ECE 271: SNES Decoder Design Project
Group 24

WeiHao Kuang, Juwan Forman, Suyang Liu, Yue Fan, Chih-Hsuan Su

June 8, 2018

Contents

1 Project Description

The purpose of this project is to design a decoder using the FPGA to translate signals provided from an 12-bit button board, a PS/2 keyboard, and an IR remote to operate a Super Nintendo Entertainment System (SNES). The signals provided from the chosen 3 sources will be translated to emulate a standard SNES controller.

The SNES is a 16 bit video game console introduced in 1990, adding X and Y buttons next to the original A and B buttons, as well as two shoulder buttons.[1] The increased bit size allows use of up to 16 buttons, however only 12 are utilized. The connector to the controller has 7 pins, however only 5 are used. Figure 1 shows the utilized pins and their connections.[1]

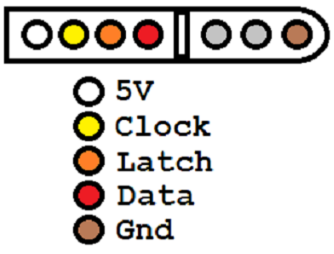


Figure 1: SNES controller pinout[1]

The SNES controller reports the state of all buttons at a frequency of 60Hz. Every frequency cycle, the CPU sends a 12us positive signal over LATCH to instruct the controller to report all button states. After a delay of 6 us, the CPU sends 16 negative pulses with a 6us width and 6us delay over the CLOCK pin. The controller sends each button state serially on the rising edge of each clock pulse over the DATA pin. The SNES reads the serial data on each falling edge of the clock. Because the clock pulse is negative, the first data bit cannot be driven on the rising edge of the clock pulse and instead must be driven earlier on the falling edge of the latch.

The SNES sends a total of 16 pulses per latch cycle, however only 12 are utilized. Each pulse represents the state of a specific button. Table 1 shows the corresponding button reported for each pulse.

Pulse	Button
1	B
2	Y
3	Start
4	Select
5	Up
6	Down
7	Left
8	Right
9	A
10	X
11	L
12	R

Table 1: Button clock pulse assignment

[1]https://gamefaqs.gamespot.com/snes/916396-super-nintendo/faqs/5395

This design will adapt an 8-bit button board, a PS/2 keyboard, or an IR remote to control an SNES. The design will read from a single input at any given time and use a button to switch between all three inputs.

The output for the design will match a standard SNES controller connector and emulate the behavior of an SNES controller.

2 High Level Description

Figure 2 shows the full block diagram.

Inputs: This top level logic block reads a PS2 Keyboard, IR signal, and 8-bit button board inputs with a select feature that can alternate between which input is used.

Outputs: The output of this logic block will be button signals that will be sent to the SNES reader which will register button presses that correspond directly to the buttons on a traditional SNES controller.

This design consists of 6 logic modules that connect the input to the outputs. The IR decoder, keyboard decoder, and button encoder translates all three input sources into the same hex code, allowing an input selector to switch between the three input sources at will.

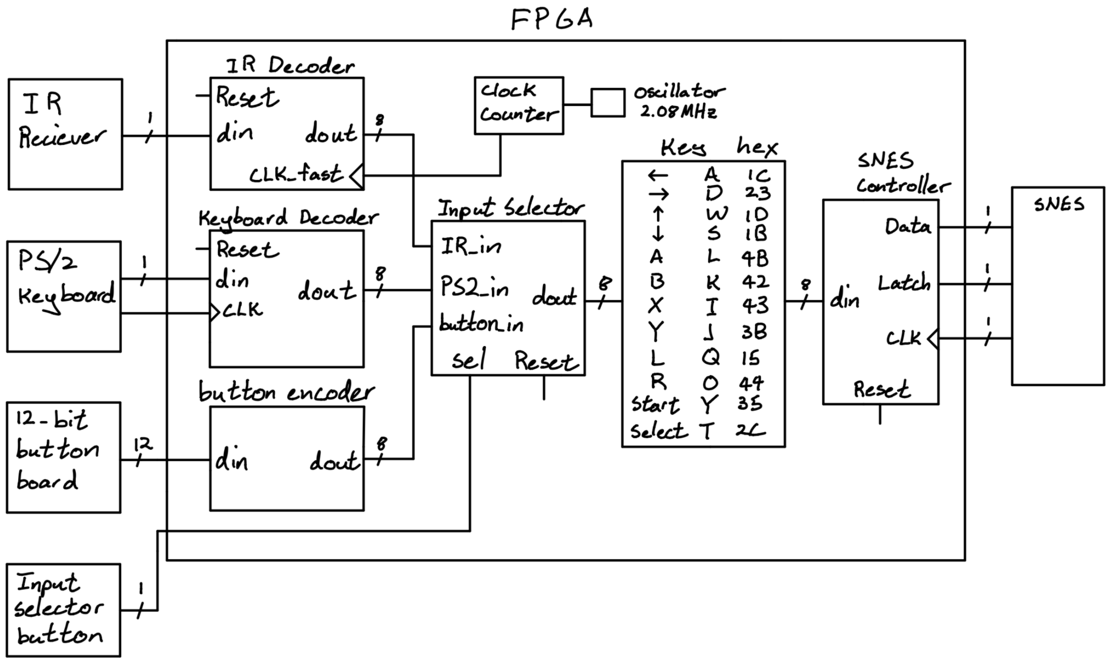


Figure 2: Full SNES controller block diagram

2.1 Hardware Diagram and FPGA Floor Plan

The hardware layout of the FPGA consists of connections to the button board, PS/2 keyboard, and IR controller. Figure 3 shows the hardware layout of the entire setup.

The button board takes the largest amount of connectors since each button requires its own data path. The button board connects 12 button pins to 12 pins on the FPGA. A single pin ties the button board to ground.

The PS/2 keyboard has two connections to the FPGA, as well as a 5V connector and ground. One of the connections to the FPGA is CLOCK, while the other is DATA. With each button press, the PS/2 keyboard sends 8 bit data packages serially over the DATA line in sync with a signal sent over CLOCK.

The IR has a single connection to the FPGA as well as a 3.3V and a GND pin. The IR receiver receives IR signals and passes the data serially over a single bit data line. The rest of the processing and decoding is done over the FPGA.

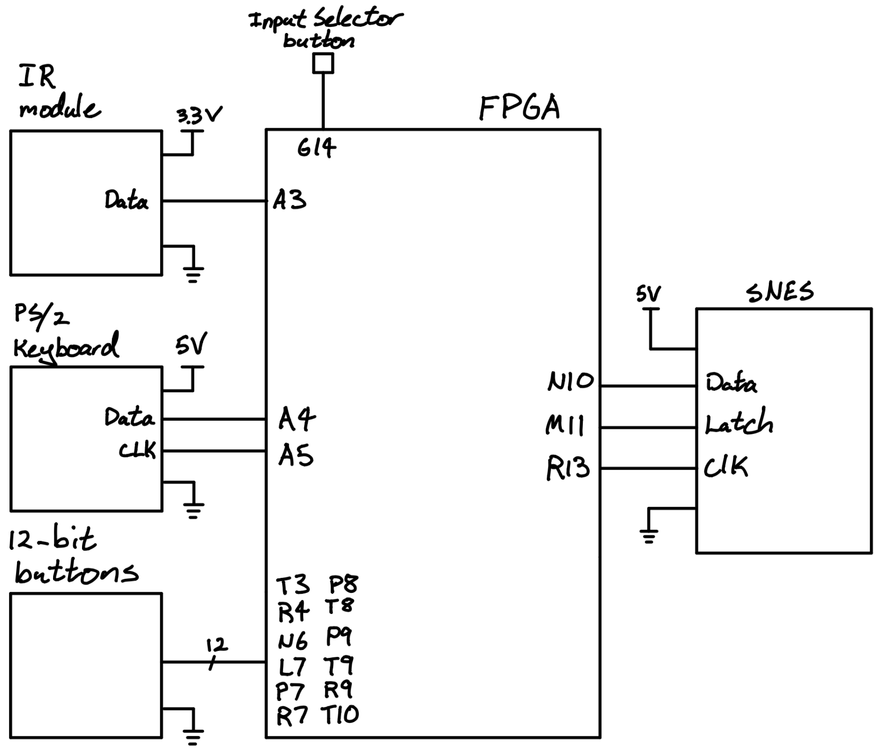


Figure 3: Full Hardware Diagram

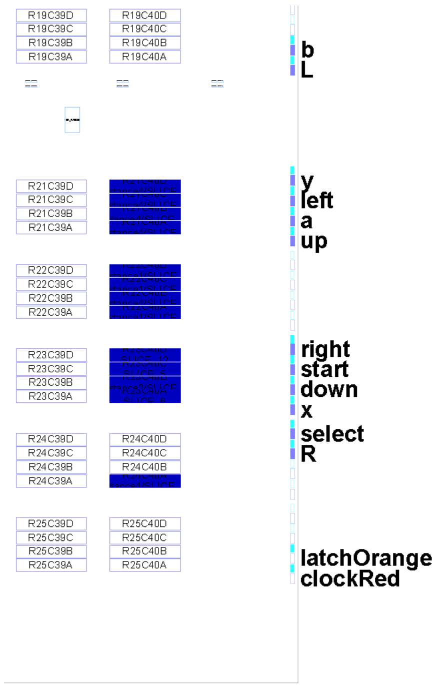


Figure 4: Full Hardware Diagram

2.2 Clock Counter

The clock counter takes the operating frequency of the FPGA and converts it to a lower frequency. The IR receiver is the only module in the FPGA that requires an internal clock. The clock counter counts through clock cycles and increments a counter that oscillates a signal at approximately 38KHz.

Inputs: 2.08MHz FPGA system clock

outputs: 38KHz clock

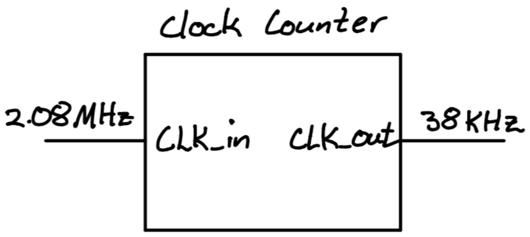


Figure 5: Clock Counter block diagram

2.2.1 Clock Counter Simulation

The clock counter simulation is done by simulating an input clock of 2.08MHz. By dividing 2.08MHz by 38KHz, the result is that the output clock should increment every 55 system clocks.

A clock rate of approximately 2.08MHz is inputted into the clock in.

