure 43: Simulation results of the vcr_decoder Functional Unit showing the transition from the READ state to the PUSH state indicating that the full 32-bit input value has been processed and an output value is produced.

2.6.1 mux2 Component Block

The mux2 individual block is used to choose whether the seven segment display shows the output of the nes decoder functional unit or the vcr_decoder functional unit.

- **Inputs:** The mux2 individual block has 2 inputs: sel and in. in is the input signal which is 2 bits wide. sel is used to select which of the values goes to the output.
- **Outputs:** The mux2 individual block has 1 output: out. This is the value that is selected by the sel input signal.

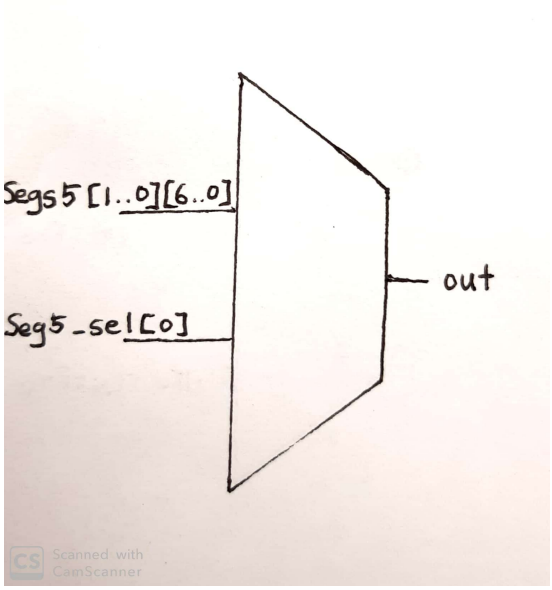


Figure 44: The block symbol of the mux2 module used in the vcr_decoder functional unit.

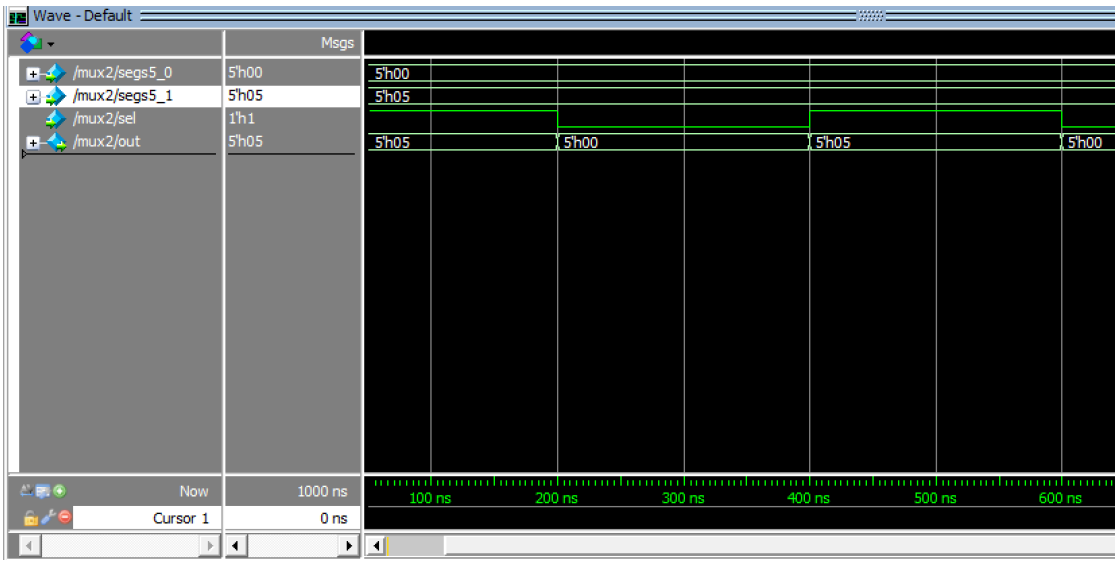


Figure 45: The simulation results of the mux2 individual block used in the vcr_decoder functional unit.

2.6.2 ReadState Component Block

The ReadState Individual Block is used to implement the READ state within the vcr_decoder Module. The logic symbol follows in **Figure 29**.

- **Inputs:** The ReadState Individual Block has 3 inputs: clk, IR, and start. clk is a 10 KHz clock that drives the ReadState block. IR is the infrared signal generated by the VCR remote on a given button push. Start is a control signal for the ReadState Individual Block. When start is 1, the block will operate.
- **Outputs:** The ReadState Individual Block has 2 outputs: pushOutput and outputValue. pushOutput is a control signal that is used to communicate that the ReadState block is ready to output the value of outputValue. outputValue is the 32-bit value that identifies the IR signal that reciieved by the block.

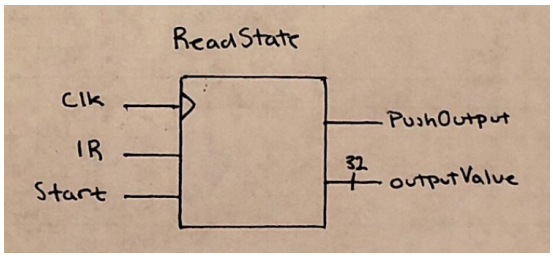


Figure 46: The block symbol of the ReadState module used in the vcr_decoder functional unit.

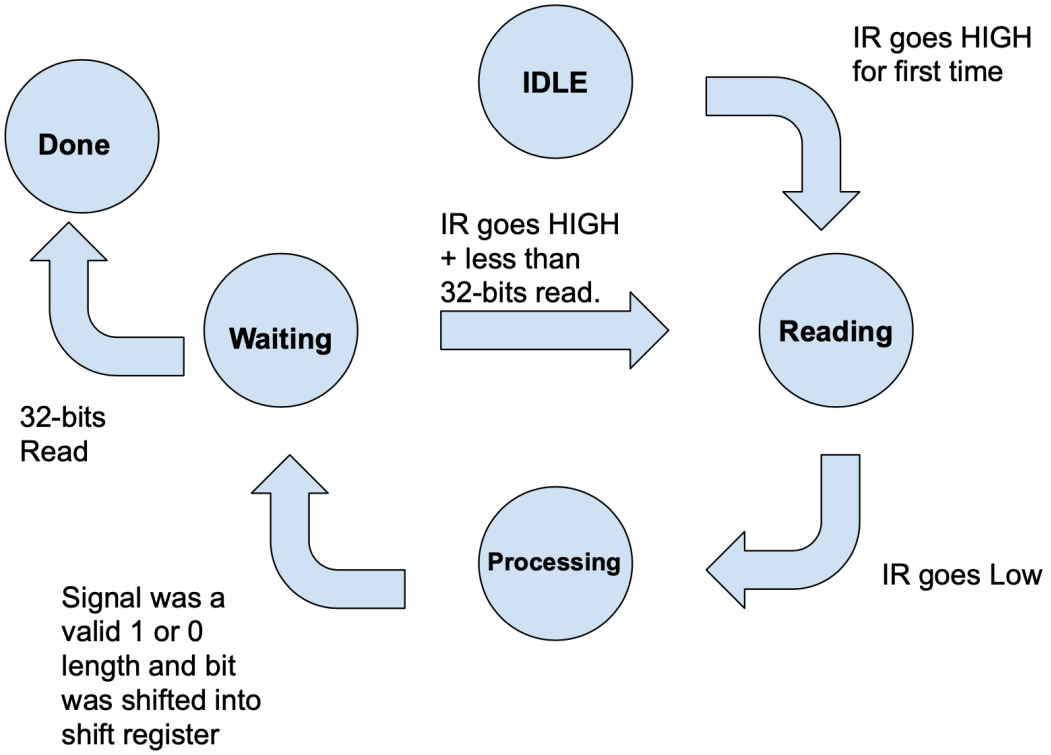


Figure 47: State Diagram for the ReadState Individual Unit. The state machine has five states: IDLE, Reading, Processing, Waiting, and Done. The IDLE state represents the initial logic LOW pulse at the beginning of the 32-bit sequence. Once this signal goes HIGH, the state machine will enter a cycle of Reading, Processing, and Waiting for each HIGH signal. The Reading state will measure the length of the pulse, the Processing state will assign the length either a 1 or 0 bit and shift the bit into the ShiftRegister Individual Unit, and the Waiting state will reset for the next HIGH signal. Once 32-bits have been read, the state machine will enter the Done state and outpit the 32-bit value as a hexadecimal value that uniquely identifies the button that was pressed on the VCR remote.

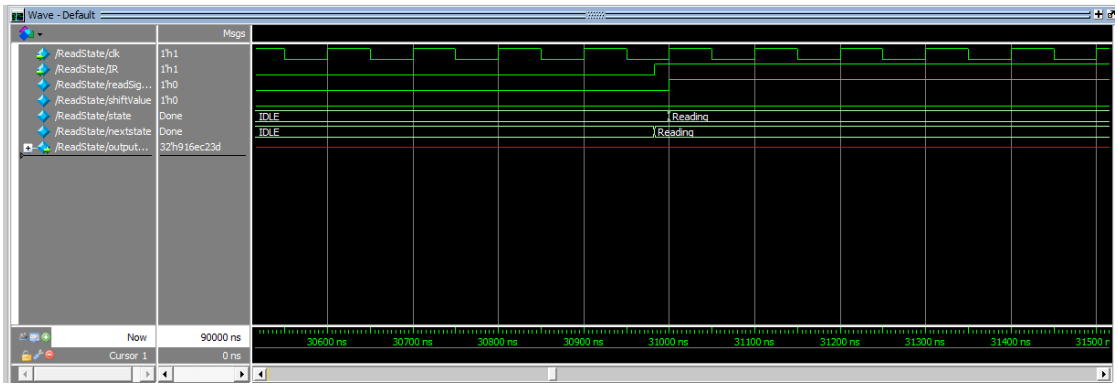


Figure 48: Simulation results of the ReadState individual block used in the vcr_decoder functional unit. The simulation shows the transition from the IDLE state to the Reading State within the ReadState Module. This transition is in response to the initial HIGH IR signal that represents the 32-bit value encoding the button that was pressed.

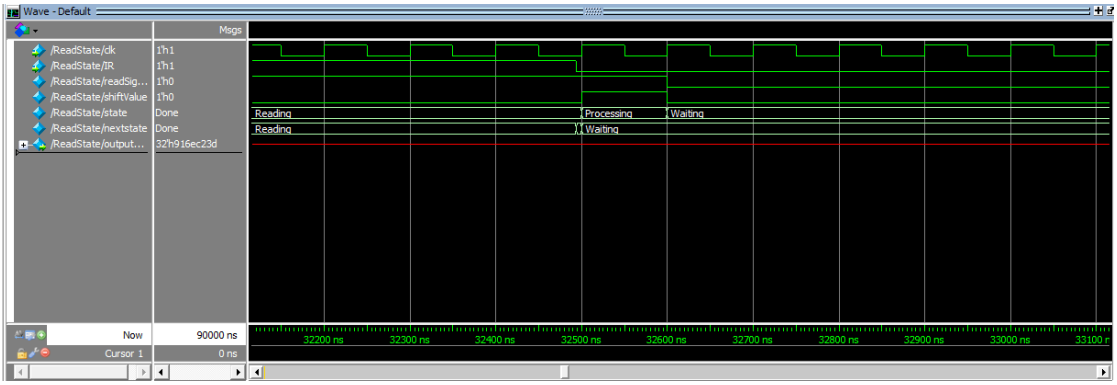


Figure 49: Simulation results of the ReadState individual block used in the vcr_decoder functional unit. The simulation shows Processing State of the ReadState Module. When the state machine is in the Reading state, that means it is actively reading a HIGH signal for IR. As seen in this waveform, when IR goes LOW again, the state machine switches to the Processing state in which it checks the length of the signal to see if it is a logic 1 or a logic 0. The 1 or 0 is then shifted into the shift register on the rising edge of the shiftValue signal, as seen in the waveform. The state machine then switches into the Waiting state, while it waits for IR to go HIGH again.

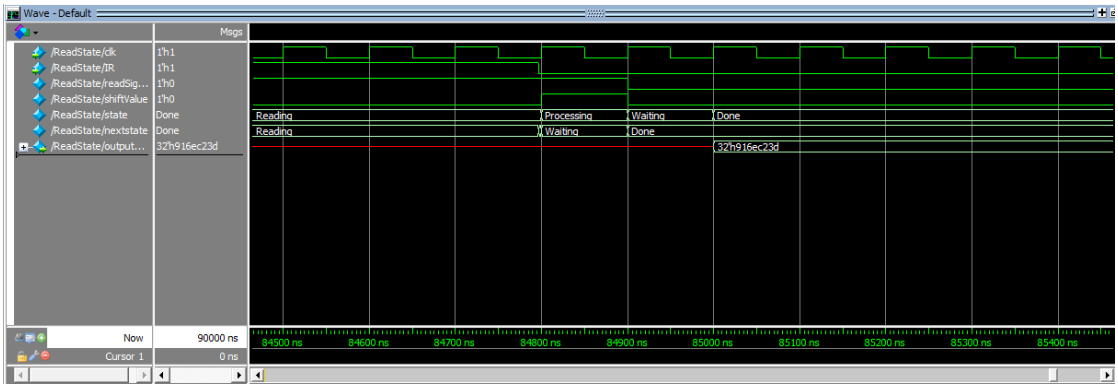


Figure 50: Simulation results of the ReadState individual block used in the vcr_decoder functional unit. The simulation shows the transition from the reading, Processing, Waiting cycle to the Done state in which it outputs the 32-bit hexadecimal value identified by the IR signal input. In each Waiting state, the state machine will check if 32-bits have been read into the shift register. If so, the state machine will switch to the Done state and output the value for the IR signal that it received as input. In addition, it will drive an output signal outputReady HIGH. This signal is used by the vcr_decoder module, to know when to output a result.

2.6.3 ShiftRegister Component Block

The ShiftRegister Individual Block is used to store the 32-bit IR signal value bit by bit as as it is read in by the ReadState Module. The logic symbol follows in **Figure 30**. The simulation results follow in **Figure 31**.

- **Inputs:** The ShiftRegister Individual Block has five inputs. The first input is a clock signal, clk. On every rising edge of this clock signal, a new bit is shifted into the register. The second input is sin. This is the bit that is being shifted into the register. the third input is d. d is a 32-bit value which can be used to fully load the the shift register. The fourth input is load. When load is 1, the value stored at d is shifted into the register. The final input is reset, which clears the shift register's contents
- **Outputs:** The ShiftRegister Individual Block has 2 outputs, q and sout. q is a 32-bit value that represents all the bits that have been shifted into the register. sout is the most significant bit in q.

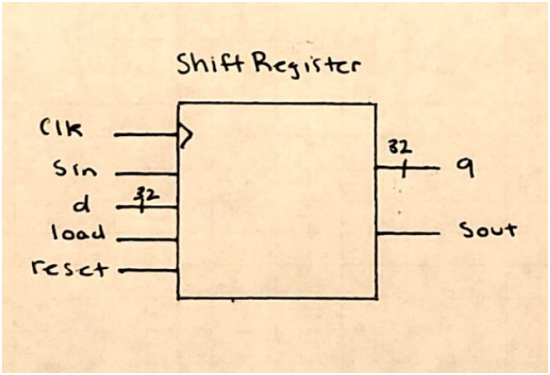


Figure 51: The block symbol of the ShiftRegister module used in the vcr_decoder functional unit.

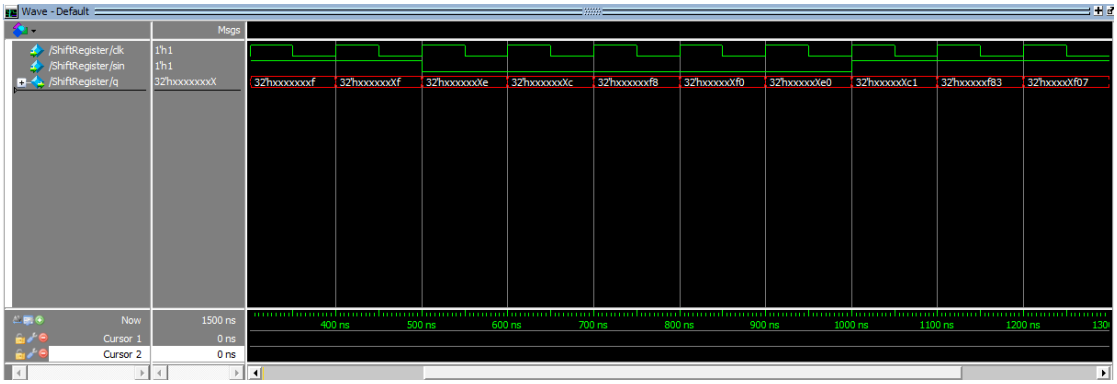


Figure 52: The simulation results for the ShiftRegister module used in the vcr_decoder functional unit.

2.6.4 signalDecoder Component Block

The SignalDecoder Individual Block is used to convert the 32-bit value generated by the ReadState Individual Block into a 4-bit value by 0 and 15 that indicated which button on the VCR remote was pressed.

- **Inputs:** The SignalDecoder Individual Block takes a single 32-bit input that corresponds to the complete IR signal that was read in by the ReadState Individual Block.
- **Outputs:** The SignalDecoder Individual Block outputs a 4-bit value betwee 0 and 15.

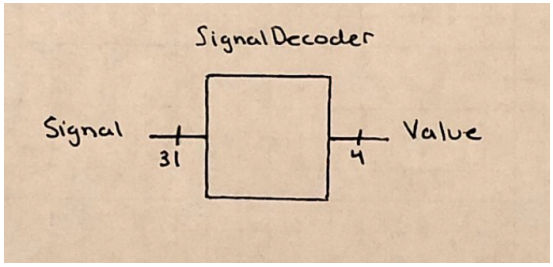


Figure 53: The block symbol of the SignalDecoder module used in the vcr_decoder functional unit.

