ECE 271: Design Project Report

Chase Denecke, Chris Pavlovich, John Davis, Yang Yang

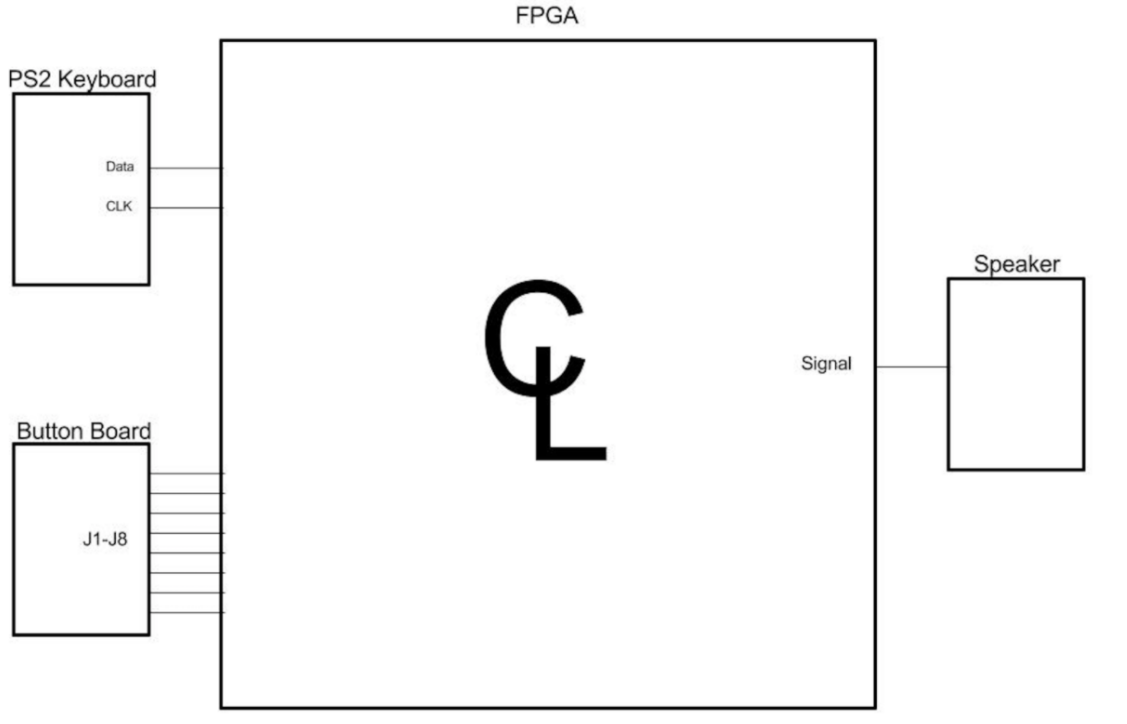
December 1, 2017

1 Introduction

The job of our final project group was to attach various controllers to a speaker so that the speaker could be made to play various frequencies using the controllers. Our three controllers were a button board with 8 buttons, a PS2 keyboard, and an SNES controller. We used two intermediate pieces of hardware to connect the controllers to the speaker: A Field Programmable Gate Array (FPGA), and a Digital-to-Analog-Converter (DAC). Our main task was to write code in SystemVerilog that would take input signals from the controllers and convert those signals into an output to the DAC.

The idea of this project was to test our FPGA programming skills in a more rigorous way than they had been in our labs. I think we can all attest that this project did so.

2 High Level



Inputs: This reads signals from either a PS_2 keyboard or an 8-button push button board, the one provided in the lab portion of this class.

Outputs: This design outputs a sine wave signal to a speaker which will play one of the eight preset frequencies, the signal of which is selected by the inputs.

Description: As shown in the diagram above, the signals are taken from either of the two input options and then sent to the FPGA. In the case of the PS2 keyboard, a data line connects the keyboard to the FPGA using one of the input pins on the FPGA. This data line provides a flow of binary inputs that are read into the logic on the FPGA. In the case of the button board, the buttons J1-J8 are connected to 8 of the input pins on the FPGA, when pressed they will send a logic low reading to the FPGA, if they are not pressed they will send a logic high. When a button is pressed, the FPGA will choose which frequency is played based on the origin of the logic low reading. The speaker has one connection with the FPGA which with send a sine wave through one of the output pins on the FPGA.