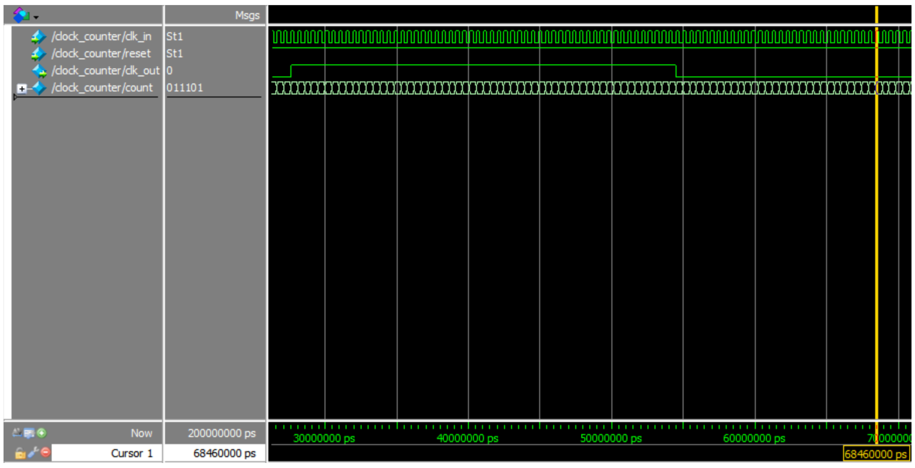


Figure 6: Clock counter simulation

2.3 Button Encoder

The button encoder accepts 12 separate button inputs over a 12 bit parallel data line. For each button that is pressed, the button encoder translates that specific button into a corresponding 8-bit hex value. The hex values are listed in Figure 2.

Inputs: 12-bit parallel data in from a 12 button board.

output: 8-bit hex value.

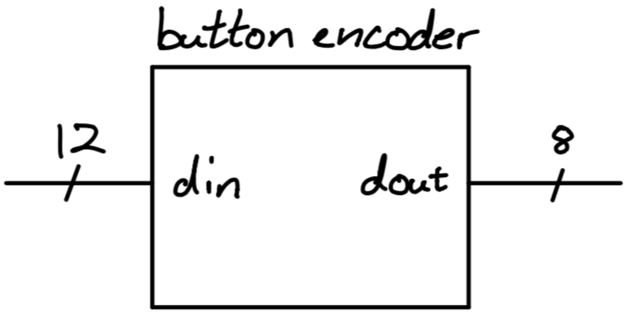


Figure 7: Button encoder block diagram

2.3.1 Button Encoder Simulation

Each button is active low and outputs a value of 0 when pressed. Each button pressed is simulated by forcing a value of zero through each data line. The resulting data out is a predetermined 8-bit hex value.

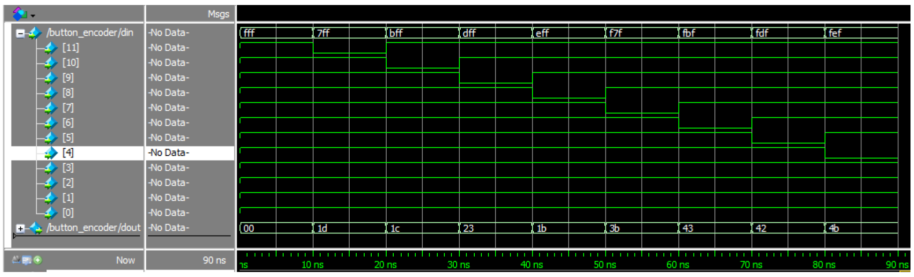

Msgs
/button_encoder/din -No Data-
[11] -No Data-
[10] -No Data-
[9] -No Data-
[8] -No Data-
[7] -No Data-
[6] -No Data-
[5] -No Data-
[4] -No Data-
[3] -No Data-
[2] -No Data-
[1] -No Data-
[0] -No Data-
/button_encoder/dout -No Data-
fff 7ff bff dff eff f7f fbf fdf fef
00 1d 1c 23 1b 3b 43 42 4b
Now 90 ns
ns 10 ns 20 ns 30 ns 40 ns 50 ns 60 ns 70 ns 80 ns 90 ns

Figure 8: Button encoder simulation

2.4 Keyboard Decoder

The keyboard module will be used to decode the waveform that will be sent by the PS2 keyboard on a button press. The waveform will be sent via a data line that is one bit, and will have a waveform that will be read using the clock signal that is generated by the on board clock oscillator on the PS2 keyboard. In doing so the decoded value will be then be sent to the input selector from figure 20 which will be used to pass an 8-bit hexa-decimal value to the SNES controller.

Inputs: The input of the PS2 keyboard is one bit serial data that will get shifted in with a clock signal.

Output: The output of the PS2 decoder will be 8-bit hexa-decimal thats used for determining what button what pressed on the SNES controller.

2.4.1 PS2 Keyboard Decoder (serial in parallel out shift register)

The shift register is essentially the decoder for the PS2 in that is essentially takes in the wave form generated by the keyboard and stores the 8 data bits that correspond to a key on the board. And then sends that signal to an input selector from figure 20 which will then send the appropriate signal so that the keyboard press can be recognized as a button press corresponding to the SNES controller layout.

Inputs: The inputs to the PS2 decoder module is data and clock from the keyboard.

Outputs: The decoder module will output a 8-bit hexa-decimal value.

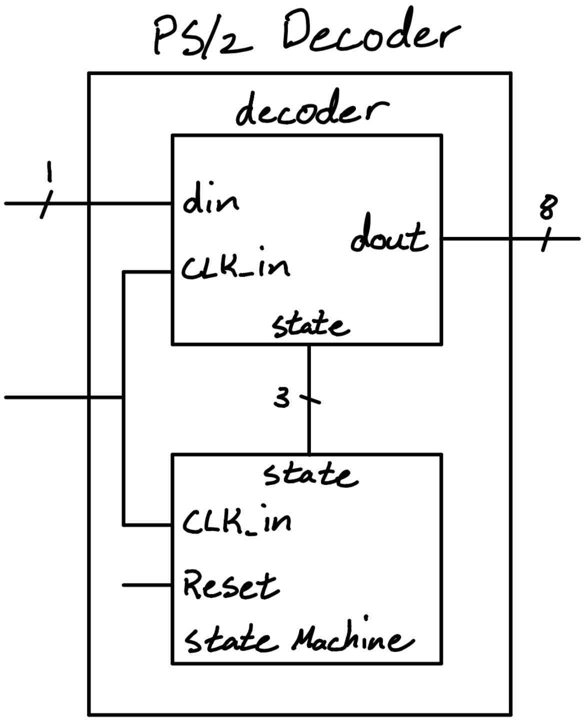


Figure 9: PS2 Keyboard Decoder Module

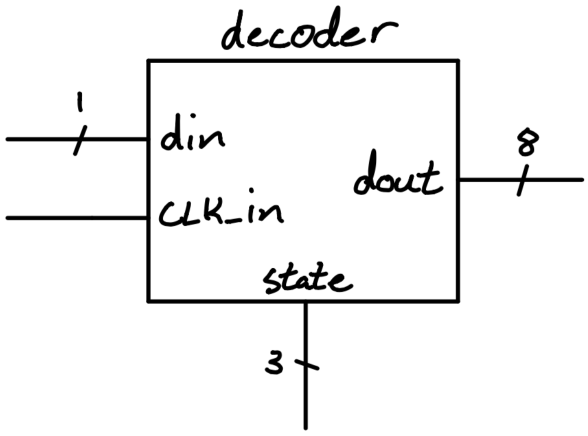


Figure 10: PS2 Keyboard Decoder Shift Register Module

2.4.2 State machine For PS2 Keyboard

The state machine that will running the decoder will have 6 states: idle, start-bit, data-read bit, stop-read bit and verify bit and send bit. Using these 6 states the shift register will take in 8-bits of information, once the data is collected the the verification state will kick in and also in this state the information will be sent if the if the verification process is successful. Also the 3rd state of this state machine is also used to register that if a button is pressed and held it will catch that and directly goes to verification if a buttons is held.

Input: PS2 clock, which will be used to determine the states.

Output: The state of the machines will be outputted to the shift register, the signal will be in a for form of a 3 bit bus.

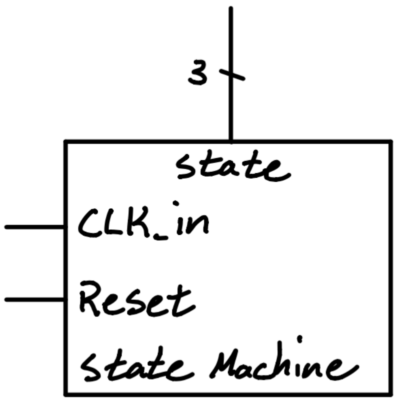


Figure 11: PS2 Keyboard Decoder Shift Register Module

2.4.3 State Diagram for PS2 State Machine

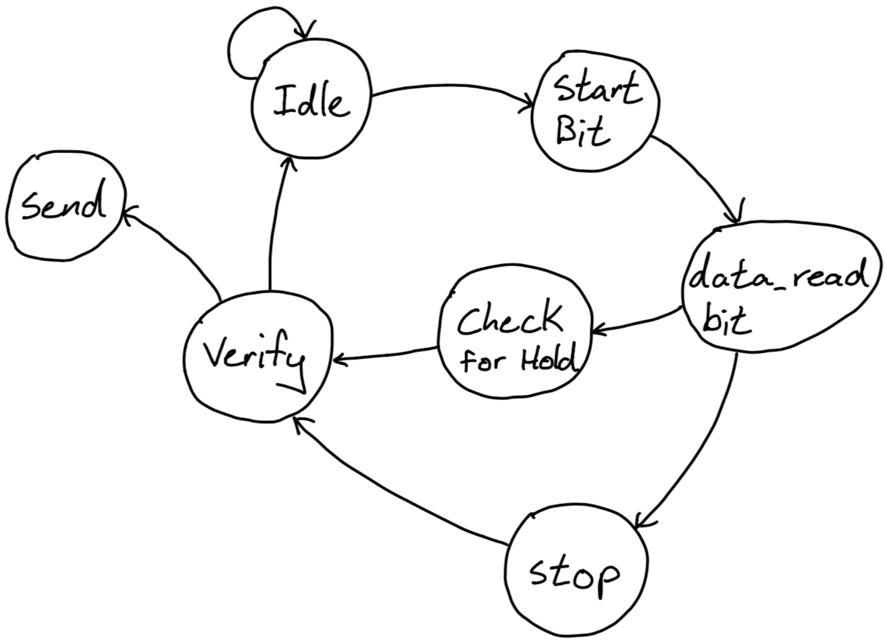


Figure 12: PS/2 Keyboard State Diagram

Figure 12 shows the 7-state state diagram used for the decoding of the PS2 keyboard waveforms, using this diagram the state machines states how what should happen at each start and how to get to each state will be shown on this diagram.