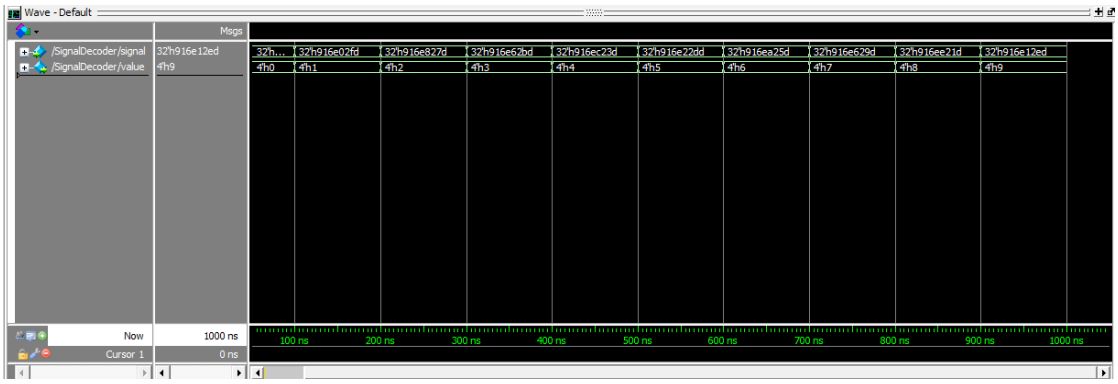


Figure 54: The simulation results of the SignalDecoder individual block used in the vcr_decoder functional unit.

2.7 ps2_keyboard Functional Unit

PS/2 is an interface for keyboards and mice to PC via a 6-pin Mini-DIN connector. The system must provide the keyboard or mouse with 5V source and ground connections. Communication occurs over two wires, a clock line and a data line. The clock has a frequency between 10 kHz and 16.7 kHz. The data has a start bit, which is set as logic low, one byte of data, a parity bit, and a stop bit, which is set as logic high. Each bit is read on the falling edge of the clock signal. In the schematics, ps2_data and ps2_clk first synchronized and debounced. The data signal is then loaded to a shift register on falling edges of the PS/2 clock so that the flip flops can store the data. When the data transmission is finished and the clock remains high for more than 55us, the time longer than half of the worst-case PS/2 clock period, idle counter sends logic high to the AND gate. If there is no error and the signal from the idle counter is high, ps2_code_new becomes logic high indicating that the block is ready for the next signal. The ps2_code[7..0] gets ps2_word[8..1] because flip flops in that position contain the data. In the simulation, the waves indicate clock with 50MHz, ps2_clk, ps2_data, ps2_code_new, and ps2_code[7..0] in sequence. The signals for ps2_clk and ps2_data are gotten from the csv file provided. After ps2_word[10..0] stores the signal of ps2_data at the falling edge of the ps2_clk, ps2_code_new becomes 1 and ps2_word[8..1] is sent to ps2_code[7..0] if there is no error. The simulation figures below shows the simulation of pushing the keys of a, s, d, f, w, e, r, and t. The keys in PS/2 Keyboard have make code in hexadecimal. The code for the keys used in this project is 0x1C, 0x1B, 0x23, 0x2B, 0x1D, 0x24, 0x2D, and 0x2C respectively. The codes are shown in the simulation in the binary format. A block diagram follows in **Figure 32**. The logic symbol follows in **Figure 33**. The simulation results follow in **Figure 34**.

- **Inputs:** The PS/2 keyboard module has three inputs, clk, ps2_data, ps2_clk. The clk is the clock signal of 50MHz, ps2_clk is the clock signal sent from the PS/2 keyboard to read the data, and the ps2_data is the data sent from PS/2 keyboard.
- **Outputs:** The PS/2 keyboard module has two outputs, ps2_code[7..0], ps2_code_new. The ps2_code[7..0] is the make code data for keys on the keyboard, and ps2_code_new indicates whether the module is ready for the next data or not.

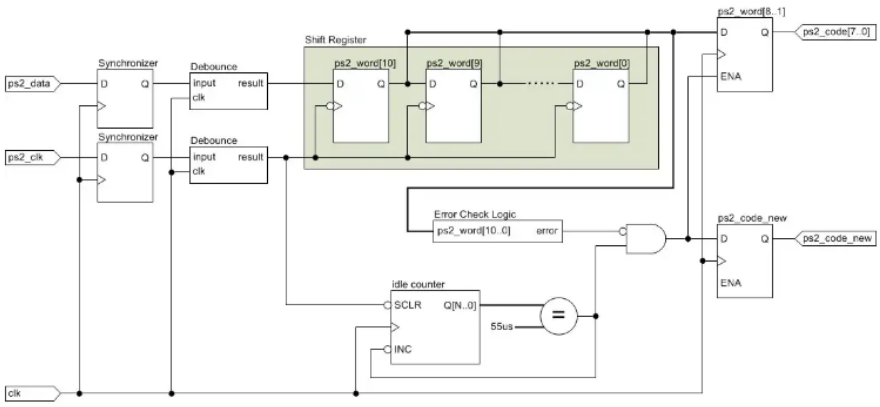


Figure 55: The logic design of the ps2_keyboard functional unit used in the final design.

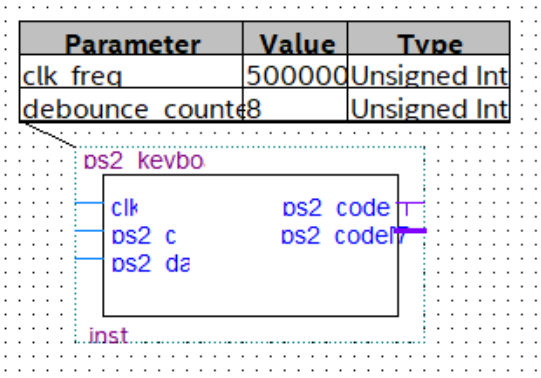


Figure 56: The block symbol of the ps2_keyboard functional unit used in the final design.

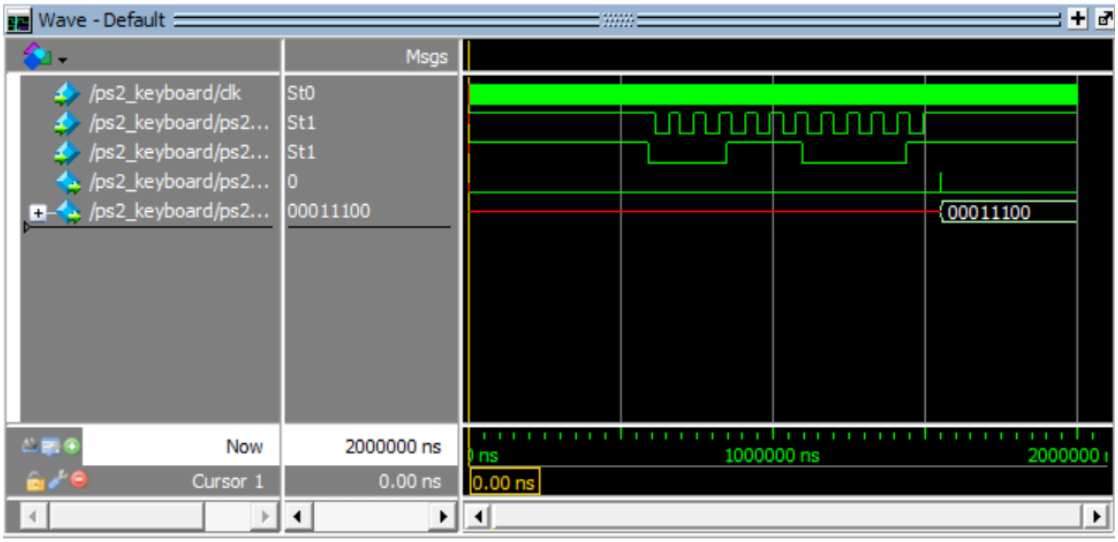


Figure 57: The simulation results for the ps2_keyboard module.

2.7.1 debounce Component Block

When using mechanical switches, the signals often rebound, or bounce, off one another before settling into a stable state. The debounce component is used for successfully sending the data in a stable state. Two flip flops in the design check whether the data changes or not via the XOR gate. If the data changes, the counter block receives the signal to reset the counter. If not, the counter increases until it reaches the specified time and enables the output register. According to simulation, when the data remains stable for 40 ns, the data is sent to the last flip flop and the result signal is updated. A block diagram follows in **Figure 35**. The simulation results follow in **Figure 36**.

- **Inputs:** The debounce block has one input: button. It indicates the input from the keyboard.
- **Outputs:** The debounce block has one output: result. It has the debounced value of the button.

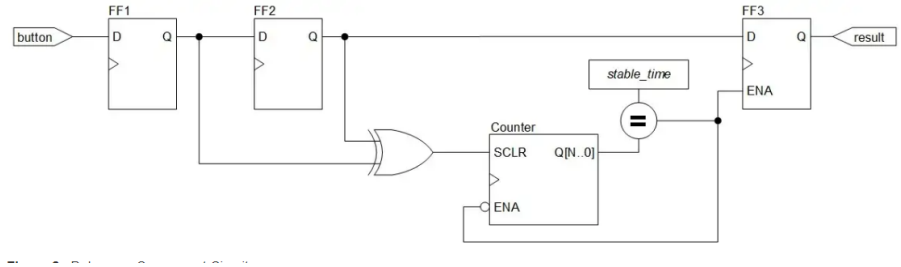


Figure 58: The logic design of the debounce module used in the ps2_keyboard functional unit.

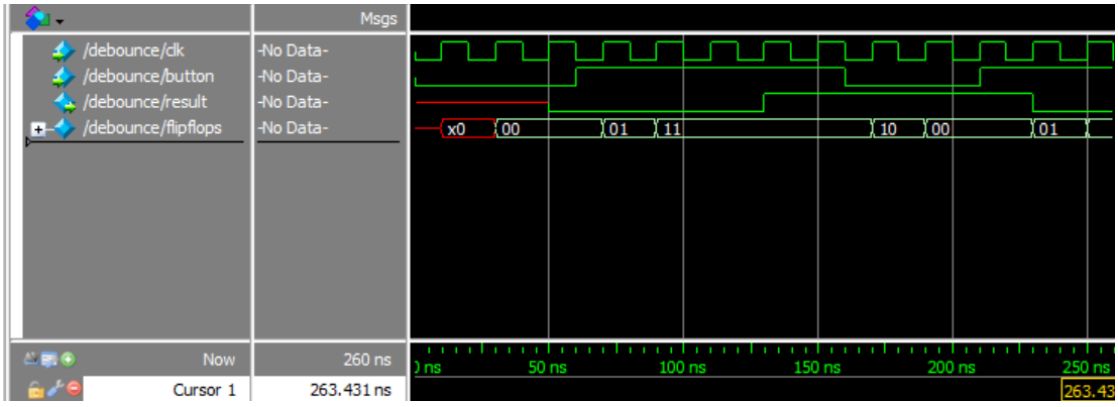


Figure 59: The simulation results for the debounce module used in the ps2_keyboard functional unit.

2.8 nes_decoder Functional Unit

The nes_decoder module interfaces with inputs from and outputs to the nes controller hardware to parse the states (active low) of the eight buttons on the controller. The logic symbol follows in **Figure 37**. The simulation results follow in **Figure 38**.

- **Inputs:** An input clock (at 50MHz for this design) which determines how quickly the system reads the nes inputs; an nes_data input that is the serialized data from the nes controller which describes the buttons pushed at any given moment; a reset signal that sets the system back to a default state; and a read_data input that, when driven high while the system is ready to read, starts the reading process. The nes_data input is the only input connected to the nes controller.
- **Outputs:** Two outputs (nes_latch and nes_clock) that are sent to the nes controller and signal to the controller to start sending the first or next button states along nes_data, respectively; ready_to_read, which is driven high by the module after the system has completed an input reading period and is ready for another round of reading; and eight outputs describing the most recently read states of each of the buttons on the nes controller (active low, they are the A, B, Start, Select, Up, Down, Left, and Right Buttons).

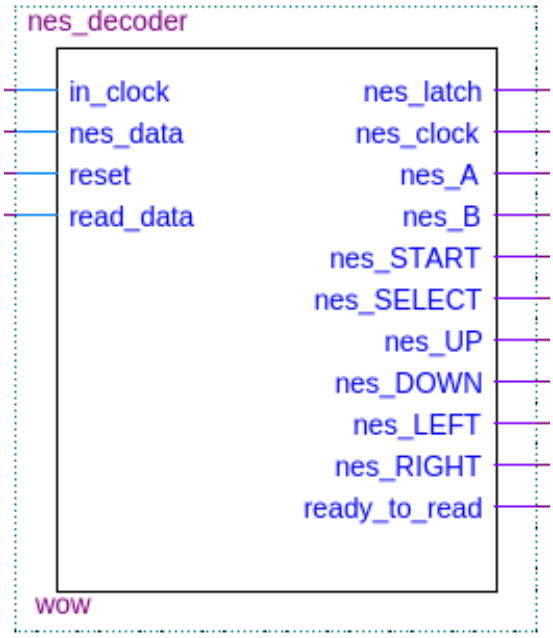


Figure 60: The block symbol of the nes_decoder functional unit used in the final design.

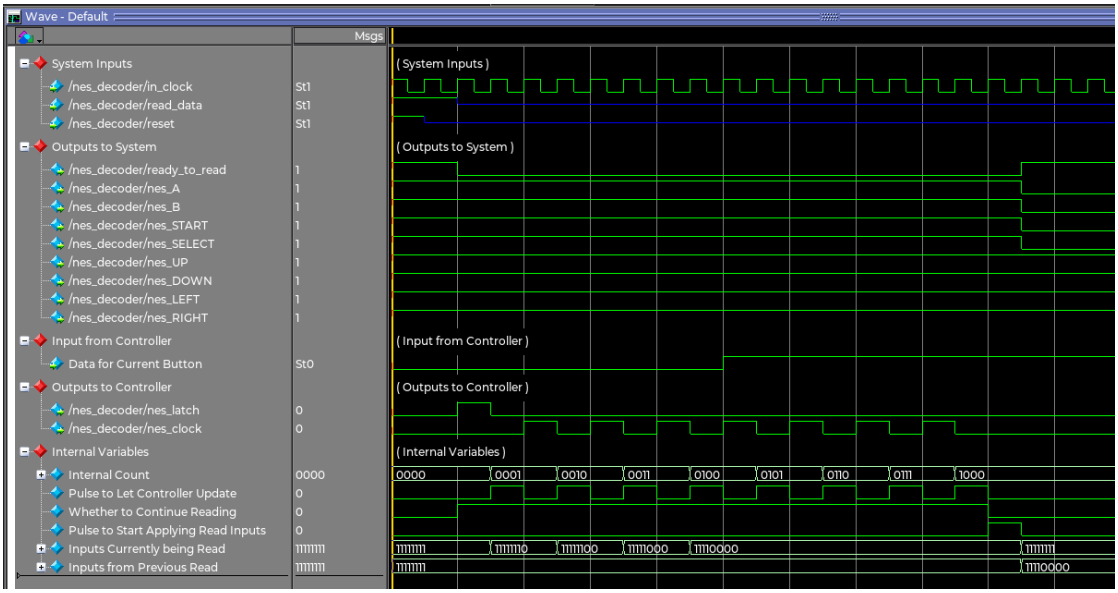


Figure 61: The simulation results for the nes_decoder module.

2.9 parsed_clock Functional Unit

The parsed_clock module takes in a clock and some control signals that start, modify the speed of, and reset a clock that is accurate to 1Hz. These outputs are parsed into seconds, minutes, and hours (in decimal notation) to match the typcial clock format of (HH.MM.SS). The inputs come from the nes_decoder and the FPGA's internal pins, and the outputs all go to the seven_seg_6 (six-digit seven segment display decoder) module in the design. A block diagram follows in **Figure 39**. The logic symbol follows in **Figure 40**. The simulation results follow in **Figure 41**.

- **Inputs:** A 50Mhz clock from the FPGA's internal pins; a reset signal from either the active-high button state of the B button on the nes controller or the overall system_reset signal that is the active-high button state of a button on the FPGA itself; and four more active-high button states from the nes controller (A Button acting as the start (toggle) enable signal of the clock, Start Button acting as the double-speed mode input of the parsed clock while held, and the Select Button acting as the super-speed mode input of the parsed clock while held).
- **Outputs:** Six 4-bit binary value outputs describing the one's and ten's places of the seconds, minutes, and hours of the parsed clock. These values never exceed 59 seconds, 59 minutes, and 23 hours, so overflow is impossible. These outputs all directly lead to the seven_seg_6 module that converts these 4-bit values into seven segment endcoded values.

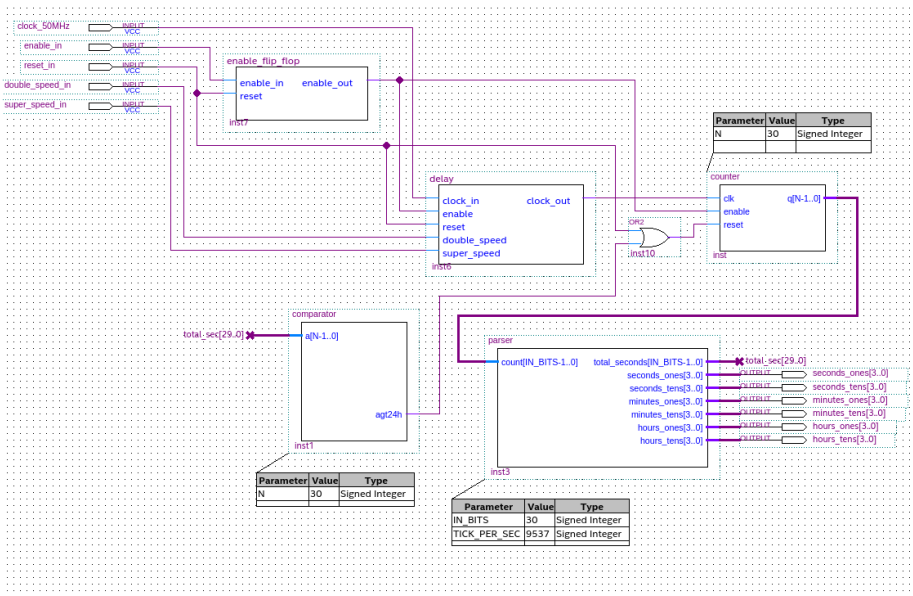


Figure 62: The logic design of the parsed_clock functional unit used in the final design.

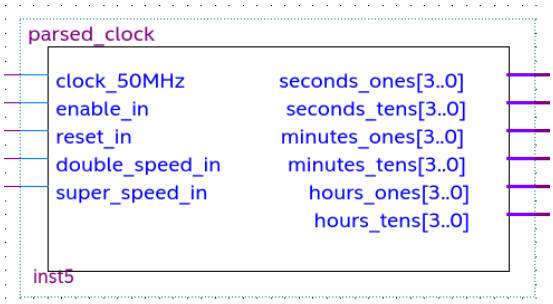


Figure 63: The block symbol of the parsed_clock functional unit used in the final design.