3 Input Boards

3.1 PS2 Keyboard

Figure 1: PS_2 Keyboard

floats

The first input that our design takes in is a PS2 keyboard (pictured above). The keyboard has one wire that connects to board itself to the logic. This wire is a data wire that sends data bit by bit to the logic. This type of communication protocol is known as parallel communication. Since the data is sent bit by bit, a shift register is required to pick up all the bits that are sent. This shift register then sends the N-bits of data when the register is full, then wipes the registers clear and starts reading bits in again. The shift register in our design is 11-bits because 11 bits of data represents a pressed key. Three of those eleven bits are erranous and don't have a particular use in our design. The middle eight bits hold the information for which key was pressed. However, the data does not start flowing instantly upon connection. A clock value has to be started in order for the data wire to start sending signals. The same clock value is used in the register to update the inputted data.

3.2 Button Board

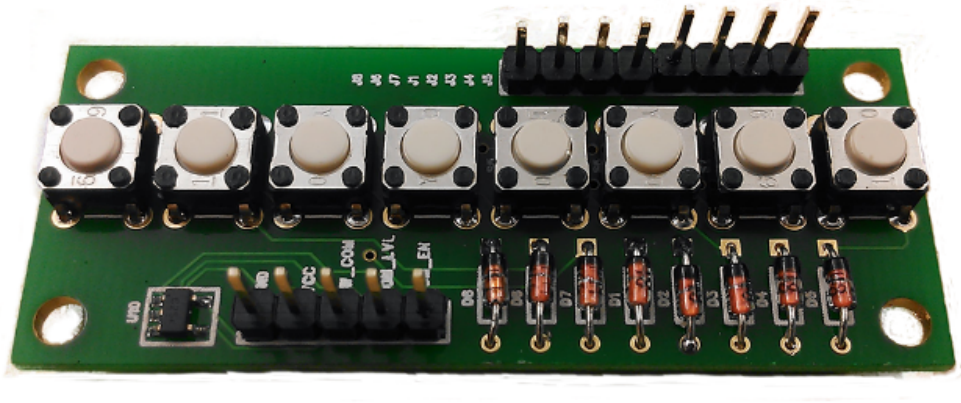
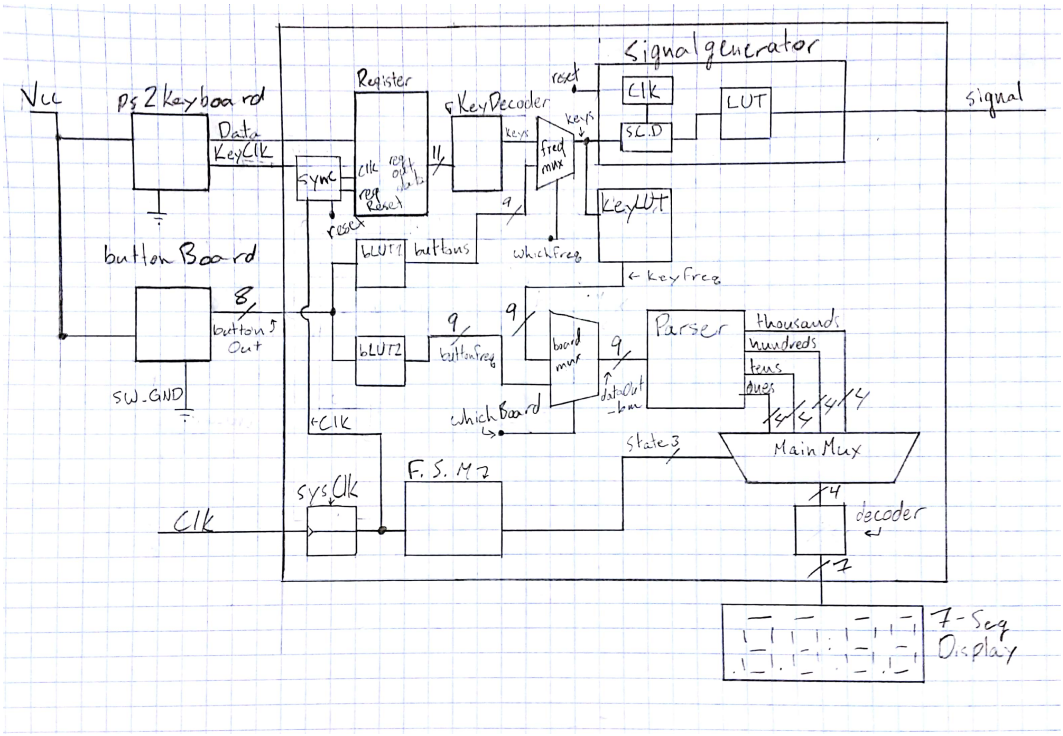


Figure 2: buttonBoard

The second input that our design is capable of reading is the 8-button push-button board. The picture above is the same model as the device used in the lab portion of this class. This is a very simple implementation of parallel communication protocol which can send 8 bits of information at a time. The controller is used in this design to switch between the 8 pre-set frequency values. There is also a command from the button board to pause and play the currently selected frequency through the speaker.