2.5 IR Decoder

This section describes the functioning for a remote a control. Remote controls send out signals via an infrared square wave. At the wave peak (1), the remote is off, and at wave trough (0), the remote is sending out a signal. The wavelength of the wave determines the logical one or zero which is sent out of the remote. Each button on the remote possesses a distinct signal which are set at along the wave in chains of thirty-two bits. Only the last eight bits are used which are then forwarded as inputs to the module. These are individually coded and designed to be implemented

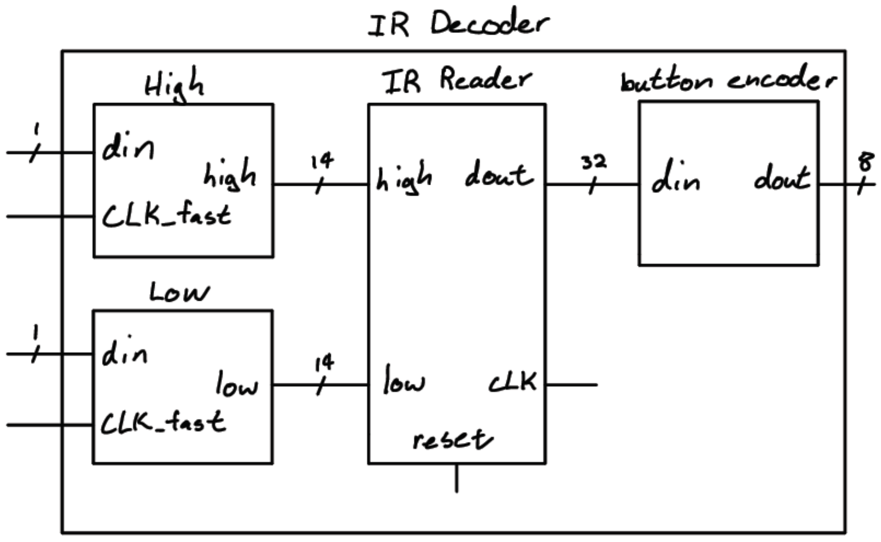


Figure 13: Block Diagram

with corresponding keys on the SNES.

Input: The input of the IR top module is the din value from the IR remote

Output: The output of the IR top module is an 8-bit hexa-decimal number that will be sent to he input selector.

2.5.1 High

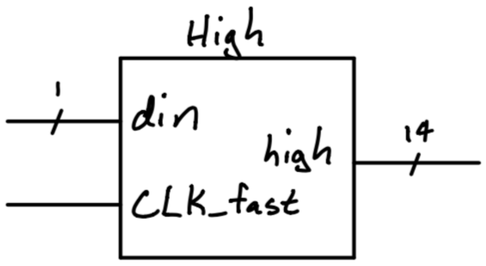


Figure 14: Block Diagram

The High count module for incrementing high count when the IR input is HIGH. This module need input the value for the input number which is din, and the clock. It will out put the 14 bits number for record the high. Adding some value like active to remember when should the count work, and the using the right to save the right state. Every time the input number from 0 to 1, the active should become 1. This is mean change the state change to active, and it can start count the high value.

Input: The the input is a single bit serial, is based from a waveform from the IR.

Output: The output of this module is an 14-bit number.

2.5.2 Low

Module for incrementing low count when the IR input is LOW. This module is very same like the high count module. For this module, it also need input numbers and the clock. The module will out the low count value. This part also have active and right for saving the state. When the

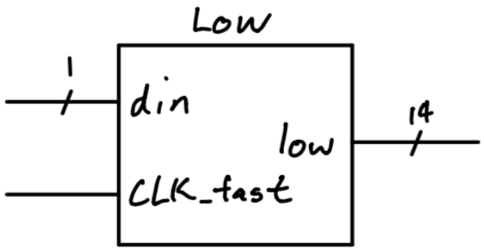


Figure 15: Block Diagram

input numbers for 1 to the 0, the active become 1, and it start count the number for low. After the work, the low value will save the count numbers and pass it.

Input: The input is a single bit serial, is based from a waveform from the IR.

Output: The output of this module is an 14-bit number.

2.5.3 Reader

The reader receives inputs from the high and low counts and sends a signal onwards to the button encoder. The clock input exists to control and differentiate between the two inputs. It is an input for the instantiated finite state machine.

Input: The inputs are 2 individual 114 bit buses, from the high and low modules. It also takes in a clock signal and a reset signal as well.

Output: The output of this module is an 32-bit number that will get inputted into the button encoder module.

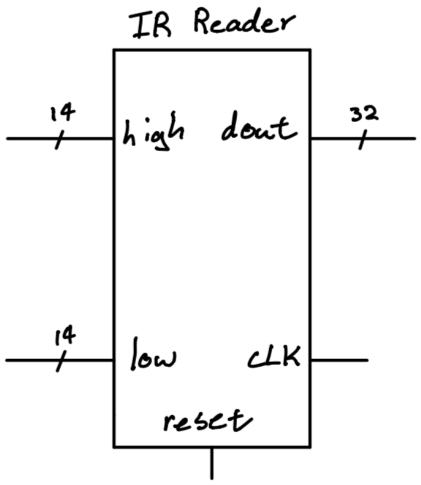


Figure 16: Block Diagram

2.5.4 Button Encoder

The button encoder takes in a 32 bit but uses only eight of those 32 bits. Depending on the binary input received by the encoder, a distinct code is sent onwards to the SNES. Each code received by the encoder corresponds to one on the SNES encoder. These values sent onwards and interpreted by the subsequent modules.

Input: The input of this module is a 32-bit signal input.

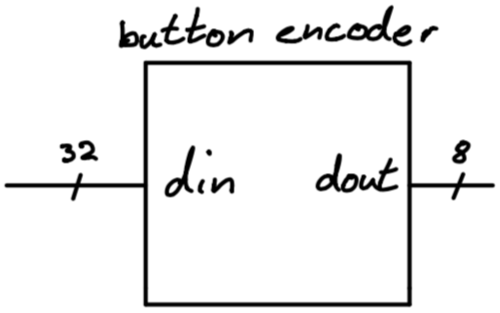
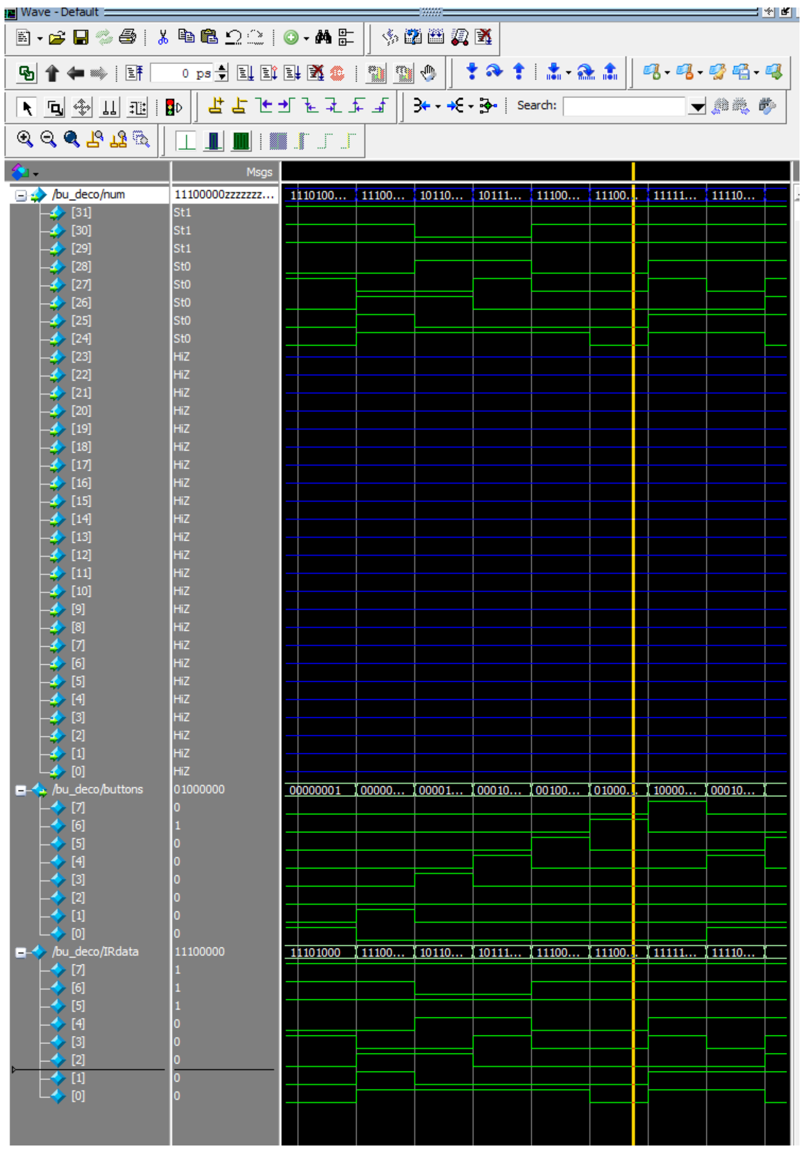


Figure 17: Block Diagram

Output: The output of this module will be encoded into an output that will.

2.5.5 IR Decoder Simulation

There are three sections to this simulation. The first segment, (all the sections above straight blue lines) are the inputs from the infrared square wave. These are further processed by the second column, which consists of those individually selected button board. The last column shows all the relevant outputs.



```
/bu_deco/num       11100000zzzzzzz...   1110100... 11100... 10110... 10111... 11100... 11100... 11111... 11110...
  [31]  St1
  [30]  St1
  [29]  St1
  [28]  St0
  [27]  St0
  [26]  St0
  [25]  St0
  [24]  St0
  [23]  HiZ
  [22]  HiZ
  [21]  HiZ
  [20]  HiZ
  [19]  HiZ
  [18]  HiZ
  [17]  HiZ
  [16]  HiZ
  [15]  HiZ
  [14]  HiZ
  [13]  HiZ
  [12]  HiZ
  [11]  HiZ
  [10]  HiZ
  [9]   HiZ
  [8]   HiZ
  [7]   HiZ
  [6]   HiZ
  [5]   HiZ
  [4]   HiZ
  [3]   HiZ
  [2]   HiZ
  [1]   HiZ
  [0]   HiZ
/bu_deco/buttons   01000000             00000001 00000... 00001... 00010... 00100... 01000... 10000... 00010...
  [7]   0
  [6]   1
  [5]   0
  [4]   0
  [3]   0
  [2]   0
  [1]   0
  [0]   0
/bu_deco/IRdata    11100000             11101000 11100... 10110... 10111... 11100... 11100... 11111... 11110...
  [7]   1
  [6]   1
  [5]   1
  [4]   0
  [3]   0
  [2]   0
  [1]   0
  [0]   0
```

Figure 18: IR Module Simulation

2.6 Input Selector

The input selector allows the system to switch between the three input sources connected to the FPGA. An 8-bit 3 input multiplexer allows all three sources to transmit an 8-bit hex value but only allows one to be active at a time. A state machine alternates between three different states on the falling edge of an active low button press.