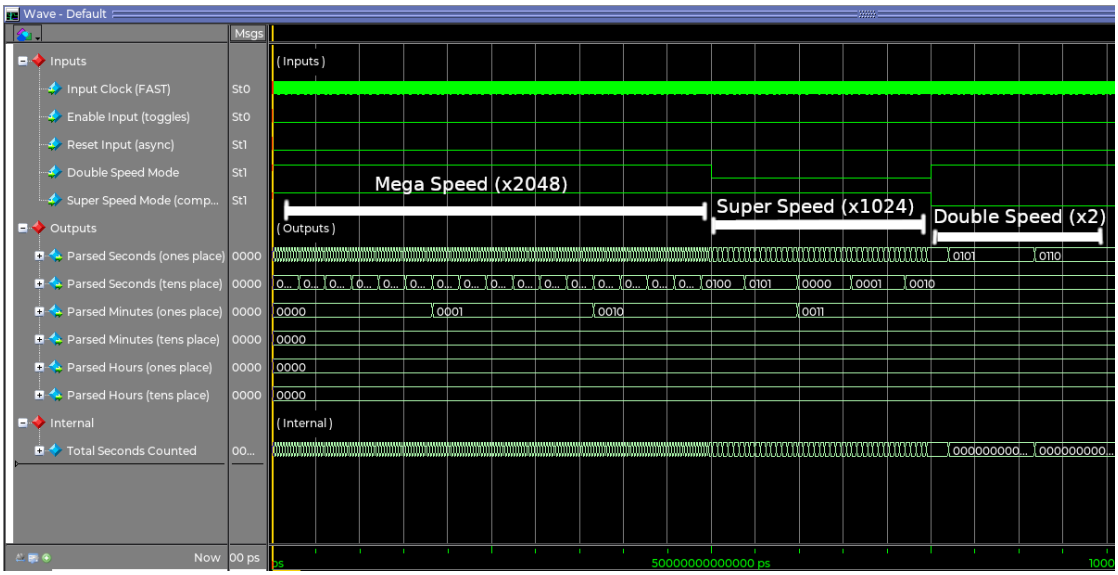


Figure 64: The simulation results for the parsed_clock module.

2.9.1 enable_flip_flop Component Block

The enable_flip_flop submodule acts to toggle whether the clock is enabled (should count). It does this by taking in an enable input from the design, and on the positive edge of this enable, toggling it's enable_out output. It can be reset. The logic symbol follows in **Figure 42**. The simulation results follow in **Figure 43**.

- **Inputs:** An input enable_in that, on its rising edge, toggles the outputted enable value of the module; and a reset input that resets the flip_flop to logic low.
- **Outputs:** An output that represents whether the clock should count or not (the submodule's enable value). This goes to the delay and counter blocks.

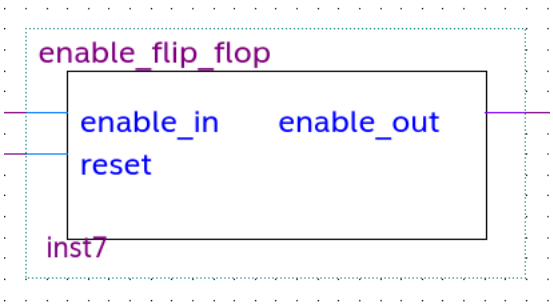


Figure 65: The block symbol of the enable_flip_flop module used in the parsed_clock functional unit.

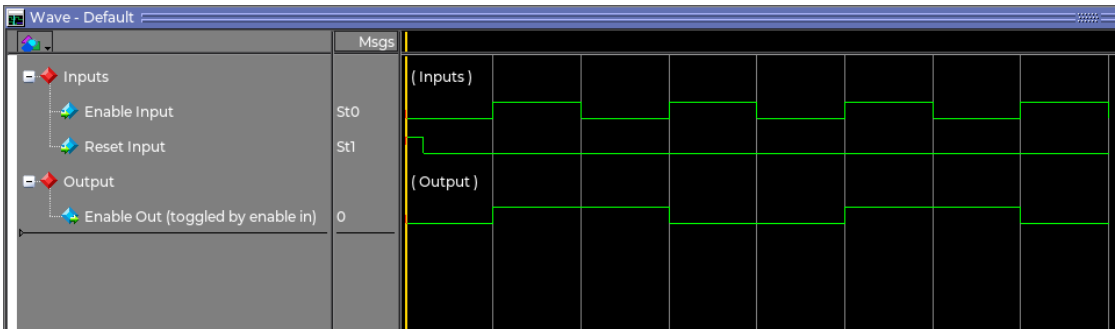


Figure 66: The simulation results for the enable_flip_flop module used in the parsed_clock functional unit.

2.9.2 delay Component Block

The delay submodule, by default, slows an input clock down by a factor of 1/2 to the 10. This is so that an input clock of 50MHz (as in this design) is slowed to a frequency that is easier converted to 1Hz later on through division. This submodule also allows for 4 modes of speed (MEGA, Super, Double, and Normal speeds). MEGA is 2048, Super is 1024, and Double is 2 times as fast as the Normal speed. This is controlled by the values of two speed inputs that act as a select signal between these options. A block diagram follows in extbfFigure 44. The logic symbol follows in **Figure 45**. The simulation results follow in **Figure 46**.

- **Inputs:** An input clock to be slowed (50Mhz in this design); an active-high enable that allows the submodule to propagate signal; a reset signal to reset all flip_flops to 1 (such that the first signal overflows and sends all held values to 0); and two inputs (double_speed and super_speed) that combine as a select signal to choose between four possible output clock speeds (MEGA 1,1 , Super 1,0 , Double 0,1 , and Normal 0,0 speeds for the select signal super_speed, double_speed).
- **Outputs:** A clock that is 1/2 to the 10 slower than the input clock in normal mode, somewhat faster depending on the speed mode selected (described above).

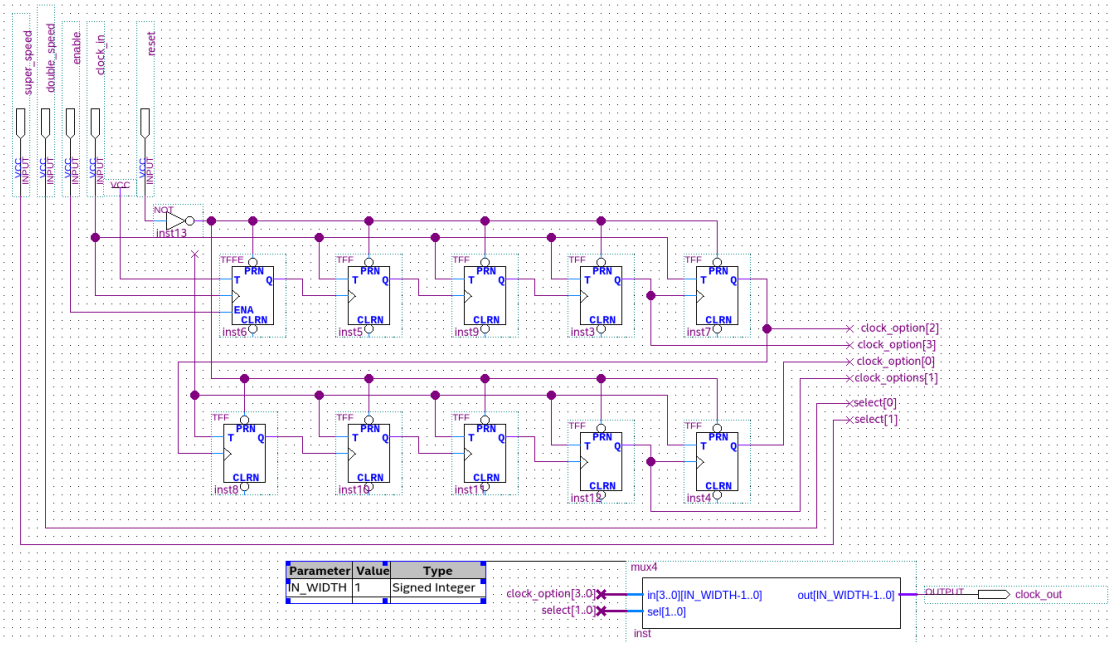


Figure 67: The logic design of the delay module used in the parsed_clock functional unit.

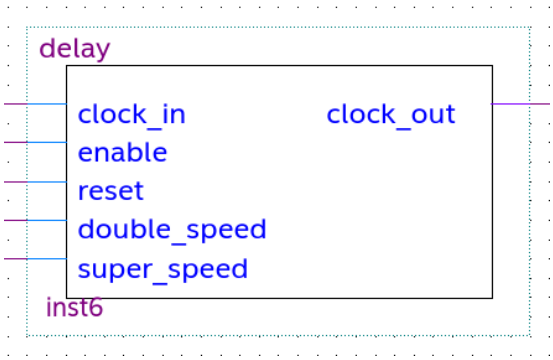


Figure 68: The block symbol of the delay module used in the parsed_clock functional unit.

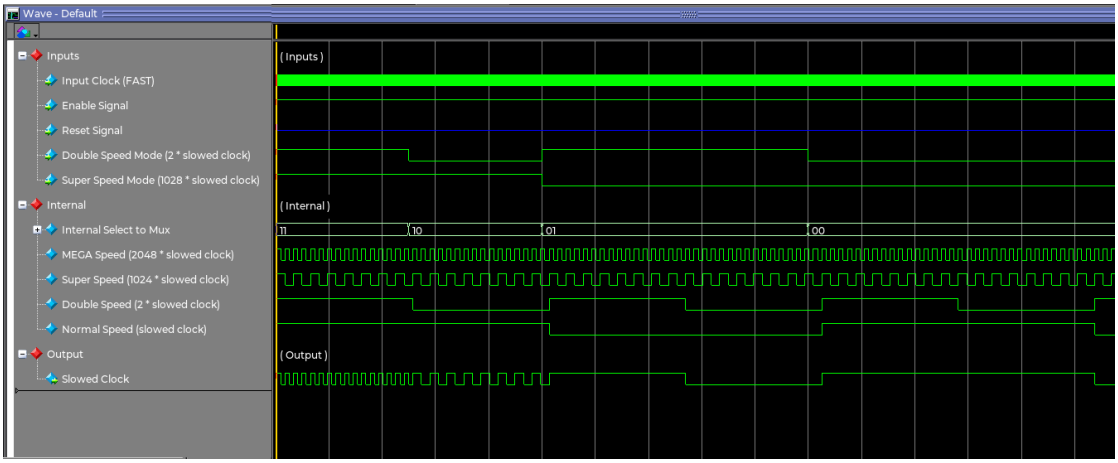


Figure 69: The simulation results for the delay module used in the parsed_clock functional unit.

2.9.3 mux4 Component Block

The mux4 submodule is a 4-option mux (where each option in this design is a one-wide clock speed option). This accounts for the select signals in the delay block. The logic symbol follows in **Figure 47**. The simulation results follow in **Figure 48**.

- **Inputs:** Four one-bit options to send to the output (the speed options of the delay block); and a 2-bit select signal that maps select values 11, 10, 01, and 00 to sending clock_options 3, 2, 1, and 0.
- **Outputs:** One 1-bit output from the 4 options presented in the inputs, selected by the select signal. This is also the output of the delay block.

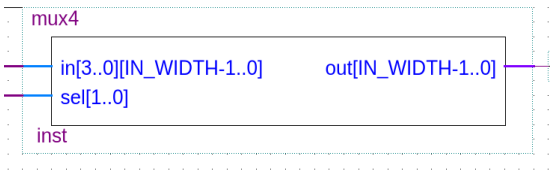


Figure 70: The block symbol of the mux4 module used in the parsed_clock functional unit.

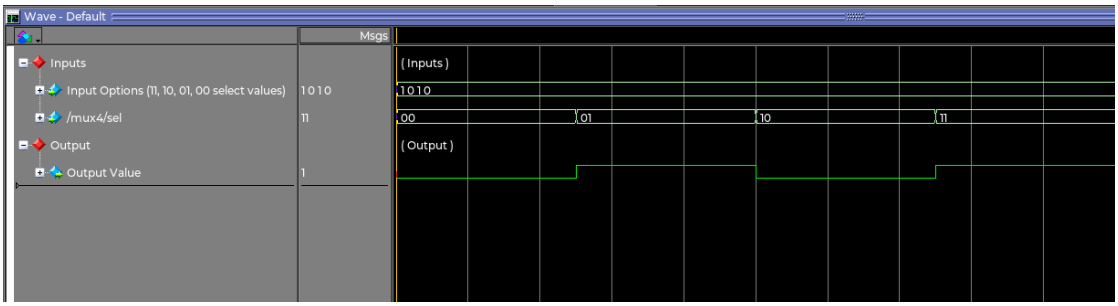


Figure 71: The simulation results for the mux4 module used in the parsed_clock functional unit.

2.9.4 counter Component Block

The counter submodule is a simple one-by counter that increments it's output q on every clock cycle. It has a default max width of q of 30-bits and can be reset and enabled via its inputs. The logic symbol follows in **Figure 49**. The simulation results follow in **Figure 50**.

- **Inputs:** A clock signal (clk) on which to increment the count held in the q output; a reset signal which sets the q output back to 0; and an enable signal that determines whether the counter should react to the clock.
- **Outputs:** The count output held in the output q, with a default maximum width of 30-bits.

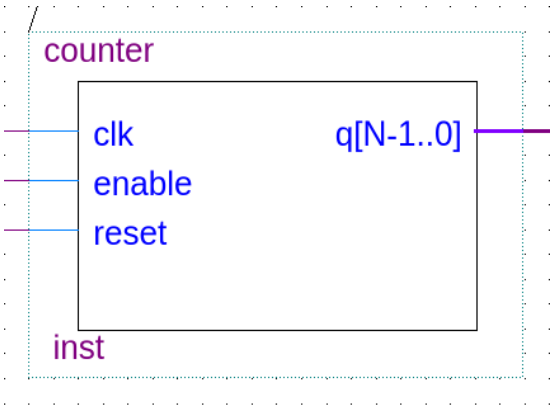


Figure 72: The block symbol of the counter module used in the parsed_clock functional unit.

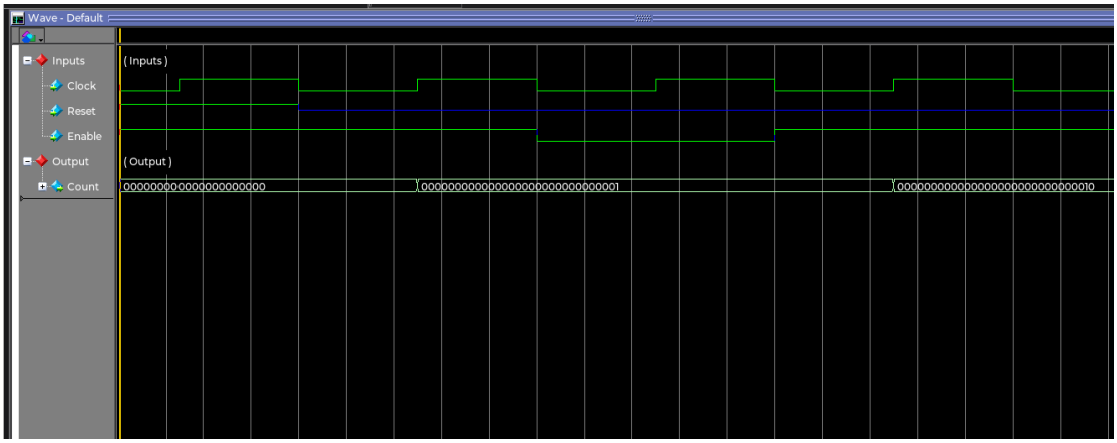


Figure 73: The simulation results for the counter module used in the parsed_clock functional unit.

2.9.5 parser Component Block

The parser submodule takes in a binary value up to 30-bits wide of slowed clock ticks, divides it by the number of clock ticks per second (9537 in this design, due to the delay block), and parses it into seconds, minutes, and hours (in decimal notation) to match the typcial clock format of (HH.MM.SS). The logic symbol follows in **Figure 51**. The simulation results follow in **Figure 52**.

- **Inputs:** The count of clock ticks of the parsed_clock module, outputted by the counter submodule.
- **Outputs:** The total seconds counted so far (used for resetting at 24 hours), and six 4-bit binary value outputs describing the one's and ten's places of the seconds, minutes, and hours of the parsed clock.

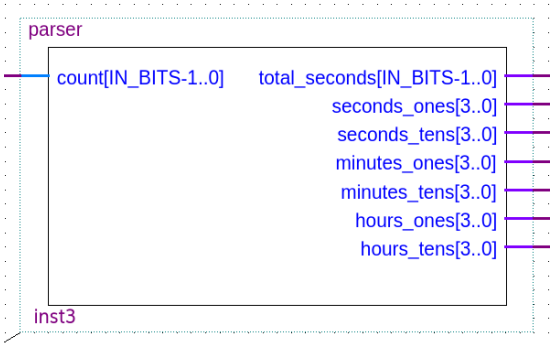


Figure 74: The block symbol of the parser module used in the parsed_clock functional unit.

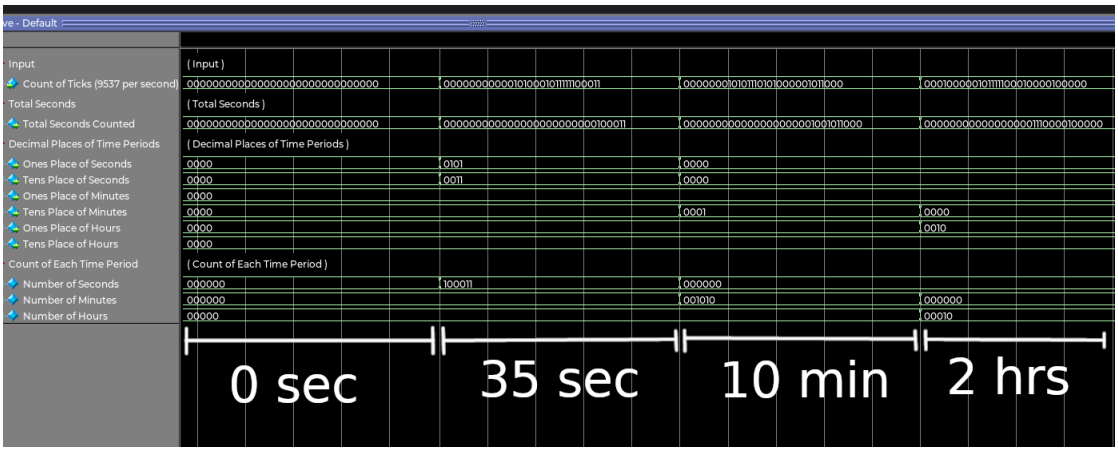


Figure 75: The simulation results for the parser module used in the parsed_clock functional unit.