4 Modules

4.1 Top Module

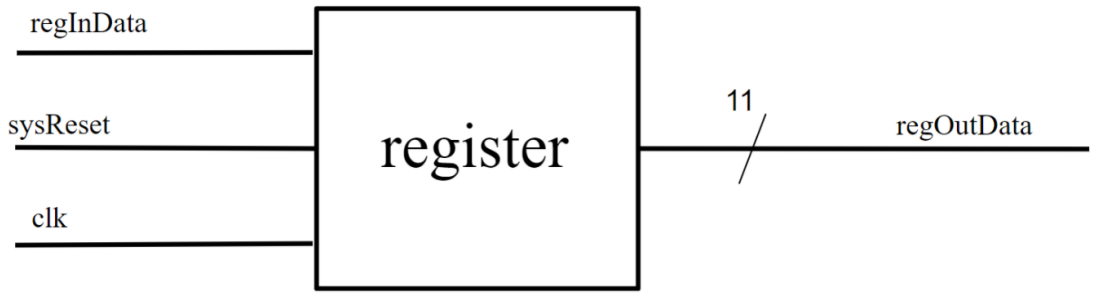


Inputs: This module intakes a 4-bit number representing the decimal value of the current digit being displayed.

Outputs: This module outputs a 7-bit code that is sent to the 7-segment display that controls which of the segments on the display are illuminated.

Description: The module is built from a case statement that looks at what the input value represents in decimal form then decides which code to send to the 7-segment display to turn on the right segments to display the decimal number.

4.2 Register

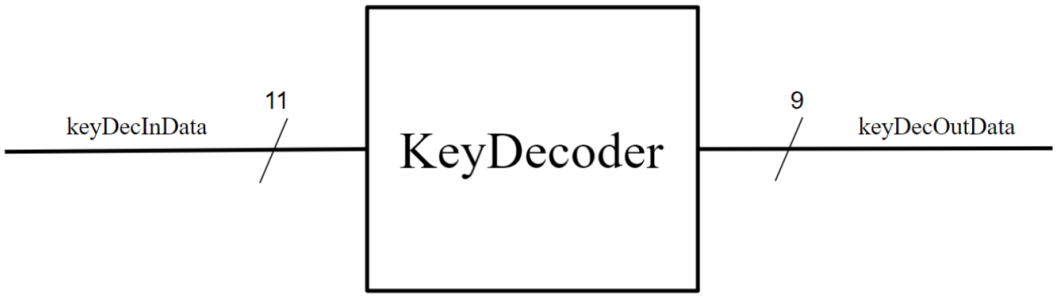


Inputs:
regInData from the PS2 keyboard;
Clk generated from the PS2 keyboard;
A reset input from clock sync module.

Outputs: 11-bit data line.

Description: The register takes the keyboard input regInData which comes 1 byte at a time; the register also takes a clk input from the keyboard. It stores 1 byte each time at the rising clock edge; after 11 ticks the 11-bit data line is output as regOutData. It also has a reset input so it can reset the values if the input data is invalid.