Input: 8-bit input from IR, PS/2, button board. Select signal. Reset signal.

Output: 8-bit output.

The input selector has a total of 3 states shown in Figure 19.

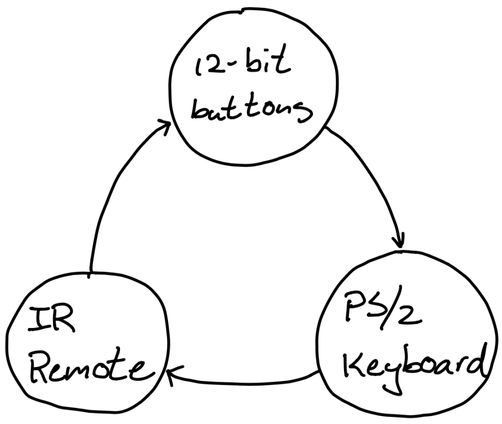


Figure 19: Input Selector States

Figure 20 shows the top module

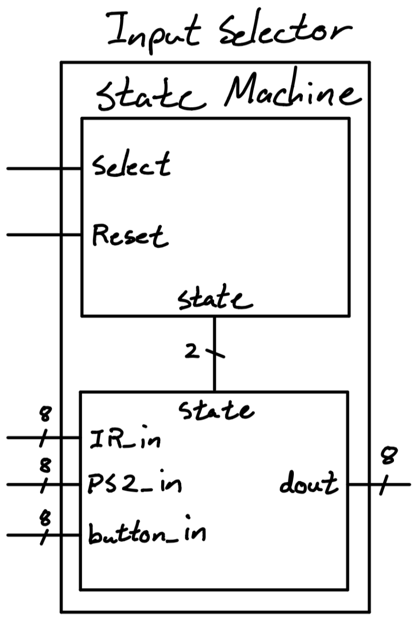


Figure 20: Input Selector Top Module

2.6.1 State Machine

The state machine takes in a single bit select signal that serves as the falling edge to transition states. A reset signal initiates the module and sets the button board as the initial mode of input. Up to 3 states exist, requiring a 2-bit output value.

Inputs: Single bit select, single bit reset.

Outputs: 2-bit state.

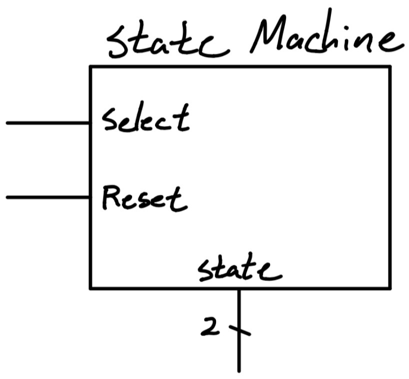


Figure 21: Input Selector State Machine

To simulate the state machine, the module is first initiated with a reset signal. A select signal oscillates at an arbitrary frequency to demonstrate each state change.

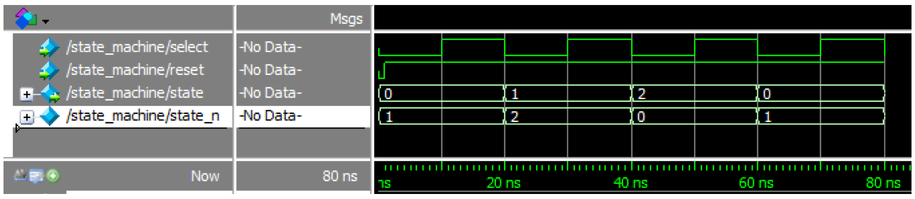


Figure 22: Input Selector State Machine Simulation

2.6.2 3:1 Multiplexer

An 3:1 multiplexer allows the state machine to select between three different inputs. Because each input module outputs the same 8-bit hex value, an 3:1 multiplexer allows easy switching between input sources.

Input: 8-bit IR, PS/2, and button in. 2-bit state.

Output: 8-bit value.

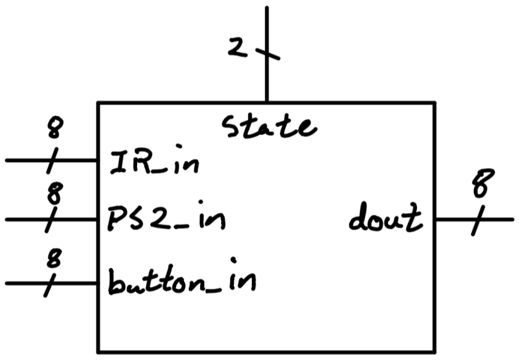


Figure 23: Input Selector 8-bit Multiplexer

The 3:1 multiplexer is simulated by arbitrarily providing different values to each 8-bit input. Three states are cycled through to show that the output changes respectively.

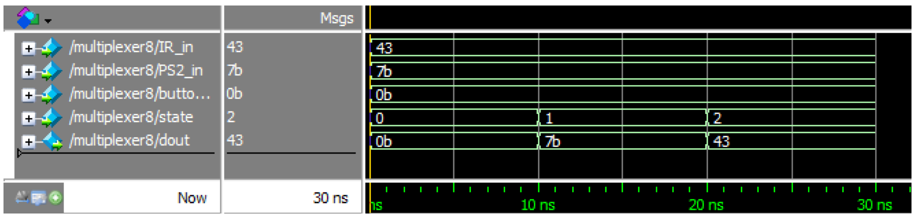


Figure 24: Input Selector 8-bit Multiplexer Simulation

2.6.3 Input Selector Simulations

The full input selector is simulated by supplying an arbitrary hex value to each 8-bit input. The reset signal is sent initially to initialize the state machine.

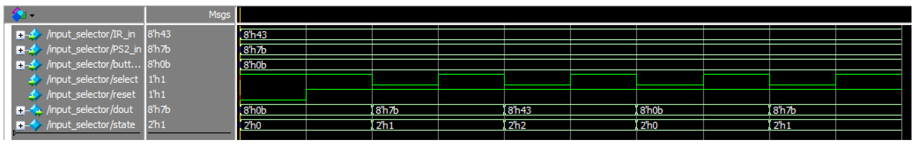

Msgs
/input_selector/IR_in 8'h43
/input_selector/PS2_in 8'h7b
/input_selector/butt... 8'h0b
/input_selector/select 1'h1
/input_selector/reset 1'h1
/input_selector/dout 8'h7b
/input_selector/state 2'h1
8'h43
8'h7b
8'h0b
8'h0b 8'h7b 8'h43 8'h0b 8'h7b
2'h0 2'h1 2'h2 2'h0 2'h1

Figure 25: Input Selector Top Module Simulation

2.7 SNES Controller

This block diagram on figure 26 illustrates the top level logic for how the SNES controller is connected internally, where there is a button reader and the shift register are connected to mimic the operations of traditional SNES controller. The way that the SNES controller works is by having 2 signals coming into the controller; the first one being a clock signal that pulses 16 times, where on every falling edge the data that is in the shift register of the controller is sampled by the SNES console. The that is and latch signal that are active low. The second signal that is sent to the controller is the latch signal which is around 60 Hz that is around 12 microseconds wide, this signal tells that controller's shift register when to send the button states (or the current button pressed) to the SNES Console where the buttons will be read.[2]

Input: The input for figure 26 is an 8-bit hexa-decimal signal that corresponds to a button press on the SNES controller, the input signal comes from the input selector in figure 20 where the signal is chosen based on what the user wants.

Output: The output for figure 26 is a serial bit by bit output into the SNES reader that will be responsible for interpreting button presses, each of the 12-bits that are transferred have the are values that signify whether a button is pressed or idle. Zero signifying a button press and One signifying a button idle state, since the SNES console runs on active low logic. [2]

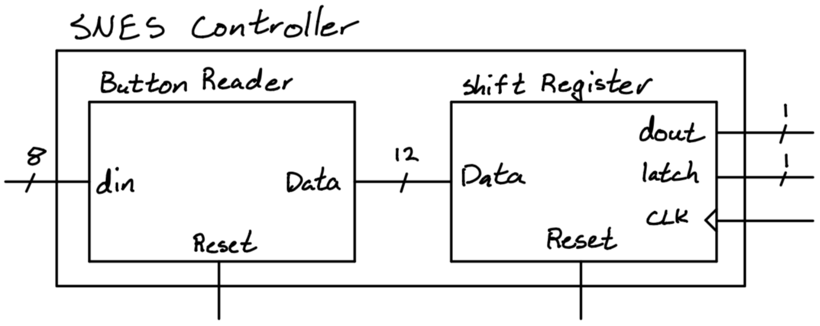


Figure 26: SNES Controller Block

2.7.1 SNES Button Reader

The block diagram in figure 27 is the button reader part of the SNES block which is responsible for interpreting the signals from the inputs: PS2 keyboard, IR remote, or the 8-bit button board.

Input: The input for figure 27 is an 8-bit hexa-decimal signal that is sent by the figure 20 which is the module that is responsible for selecting when input to use.

Output: This output for figure 27 is a 12-bit number that is responsible for showing which button of the SNES controller is being pressed.

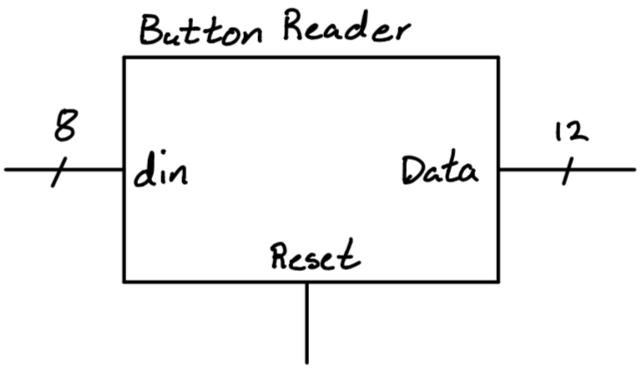


Figure 27: SNES Controller Block

2.7.2 SNES Shift Register

Figure 28 is the block diagram that is responsible for taking the parallel output of the button reader from figure 27 and then based on a clock signal and latch signal are inputted from the SNES console it will serially shift out a button signal based on the clock edge, which will in turn get translated into a button press of the SNES controller.

Input: The input for figure 28 is a 12-bit signal that is outputted from the button reader from figure 27, also this block gets a clock signal and latch signal that will be coming from SNES Reader (Emulated SNES Console) in figure 32 which will be responsible for serially outputting the buttons that were pressed on the SNES button reader module.And also the there is a reset input that is responsible for clearing the current signals in the shift register so that values are known.

Output: The output of the shift register inside the SNES controller is a dout which is the bit that has information that is serially transmitted into the the dataYellow of the SNES console.