2.9.6 comparator Component Block

The comparator submodule takes in the total seconds counted by the parsed clock since the last reset, and sends a signal to reset the system when it is greater than or equal to the number of seconds in 24 hours. This prevents the clock from outputting values over what is possible in the typical clock format (HH.MM.SS). The logic symbol follows in **Figure 53**. The simulation results follow in **Figure 54**.

- **Inputs:** The total seconds counted by the parsed clock since the last reset (default max width of 30-bits, called a).
- **Outputs:** Whether the total seconds input (a) is greater than or equal to the number of seconds in a day ($60 * 60 * 24 = 3600$).

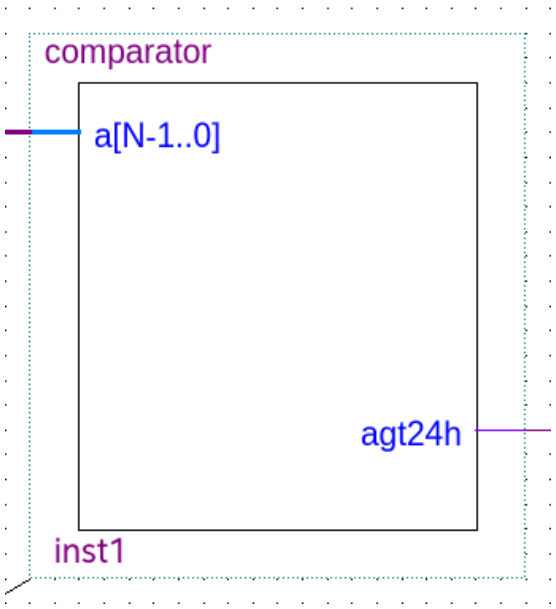


Figure 76: The block symbol of the comparator module used in the parsed_clock functional unit.

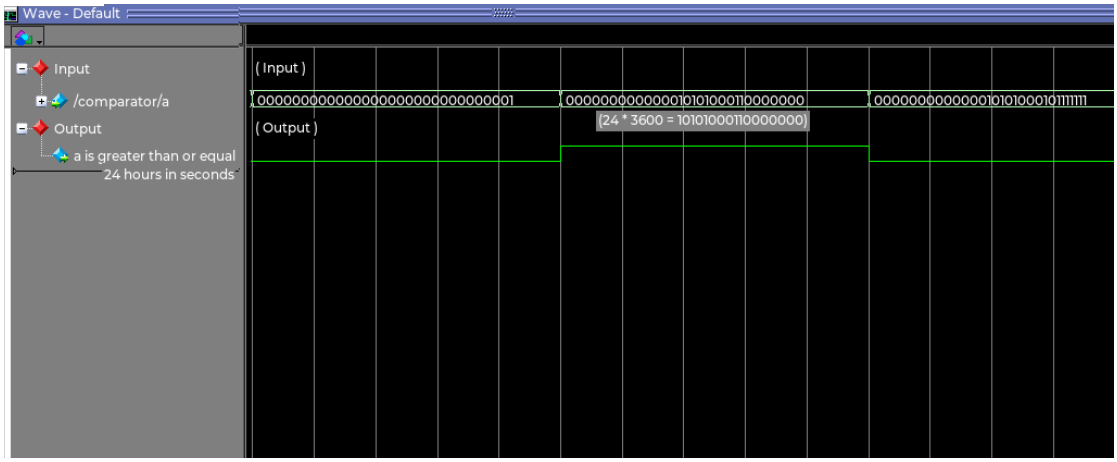


Figure 77: The simulation results for the comparator module used in the parsed_clock functional unit.

2.10 SquareWaveGenerator Functional Unit

Square Wave Generator generates square wave and it is used to generate tones in the piano in this project. When ps2_code[7..0] is read from the PS/2 Keyboard functional unit, a decoder decodes the data into three-digit binary data. The SquareWaveGenerator block receives the data and converts it into the square wave with specific Hz that indicates notes in the piano.

- **Inputs:** The square wave generator schematic has three inputs, clk_50MHz, ps2_code[7..0], en. The clk_50MHz is the clock with a frequency of 50MHz, ps2_code[7..0] is the 8-bit data sent from PS/2 keyboard, and en is the switch that allows the SquareWaveGenerator block to send a data to output signal sqWave.
- **Outputs:** The square wave generator schematic has one output, sqWave. It shows the square wave with a specific Hz to generate the tone in the piano.

2.10.1 SquareWaveGenerator Component Block

The square wave generator block converts eight types of 3-bit data from 0b000 to 0b111 into a square wave. In the SystemVerilog file, the internal counter div_cnt is compared to max_count which indicates the period taken to reach specific hz. If div_cnt reaches max_count, tmp_clk changes to make a square wave. The en signal determines whether the tmp_clk is sent to speaker or not. To make it clear, max_count for middle C is 95419 because the period for 262Hz is 3816973ns. Considering that the clock has the frequency of 50MHz, the signal is read in every 20ns and the square wave should be changed after it passes half of the period. Therefore, after calculating 3816973/20/2=95419. The max_count for other notes is also derived from this calculation. The notes from the middle C to the C with next octave have the frequency of 262 Hz, 294 Hz, 330 Hz, 350 Hz, 392 Hz, 440 Hz, 494 Hz, and 523 Hz in sequence. This means that each note has the period of 381673 ns, 3401360 ns, 3030303 ns, 2857142 ns, 2551020 ns, 2272727 ns, 2024291 ns, and 1912045 ns respectively. When they are simulated in the period, the square wave changes only once at the half of the period as shown below. The logic symbol follows in **Figure 55**. The simulation results follow in **Figure 56**.

- **Inputs:** The SquareWaveGenerator block has four inputs, clk, note[2..0], en, and reset. The clk is the clock with a frequency of 50MHz, and note[2..0] is the decoded data from the decoder to determine the note to generate. The en and reset performs as a switch for the block.
- **Outputs:** The SquareWaveGenerator block has one input, speaker. It shows the square wave generated from the block.

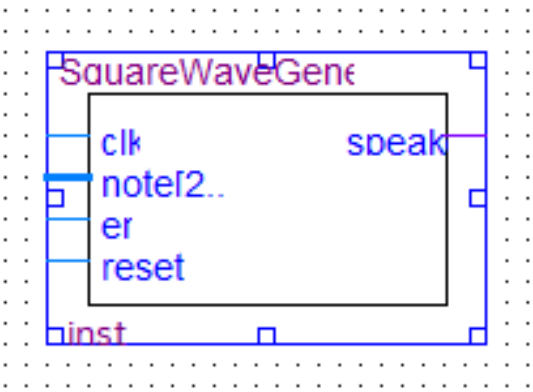


Figure 78: The block symbol of the SquareWaveGenerator module used in the SquareWaveGenerator functional unit.

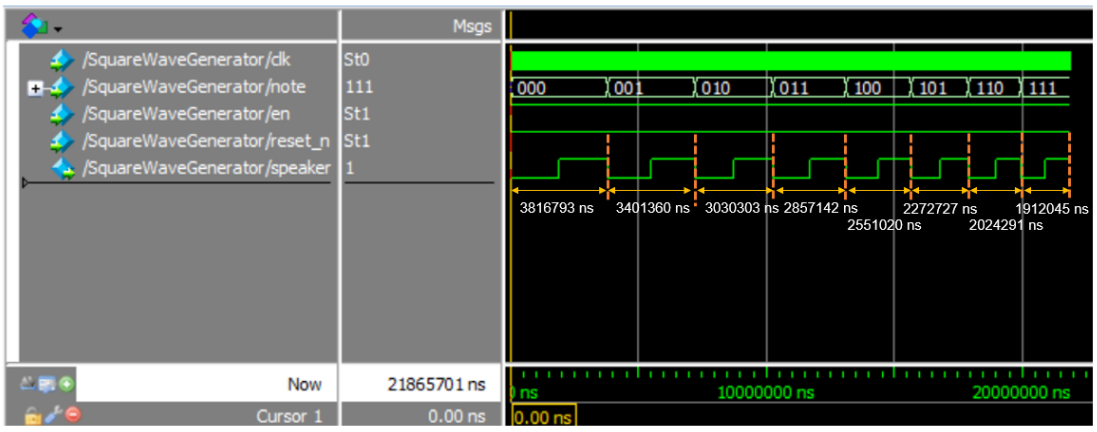


Figure 79: The simulation results for the SquareWaveGenerator module used in the SquareWaveGenerator functional unit.

2.10.2 decoder Component Block

The decoder block decodes the data from the PS/2 keyboard to the three-bit data. To match the keys in keyboard from a to t with the piano notes from middle C to the upper octave C, the hexadecimal code of the keyboard is converted to 3-bit data as shown in the simulation. The logic symbol follows in **Figure 57**. The simulation results follow in **Figure 58**.

- **Inputs:** The decoder block has one input, a[7..0]. It indicates the input 8-bit data, which is the make code for keys on the keyboard.
- **Outputs:** The decoder block has one input, y[2..0]. It is converted from 8-bit to 3-bit data and is used to determine the note to generate.

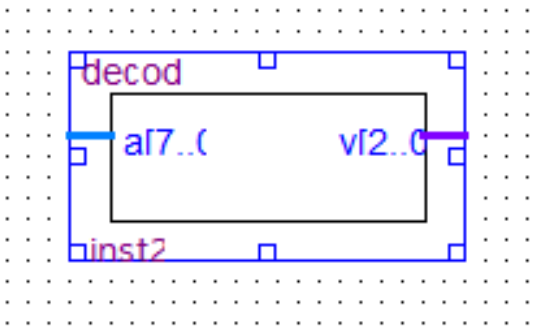


Figure 80: The block symbol of the decoder module used in the SquareWaveGenerator functional unit.

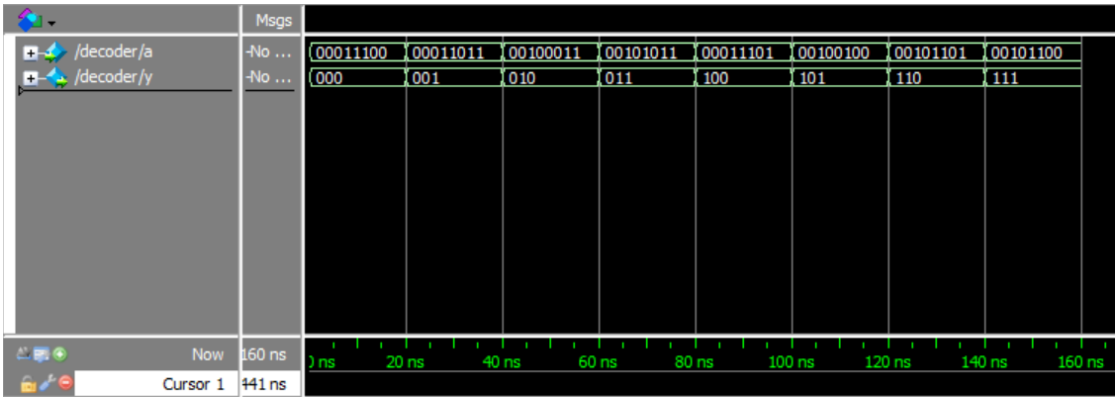


Figure 81: The simulation results for the decoder module used in the SquareWaveGenerator functional unit.

2.11 nes_to_motor Functional Unit

The nes_to_motor module converts nes button states coming from the nes_deccoder module and converts them to the signals used internally in the L293D_encoder module to interface with the L293D DC motor controller. This is a module completely internal to the design that converts between the output and input signal conventions of other modules. The logic symbol follows in **Figure 59**. The simulation results follow in **Figure 60**.

- **Inputs:** Four active-high inputs describing that a system of two motors (like a tank) should turn left (left), turn right (right), drive forward (forward), or drive backward (backward), respectively. In this design, these inputs come from the inverted left, right, up, and down button states of the nes controller.
- **Outputs:** Whether motors 1 (left) and 2 (right) should be spinning (motor_1_on and motor_2_on); and which direction each motor should spin if also enabled, where logic high correlates to clockwise motion and low to counterclockwise (motor_1_dir and motor_2_dir for each motor, respectively).

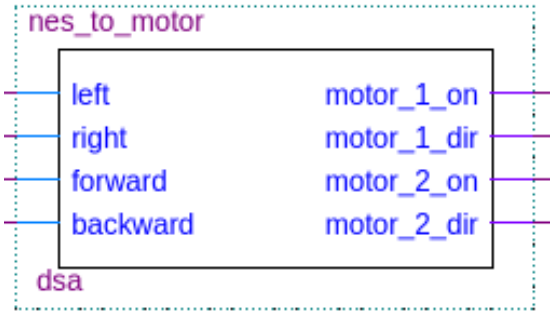


Figure 82: The block symbol of the nes_to_motor functional unit used in the final design.

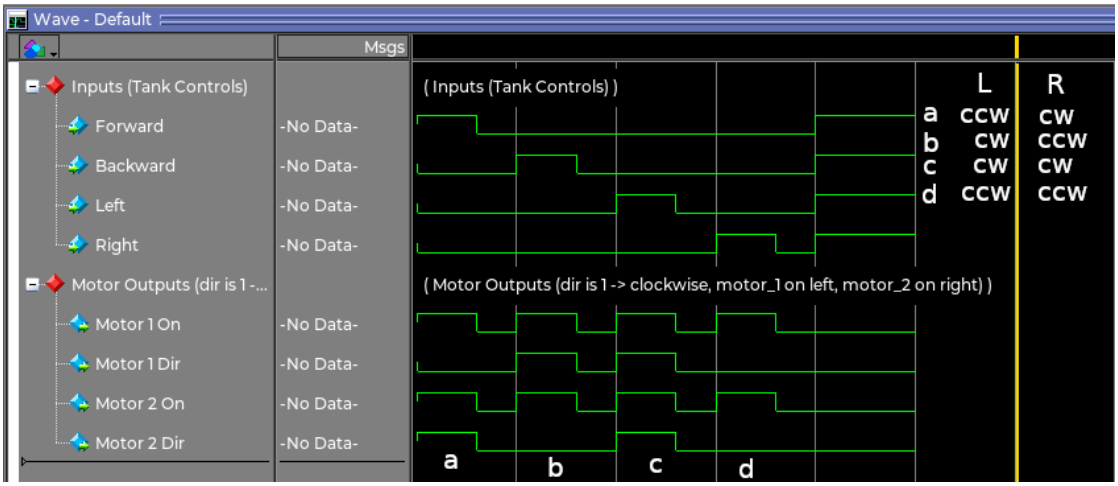


Figure 83: The simulation results for the nes_to_motor module.

2.12 Comparator Functional Unit

The Comparator Functional Unit is used to create a clock that operates at 10 KHz, to drive the vcr_decoder Functional Unit. The FPGA clock is 50 MHz so in order to create a 10 KHz clock, we only want a rising edge every 5000 cycles of the FPGA clock. The logic symbol follows in **Figure 61**. The simulation results follow in **Figure 62**.

- **Inputs:** The Comparator Functional Unit has a single input: a. a is a 23-bit value that is the output of the Counter Functional Unit
- **Outputs:** The Comparator Functional Unit has a single output: eq. eq is one if the input value, a, is equal to 5000.

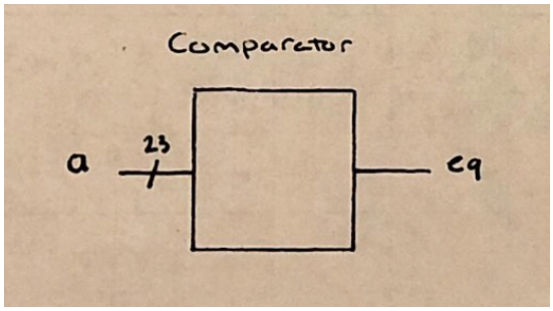


Figure 84: The block symbol of the Comparator functional unit used in the final design.

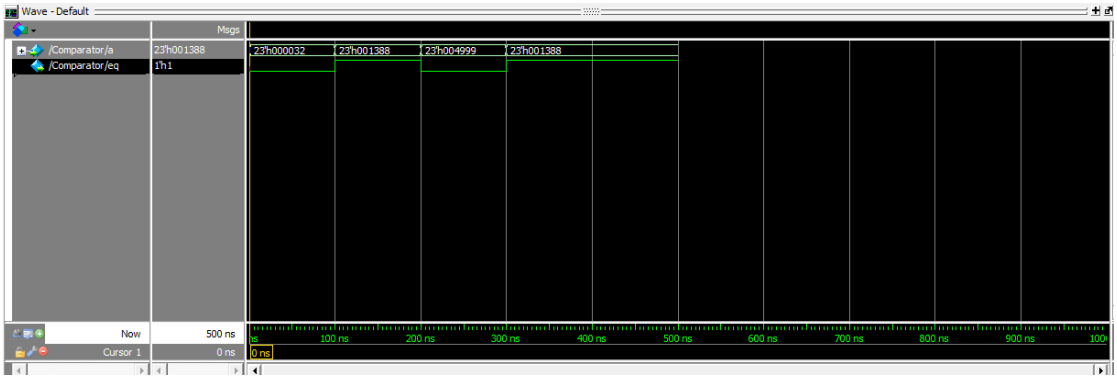


Figure 85: The simulation results for the Comparator module.

2.13 DisplayDecoder Functional Unit

The DisplayDecoder Module converts a 4-bit input value into a display value on the seven segment display of the FPGA. The DisplayDecoder isable to output a hexadecimal display value between 0 and F. A block diagram of the unit follows in **Figure A** and the simulation results for the unit follows in **Figure B**.

- **Inputs:** The DisplayDecoder module takes a 4-bit binary value input, data, as its only input.
- **Outputs:** The DisplayDecoder module outputs a 7-bit binary value that is used to activate specific segments in the FPGA seven segment display.