4.3 Keyboard Decoder

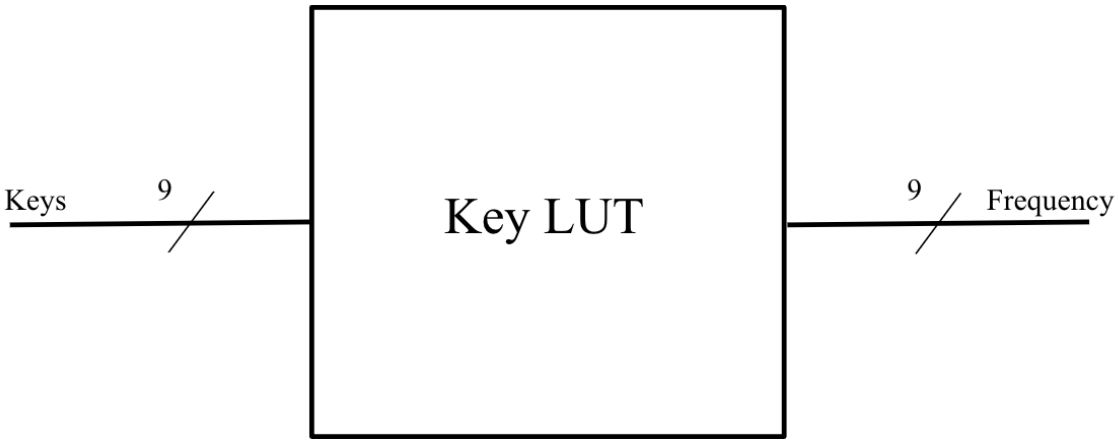


Inputs: This module intakes a 4-bit number representing the decimal value of the current digit being displayed.

Outputs: This module outputs a 7-bit code that is sent to the 7-segment display that controls which of the segments on the display are illuminated.

Description: The module is built from a case statement that looks at what the input value represents in decimal form then decides which code to send to the 7-segment display to turn on the right segments to display the decimal number.

4.4 Key Look Up Table

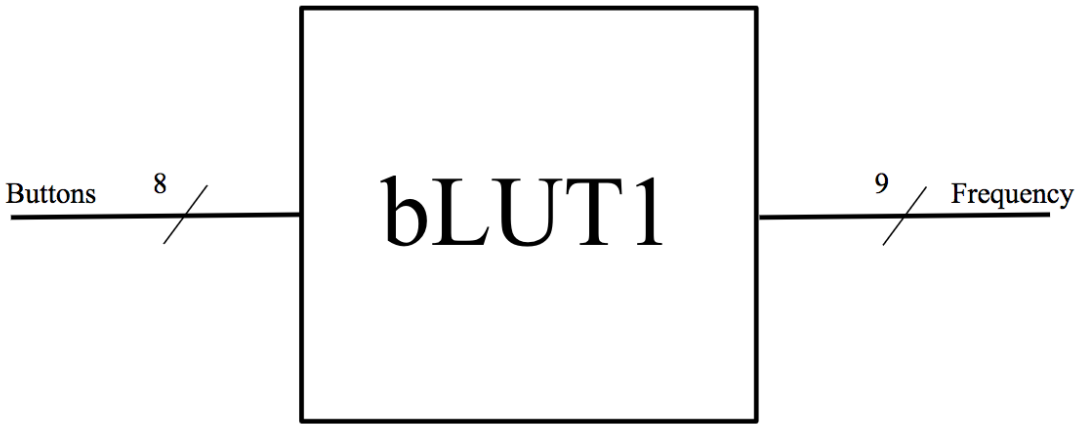


Inputs: This module inputs a 9-bit frequency key that is related to an actual frequency value.

Outputs: This module outputs a 9-bit number that represents the decimal value for the selected frequency.

Description: The module is constructed from a case statement that selects which frequency value to send out based on what frequency key was inputted. If the frequency key was inputted for the first frequency, then the output of the module would be the binary representation of the decimal value for the first frequency.

4.5 Button Look Up Table 1

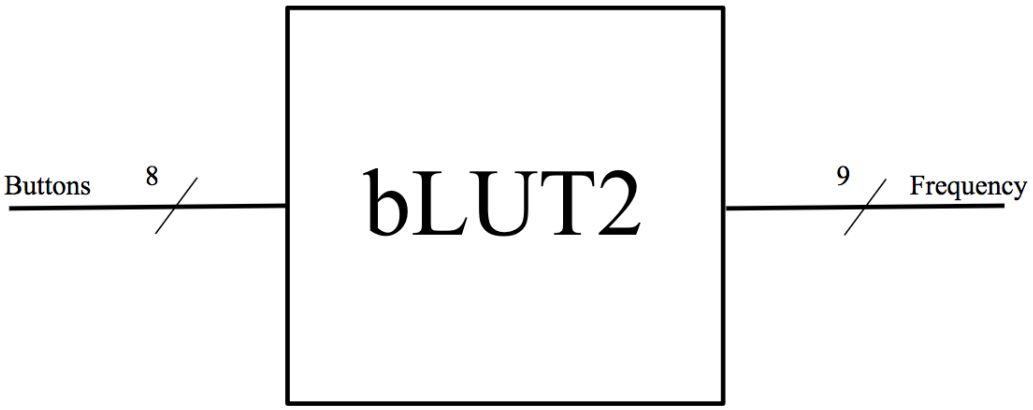


Inputs: This module intakes 8-bits of data from the button board.