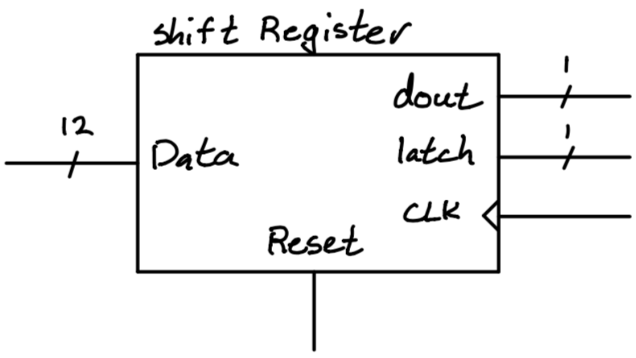


Figure 28: SNES Shift Register Block

2.7.3 SNES Controller Simulation

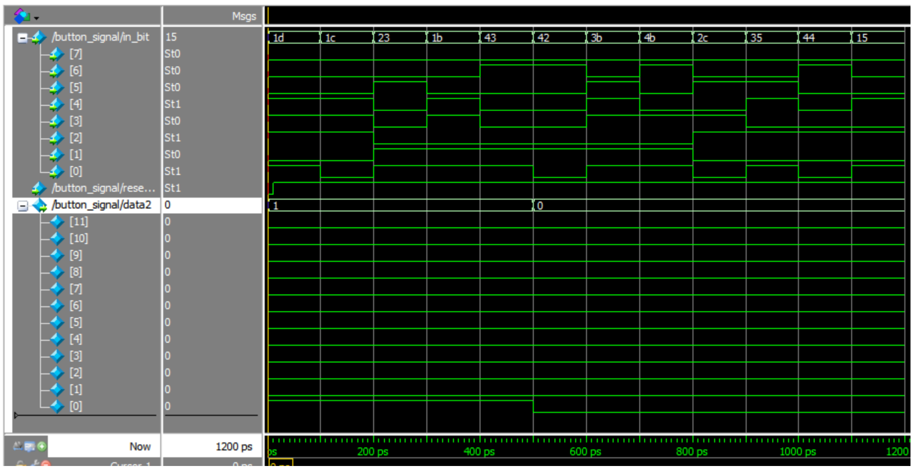


Figure 29: SNES Buttons Reader Simulation

Figure 29 above is a simulation of the button reader block which shows that the output based on the 12-bit input from the input selector from figure 20 which can choose which input mechanism is used.

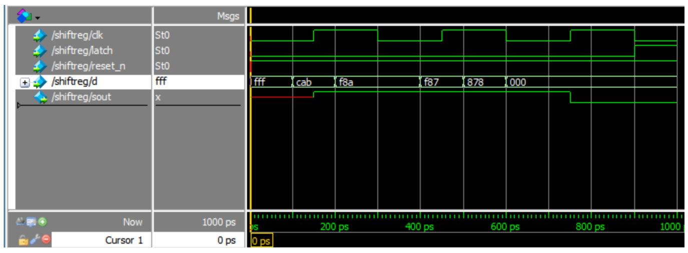


Figure 30: SNES Shift Register Simulation

Figure 30 is the ModelSim simulation of the shift register that is used in the the SNES controller, this simulation is shows the output as a parallel bus load from the button reader from 27. The output sout is "0" when the that button is pressed and "1" when the button is released.

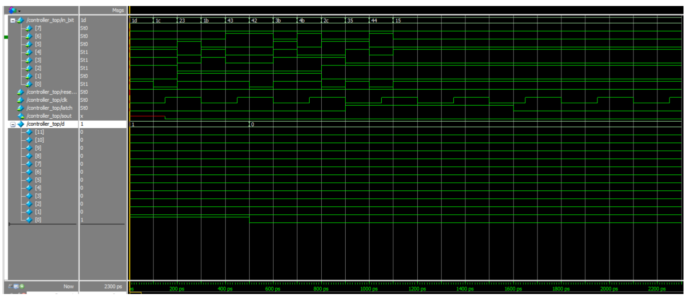


Figure 31: SNES Controller Block Simulation

Figure 31 is the ModelSim simulation of the whole SNES controller where all of the leads and connection are connected like the block diagram shown in 26. This ModelSim simulation shows that the SNES controller block functions as intend for the design needs.

All of the DO files that are associated with the the simulation of the SNES Controller module and inner modules will can be found in the Appendix at the end of the report.

2.8 SNES Console (Reader)

In figure 32 the block diagram shows an emulated version of the SNES console, the emulated version of the console works by interfacing for the controller and requesting the state of the buttons pressed via a clock and latch signal that are both active low.

Inputs: dataYellow which is the serial bit that get inputted into the SNES console on every clock edge of the SNES clock signal, this data input is shifted serially from the SNES shift register from figure 28 using a clock and latch signal to interface with the shift register that is in the SNES controller.[3]

Outputs: In SNES console module there is the SNES clock output, the latch output both of which are used for shifting the data from the shift register from figure 28. Along with that there are 12 other output signals where each one represents an actual button press on the actual SNES controller.

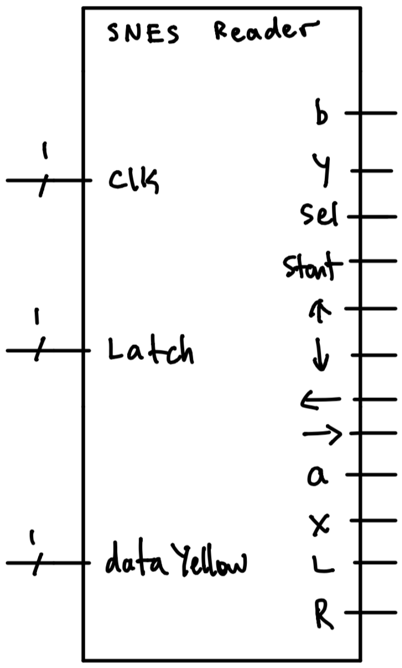
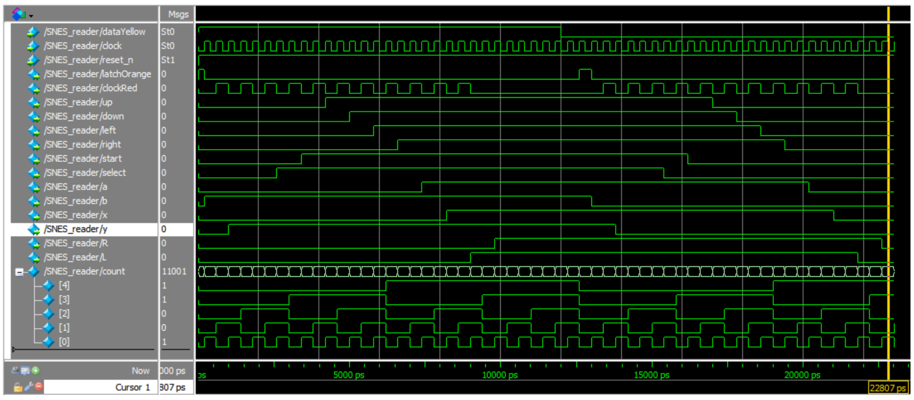


Figure 32: SNES Console Block

2.8.1 SNES Console (Reader) Simulation

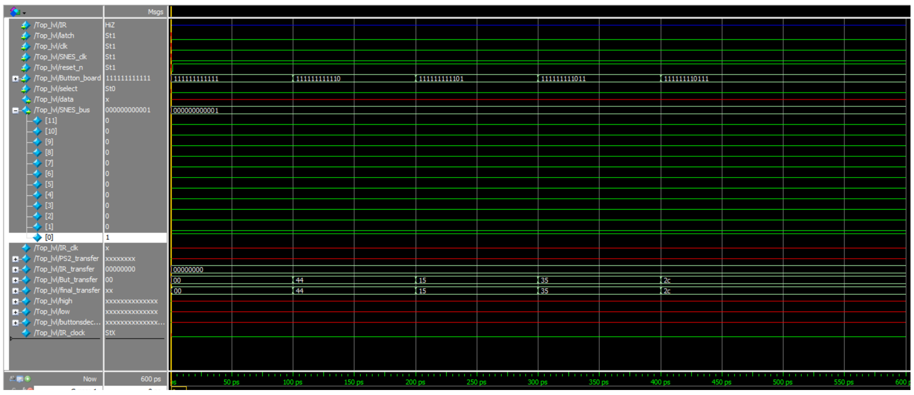


```
                              Msgs
/SNES_reader/dataYellow       St0
/SNES_reader/clock            St0
/SNES_reader/reset_n          St1
/SNES_reader/latchOrange      0
/SNES_reader/clockRed         0
/SNES_reader/up               0
/SNES_reader/down             0
/SNES_reader/left             0
/SNES_reader/right            0
/SNES_reader/start            0
/SNES_reader/select           0
/SNES_reader/a                0
/SNES_reader/b                0
/SNES_reader/x                0
/SNES_reader/y                0
/SNES_reader/R                0
/SNES_reader/L                0
/SNES_reader/count            11001
    [4]                       1
    [3]                       1
    [2]                       0
    [1]                       0
    [0]                       1
                   Now        000 ps
                   Cursor 1   307 ps
```

Figure 33: SNES Console Block

Figure 33 is the ModelSim simulation of the SNES console that is responsible for reading the inputs from the SNES shift register from figure 28, the outputs of the SNES console should be "0" is the button is logically pressed and "1" if the the button is at an idle state.