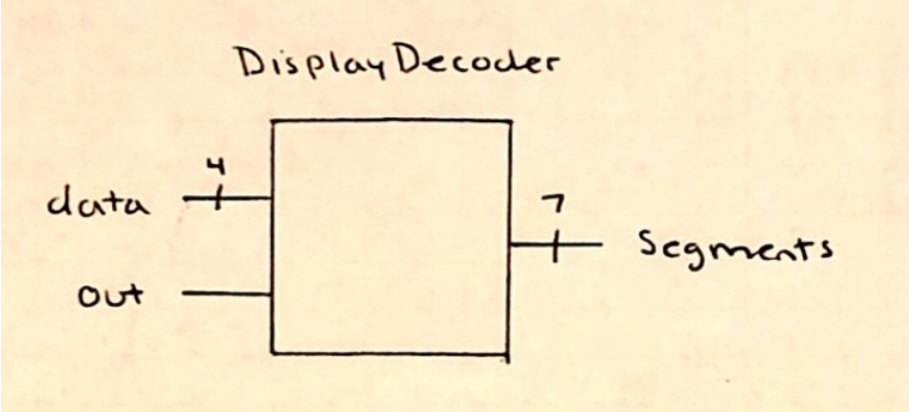


Figure 86: The logic design of the DisplayDecoder functional unit used in the final design.

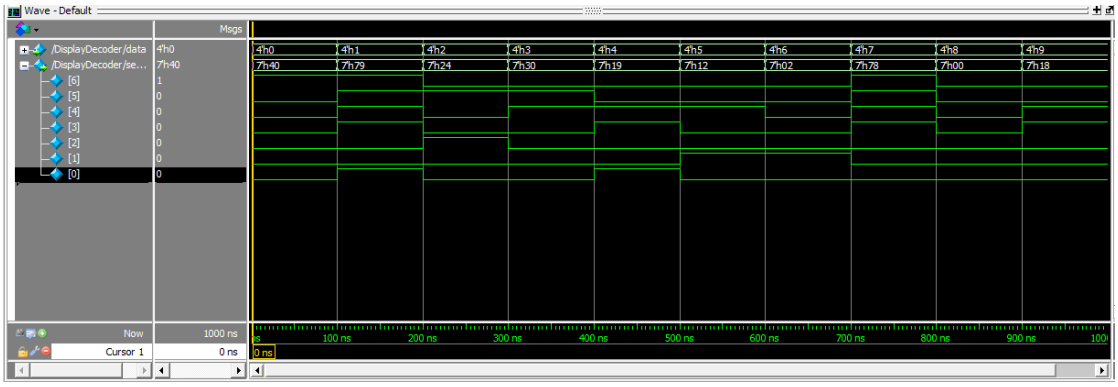


Figure 87: The simulation results for the DisplayDecoder Individual Unit.

A SystemVerilog Files

This appendix will list the SystemVerilog code used for each block used in the design project.

A.1 decoder

decoder.sv

```
module decoder(input logic[7:0]a,
               output logic [2:0] y);

   always_comb
      case(a)
         8'h1C: y = 3'b000;
         8'h1B: y = 3'b001;
         8'h23: y = 3'b010;
         8'h2B: y = 3'b011;
         8'h1D: y = 3'b100;
         8'h24: y = 3'b101;
         8'h2D: y = 3'b110;
         8'h2C: y = 3'b111;
         default: y = 3'bxxx;

      endcase

endmodule
```

A.2 seven_seg_6

A.2.1 Component Blocks of seven_seg_6 Unit

sevenseg.sv

```
module sevenseg(input   logic [3:0] data,
                output  logic [6:0] segments);

   always_comb
      case(data)
         //                      gfe_dcba
         0:       segments =  7'b100_0000;
         1:       segments =  7'b111_1001;
         2:       segments =  7'b010_0100;
         3:       segments =  7'b011_0000;
         4:       segments =  7'b001_1001;
         5:       segments =  7'b001_0010;
         6:       segments =  7'b000_0010;
         7:       segments =  7'b111_1000;
         8:       segments =  7'b000_0000;
         9:       segments =  7'b001_1000;
         default: segments =  7'b111_1111;
      endcase

endmodule
```

A.3 L293D_encoder

L293D_encoder.sv

```

/* Takes input as 4-bit register describing whether
   each motor is on and its direction */

module L293D_encoder (
   input logic    motor_1_on,    // whether the first motor should spin
                  motor_1_dir,   // the direction of spin (0 is back, 1 is forward)

                  motor_2_on,    // whether the second motor should spin
                  motor_2_dir,   // the direction of spin (0 is back, 1 is forward)

   output logic   enable_1,   // pin 1    -  enable for inputs 1 and 2
                  input_1,    // pin 2    -  input 1 (forward for motor 1)
                  input_2,    // pin 7    -  input 2 (backward for motor 1)

                  enable_2,   // pin 9    -  enable for inputs 3 and 4
                  input_3,    // pin 10   -  input 3 (forward for motor 2)
                  input_4     // pin 15   -  input 4 (backward for motor 2)
   );

   always_comb begin
      enable_1    <=  motor_1_on;
      input_1     <= (motor_1_on &&  (motor_1_dir));
      input_2     <= (motor_1_on && ~(motor_1_dir));

      enable_2    <=  motor_2_on;
      input_3     <= (motor_2_on &&  (motor_2_dir));
      input_4     <= (motor_2_on && ~(motor_2_dir));
   end

endmodule
```

A.4 Counter

Counter.sv

```
module Counter #(parameter n = 23)
               (input logic clk,
               input logic reset,
               output logic [n-1:0] q);

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

A.5 vga_encoder

A.5.1 Component Blocks of vga_encoder Unit

vga_640x480.sv

```

module vga_640x480   (input  logic clock_25, reset_frame,
                      output logic h_sync, v_sync, h_display, v_display);

      // Parameters
      localparam line   = 800;   // complete line
      localparam screen = 525;   // complete screen

```

```
9         localparam h_front_porch   = 16;
10        localparam h_sync_dur      = 96;
11        localparam h_back_porch    = 48;
12        localparam h_active_dur    = 640;
13
14        localparam v_front_porch   = 10;
15        localparam v_sync_dur      = 2;
16        localparam v_back_porch    = 33;
17        localparam v_active_dur    = 480;
18
19        localparam h_sync_start    = h_front_porch; // time h sync starts (after)
20        localparam h_sync_end      = h_front_porch + h_sync_dur; // time h sync ends (after)
21        localparam h_active_start  = h_front_porch + h_sync_dur + h_back_porch;
22
23        localparam v_sync_start    = v_front_porch; // time h sync starts (after)
24        localparam v_sync_end      = v_front_porch + v_sync_dur; // time h sync ends (after)
25        localparam v_active_start  = v_front_porch + v_sync_dur + v_back_porch;
26
27        reg [9:0] h_count; // current column
28        reg [9:0] v_count; // current row
29
30
31        // Sync Signals
32        assign h_sync = ~( (h_count >= h_sync_start) && (h_count < h_sync_end) );
33        assign v_sync = ~( (v_count >= v_sync_start) && (v_count < v_sync_end) );
34
35
36        // Display Signal - when pixels can display
37        assign h_display = h_count >= h_active_start;
38        assign v_display = v_count >= v_active_start;
39
40
41        // Flip flop for incrementing
42        always_ff @(posedge clock_25)
43        begin
44
45           if (clock_25)
46           begin
47              v_count <= (v_count >= screen) ? 0 : ((h_count >= line) ? v_count + 1: v_count)
48              h_count <= (clock_25 && (h_count >= line | v_count >= screen)) ? 0 :
    h_count + 1;
49           end
50
51
52 //        if (clock_25 || reset_frame)
53 //           begin
54 //              if (h_count >= line) // end of line
55 //                 begin
56 //                    h_count = 0;
57 //                    v_count = v_count + 1;
58 //                 end
59 //              else
60 //                 begin
61 //                    h_count = h_count + 1;
62 //                    v_count = v_count;
63 //                 end
64 //
65 //              if (v_count >= screen || reset_frame) // end of screen or reset
66 //                 begin
67 //                    h_count = 0;
68 //                    v_count = 0;
69 //                 end
70 //           end
```

```
71
72          end
73
74  endmodule
```

rgb_decoder.sv

```
1   module rgb_decoder (input  logic [1:0] red_in,
2                       input  logic [1:0] green_in,
3                       input  logic [1:0] blue_in,
4
5                       output logic [3:0] red_out,
6                       output logic [3:0] green_out,
7                       output logic [3:0] blue_out);
8
9       always_comb
10      begin
11         red_out = red_in * 85;
12         green_out = green_in * 85;
13         blue_out = blue_in * 85;
14      end
15
16  endmodule
```

clock_modulator.sv

```
1   module clock_modulator (input  logic clock_in,
2                           output logic clock_out);
3
4          logic count = 0;
5
6          always_ff @(posedge clock_in)
7          begin
8             if (clock_in)     count <= count+1;
9          end
10
11         always_comb
12         begin
13            if (count == 1)   clock_out = 1;
14            else              clock_out = 0;
15         end
16
17  endmodule
```

A.6 vcr_decoder

vcr_decoder.sv

```
//VCR Decoder Module
module vcr_decoder(input logic clk_10KHz, //input clock with 10KHz frequency
                   input logic IR, //input IR signal from VCR remote
                   output logic outputting,
                   output logic[3:0] vcr_out); //output value for vcr signal

    //declare states
    typedef enum logic [2:0] {IDLE, C1, C2, READ, PUSH} statetype;
    statetype state, nextstate;

    //Internal
    logic stateReset = 0; //value to reset state machine to IDLE
    logic readControl = 0; //TRUE when reading a control signal
    logic [8:0]controlCounter = 0; //counter for length of control signal
    logic controlLength = 0; //length of control signal
    logic readyToRead = 0; //set to 1 to activate the ReadState state machine
    logic outputReady = 0; //set to 1 when ReadState is ready to output 32 bit value
    logic [31:0] tempOutput = 0; //gets output from ReadState state machine module
    logic [3:0] temp_vcr_out = 0; //gets output from SignalDecoder module

    logic init = 1; //used to initialize state to IDLE


    //READ State Machine
    /*This module reads each pulse of the IR signal and outputs a 32-bit value when
    32 bits have been read into the shift register*/
    ReadState reader(
                      .clk(clk_10KHz),
                      .IR(IR),
                      .start(readyToRead),
                      .pushOutput(outputReady),
                      .outputValue(tempOutput)
                      );

    //Signal Decoder
    //takes 32-bit value and converts it to a 4 bit value between 0 and 15
    SignalDecoder decoder(
                            .signal(tempOutput),
                            .value(temp_vcr_out)
                         );


    //State register
    always_ff@(posedge clk_10KHz, posedge stateReset)
       begin

          //if reading a control signal, increment counter with the clock to count length of
          if(readControl)
             controlCounter <= controlCounter + 1;
          else
             controlCounter <= 0;

          //Then either reset the state to IDLE or move to the next state
          if(stateReset || init)
             state <= IDLE; //set state to IDLE
          else
             state <= nextstate;

       end

```

```
61
62      //next state logic
63      always_comb
64         begin
65            case(state)
66               IDLE:
67                  begin
68                     init = 0;
69                     outputReady = 0;
70                     readyToRead = 0;
71                     if(!IR) //IR signal goes low for first control signal
72                        nextstate = C1;
73                     else //otherwise just stay in IDLE
74                        nextstate = IDLE;
75
76                     readControl = 0; //Not reading control signal
77                  end
78               C1:
79                  begin
80                     readControl = 1; //start reading the control signal
81                     readyToRead = 0;
82                     if(IR) begin //IR goes high signalling end of first control signal and st
83                        if(controlCounter > 80 && controlCounter < 110) begin //good control s
84                           nextstate = C2; //move to next control state
85                           readControl = 0; //No longer reading control signal
86                        end
87                        else begin
88                           nextstate = IDLE; //if bad control signal, move back to IDLE
89                           readControl = 0; //No longer reading control signal
90                        end
91                     end
92                     else //otherwise just stay in C1
93                        nextstate = C1;
94                  end
95               C2:
96                  begin
97                     readControl = 1; //start reading control signal
98                     readyToRead = 0;
99                     if(!IR) begin //IR signal goes low signalling end of second control signa
100                       if(controlCounter > 35 && controlCounter < 50) begin
101                          nextstate = READ;
102                          readControl = 0;
103                       end
104                       else begin
105                          nextstate = IDLE;
106                          readControl = 0;
107                       end
108                    end
109                    else
110                       nextstate = C2;
111                 end
112              READ:
113                 begin
114                    readyToRead = 1; //activate ReadState state machine module
115                    if(outputReady) //when ready to output move to PUSH state
116                       nextstate = PUSH;
117                    else
118                       nextstate <= READ; //Otherwise just stay in READ
119                 end
120              PUSH:
121                 begin
122                    outputting = 1;
123                    vcr_out = temp_vcr_out; //Set output of vcr_decoder module
```

```
124                end
125             default:
126                begin
127                   nextstate = IDLE; //default state is IDLE
128                end
129          endcase
130       end
131
132 endmodule
```

A.6.1 Component Blocks of vcr_decoder Unit

ReadState.sv

```
1  ////////////////////
2  //READ State Machine//
3  ////////////////////
4  module ReadState(input logic clk,
5                   input logic IR, //VCR signal
6                   input logic start, //input signal to activate the ReadState state machine
7                   output logic pushOutput, //output signal that a full 32 bit value is ready
8                   output logic [31:0] outputValue); //32 bit output value
9
10    //declare states
11    typedef enum logic [2:0] {Reading, Processing, Waiting, IDLE, Done} statetype;
12    statetype state, nextstate;
13
14    //Internal
15    logic readReset = 0; //resets state machine to waiting
16    logic readSignal = 0; //TRUE when reading a logic HIGH
17    logic [8:0]signalCounter = 0; //Measures length of HIGH signal
18    logic value = 0; //value 1 or 0 based on HIGH signal length
19    logic shiftValue = 0; //when TRUE a new bit will be shifted into the shift register
20    logic [8:0]bitCounter = 0; //counts the number of bits being read
21    logic [31:0] tempSignalValue; //will temporarily store the output value
22    logic [31:0] signalValue; //stores shift register values
23    logic init = 1; //used to set the initial state to IDLE
24
25    //Shift Register
26    ShiftRegister shiftReg(
27                            .clk(shiftValue), //shift register clock is shiftValue
28                            .reset(readReset),   //reset shift register
29                            .sin(value), //shifts current signal value in
30                            .q(signalValue) //output is 32 bits value for entire signal
31                            );
32
33
34    //State Register
35    always_ff@(posedge clk, posedge readReset, posedge shiftValue)
36       begin
37
38          if(start) begin
39
40             //if reading signal, increment clock
41             if(readSignal)
42                signalCounter <= signalCounter + 1;
43             else
44                signalCounter <= 0;
45
46             //If bit is being added shift register, increment the bit counter
47             if(shiftValue)
48                bitCounter <= bitCounter + 1;
49
50             //Then either reset, or move to the next state
```

```
51              if(readReset || init)
52                  state <= IDLE;
53              else
54                  state <= nextstate;
55          end
56
57      end
58
59      //Next State Logic
60      always_comb
61          begin
62              case(state)
63                  IDLE: //Initial state where no IR is being read
64                      begin
65                          init = 0;
66                          pushOutput = 0;
67                          readSignal = 0;
68                          readReset = 0;
69                          shiftValue = 0;
70
71                          if(IR) //when IR goes high, switch to reading state
72                              nextstate <= Reading;
73                          else
74                              nextstate <= IDLE;
75                      end
76                  Waiting: //Low signal between processing and Reading
77                      begin
78                          shiftValue = 0;
79                          readReset = 0; //while waiting, not resetting read signal
80                          readSignal = 0; //while waiting, not reading signal
81
82                          if(bitCounter == 64)
83                              nextstate <= Done;
84                          else if(IR)
85                              nextstate <= Reading;
86                          else
87                              nextstate <= Waiting;
88                      end
89                  Reading:
90                      begin
91                          readSignal = 1; //counting signal length
92
93                          if(!IR) //when IR signal goes low again, move to processing state
94                              nextstate <= Processing;
95                          else if(IR)
96                              nextstate <= Reading;
97                          else
98                              nextstate <= IDLE;
99                      end
100                 Processing:
101                     begin
102                         if(signalCounter <= 18 && signalCounter > 12) begin //logic 1
103                             value = 1; //store value
104                             shiftValue = 1; //shift into register
105                             nextstate <= Waiting;
106                         end
107                         else if(signalCounter <= 9 && signalCounter >= 1) begin //logic 0
108                             value = 0; //store value
109                             shiftValue = 1; //shift into register
110                             nextstate <= Waiting;
111                         end
112                         else begin
113                             readReset = 1;
```

```
                    nextstate <= IDLE;
                end
            end
        Done:
            begin
                tempSignalValue = signalValue;
                outputValue = tempSignalValue;
                pushOutput = 1;
            end
        endcase
    end

endmodule
```

ShiftRegister.sv

```
//////////////////////////
//Shift Register Module//
//////////////////////////
module ShiftRegister #(parameter N=32)
                      (input logic clk,
                      input logic reset, load,
                      input logic sin,
                      input logic [N-1:0]d,
                      output logic [N-1:0]q,
                      output logic sout);

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

   assign sout = q[N-1];

endmodule
```

mux2.sv

```
// mux2 module

module mux2 #(parameter IN_WIDTH = 7)                    // Width of each input option

             (input  logic  [1:0] [IN_WIDTH-1:0] in,     // 2, (IN_WIDTH-1)-bit inputs (sel
              input  logic  sel,                         // input sel that selects between

              output logic    [IN_WIDTH-1:0] out);       // (IN_WIDTH-1)-bit output based

   // Whenever an input or select changes, reassign output
   always @ (in or sel) begin

         out <= in[sel];

   end

endmodule
```

SignalDecoder.sv

```
//////////////////////////////////
///// Signal Decoder //////////////
//////////////////////////////////
module SignalDecoder(input logic [31:0]signal,
                      output logic [3:0] value);
   always_comb
      case(signal)
         32'h916E926D : value = 0;
```

```
9           32'h916E02FD : value = 1;
10          32'h916E827D : value = 2;
11          32'h916E62BD : value = 3;
12          32'h916EC23D : value = 4;
13          32'h916E22DD : value = 5;
14          32'h916EA25D : value = 6;
15          32'h916E629D : value = 7;
16          32'h916EE21D : value = 8;
17          32'h916E12ED : value = 9;
18          default : value = 15;
19      endcase
20  endmodule
```

A.7 ps2_keyboard

ps2_keyboard.sv

```
module ps2_keyboard  #(parameter clk_freq = 50_000_000,
                       debounce_counter_size = 8)
                       (input logic clk,     //system clock
                          input logic ps2_clk,     //clock signal from PS/2 keyboard
                          input logic ps2_data,    //data signal from PS/2 keyboard
                          output logic ps2_code_new,        //flag that new PS/2 code is a
                          output logic [7:0] ps2_code);       // code received from PS/2

      logic [1:0] sync_ffs;    //synchronizer flip-flops for PS/2 signals
      logic ps2_clk_int;       //debounced clock signal from PS/2 keyboard
      logic ps2_data_int;      //debounced data signal from PS/2 keyboard
      logic [10:0] ps2_word;   //stores the ps2 data word
      logic error;            //validate parity, start, and stop bits
      logic [2778:0] count_idle;   //counter to determine PS/2 is idle

      always_ff @(posedge clk)
         begin
            sync_ffs[0] <= ps2_clk;
            sync_ffs[1] <= ps2_data;
         end

      //input clock
      //input signal to be debounced
      //debounced signal
      always_ff @(posedge clk)
         begin
            ps2_clk_int <= sync_ffs[0];
            ps2_data_int <= sync_ffs[1];
         end


      //input PS2 data
      always_ff @(negedge ps2_clk_int)
         begin
            ps2_word <= {ps2_data_int, ps2_word[10:1]};
         end

      //verify that parity, start, and stop bits are all correct
      assign error = ~ (~ps2_word[0] && ps2_word[10] && (ps2_word[9] ^ ps2_word[8] ^
        ps2_word[7] ^ ps2_word[6] ^ ps2_word[5] ^ ps2_word[4] ^ ps2_word[3] ^
        ps2_word[2] ^ ps2_word[1]));

      //determine if PS2 port is idle (i.e. last transaction is finished) and output result
      always_ff @ (posedge clk)
         begin
            if(ps2_clk_int == 0) //low PS2 clock, PS/2 is active
               count_idle <= 2779'b0;  //reset idle counter
            else //PS2 clock has been high less than a half clock period (<55us)
               count_idle <= count_idle + 2779'b1; //continue counting

            if(count_idle == 2750 && error == 0) //idle threshold reached and no errors det
               begin
                  ps2_code_new <= 1'b1 ; //set flag that new PS/2 code is available
                  ps2_code <= ps2_word[8:1]; //output new PS/2 code
               end
            else  //PS/2 port active or error detected
               ps2_code_new <= 1'b0;   //set flag that PS/2 transaction is in progress
         end

endmodule
```

A.7.1 Component Blocks of ps2_keyboard Unit

debounce.sv

```
module debounce #(counter_size = 19)
                 (input logic clk,     //input clock
                 input logic button,  //input signal to be debounced
                 output logic result);  //debounced signal

   logic [1:0] flipflops;   //input flip flops
   logic counter_set;
   logic [0:counter_size]counter_out = 0; //counter output

   always_ff @(posedge clk)
      begin
         flipflops[0] <= button;
         flipflops[1] <= flipflops[0];
         counter_set <= (flipflops[0] ^ flipflops[1]); //sync reset to zero
         if (counter_set == 1) counter_out <= 0;
         else if (counter_out[counter_size] == 1) counter_out <= counter_out + 1'b1;
         else result <= flipflops[1];
      end
endmodule
```

A.8 nes_decoder

nes_decoder.sv

```
1
2  module nes_decoder (input  logic     nes_data,    // the current button data coming from cont
3                                       in_clock,    // will synchronize output to the positive
4                                       read_data,   // will read the data on the positive edge
5                                       reset,       // does nothing so far
6
7                      output logic     nes_latch,   // sent to controller, begins process of re
8                                       nes_clock,   // sent to controller, tells controller to
9
10                     output logic     nes_A,       // A button output, active low
11                                      nes_B,       // B button output, active low
12                                      nes_START,   // START button output, active low
13                                      nes_SELECT,  // SELECT button output, active low
14                                      nes_UP,      // UP button output, active low
15                                      nes_DOWN,    // DOWN button output, active low
16                                      nes_LEFT,    // LEFT button output, active low
17                                      nes_RIGHT,   // RIGHT button output, active low
18
19                     output logic     ready_to_read); // output telling system whether the mo
20
21
22       // Parameter
23       localparam                 NUM_BUTTONS = 8;
24
25       // Internal
26       logic  [3:0]               count = 3'b0000;
27       logic                      pause = 0;
28       logic                      next = 0;
29       logic                      apply = 0;
30       logic  [NUM_BUTTONS-1:0]   tmp_buttons = {NUM_BUTTONS{1'b1}};
31       logic  [NUM_BUTTONS-1:0]   prev_input  = {NUM_BUTTONS{1'b1}};
32       logic                      tmp_ready   = 0;
33
34       // Start reading process if latch, read until all buttons read, apply after
35       always_ff @(posedge in_clock, posedge read_data, posedge reset) begin
36
37             if (reset) begin
38                /* On reset, set to default values */
39
40                   tmp_ready      = 1;
41                   nes_latch      = 0;
42                   nes_clock      = 0;
43                   tmp_buttons    = {NUM_BUTTONS{1'b1}};
44                   prev_input     = {NUM_BUTTONS{1'b1}};
45                   next           = 0;
46                   apply          = 0;
47                   count          = 0;
48                   tmp_ready      = 1;
49
50             end
51             else if (read_data) begin
52                /* On start (read_data) if the system is ready to read (ready_to_read),
53                 * send nes_latch signal to controller to start reading process, signal syst
54                 * is no longer ready to read, and set that it should read the next value. *
55
56                   if (tmp_ready) begin
57                      nes_latch      =  1;
58                      tmp_ready      =  0;
59                      count          =  0;
60                      tmp_buttons    =  {NUM_BUTTONS{1'b1}};
```

```
61                      next           =  1;
62                  end
63
64              end
65              else if (pause) begin
66                 /* In between each read (after each [next] case where a button is read),
67                  * either send the system to the apply state or to the next button read
68                  * depending on if we have read all buttons; gives nes_clock time to be 0. *
69
70                      next        = (count < NUM_BUTTONS);
71                      nes_clock   = (count < NUM_BUTTONS);
72                      apply       = ~(count < NUM_BUTTONS);
73                      pause       = 0;
74
75              end
76              else if (next) begin
77                 /* Read the next button and set its place in the tmp_button reg. */
78
79                      // Set pulsed items to 0
80                      nes_latch   = 0;
81                      nes_clock   = 0;
82
83                      // Read data into current button
84                      tmp_buttons[count] = nes_data;
85
86                      // Trigger next execution logic (pause)
87                      pause = 1;
88
89                      // Increment count
90                      count = (count < NUM_BUTTONS) ? count + 1 : 0;
91
92              end
93              else if (apply) begin
94                 /* Apply state, send out the buttons read so far by the decoder
95                  * to prev_input to be sent to the outputs. */
96
97                      prev_input     = tmp_buttons;
98                      tmp_buttons    = {NUM_BUTTONS{1'b1}};
99                      apply          = 0;
100                     tmp_ready      = 1;
101
102             end
103
104         end
105
106         always_comb begin
107
108                     nes_A          = prev_input[0];
109                     nes_B          = prev_input[1];
110                     nes_START      = prev_input[2];
111                     nes_SELECT     = prev_input[3];
112                     nes_UP         = prev_input[4];
113                     nes_DOWN       = prev_input[5];
114                     nes_LEFT       = prev_input[6];
115                     nes_RIGHT      = prev_input[7];
116                     ready_to_read  = tmp_ready;
117
118         end
119
120 endmodule
```

A.9 parsed_clock

A.9.1 Component Blocks of parsed_clock Unit

parser.sv

```
module parser #(parameter IN_BITS=30, TICK_PER_SEC=9537)
               (input logic [IN_BITS-1:0] count,
                output logic [IN_BITS-1:0] total_seconds,
                output logic [3:0] seconds_ones,
                output logic [3:0] seconds_tens,
                output logic [3:0] minutes_ones,
                output logic [3:0] minutes_tens,
                output logic [3:0] hours_ones,
                output logic [3:0] hours_tens);

   reg   [5:0]          seconds;
   reg   [5:0]          minutes;
   reg   [4:0]          hours;

   always_comb
      begin
         total_seconds = count / TICK_PER_SEC;

         seconds = total_seconds % 60;
         seconds_ones = seconds % 10;
         seconds_tens = seconds / 10;

         minutes = (total_seconds / 60) % 60;
         minutes_ones = minutes % 10;
         minutes_tens = minutes / 10;

         hours = (total_seconds / 3600) % 24;
         hours_ones = hours % 10;
         hours_tens = hours / 10;
      end

endmodule
```

comparator.sv

```
module comparator #(parameter N=30)
                   (input  logic [N-1:0] a, b,
                    output logic eq, neq, lt, lte, gt, gte, agt24h);

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

   assign agt24h =   (a >= 24*3600);

endmodule
```

mux4.sv

```
// mux4 module

module mux4 #(parameter IN_WIDTH = 1)                    // Width of each input option

             (input  logic    [3:0] [IN_WIDTH-1:0] in,   // 4, 4-bit inputs (select 00, 01,
              input  logic  [1:0] sel,                   // input sel that selects between

              output logic    [IN_WIDTH-1:0] out);       // 4-bit output based on input sel

```

```
10      // Whenever an input or select changes, reassign output
11      always @ (in or sel) begin
12
13            out <= in[sel];
14
15      end
16
17  endmodule
```

counter.sv

```
1   module counter #(parameter N=30)
2                   (input  logic clk, reset, enable,
3                    output logic [N-1:0]  q);
4
5      always_ff@(posedge clk, posedge reset)
6         begin
7            if(reset)          q <= 0;
8            else if(enable)    q <= q+1;
9         end
10
11  endmodule
```

A.10 SquareWaveGenerator

A.10.1 Component Blocks of SquareWaveGenerator Unit

decoder.sv

```
module decoder(input logic[7:0]a,
               output logic [2:0] y);

   always_comb
      case(a)
         8'h1C: y = 3'b000;
         8'h1B: y = 3'b001;
         8'h23: y = 3'b010;
         8'h2B: y = 3'b011;
         8'h1D: y = 3'b100;
         8'h24: y = 3'b101;
         8'h2D: y = 3'b110;
         8'h2C: y = 3'b111;
         default: y = 3'bxxx;

      endcase

endmodule
```

SquareWaveGenerator.sv

```
module SquareWaveGenerator(input logic clk,
                           input logic [2:0] note,
                           input logic en,
                           input logic reset_n,
                           output logic speaker);

   logic tmp_clk;

   logic [20:0] div_cnt, max_count;
   /*initial begin
      tmp_clk=0;
      div_cnt=0;
   end */

   //select correct frequency based on input note
   always_comb
      begin

         case (note)
            3'b000: max_count = 95419; //C(middle)
            3'b001: max_count = 85034; //D
            3'b010: max_count = 75757; //E
            3'b011: max_count = 71428; //F
            3'b100: max_count = 63775; //G
            3'b101: max_count = 56818; //A
            3'b110: max_count = 50607; //B
            3'b111: max_count = 47800; //C
         endcase
      end

   always_ff @ (posedge(clk), negedge(reset_n), posedge(en))
      begin
         if (!reset_n)
            begin
               tmp_clk <= 0;
               div_cnt <= 0;
            end
         else
            begin
```

```
40                 if (div_cnt >= max_count)
41                    begin
42                       tmp_clk <= ~tmp_clk;
43                       div_cnt <= 21'b0;
44                    end
45                 else
46                    div_cnt <= div_cnt + 1;
47              end
48           if (en) speaker <= tmp_clk;
49           else speaker <= 0;
50        end
51
52
53  endmodule
```

A.11 nes_to_motor

nes_to_motor.sv

```
1
2  module nes_to_motor (input logic    forward,
3                                      backward,
4                                      left,
5                                      right,
6
7                       output logic   motor_1_on,
8                                      motor_1_dir,
9                                      motor_2_on,
10                                     motor_2_dir);
11
12     // Internal (NO DOUBLE INPUTS)
13     logic int_forward    = 0;
14     logic int_backward   = 0;
15     logic int_left       = 0;
16     logic int_right      = 0;
17
18     always_comb begin
19
20        int_forward    = forward   && (forward ^ backward ^ left ^ right);
21        int_backward   = backward  && (forward ^ backward ^ left ^ right);
22        int_left       = left      && (forward ^ backward ^ left ^ right);
23        int_right      = right     && (forward ^ backward ^ left ^ right);
24
25        motor_1_on  =  int_left || int_right || int_forward || int_backward;
26        motor_1_dir =  int_backward || int_left;
27
28        motor_2_on  =  int_left || int_right || int_forward || int_backward;
29        motor_2_dir =  int_forward || int_left;
30
31     end
32
33 endmodule
```

A.12 Comparator

Comparator.sv

```
1  module Comparator #(parameter n = 23, m = 5000)
2                     (input logic [n-1:0] a,
3                     output logic eq);
4
5
6
7     assign eq = (a == m);
8  endmodule
```

A.13 DisplayDecoder

DisplayDecoder.sv

```
1  //Seven Segment Display Decoder
2  module DisplayDecoder(input logic [3:0] data,
3                        input logic out,
4                        output logic [6:0]segments);
5     always_comb
6        if(out) begin
7           case(data)      //   abc_defg
8              0: segments = 7'b100_0000;
9              1: segments = 7'b111_1001;
```

```
10              2: segments = 7'b010_0100;
11              3: segments = 7'b011_0000;
12              4: segments = 7'b001_1001;
13              5: segments = 7'b001_0010;
14              6: segments = 7'b000_0010;
15              7: segments = 7'b111_1000;
16              8: segments = 7'b000_0000;
17              9: segments = 7'b001_1000;
18              default: segments = 7'b111_1111;
19          endcase
20      end
21      else
22          segments = 7'b111_1111;
23  endmodule
```

B Simulation Files (Do Scripts)

This appendix will list the Do Scripts used to simulate each block used in the design project.

B.1 decoder

decoder_sim.do

```
vsim work.decoder

add wave a
add wave y

restart -f
force a 00011100
run 20 ns

force a 00011011
run 20 ns

force a 00100011
run 20 ns

force a 00101011
run 20 ns

force a 00011101
run 20 ns

force a 00100100
run 20 ns

force a 00101101
run 20 ns

force a 00101100
run 20 ns
```

B.2 seven_seg_6

seven_seg_6.do

```
vsim design-project.seven_seg_6
add wave -position insertpoint sim:/seven_seg_6/*

force in0 10#0 25, 10#1 50, 10#2 75, 10#3 100, 10#4 125, 10#5 150, 10#6 175, 10#7 200
force in1 10#0 25, 10#1 50, 10#2 75, 10#3 100, 10#4 125, 10#5 150, 10#6 175, 10#7 200
force in2 10#0 25, 10#1 50, 10#2 75, 10#3 100, 10#4 125, 10#5 150, 10#6 175, 10#7 200
force in3 10#0 25, 10#1 50, 10#2 75, 10#3 100, 10#4 125, 10#5 150, 10#6 175, 10#7 200
force in4 10#0 25, 10#1 50, 10#2 75, 10#3 100, 10#4 125, 10#5 150, 10#6 175, 10#7 200
force in5 10#0 25, 10#1 50, 10#2 75, 10#3 100, 10#4 125, 10#5 150, 10#6 175, 10#7 200

run 275
```

B.2.1 Component Blocks of seven_seg_6 Unit

sevenseg.do

```
vsim design-project.sevenseg
add wave -position insertpoint sim:/sevenseg/*
```

```
force -freeze sim:/sevenseg/data 0000 0
run 50

force -freeze sim:/sevenseg/data 0001 0
run 50

force -freeze sim:/sevenseg/data 0010 0
run 50

force -freeze sim:/sevenseg/data 0011 0
run 50

force -freeze sim:/sevenseg/data 0100 0
run 50

force -freeze sim:/sevenseg/data 0101 0
run 50

force -freeze sim:/sevenseg/data 0110 0
run 50

force -freeze sim:/sevenseg/data 0111 0
run 50

force -freeze sim:/sevenseg/data 1000 0
run 50

force -freeze sim:/sevenseg/data 1001 0
run 50
```

B.3 L293D_encoder

L293D_encoder.do

```
vsim design-project.L293D_encoder

add wave -position insertpoint sim:/L293D_encoder/motor_1_on
add wave -position insertpoint sim:/L293D_encoder/motor_2_on

add wave -position insertpoint sim:/L293D_encoder/motor_1_dir
add wave -position insertpoint sim:/L293D_encoder/motor_2_dir

add wave -position insertpoint sim:/L293D_encoder/enable_1
add wave -position insertpoint sim:/L293D_encoder/input_1
add wave -position insertpoint sim:/L293D_encoder/input_2

add wave -position insertpoint sim:/L293D_encoder/enable_2
add wave -position insertpoint sim:/L293D_encoder/input_3
add wave -position insertpoint sim:/L293D_encoder/input_4

force -freeze motor_1_on    0 0, 1 100, 0 {300}
force -freeze motor_2_on    0 0, 1 100, 0 {300}
force -freeze motor_1_dir   0 0, 1 {200}
force -freeze motor_2_dir   0 0, 1 {200}

run 400
```

B.4 Counter

CounterTest.do

```
vsim work.Counter

add wave clk
add wave reset
add wave q

force clk 1 0, 0 50 {50 ns} -r 100
force reset 1 0, 0 10, 1 1000

run 1200
```

B.5 vga_encoder

B.5.1 Component Blocks of vga_encoder Unit

vga_640x480_sim.do

```
vsim work.vga_640x480

add wave clock_25
add wave reset_frame

add wave h_sync
add wave v_sync
add wave h_display
add wave v_display

restart -f
force clock_25 0 0, 1 {20ns} -r {40ns}
```

```
force reset_frame 0
run 10ns

force reset_frame 1
run 38400ns
```

clock_modulator_sim.do

```
vsim work.clock_modulator

add wave clock_in
add wave clock_out

restart -f
force clock_in 0 0, 1 {10 ns} -r 20ns
run 100ns
```

vga_640x480_sim_Vsync.do

```
vsim work.vga_640x480

add wave clock_25
add wave reset_frame

add wave h_sync
add wave v_sync
add wave h_display
add wave v_display

restart -f
force clock_25 0 0, 1 {20ns} -r {40ns}
force reset_frame 0
run 10ns

force reset_frame 1
run 18240000ns
```

color_mux_sim.do

```
vsim work.color_mux

add wave hdisplay
add wave vdisplay
add wave display_in
add wave blue_in
add wave green_in
add wave red_in

add wave blue_out
add wave green_out
add wave red_out

restart -f
force hdisplay 0
force vdisplay 0
force display_in 0

force blue_in 11
force green_in 11
force red_in 11
run 10ns
```

```
force hdisplay 1
run 10ns

force vdisplay 1
run 10ns

force hdisplay 0
run 10ns

force display_in 1
force hdisplay 0
force vdisplay 0
run 10ns

force hdisplay 1
run 10ns
force vdisplay 1
run 10ns
force hdisplay 0
run 10ns
```

rgb_decoder_sim.do