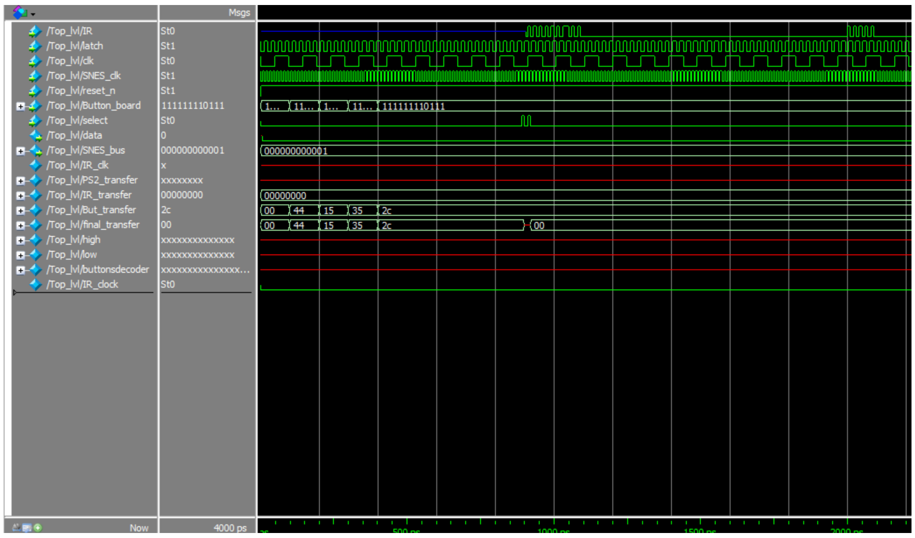
3 Top Level Hardware simulation



```
                              Msgs
/Top_lvl/IR                   HiZ
/Top_lvl/latch                St1
/Top_lvl/clk                  St1
/Top_lvl/SNES_clk             St1
/Top_lvl/reset_n              St1
/Top_lvl/Button_board         111111111111
/Top_lvl/select               St0
/Top_lvl/data                 x
/Top_lvl/SNES_bus             000000000001
    [11]                      0
    [10]                      0
    [9]                       0
    [8]                       0
    [7]                       0
    [6]                       0
    [5]                       0
    [4]                       0
    [3]                       0
    [2]                       0
    [1]                       0
    [0]                       1
/Top_lvl/IR_clk               x
/Top_lvl/PS2_transfer         xxxxxxxx
/Top_lvl/IR_transfer          00000000
/Top_lvl/But_transfer         00
/Top_lvl/final_transfer       xx
/Top_lvl/high                 xxxxxxxxxxxxxx
/Top_lvl/low                  xxxxxxxxxxxxxx
/Top_lvl/buttonsdec...        xxxxxxxxxxxxxx...
/Top_lvl/IR_clock             StX
                   Now        600 ps
```

Figure 34: Top Level Simulation with only Button Board

Figure reffig:Topbutsim is a top level hardware simulation that tests how all of the module works when they are all put in and interfaced with one another. In this simulation there only the the button board is used and because the initial state of the input selector is using the button input, and the state only changes when there are pulses initiated by the button board.



```
                                Msgs
/Top_lvl/IR                     St0
/Top_lvl/latch                  St1
/Top_lvl/clk                    St0
/Top_lvl/SNES_clk               St1
/Top_lvl/reset_n                St1
/Top_lvl/Button_board           111111110111
/Top_lvl/select                 St0
/Top_lvl/data                   0
/Top_lvl/SNES_bus               000000000001
/Top_lvl/IR_clk                 x
/Top_lvl/PS2_transfer           xxxxxxxx
/Top_lvl/IR_transfer            00000000
/Top_lvl/But_transfer           2c
/Top_lvl/final_transfer         00
/Top_lvl/high                   xxxxxxxxxxxxxx
/Top_lvl/low                    xxxxxxxxxxxxxx
/Top_lvl/buttonsdecoder         xxxxxxxxxxxxxx...
/Top_lvl/IR_clock               St0
Now                             4000 ps
```

Figure 35: Simulation of every module

Figure reffig:Topeverysim is the ModelSim simulation of the top level hardware block with every module included where and the is shows the input selector cycling through its states where "0" is the button board, "1" is the PS2 Keyboard and "3" is the IR remote.

A System Verilog Files

A.1 Counter System Verilog Files

Counter System Verilog

```
module clock_counter(
  input logic clk_in,  //2.08MHz
  input reset,

  output logic clk_out
    );

    logic [5:0] count; //count to a max value of 63

                always_ff @ (posedge clk_in, negedge reset)
                        begin
                                count <= count + 1;
                                if(!reset)
                                        begin
                                                clk_out <= 0;
                                                count <= 0;
                                        end
                                else
                                        if(count >= 55) //freq out of 38KHz, flip count approx every 55 cycles
                                                begin
                                                        clk_out <= ~clk_out; //Flip slow clock
                                                        count <= 0;                    //Reset the counter
                                                end
                        end
endmodule
```

A.2 Button Encoder System Verilog Files

Button Encoder Module

```
module button_encoder(
        input logic [11:0] din,

        output logic [7:0] dout
                );

                always_comb
                        begin
                                if (!din[11]) begin
                                        dout = 8'h1D;
                                end
                                else if (!din[10]) begin
```

```
                                        dout = 8'h1C;
                                end
                                else if (!din[9]) begin
                                        dout = 8'h23;
                                end
                                else if (!din[8]) begin
                                        dout = 8'h1B;
                                end
                                else if (!din[7]) begin
                                        dout = 8'h3B;
                                end
                                else if (!din[6]) begin
                                        dout = 8'h43;
                                end
                                else if (!din[5]) begin
                                        dout = 8'h42;
                                end
                                else if (!din[4]) begin
                                        dout = 8'h4B;
                                end
                                else if (!din[3]) begin
                                        dout = 8'h2C;
                                end
                                else if (!din[2]) begin
                                        dout = 8'h35;
                                end
                                else if (!din[1]) begin
                                        dout = 8'h15;
                                end
                                else if (!din[0]) begin
                                        dout = 8'h44;
                                end
                                else begin
                                        dout = 8'h00;
                                end
                        end

endmodule
```

A.3 PS2 Keyboard Decoder System Verilog Files

Keyboard Decoder Shift Register

```
module shift(input logic din, input logic clk, output logic [7:0] dout)
    ;
        logic [10:0] s_reg;
        logic hold;
        logic [7:0] memory;
/*logic clk;
OSCH #("2.08") osc_int(
        .STDBY(1'b0),
        .OSC(clk),
        .SEDSTDBY()
);*/
        always @(negedge clk)begin
                s_reg <= {s_reg[9:0],din};
                        if(s_reg[10] == 0)begin
                                if(s_reg[9:2] == 8'b00001111)begin
                        hold = 1;
                        dout <= 0;
                        memory <= 0;
                        s_reg <= {10'bx,1'b0};
                    end
                else if(hold != 1)begin
                        dout <= {s_reg[2],s_reg
                          [3],s_reg[4],s_reg
                          [5],s_reg[6],s_reg
                          [7],s_reg[8],s_reg
                          [9]};
                        memory <={s_reg[2],
                          s_reg[3],s_reg[4],
                          s_reg[5],s_reg[6],
                          s_reg[7],s_reg[8],
                          s_reg[9]};
                        s_reg <= {10'bx,1'b0};
                    end
                    else if(hold == 1)begin
                        dout <= 11'b0;
                        memory <=11'b0;


                        s_reg <= {10'bx,1'b0};
                        //hold <= 0;
                    end
                else begin
                        dout <= {s_reg[2],s_reg
                          [3],s_reg[4],s_reg
                          [5],s_reg[6],s_reg
                          [7],s_reg[8],s_reg
                          [9]};
                        memory <={s_reg[2],
                          s_reg[3],s_reg[4],
                          s_reg[5],s_reg[6],
                          s_reg[7],s_reg[8],
                          s_reg[9]};
                        s_reg <= {10'bx,1'b0};

                        hold = 0;
                    end
                end
            else begin
                    dout <= memory;
                end
        end

endmodule
```

A.4 IR Decoder System Verilog Files

Button Decoder

```
//IR
// button  decoder.sv
//
// Decodes 32 bit decoded result from the IR component further
// into 12 possible buttons inputs for our SNES encoder
//

module bu_deco(
                input logic [31:0] num,
                output logic [7:0] buttons
);

        logic [7:0] IRdata;

        always_comb
                begin
                        IRdata <= num[31:24];

                        // determine if any of the 12 buttons we're
                           looking for were pressed
                        case(IRdata)
                                // SNES BUTTON / IR BUTTON
                                // B / 6
                                8'b11111010:
                                        buttons = 8'b00000001;

                                // Y / 5
                                8'b11101000:
                                        buttons = 8'b00000010;

                                //  Up
                                8'b11101000:
                                        buttons = 8'b00000100;

                                //  Down
                                8'b11100111:
                                        buttons = 8'b00001000;

                                // up / up
                                8'b10110101:
                                        buttons = 8'b00010000;

                                // Down / Down
                                8'b10111001:
                                        buttons = 8'b00100000;

                                // Left / Left
                                8'b11100001:
                                        buttons = 8'b01000000;

                                // Right / Right
                                8'b11100000:
                                        buttons = 8'b10000000;

                                // A / 4
                                8'b11111011:
                                        buttons = 8'b00010001;

                                // X / 0
                                8'b11110011:
                                        buttons = 8'b00100001;

                // L / 2
                8'b11111110:
                    buttons = 8'b01000001;

                // R / 8
                8'b11110110:
                    buttons = 8'b10000001;


                default:
                    buttons = 8'b00000000;
            endcase
        end
endmodule
```

High Count

```

//IR
// high counter
//
// Module for incrementing high count when the ir input is HIGH
//

module high(
    input logic innum,
    input logic clock,
    output logic[13:0] high
);
    // whether this count is active
    logic active;

    logic [13:0]right;

    always_ff@(posedge innum)
        begin
            active <= 1;
            right <= 0;
        end


    always_ff@(negedge innum)
        begin
            active <= 0;
            high <= right;
        end

    always_ff@(posedge clock)
        if(active)
            right <= right + 1;


endmodule
```

Low Count

```
//IR
// low count
//
// Module for incrementing low count when the ir input is LOW
```

```
//

module low(
        input logic innum,
        input logic clock,
        output logic[13:0] low
);

        logic active;

        logic [13:0] right;

        always_ff@(negedge innum)
                begin
                        active <= 1;
                        right <= 0;
                end

        always_ff@(posedge innum)
                begin
                        active <= 0;
                        low <= right;
                end

        always_ff@(posedge clock)
                if(active)
                        right <= right + 1;


endmodule
```

IR Reader

```
//IR
// reader
//
//


module reader(
        input logic re_data,
        input logic [13:0] high,
        input logic [13:0] low,
        input logic clock,
        output logic [31:0] out_deco
);


        typedef enum logic[1:0] {S0,S1,S2}statetype;

        statetype[1:0] state, state1;

        logic [4:0] readB;


        always_ff @ (posedge clock, posedge re_data)
                if(re_data) state <= S0;
                else      state <= state1;

```

```

    always_comb
        begin
            case(state)
                S0:

                    if(low < 8000)
                        state1 <= S1;
                    else
                        state1 <= S0;

                S1:
                    if(high < 4000)
                        begin
                            readB <= 0;
                            out_deco <= 0;
                            state1 <= S2;
                        end
                    else
                        state1 <= S0;

                S2:
                    begin
                        if(high > 1500 && high < 1650)
                            begin
                                out_deco[readB] <= 1;
                                readB++;
                            end


                        if(high > 650 && high < 500)
                            begin
                                out_deco[readB] <= 0;
                                readB++;
                            end


                        if(readB < 31)
                            state1 <= S2;
                        else
                            state1 <= S0;
                    end

                default: state1 <= S0;
```

```

                                endcase
                        end
                //assign out_deco =
endmodule
```