```
vsim work.rgb_decoder

add wave red_in
add wave green_in
add wave blue_in

add wave red_out
add wave green_out
add wave blue_out

restart -f
force red_in 00
force green_in 01
force blue_in 10
run 20ns

force red_in 01
force green_in 10
force blue_in 11
run 20ns

force red_in 00
force green_in 00
force blue_in 00
run 20ns

force red_in 11
force green_in 11
force blue_in 11
run 20ns
```

B.6 vcr_decoder

vcr_decoder_Sim_4.do

```
vsim work.vcr_decoder

add wave clk_10KHz
add wave IR
add wave readControl
add wave readyToRead
add wave outputReadyadd wave state
add wave nextstate
add wave vcr_out

force clk 1 0, 0 {50 ns} -r 100
force stateReset 1 0, 0 100

force IR 1 0, 0 16901.0, 1 26010.0, 0 30236.0, 1 30982.999999999996, 0 32494.99999999

run 90000
```

vcr_decoder_Sim_0.do

```
vsim work.vcr_decoder

add wave clk_10KHz
add wave IR
add wave readControl
add wave readyToRead
add wave outputReadyadd wave state
add wave nextstate
add wave vcr_out

force clk 1 0, 0 {50 ns} -r 100
force stateReset 1 0, 0 100

force IR 1 0, 0 16901.0, 1 26001.999999999996, 0 30257.0, 1 31009.999999999996, 0 325

run 90000
```

vcr_decoder_Sim_3.do

```
vsim work.vcr_decoder

add wave clk_10KHz
add wave IR
add wave readControl
add wave readyToRead
add wave outputReadyadd wave state
add wave nextstate
add wave vcr_out

force clk 1 0, 0 {50 ns} -r 100
force stateReset 1 0, 0 100

force IR 1 0, 0 16901.0, 1 25972.0, 0 30258.0, 1 31024.999999999996, 0 32520.99999999

run 90000
```

vcr_decoder_Sim_1.do

```
vsim work.vcr_decoder
```

```
add wave clk_10KHz
add wave IR
add wave readControl
add wave readyToRead
add wave outputReadyadd wave state
add wave nextstate
add wave vcr_out

force clk 1 0, 0 {50 ns} -r 100
force stateReset 1 0, 0 100

force IR 1 0, 0 16901.0, 1 26005.999999999996, 0 30261.999999999996, 1 31042.99999999

run 90000
```

vcr_decoder_Sim_9.do

```
vsim work.vcr_decoder

add wave clk_10KHz
add wave IR
add wave readControl
add wave readyToRead
add wave outputReadyadd wave state
add wave nextstate
add wave vcr_out

force clk 1 0, 0 {50 ns} -r 100
force stateReset 1 0, 0 100

force IR 1 0, 0 16901.0, 1 25989.0, 0 30258.0, 1 30994.0, 0 32501.000000000004, 1 332

run 90000
```

vcr_decoder_Sim_5.do

```
vsim work.vcr_decoder

add wave clk_10KHz
add wave IR
add wave readControl
add wave readyToRead
add wave outputReadyadd wave state
add wave nextstate
add wave vcr_out

force clk 1 0, 0 {50 ns} -r 100
force stateReset 1 0, 0 100

force IR 1 0, 0 16901.0, 1 25979.0, 0 30260.999999999996, 1 31010.999999999996, 0 325

run 90000
```

B.6.1 Component Blocks of vcr_decoder Unit

ReadState_Sim_2.do

```
vsim work.ReadState

add wave clk
add wave IR
```

```
add wave readSignal
add wave shiftValue
add wave state
add wave nextstate
add wave outputValue

force clk 1 0, 0 {50 ns} -r 100
force start 1 0

force IR 0 1, 1 30982.999999999996, 0 32519.0, 1 33262.0, 0 33666.0, 1 34408.99999999

run 90000
```

ReadState_Sim_3.do

```
vsim work.ReadState

add wave clk
add wave IR
add wave readSignal
add wave shiftValue
add wave state
add wave nextstate
add wave outputValue

force clk 1 0, 0 {50 ns} -r 100
force start 1 0

force IR 0 1, 1 31024.999999999996, 0 32520.999999999993, 1 33242.99999999999, 0 3366

run 90000
```

ReadState_Sim_9.do

```
vsim work.ReadState

add wave clk
add wave IR
add wave readSignal
add wave shiftValue
add wave state
add wave nextstate
add wave outputValue

force clk 1 0, 0 {50 ns} -r 100
force start 1 0

force IR 0 1, 1 33269.0, 0 33664.0, 1 34434.99999999999, 0 34810.99999999999, 1 35578

run 90000
```

shiftRegisterTest.do

```
vsim work.ShiftRegister

add wave clk
add wave sin
add wave q

force clk 1 0, 0 {50 ns} -r 100
force sin 1 0, 0 500, 1 1000
```

```
run 1500
```

ReadState_Sim_8.do

```
vsim work.ReadState

add wave clk
add wave IR
add wave readSignal
add wave shiftValue
add wave state
add wave nextstate
add wave outputValue

force clk 1 0, 0 {50 ns} -r 100
force start 1 0

force IR 0 1, 1 31016.0, 0 32518.0, 1 33296.0, 0 33664.0, 1 34456.0, 0 34807.0, 1 355

run 90000
```

ReadState_Sim_7.do

```
vsim work.ReadState

add wave clk
add wave IR
add wave readSignal
add wave shiftValue
add wave state
add wave nextstate
add wave outputValue

force clk 1 0, 0 {50 ns} -r 100
force start 1 0

force IR 0 1, 1 31017.999999999996, 0 32500.0, 1 33308.0, 0 33664.0, 1 34433.0, 0 348

run 90000
```

ReadState_Sim_0.do

```
vsim work.ReadState

add wave clk
add wave IR
add wave readSignal
add wave shiftValue
add wave state
add wave nextstate
add wave outputValue

force clk 1 0, 0 {50 ns} -r 100
force start 1 0

force IR 0 1, 1 31009.999999999996, 0 32500.0, 1 33239.0, 0 33663.0, 1 34417.0, 0 348

run 90000
```

ReadState_Sim_4.do

```
vsim work.ReadState
```

```
add wave clk
add wave IR
add wave readSignal
add wave shiftValue
add wave state
add wave nextstate
add wave outputValue

force clk 1 0, 0 {50 ns} -r 100
force start 1 0

force IR 0 1, 1 30982.999999999996, 0 32494.999999999996, 1 33248.0, 0 33663.0, 1 344

run 90000
```

ReadState_Sim_5.do

```
vsim work.ReadState

add wave clk
add wave IR
add wave readSignal
add wave shiftValue
add wave state
add wave nextstate
add wave outputValue

force clk 1 0, 0 {50 ns} -r 100
force start 1 0

force IR 0 1, 1 31010.999999999996, 0 32520.0, 1 33271.99999999999, 0 33667.0, 1 3441

run 90000
```

signalDecoder.do

```
vsim work.SignalDecoder

add wave signal
add wave value

force signal 916E926D 0, 916E02FD 100, 916E827D 200, 916E62BD 300, 916EC23D 400, 916E2

run 1000
```

ReadState_Sim_1.do

```
vsim work.ReadState

add wave clk
add wave IR
add wave readSignal
add wave shiftValue
add wave state
add wave nextstate
add wave outputValue

force clk 1 0, 0 {50 ns} -r 100
force start 1 0

force IR 0 1, 1 31042.999999999996, 0 32500.0, 1 33287.0, 0 33668.0, 1 34448.0, 0 348
```

```
run 90000
   ReadState_Sim_6.do
vsim work.ReadState

add wave clk
add wave IR
add wave readSignal
add wave shiftValue
add wave state
add wave nextstate
add wave outputValue

force clk 1 0, 0 {50 ns} -r 100
force start 1 0

force IR 0 1, 1 31009.999999999996, 0 32520.999999999993, 1 33319.0, 0 33668.0, 1 344

run 90000
   vcr_decoder_Sim_7.do
vsim work.vcr_decoder

add wave clk_10KHz
add wave IR
add wave readControl
add wave readyToRead
add wave outputReadyadd wave state
add wave nextstate
add wave vcr_out

force clk 1 0, 0 {50 ns} -r 100
force stateReset 1 0, 0 100

force IR 1 0, 0 16901.0, 1 26005.999999999996, 0 30237.0, 1 31017.999999999996, 0 325

run 90000
   vcr_decoder_Sim_6.do
vsim work.vcr_decoder

add wave clk_10KHz
add wave IR
add wave readControl
add wave readyToRead
add wave outputReadyadd wave state
add wave nextstate
add wave vcr_out

force clk 1 0, 0 {50 ns} -r 100
force stateReset 1 0, 0 100

force IR 1 0, 0 16899.9999999962, 1 25987.999999999996, 0 30239.999999999996, 1 31009

run 90000
   vcr_decoder_Sim_8.do
vsim work.vcr_decoder
```

```
add wave clk_10KHz
add wave IR
add wave readControl
add wave readyToRead
add wave outputReadyadd wave state
add wave nextstate
add wave vcr_out

force clk 1 0, 0 {50 ns} -r 100
force stateReset 1 0, 0 100

force IR 1 0, 0 16901.0, 1 26019.999999999996, 0 30258.0, 1 31016.0, 0 32518.0, 1 332

run 90000
```

vcr_decoder_Sim_2.do

```
vsim work.vcr_decoder

add wave clk_10KHz
add wave IR
add wave readControl
add wave readyToRead
add wave outputReadyadd wave state
add wave nextstate
add wave vcr_out

force clk 1 0, 0 {50 ns} -r 100
force stateReset 1 0, 0 100

force IR 1 0, 0 16901.0, 1 25990.0, 0 30239.0, 1 30982.999999999996, 0 32519.0, 1 332

run 90000
```

B.7 ps2_keyboard

ps2_keyboard_sim_3.do

```
vsim work.ps2_keyboard
# Do file for E

add wave ps2_keyboard/clk
add wave ps2_keyboard/ps2_clk
add wave ps2_keyboard/ps2_data

add wave ps2_keyboard/ps2_code_new
add wave ps2_keyboard/ps2_code

restart -f
force ps2_keyboard/ps2_clk 1 @ 0
force ps2_keyboard/ps2_clk 0 @ {615us}
force ps2_keyboard/ps2_clk 1 @ {659us}
force ps2_keyboard/ps2_clk 0 @ {698us}
force ps2_keyboard/ps2_clk 1 @ {742us}
force ps2_keyboard/ps2_clk 0 @ {782us}
force ps2_keyboard/ps2_clk 1 @ {826us}
force ps2_keyboard/ps2_clk 0 @ {865us}
force ps2_keyboard/ps2_clk 1 @ {910us}
force ps2_keyboard/ps2_clk 0 @ {949us}
force ps2_keyboard/ps2_clk 1 @ {993us}
force ps2_keyboard/ps2_clk 0 @ {1033us}
force ps2_keyboard/ps2_clk 1 @ {1077us}
force ps2_keyboard/ps2_clk 0 @ {1117us}
force ps2_keyboard/ps2_clk 1 @ {1161us}
force ps2_keyboard/ps2_clk 0 @ {1200us}
force ps2_keyboard/ps2_clk 1 @ {1244us}
force ps2_keyboard/ps2_clk 0 @ {1284us}
force ps2_keyboard/ps2_clk 1 @ {1328us}
force ps2_keyboard/ps2_clk 0 @ {1367us}

force ps2_keyboard/ps2_clk 1 @ {1411us}
force ps2_keyboard/ps2_clk 0 @ {1451us}
force ps2_keyboard/ps2_clk 1 @ {1495us}

force -freeze sim:ps2_keyboard/clk 0 0, 1 {10 ns} -r 20ns

force ps2_keyboard/ps2_data 1 @ 0
force ps2_keyboard/ps2_data 0 @ {595 us}
force ps2_keyboard/ps2_data 1 @ {847 us}
force ps2_keyboard/ps2_data 0 @ {930 us}
force ps2_keyboard/ps2_data 1 @ {1099 us}
force ps2_keyboard/ps2_data 0 @ {1181 us}
force ps2_keyboard/ps2_data 1 @ {1350 us}

run 2000us
```

ps2_keyboard_sim.do

```
vsim work.ps2_keyboard
# Do file for A

add wave ps2_keyboard/clk
add wave ps2_keyboard/ps2_clk
add wave ps2_keyboard/ps2_data
```

```
add wave ps2_keyboard/ps2_code_new
add wave ps2_keyboard/ps2_code

restart -f
force ps2_keyboard/ps2_clk 1 @ 0
force ps2_keyboard/ps2_clk 0 @ {615us}
force ps2_keyboard/ps2_clk 1 @ {659us}
force ps2_keyboard/ps2_clk 0 @ {698us}
force ps2_keyboard/ps2_clk 1 @ {742us}
force ps2_keyboard/ps2_clk 0 @ {782us}
force ps2_keyboard/ps2_clk 1 @ {826us}
force ps2_keyboard/ps2_clk 0 @ {865us}
force ps2_keyboard/ps2_clk 1 @ {910us}
force ps2_keyboard/ps2_clk 0 @ {949us}
force ps2_keyboard/ps2_clk 1 @ {993us}
force ps2_keyboard/ps2_clk 0 @ {1033us}
force ps2_keyboard/ps2_clk 1 @ {1077us}
force ps2_keyboard/ps2_clk 0 @ {1117us}
force ps2_keyboard/ps2_clk 1 @ {1161us}
force ps2_keyboard/ps2_clk 0 @ {1200us}
force ps2_keyboard/ps2_clk 1 @ {1244us}
force ps2_keyboard/ps2_clk 0 @ {1284us}
force ps2_keyboard/ps2_clk 1 @ {1328us}
force ps2_keyboard/ps2_clk 0 @ {1367us}

force ps2_keyboard/ps2_clk 1 @ {1411us}
force ps2_keyboard/ps2_clk 0 @ {1451us}
force ps2_keyboard/ps2_clk 1 @ {1495us}

force -freeze sim:ps2_keyboard/clk 0 0, 1 {10 ns} -r 20ns

force ps2_keyboard/ps2_data 1 @ 0
force ps2_keyboard/ps2_data 0 @ {595 us}
force ps2_keyboard/ps2_data 1 @ {847 us}
force ps2_keyboard/ps2_data 0 @ {1097 us}
force ps2_keyboard/ps2_data 1 @ {1439 us}

run 2000us
```

B.7.1 Component Blocks of ps2_keyboard Unit

debounce_sim.do

```
vsim work.debounce

add wave clk
add wave button
add wave result

add wave flipflops

restart -f
force clk 0 0, 1 {10 ns} -r 20ns
force button 0
run 60 ns

force button 1
run 100 ns
```

```
force button 0
run 50ns
force button 1
run 50ns
```

ps2_keyboard_sim_5.do

```
vsim work.ps2_keyboard
# Do file for R

add wave ps2_keyboard/clk
add wave ps2_keyboard/ps2_clk
add wave ps2_keyboard/ps2_data

add wave ps2_keyboard/ps2_code_new
add wave ps2_keyboard/ps2_code

restart -f
force ps2_keyboard/ps2_clk 1 @ 0
force ps2_keyboard/ps2_clk 0 @ {615us}
force ps2_keyboard/ps2_clk 1 @ {659us}
force ps2_keyboard/ps2_clk 0 @ {698us}
force ps2_keyboard/ps2_clk 1 @ {742us}
force ps2_keyboard/ps2_clk 0 @ {782us}
force ps2_keyboard/ps2_clk 1 @ {826us}
force ps2_keyboard/ps2_clk 0 @ {865us}
force ps2_keyboard/ps2_clk 1 @ {910us}
force ps2_keyboard/ps2_clk 0 @ {949us}
force ps2_keyboard/ps2_clk 1 @ {993us}
force ps2_keyboard/ps2_clk 0 @ {1033us}
force ps2_keyboard/ps2_clk 1 @ {1077us}
force ps2_keyboard/ps2_clk 0 @ {1117us}
force ps2_keyboard/ps2_clk 1 @ {1161us}
force ps2_keyboard/ps2_clk 0 @ {1200us}
force ps2_keyboard/ps2_clk 1 @ {1244us}
force ps2_keyboard/ps2_clk 0 @ {1284us}
force ps2_keyboard/ps2_clk 1 @ {1328us}
force ps2_keyboard/ps2_clk 0 @ {1367us}

force ps2_keyboard/ps2_clk 1 @ {1411us}
force ps2_keyboard/ps2_clk 0 @ {1451us}
force ps2_keyboard/ps2_clk 1 @ {1495us}

force -freeze sim:ps2_keyboard/clk 0 0, 1 {10 ns} -r 20ns

force ps2_keyboard/ps2_data 1 @ 0
force ps2_keyboard/ps2_data 0 @ {595 us}
force ps2_keyboard/ps2_data 1 @ {680 us}
force ps2_keyboard/ps2_data 0 @ {763 us}
force ps2_keyboard/ps2_data 1 @ {848 us}
force ps2_keyboard/ps2_data 0 @ {1014 us}
force ps2_keyboard/ps2_data 1 @ {1099 us}
force ps2_keyboard/ps2_data 0 @ {1181 us}
force ps2_keyboard/ps2_data 1 @ {1350 us}

run 2000us
```

ps2_keyboard_sim_2.do

```
vsim work.ps2_keyboard
# Do file for D
```

```
add wave ps2_keyboard/clk
add wave ps2_keyboard/ps2_clk
add wave ps2_keyboard/ps2_data

add wave ps2_keyboard/ps2_code_new
add wave ps2_keyboard/ps2_code

restart -f
force ps2_keyboard/ps2_clk 1 @ 0
force ps2_keyboard/ps2_clk 0 @ {615us}
force ps2_keyboard/ps2_clk 1 @ {659us}
force ps2_keyboard/ps2_clk 0 @ {698us}
force ps2_keyboard/ps2_clk 1 @ {742us}
force ps2_keyboard/ps2_clk 0 @ {782us}
force ps2_keyboard/ps2_clk 1 @ {826us}
force ps2_keyboard/ps2_clk 0 @ {865us}
force ps2_keyboard/ps2_clk 1 @ {910us}
force ps2_keyboard/ps2_clk 0 @ {949us}
force ps2_keyboard/ps2_clk 1 @ {993us}
force ps2_keyboard/ps2_clk 0 @ {1033us}
force ps2_keyboard/ps2_clk 1 @ {1077us}
force ps2_keyboard/ps2_clk 0 @ {1117us}
force ps2_keyboard/ps2_clk 1 @ {1161us}
force ps2_keyboard/ps2_clk 0 @ {1200us}
force ps2_keyboard/ps2_clk 1 @ {1244us}
force ps2_keyboard/ps2_clk 0 @ {1284us}
force ps2_keyboard/ps2_clk 1 @ {1328us}
force ps2_keyboard/ps2_clk 0 @ {1367us}