Top-Level

```
//IR
// module.sv
//
//
//

module IR(
                        input logic innum,
                        input logic inclock,
                        input logic re_data,
                        output logic[7:0]data
);

                // internal 6 bit counter & 3 bits of state
                //logic [5:0] count;
                //logic [1:0] state;

                logic [13:0] high;
                logic [13:0] low;

                logic [31:0] buttonsdecoder;

                // handle high bit count
                high hc(

                        .innum(innum),
                        .clock(inclock),
                        .high(high)
                );

                // handle low bit count
                low ic(

                        .innum(innum),
                        .clock(inclock),
                        .low(low)
                );

                // handle state machine reader
                reader re(

                        .re_data(re_data),
                        .high(high),
                        .low(low),
                        .clock(inclock),
                        .out_deco(buttonsdecoder)
                );


                bu_deco deco(
                        .num(buttonsdecoder),
                .buttons(data)
        );

endmodule
```

A.5 Input Selector System Verilog Files

Input Selector State Machine Module

```
module state_machine(
        input logic select, reset,

        output logic [1:0] state
                );

                logic [1:0] state_n;

                parameter S0 = 2'b00;
                parameter S1 = 2'b01;
                parameter S2 = 2'b10;

                always_ff @ (negedge select, negedge reset)
                        begin
                                if(!reset)
                                        state = S0;
                                else
                                        state = state_n;
                        end

                always_comb
                        case(state)
                                S0: state_n = S1;
                                S1: state_n = S2;
                                S2: state_n = S0;
                        endcase
endmodule
```

Input Selector 3:1 Multiplexer Module

```
module multiplexer8(
        input logic [7:0] IR_in,
        input logic [7:0] PS2_in,
        input logic [7:0] button_in,
        input logic [1:0] state,

        output logic [7:0] dout
                );

                parameter S0 = 2'b00;
                parameter S1 = 2'b01;
                parameter S2 = 2'b10;

                always_comb
                        case(state)
                                S0: dout = button_in;
                                S1: dout = PS2_in;
                                S2: dout = IR_in;
                        endcase
```

```
endmodule
```

Input Selector Top Module

```
module input_selector(
        input logic [7:0] IR_in,
        input logic [7:0] PS2_in,
        input logic [7:0] button_in,
        input logic select, reset,

        output logic [7:0] dout
                );

                logic [1:0] state;

                state_machine sm1(
                .select(select),
                .reset(reset),
                .state(state));

                multiplexer8 mux1(
                .IR_in(IR_in),
                .PS2_in(PS2_in),
                .button_in(button_in),
                .state(state),
                .dout(dout));
endmodule
```

A.6 SNES Controller System Verilog Files

SNES Button Reader

```
module button_signal (
        input logic [7:0] in_bit,
        input logic reset_n,
        output logic [11:0] data2
        );


parameter W = 8'h1d;
parameter A = 8'h1c;
parameter S = 8'h23;
parameter D = 8'h1b;
parameter i = 8'h43;
parameter k = 8'h42;
parameter j = 8'h3b;
parameter l = 8'h4b;
parameter Sel_t = 8'h2c;
parameter Start_y = 8'h35;
parameter L_q = 8'h44;
parameter R_o = 8'h15;




        always_comb

                if (!reset_n)
```

```
                                data2 <= 1;
                        else
                                case( in_bit )

                                        k: data2[0] <= 1'b0;
                                        j: data2[1] <= 1'b0;
                                        Sel_t: data2[2] <= 1'b0;
                                        Start_y: data2[3] <= 1'b0;
                                        W: data2[4] <= 1'b0;
                                        S: data2[5] <= 1'b0;
                                        A: data2[6] <= 1'b0;
                                        D: data2[7] <= 1'b0;
                                        l: data2[8] <= 1'b0;
                                        i: data2[9] <= 1'b0;
                                        L_q: data2[10] <= 1'b0;
                                        R_o: data2[11] <= 1'b0;
                                        default: data2 <= 1'b1;

                                endcase




endmodule
```

SNES Shift Register

```
module shiftreg #(parameter N = 12) (
        input logic clk,
        input logic latch, reset_n,
        input logic [N-1:0] d,
        output logic sout

);
        always_ff @(posedge clk)


                if (!reset_n)
                        sout = 0;

                else if (!latch)

                        sout = d[N-1];


endmodule
```

SNES Controller Top Module

```
module controller_top (
        input logic [7:0] in_bit,
        input logic reset_n,
        input logic clk,
        input logic latch,
        output logic sout
);

logic [11:0] d;


```

```
        button_signal dank (
                .in_bit(in_bit),
                .reset_n(reset_n),
                .data2(d)
        );

        shiftreg dank1 (
                .clk(clk),
                .latch(latch),
                .d(d),
                .sout(sout)
        );

endmodule
```

A.7 SNES Console System Verilog Files

SNES Reader Module

```
module SNES_reader(
  input logic dataYellow,
  input logic clock,
  input logic reset_n,
  output logic latchOrange,
  output logic clockRed,
  output logic up,
  output logic down,
  output logic left,
  output logic right,
  output logic start,
  output logic select,
  output logic a,
  output logic b,
  output logic x,
  output logic y,
  output logic R,
  output logic L
  );
  logic [4:0] count;

  Counter4 instance1(
    .clk              (clock),
    .reset_n          (reset_n),
    .count            (count)
  );

  SNES_ClockStateDecoder instance2(
    .controllerState  (count),
    .SNES_clk         (clockRed)
  );

  SNES_LatchStateDecoder instance3 (
    .controllerState  (count),
    .SNES_Latch         (latchOrange)
  );

  SNES_DataReceiverDecoder instance4 (
    .dataYellow       (dataYellow),
    .reset_n            (reset_n),
    .controllerState   (count),
    .readButtons       ({b, y, select, start, up, down, left, right, a,
       x, L, R})
  );
endmodule


module Counter4(
  input logic clk, reset_n,
  output logic [4:0] count);

  always_ff @ (posedge clk, negedge reset_n)
    if(!reset_n) count <= 5'b0_0000;
    else count <= count + 1;
endmodule


module SNES_LatchStateDecoder(
  input logic [4:0] controllerState,
  output logic SNES_Latch);

  always_comb
    case(controllerState)
      5'b0_0000: SNES_Latch = 1;
      default: SNES_Latch = 0;
    endcase
endmodule


module SNES_ClockStateDecoder(
  input logic [4:0] controllerState,
  output logic SNES_clk);

  always_comb
    case (controllerState)
      5'b0_0010: SNES_clk = 1;
      5'b0_0100: SNES_clk = 1;
      5'b0_0110: SNES_clk = 1;
      5'b0_1000: SNES_clk = 1;
      5'b0_1010: SNES_clk = 1;
      5'b0_1100: SNES_clk = 1;
      5'b0_1110: SNES_clk = 1;
      5'b1_0000: SNES_clk = 1;
      5'b1_0010: SNES_clk = 1;
      5'b1_0100: SNES_clk = 1;
      5'b1_0110: SNES_clk = 1;

      default: SNES_clk = 0;
    endcase
endmodule


module SNES_DataReceiverDecoder(
  input logic dataYellow,
  input logic reset_n,
  input logic [4:0] controllerState,
  output logic [11:0] readButtons);

  always_ff @ (posedge controllerState[0], negedge reset_n)
    if(!reset_n) readButtons <= 11'b0;
    else case(controllerState[4:0])

      5'b0_0001: readButtons[11] <= dataYellow; //b button
      5'b0_0011: readButtons[10] <= dataYellow; //y button
      5'b0_0111: readButtons[9] <= dataYellow;  //select button
      5'b0_1001: readButtons[8] <= dataYellow;  //start button
      5'b0_1011: readButtons[7] <= dataYellow;  //up button
      5'b0_1101: readButtons[6] <= dataYellow;  //down button
      5'b0_1111: readButtons[5] <= dataYellow;  //left button
      5'b1_0001: readButtons[4] <= dataYellow;  //right button
      5'b1_0011: readButtons[3] <= dataYellow;  //a button
      5'b1_0101: readButtons[2] <= dataYellow;  //x button
      5'b1_0111: readButtons[1] <= dataYellow;  //L button
      5'b1_1001: readButtons[0] <= dataYellow;  //R button

      default: readButtons <= readButtons;
    endcase

endmodule
```

A.8 Top Level System Verilog Files

```
module Top_lvl (
        input logic IR,
        input logic latch,
        input logic SNES_clk,
        input logic reset_n,
        input logic [11:0] Button_board,
        input logic select,
        output logic data,
        output logic [11:0] SNES_bus
);
logic IR_clk;
logic clk;
logic [7:0] PS2_transfer;
logic [7:0] IR_transfer;
logic [7:0] But_transfer;
logic [7:0] final_transfer;
logic [13:0] high;
logic [13:0] low;
logic [31:0] buttonsdecoder;



OSCH #("2.08") osc_int (        //"2.08" specifies the operating
    frequency, 2.08 MHz.
                                                                  //
                                                                     Other

                                                                     clock

                                                                     frequencie

                                                                     can
```

```
                                                                                be

                                                                                found

                                                                                in

                                                                                the

                                                                                MachX02
                                                                                '
                                                                                s

                                                                                documentat
                    .STDBY(1'b0),                       //Specifies
                       active state
                    .OSC(clk),                                  //
                       Outputs clock signal to 'clk' net
                    .SEDSTDBY());                       //Leaves
                       SEDSTDBY pin unconnected*/


button_signal danky(
                    .in_bit(final_transfer),
                    .reset_n(reset_n),
                    .data2(SNES_bus)
                    );

shiftreg shifty (
                    .clk(SNES_clk),
                    .latch(latch),
                    .reset_n ( reset_n ),
                    .d(SNES_bus),
                    .sout(data)
                    );

clock_counter clocky(
                    .clk_in(clk),
                    .reset(reset_n),
                    .clk_out(IR_clock)
                    );


button_encoder dna (
                    .din(Button_board),
                    .dout(But_transfer)
                    );

IR dankmemes (
                    .innum(IR),
                    .inclock(clk),
                    .re_data(reset_n),
                    .data(IR_transfer)
                    );

input_selector instanbul (
        .IR_in(IR_transfer),
          .PS2_in(PS2_transfer),
          .button_in(But_transfer),
          .select(select),
          .reset(reset_n),
          .dout(final_transfer)

);

endmodule
```

B Simulation Files (Do scripts)

B.0.1 Counter Do Files

Do File for Counter

```
restart

force reset 0 0us, 1 1us
force clk_in 0 0ns, 1 240ns -r 480ns

run 10 us
```

B.0.2 Button Encoder Do File

Do File for Button Encoder

```
restart

force din 2#111111111111 0, 2#011111111111 10, 2#101111111111 20,
    2#110111111111 30, 2#111011111111 40, 2#111101111111 50,
    2#111110111111 60, 2#111111011111 70, 2#111111101111 80

run 90
```