force ps2_keyboard/ps2_clk 1 @ {1411us}
force ps2_keyboard/ps2_clk 0 @ {1451us}
force ps2_keyboard/ps2_clk 1 @ {1495us}

force -freeze sim:ps2_keyboard/clk 0 0, 1 {10 ns} -r 20ns

force ps2_keyboard/ps2_data 1 @ 0
force ps2_keyboard/ps2_data 0 @ {595 us}
force ps2_keyboard/ps2_data 1 @ {680 us}
force ps2_keyboard/ps2_data 0 @ {846 us}
force ps2_keyboard/ps2_data 1 @ {1097 us}
force ps2_keyboard/ps2_data 0 @ {1181 us}
force ps2_keyboard/ps2_data 1 @ {1432 us}

run 2000us
```

ps2_keyboard_sim_7.do

```
vsim work.ps2_keyboard
# Do file for T

add wave ps2_keyboard/clk
add wave ps2_keyboard/ps2_clk
add wave ps2_keyboard/ps2_data

add wave ps2_keyboard/ps2_code_new
add wave ps2_keyboard/ps2_code

restart -f
force ps2_keyboard/ps2_clk 1 @ 0
```

```
force ps2_keyboard/ps2_clk 0 @ {615us}
force ps2_keyboard/ps2_clk 1 @ {659us}
force ps2_keyboard/ps2_clk 0 @ {698us}
force ps2_keyboard/ps2_clk 1 @ {742us}
force ps2_keyboard/ps2_clk 0 @ {782us}
force ps2_keyboard/ps2_clk 1 @ {826us}
force ps2_keyboard/ps2_clk 0 @ {865us}
force ps2_keyboard/ps2_clk 1 @ {910us}
force ps2_keyboard/ps2_clk 0 @ {949us}
force ps2_keyboard/ps2_clk 1 @ {993us}
force ps2_keyboard/ps2_clk 0 @ {1033us}
force ps2_keyboard/ps2_clk 1 @ {1077us}
force ps2_keyboard/ps2_clk 0 @ {1117us}
force ps2_keyboard/ps2_clk 1 @ {1161us}
force ps2_keyboard/ps2_clk 0 @ {1200us}
force ps2_keyboard/ps2_clk 1 @ {1244us}
force ps2_keyboard/ps2_clk 0 @ {1284us}
force ps2_keyboard/ps2_clk 1 @ {1328us}
force ps2_keyboard/ps2_clk 0 @ {1367us}

force ps2_keyboard/ps2_clk 1 @ {1411us}
force ps2_keyboard/ps2_clk 0 @ {1451us}
force ps2_keyboard/ps2_clk 1 @ {1495us}

force -freeze sim:ps2_keyboard/clk 0 0, 1 {10 ns} -r 20ns

force ps2_keyboard/ps2_data 1 @ 0
force ps2_keyboard/ps2_data 0 @ {595 us}
force ps2_keyboard/ps2_data 1 @ {847 us}
force ps2_keyboard/ps2_data 0 @ {1013 us}
force ps2_keyboard/ps2_data 1 @ {1097 us}
force ps2_keyboard/ps2_data 0 @ {1180 us}
force ps2_keyboard/ps2_data 1 @ {1430 us}

run 2000us
```

ps2_keyboard_sim_6.do

```
vsim work.ps2_keyboard
# Do file for S

add wave ps2_keyboard/clk
add wave ps2_keyboard/ps2_clk
add wave ps2_keyboard/ps2_data

add wave ps2_keyboard/ps2_code_new
add wave ps2_keyboard/ps2_code

restart -f
force ps2_keyboard/ps2_clk 1 @ 0
force ps2_keyboard/ps2_clk 0 @ {615us}
force ps2_keyboard/ps2_clk 1 @ {659us}
force ps2_keyboard/ps2_clk 0 @ {698us}
force ps2_keyboard/ps2_clk 1 @ {742us}
force ps2_keyboard/ps2_clk 0 @ {782us}
force ps2_keyboard/ps2_clk 1 @ {826us}
force ps2_keyboard/ps2_clk 0 @ {865us}
force ps2_keyboard/ps2_clk 1 @ {910us}
force ps2_keyboard/ps2_clk 0 @ {949us}
force ps2_keyboard/ps2_clk 1 @ {993us}
```

```
force ps2_keyboard/ps2_clk 0 @ {1033us}
force ps2_keyboard/ps2_clk 1 @ {1077us}
force ps2_keyboard/ps2_clk 0 @ {1117us}
force ps2_keyboard/ps2_clk 1 @ {1161us}
force ps2_keyboard/ps2_clk 0 @ {1200us}
force ps2_keyboard/ps2_clk 1 @ {1244us}
force ps2_keyboard/ps2_clk 0 @ {1284us}
force ps2_keyboard/ps2_clk 1 @ {1328us}
force ps2_keyboard/ps2_clk 0 @ {1367us}

force ps2_keyboard/ps2_clk 1 @ {1411us}
force ps2_keyboard/ps2_clk 0 @ {1451us}
force ps2_keyboard/ps2_clk 1 @ {1495us}

force -freeze sim:ps2_keyboard/clk 0 0, 1 {10 ns} -r 20ns

force ps2_keyboard/ps2_data 1 @ 0
force ps2_keyboard/ps2_data 0 @ {595 us}
force ps2_keyboard/ps2_data 1 @ {680 us}
force ps2_keyboard/ps2_data 0 @ {846 us}
force ps2_keyboard/ps2_data 1 @ {929 us}
force ps2_keyboard/ps2_data 0 @ {1097 us}
force ps2_keyboard/ps2_data 1 @ {1350 us}

run 2000us
```

ps2_keyboard_sim_4.do

```
vsim work.ps2_keyboard
# Do file for F

add wave ps2_keyboard/clk
add wave ps2_keyboard/ps2_clk
add wave ps2_keyboard/ps2_data

add wave ps2_keyboard/ps2_code_new
add wave ps2_keyboard/ps2_code

restart -f
force ps2_keyboard/ps2_clk 1 @ 0
force ps2_keyboard/ps2_clk 0 @ {615us}
force ps2_keyboard/ps2_clk 1 @ {659us}
force ps2_keyboard/ps2_clk 0 @ {698us}
force ps2_keyboard/ps2_clk 1 @ {742us}
force ps2_keyboard/ps2_clk 0 @ {782us}
force ps2_keyboard/ps2_clk 1 @ {826us}
force ps2_keyboard/ps2_clk 0 @ {865us}
force ps2_keyboard/ps2_clk 1 @ {910us}
force ps2_keyboard/ps2_clk 0 @ {949us}
force ps2_keyboard/ps2_clk 1 @ {993us}
force ps2_keyboard/ps2_clk 0 @ {1033us}
force ps2_keyboard/ps2_clk 1 @ {1077us}
force ps2_keyboard/ps2_clk 0 @ {1117us}
force ps2_keyboard/ps2_clk 1 @ {1161us}
force ps2_keyboard/ps2_clk 0 @ {1200us}
force ps2_keyboard/ps2_clk 1 @ {1244us}
force ps2_keyboard/ps2_clk 0 @ {1284us}
force ps2_keyboard/ps2_clk 1 @ {1328us}
force ps2_keyboard/ps2_clk 0 @ {1367us}
```

```
force ps2_keyboard/ps2_clk 1 @ {1411us}
force ps2_keyboard/ps2_clk 0 @ {1451us}
force ps2_keyboard/ps2_clk 1 @ {1495us}

force -freeze sim:ps2_keyboard/clk 0 0, 1 {10 ns} -r 20ns

force ps2_keyboard/ps2_data 1 @ 0
force ps2_keyboard/ps2_data 0 @ {595 us}
force ps2_keyboard/ps2_data 1 @ {680 us}
force ps2_keyboard/ps2_data 0 @ {846 us}
force ps2_keyboard/ps2_data 1 @ {932 us}
force ps2_keyboard/ps2_data 0 @ {1014 us}
force ps2_keyboard/ps2_data 1 @ {1099 us}
force ps2_keyboard/ps2_data 0 @ {1180 us}
force ps2_keyboard/ps2_data 1 @ {1350 us}

run 2000us
```

ps2_keyboard_sim_9.do

```
vsim work.ps2_keyboard
# Do file for W

add wave ps2_keyboard/clk
add wave ps2_keyboard/ps2_clk
add wave ps2_keyboard/ps2_data

add wave ps2_keyboard/ps2_code_new
add wave ps2_keyboard/ps2_code

restart -f
force ps2_keyboard/ps2_clk 1 @ 0
force ps2_keyboard/ps2_clk 0 @ {615us}
force ps2_keyboard/ps2_clk 1 @ {659us}
force ps2_keyboard/ps2_clk 0 @ {698us}
force ps2_keyboard/ps2_clk 1 @ {742us}
force ps2_keyboard/ps2_clk 0 @ {782us}
force ps2_keyboard/ps2_clk 1 @ {826us}
force ps2_keyboard/ps2_clk 0 @ {865us}
force ps2_keyboard/ps2_clk 1 @ {910us}
force ps2_keyboard/ps2_clk 0 @ {949us}
force ps2_keyboard/ps2_clk 1 @ {993us}
force ps2_keyboard/ps2_clk 0 @ {1033us}
force ps2_keyboard/ps2_clk 1 @ {1077us}
force ps2_keyboard/ps2_clk 0 @ {1117us}
force ps2_keyboard/ps2_clk 1 @ {1161us}
force ps2_keyboard/ps2_clk 0 @ {1200us}
force ps2_keyboard/ps2_clk 1 @ {1244us}
force ps2_keyboard/ps2_clk 0 @ {1284us}
force ps2_keyboard/ps2_clk 1 @ {1328us}
force ps2_keyboard/ps2_clk 0 @ {1367us}

force ps2_keyboard/ps2_clk 1 @ {1411us}
force ps2_keyboard/ps2_clk 0 @ {1451us}
force ps2_keyboard/ps2_clk 1 @ {1495us}

force -freeze sim:ps2_keyboard/clk 0 0, 1 {10 ns} -r 20ns

force ps2_keyboard/ps2_data 1 @ 0
force ps2_keyboard/ps2_data 0 @ {595 us}
```

```
force ps2_keyboard/ps2_data 1 @ {681 us}
force ps2_keyboard/ps2_data 0 @ {763 us}
force ps2_keyboard/ps2_data 1 @ {848 us}
force ps2_keyboard/ps2_data 0 @ {1097 us}
force ps2_keyboard/ps2_data 1 @ {1350 us}

run 2000us
```

B.8 nes_decoder

nes_decoder.do

```
vsim design-project.nes_decoder

add wave -position insertpoint sim:/nes_decoder/in_clock
add wave -position insertpoint sim:/nes_decoder/read_data
add wave -position insertpoint sim:/nes_decoder/reset
add wave -position insertpoint sim:/nes_decoder/nes_*
add wave -position insertpoint sim:/nes_decoder/ready_to_read
add wave -position insertpoint sim:/nes_decoder/count
add wave -position insertpoint sim:/nes_decoder/tmp_buttons
add wave -position insertpoint sim:/nes_decoder/prev_input
add wave -position insertpoint sim:/nes_decoder/pause
add wave -position insertpoint sim:/nes_decoder/next
add wave -position insertpoint sim:/nes_decoder/apply
add wave -position insertpoint sim:/nes_decoder/tmp_ready

force -freeze   in_clock    1 0,    0 {25}    -r 50
force -freeze   nes_data    0 0,    1 250     -r 500
force -freeze   read_data   0 0,    1 10,     0 {20}
force -freeze   reset       1 0,    0 {10}

run 900
```

B.9 parsed_clock

parsed_clock.do

```
vsim design-project.parsed_clock

add wave -position insertpoint sim:/parsed_clock/clock_50MHz
add wave -position insertpoint sim:/parsed_clock/enable_in
add wave -position insertpoint sim:/parsed_clock/reset_in
add wave -position insertpoint sim:/parsed_clock/double_speed_in
add wave -position insertpoint sim:/parsed_clock/super_speed_in

add wave -position insertpoint sim:/parsed_clock/total_sec

add wave -position insertpoint sim:/parsed_clock/seconds_ones
add wave -position insertpoint sim:/parsed_clock/seconds_tens
add wave -position insertpoint sim:/parsed_clock/minutes_ones
add wave -position insertpoint sim:/parsed_clock/minutes_tens
add wave -position insertpoint sim:/parsed_clock/hours_ones
add wave -position insertpoint sim:/parsed_clock/hours_tens

force -freeze clock_50MHz   0 0,        1  1us      -r {2us}
force -freeze enable_in     0 0,        1  1us,      0 {2us}
force -freeze reset_in      1 0,        0 {1us}

force -freeze double_speed_in  1 0, 0 50000000us, 1 75000000us, 0 {100000000us}
force -freeze super_speed_in   1 0, 0 75000000us

run 150000000us
```

B.9.1 Component Blocks of parsed_clock Unit

counter.do

```
vsim design-project.counter
add wave -position insertpoint sim:/counter/*
force -freeze sim:/counter/clk 0 0, 1 {25 ps} -r 50
force -freeze sim:/counter/reset 1 0 -cancel 50
force -freeze sim:/counter/enable 1 0 -cancel 100
run 100
force -freeze sim:/counter/enable 0 0 -cancel 50
run 50
force -freeze sim:/counter/enable 1 0
run 150
```

parser.do

```
vsim design-project.parser
add wave -position insertpoint sim:/parser/*

force -freeze sim:/parser/count 0 0
run 50

force -freeze sim:/parser/count 1010001011111100011 0
run 50

force -freeze sim:/parser/count 10101110101000001011000 0
run 50

force -freeze sim:/parser/count 10000010111110001000010000 0
run 50
```

enable_flip_flop.do

```
vsim design-project.enable_flip_flop

add wave -position insertpoint sim:/enable_flip_flop/enable_in
add wave -position insertpoint sim:/enable_flip_flop/reset
add wave -position insertpoint sim:/enable_flip_flop/enable_out

force -freeze enable_in      0 0,     1  50     -r {100}
force -freeze reset          1 0,     0 {10}

run 400
```

delay.do

```
vsim design-project.delay
add wave -position insertpoint sim:/delay/clock_in
add wave -position insertpoint sim:/delay/enable
add wave -position insertpoint sim:/delay/reset
add wave -position insertpoint sim:/delay/double_speed
add wave -position insertpoint sim:/delay/super_speed
add wave -position insertpoint sim:/delay/select
add wave -position insertpoint sim:/delay/clock_option
add wave -position insertpoint sim:/delay/clock_out

force -freeze sim:/delay/clock_in 0 0, 1 {50 ps} -r 100
force -freeze sim:/delay/enable 1 0
force -freeze sim:/delay/reset 1 0 -cancel 200
force -freeze sim:/delay/double_speed 1 0, 0 25000, 1 50000, 0 {100000}
force -freeze sim:/delay/super_speed  1 0, 0 50000

run 200000
```

mux4.do

```
vsim design-project.mux4
add wave -position insertpoint sim:/mux4/*


force -freeze {in[0]} 0000 {0}
force -freeze {in[1]} 0011 {0}
force -freeze {in[2]} 1100 {0}
force -freeze {in[3]} 1111 {0}

force -freeze sel 00 0, 01 50, 10 100, 11 {150}

run 200
```

comparator.do

```
vsim design-project.comparator
add wave -position insertpoint sim:/comparator/*
force -freeze sim:/comparator/a 0 0
run 100
force -freeze sim:/comparator/a 0010101000110000000 0
run 100
force -freeze sim:/comparator/a 0010101000101111111 0
run 100
```

B.10 SquareWaveGenerator

B.10.1 Component Blocks of SquareWaveGenerator Unit

SquareWaveGenerator_sim.do

```
vsim work.SquareWaveGenerator

add wave clk
add wave note
add wave en
add wave reset_n

add wave speaker

restart -f
force clk 0 0, 1 {10 ns} -r 20ns
force reset_n 0
force en 1
run 20ns

force reset_n 1
force note 000
run 3816793 ns

force note 001
run 3401360 ns

force note 010
run 3030303 ns

force note 011
run 2857142 ns

force note 100
run 2551020 ns

force note 101
run 2272727 ns

force note 110
run 2024291 ns

force note 111
run 1912045 ns
```

decoder_sim.do

```
vsim work.decoder

add wave a
add wave y

restart -f
force a 00011100
run 20 ns

force a 00011011
run 20 ns

force a 00100011
```

```
run 20 ns

force a 00101011
run 20 ns

force a 00011101
run 20 ns

force a 00100100
run 20 ns

force a 00101101
run 20 ns

force a 00101100
run 20 ns
```

B.11 nes_to_motor

nes_to_motor.do

```
vsim design-project.nes_to_motor
add wave -position insertpoint sim:/nes_to_motor/forward
add wave -position insertpoint sim:/nes_to_motor/backward
add wave -position insertpoint sim:/nes_to_motor/left
add wave -position insertpoint sim:/nes_to_motor/right
add wave -position insertpoint sim:/nes_to_motor/motor_1_on
add wave -position insertpoint sim:/nes_to_motor/motor_1_dir
add wave -position insertpoint sim:/nes_to_motor/motor_2_on
add wave -position insertpoint sim:/nes_to_motor/motor_2_dir

force -freeze   forward     1 0, 0 30,          1 {200}
force -freeze   backward    0 0, 1 50,  0 80,   1 {200}
force -freeze   left        0 0, 1 100, 0 130, 1 {200}
force -freeze   right       0 0, 1 150, 0 180, 1 {200}

run 250
```

B.12 Comparator

ComparatorTest.do

```
vsim work.Comparator

add wave a
add wave eq

force a 32 0, 1388 100, 4999 200, 1388 300

run 500
```

B.13 DisplayDecoder

DisplayDecoderTest.do

```
vsim work.DisplayDecoder

add wave data
add wave segments

force data 0 0, 1 100, 2 200, 3 300, 4 400, 5 500, 6 600, 7 700, 8 800, 9 900

run 1000
```