B.0.3 PS2 Keyboard Decoder Do File

Do File for PS2 Keyboard Shift Register

```
vsim work.shift
add wave din
add wave clk
add wave hold
add wave s_reg
add wave memory
add wave dout
force din 0,1 0.05 ms,0 0.10 ms,1 0.15 ms,1 0.20 ms,1 0.25 ms,0 0.3 ms
    ,0 0.35 ms,0 0.4 ms,1 0.45 ms,1 0.5 ms
force din 0 0.55 ms,0 0.60 ms,0 0.65 ms,0 0.70 ms,0 0.75 ms,1 0.80 ms,1
     0.85 ms,1 0.9 ms,1 0.95 ms,1 1 ms, 1 1.05 ms
force din 0 1.1 ms,0 1.15 ms,0 1.2 ms,1 1.25 ms,1 1.3 ms,1 1.35 ms,0
    1.4 ms,0 1.45 ms,0 1.5 ms,1 1.55 ms,1 1.6 ms
force clk 1, 0 0.025 ms -r 0.05 ms
run 2 ms
```

B.0.4 IR Decoder Do Files

Do File for Top Level

```
add wave *
force num(31)  1 @ 100, 1 @ 200, 1 @ 300, 1 @ 400, 1 @ 500, 1 @ 600, 1
    @ 700, 1 @ 800, 1 @ 900, 1 @ 1000, 1 @ 1100, 1 @ 1200, 1 @ 1300, 1
    @ 1400
force num(30)  1 @ 100, 1 @ 200, 1 @ 300, 1 @ 400, 1 @ 500, 1 @ 600, 0
    @ 700, 0 @ 800, 1 @ 900, 1 @ 1000, 1 @ 1100, 1 @ 1200, 1 @ 1300, 1
    @ 1400
force num(29)  1 @ 100, 1 @ 200, 1 @ 300, 1 @ 400, 1 @ 500, 1 @ 600, 1
    @ 700, 1 @ 800, 1 @ 900, 1 @ 1000, 1 @ 1100, 1 @ 1200, 1 @ 1300, 1
    @ 1400
force num(28)  1 @ 100, 1 @ 200, 1 @ 300, 0 @ 400, 0 @ 500, 0 @ 600, 1
    @ 700, 1 @ 800, 0 @ 900, 0 @ 1000, 1 @ 1100, 1 @ 1200, 1 @ 1300, 1
    @ 1400
force num(27)  1 @ 100, 0 @ 200, 1 @ 300, 1 @ 400, 1 @ 500, 0 @ 600, 0
    @ 700, 1 @ 800, 0 @ 900, 0 @ 1000, 1 @ 1100, 0 @ 1200, 1 @ 1300, 0
    @ 1400
force num(26)  1 @ 100, 1 @ 200, 0 @ 300, 0 @ 400, 0 @ 500, 1 @ 600, 1
    @ 700, 0 @ 800, 0 @ 900, 0 @ 1000, 0 @ 1100, 0 @ 1200, 1 @ 1300, 1
    @ 1400
force num(25)  1 @ 100, 1 @ 200, 1 @ 300, 0 @ 400, 0 @ 500, 1 @ 600, 0
    @ 700, 0 @ 800, 0 @ 900, 0 @ 1000, 1 @ 1100, 1 @ 1200, 1 @ 1300, 1
    @ 1400
force num(24)  0 @ 100, 0 @ 200, 0 @ 300, 0 @ 400, 0 @ 500, 1 @ 600, 1
    @ 700, 1 @ 800, 1 @ 900, 0 @ 1000, 1 @ 1100, 1 @ 1200, 0 @ 1300, 0
    @ 1400

run 1600
```

B.0.5 Input Selector Do Files

Do File for Input Selector

```
restart

force reset 0 0, 1 10

force IR_in 2#01000011 0
force PS2_in 2#01111011 0
force button_in 2#00001011 0

force select 1 0, 0 20, 1 30, 0 40, 1 50, 0 60, 1 70, 0 80, 1 90

run 100
```

B.0.6 SNES Controller Do Files

Do File for SNES Button Reader

```
vsim work.button_signal

restart

add wave *

```

```
force -drive {sim:/button_signal/in_bit[7]} 0 0, 0 100, 0 200, 0 300, 0
    400, 0 500, 0 600, 0 700, 0 800, 0 900, 0 1000, 0 1100
force -drive {sim:/button_signal/in_bit[6]} 0 0, 0 100, 0 200, 0 300, 1
    400, 1 500, 0 600, 1 700, 0 800, 0 900, 1 1000, 0 1100
force -drive {sim:/button_signal/in_bit[5]} 0 0, 0 100, 1 200, 0 300, 0
    400, 0 500, 1 600, 0 700, 1 800, 1 900, 0 1000, 0 1100
force -drive {sim:/button_signal/in_bit[4]} 1 0, 1 100, 0 200, 1 300, 0
    400, 0 500, 1 600, 0 700, 0 800, 1 900, 0 1000, 1 1100
force -drive {sim:/button_signal/in_bit[3]} 1 0, 1 100, 0 200, 1 300, 0
    400, 0 500, 1 600, 1 700, 1 800, 0 900, 0 1000, 0 1100
force -drive {sim:/button_signal/in_bit[2]} 1 0, 1 100, 0 200, 0 300, 0
    400, 0 500, 0 600, 0 700, 1 800, 1 900, 1 1000, 1 1100
force -drive {sim:/button_signal/in_bit[1]} 0 0, 0 100, 1 200, 1 300, 1
    400, 1 500, 1 600, 1 700, 0 800, 0 900, 0 1000, 0 1100
force -drive {sim:/button_signal/in_bit[0]} 1 0, 0 100, 1 200, 1 300, 1
    400, 0 500, 1 600, 1 700, 0 800, 1 900, 0 1000, 1 1100
force reset_n 1 0, 0 1, 1 10

run 1200
```

Do File for SNES Shift Register

```
vsim work.shiftreg

add wave *

force -drive sim:/shiftreg/d 111111111111 0, 110010101011 100,
   111110001010 200, 111110001010 300, 111110000111 400, 100001111000
   500, 000000000000 600
force clk 0 0, 1 150 -r 300
force latch 0 0, 1 900 -r 1600
force reset_n 0 0, 1 1, 0 2, 1 3

run 1000
```

Do File for SNES Button Reader

```
vsim work.controller_top

add wave *

force -drive {sim:/controller_top/in_bit[7]} 0 0, 0 100, 0 200, 0 300,
   0 400, 0 500, 0 600, 0 700, 0 800, 0 900, 0 1000, 0 1100
force -drive {sim:/controller_top/in_bit[6]} 0 0, 0 100, 0 200, 0 300,
   1 400, 1 500, 0 600, 1 700, 0 800, 0 900, 1 1000, 0 1100
force -drive {sim:/controller_top/in_bit[5]} 0 0, 0 100, 1 200, 0 300,
   0 400, 0 500, 1 600, 0 700, 1 800, 1 900, 0 1000, 0 1100
force -drive {sim:/controller_top/in_bit[4]} 1 0, 1 100, 0 200, 1 300,
   0 400, 0 500, 1 600, 0 700, 0 800, 1 900, 0 1000, 1 1100
force -drive {sim:/controller_top/in_bit[3]} 1 0, 1 100, 0 200, 1 300,
   0 400, 0 500, 1 600, 1 700, 1 800, 0 900, 0 1000, 0 1100
force -drive {sim:/controller_top/in_bit[2]} 1 0, 1 100, 0 200, 0 300,
   0 400, 0 500, 0 600, 0 700, 1 800, 1 900, 1 1000, 1 1100
force -drive {sim:/controller_top/in_bit[1]} 0 0, 0 100, 1 200, 1 300,
   1 400, 1 500, 1 600, 1 700, 0 800, 0 900, 0 1000, 0 1100
force -drive {sim:/controller_top/in_bit[0]} 1 0, 0 100, 1 200, 1 300,
   1 400, 0 500, 1 600, 1 700, 0 800, 1 900, 0 1000, 1 1100

force clk 0 0, 1 150 -r 300
force latch 0 0, 1 900 -r 1600
force reset_n 0 0, 1 1, 0 2, 1 3

run 2300
```

B.0.7 SNES Console Do Files

Do File for SNES Console

```
vsim work.SNES_reader

add wave *

force dataYellow 1 0, 0 12000
force reset_n 0 0, 1 5, 0 10
force clock 0 0, 1 200 -r 400
force reset_n 1 0, 0 5, 1 10


run 23000
```

B.0.8 Top Level Logic Do Files

Do File for Top Level only Buttons

```
vsim work.Top_lvl

add wave *

force -drive sim:/Top_lvl/latch 1 0, 0 1, 1 12 -r 24
force -freeze sim:/Top_lvl/clk 1 0, 0 1, 1 48 -r 96
force -drive sim:/Top_lvl/SNES_clk 1 0, 0 1, 1 6 -r 12
force -freeze sim:/Top_lvl/reset_n 1 0, 0 1, 1 2
force -drive sim:/Top_lvl/IR 0 905, 1 910, 0 920, 1 930, 0 940, 1 950,
    0 960, 1 970, 0 980, 1 990, 0 1000, 1 1010, 0 1020, 1 1030, 1 1040,
     0 1050, 1 1060, 0 1070, 1 1080, 0 1090, 1 2000, 0 2010, 1 2020, 0
    2030, 1 2040, 0 2050, 1 2060, 0 2070, 1 2080, 0 2090, 1 3000
force -drive sim:/Top_lvl/Button_board 111111111111 0, 111111111110
    100, 111111111101 200, 111111111011 300, 111111110111 400
force -drive sim:/Top_lvl/select 0 0, 1 890, 0 900, 1 910, 0 920

run 4000ps
```

Do File for Top Level with All Modules

```
vsim work.Top_lvl

add wave *

force -drive sim:/Top_lvl/latch 1 0, 0 1, 1 12 -r 24
force -freeze sim:/Top_lvl/clk 1 0, 0 1, 1 48 -r 96
force -drive sim:/Top_lvl/SNES_clk 1 0, 0 1, 1 6 -r 12
force -freeze sim:/Top_lvl/reset_n 1 0, 0 1, 1 2
force -drive sim:/Top_lvl/IR 1 0, 0 20, 1 30, 0 40, 1 50, 0 60, 1 70, 0
     80, 1 90, 0 100, 1 110, 0 120, 1 130, 1 140, 0 150, 1 160, 0 170,
    1 180, 0 190, 1 200, 0 210, 1 220, 0 230
force -drive sim:/Top_lvl/select 0 0, 1 1, 0 2, 1 3, 0 4

run 600ps
```

References

[1] Wiki, “Super nintendo entertainment system.” https://en.wikipedia.org/wiki/Super_Nintendo_Entertainment_System.

[2] gamesx, “Snes / super famicom joystick data.” https://gamesx.com/controldata/snesdat.htm.

[3] GAMEFAQS, “Super nintendo – guides and faqs.” https://gamefaqs.gamespot.com/snes/916396-super-nintendo/faqs/5395.