Outputs: This module outputs a 9-bit frequency key value

Description: The module is built from a case statement that looks at the buttons pressed, and then sends a key value to the signal generator which lets the signal generator which frequency to output.

4.6 Button Look Up Table 2

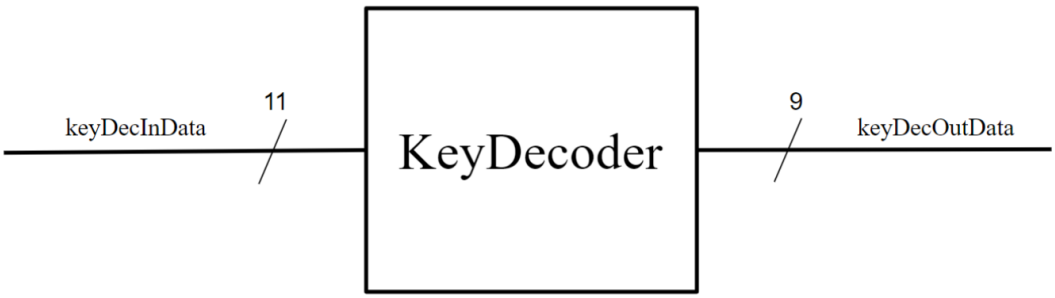


Inputs: This module intakes 8-bits of data from the button board.

Outputs: This module outputs a 9-bit value that represents the decimal value of the frequency selected.

Description: The module is built from a case statement that looks at what buttons are pressed, then decides which decimal frequency value corresponds with the button pressed, then sends the frequency value in binary form so it can be displayed by the 7-segment display decoder.

4.7 Frequency Mux

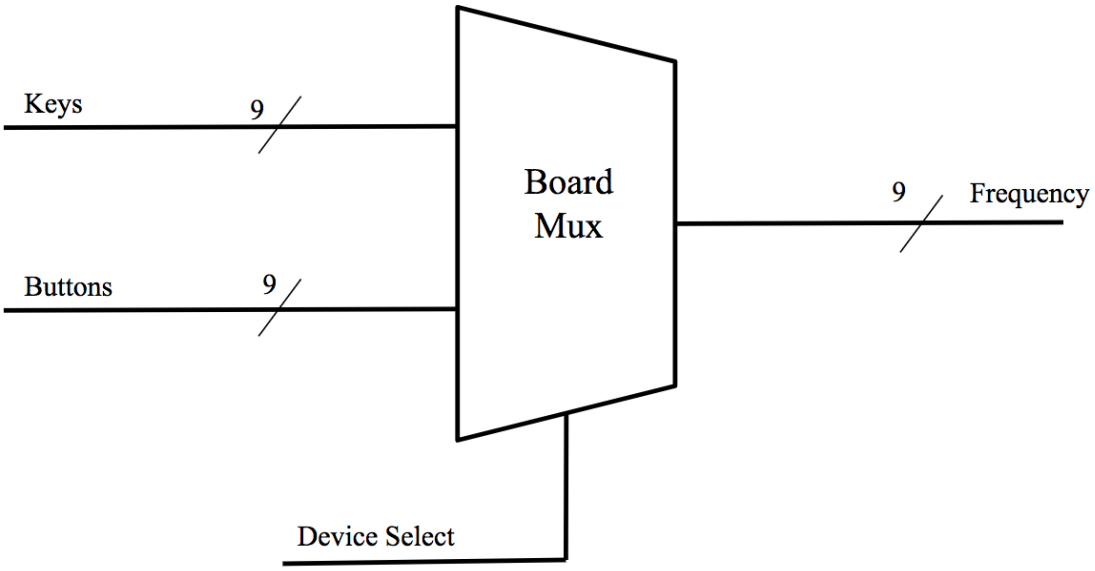


Inputs: keyFreqIn from the KeyDecoder module and bbFreqIn from the bLUT1 module;

Outputs: dataOut$_fm$;

Description: This multiplexer takes the output from the KeyDecoder (keyDecOutData) and the output from bLUT1 module (buttons), which are both binary values; then it selects one between the two to pass straight through the output (dataOut$_fm$). When the 1-bit input whichFreqOut is '0', the keyboard data is being passed; and when it is '1', the button board data is being passed. The output goes to keyLUT which determines the binary value of the frequency being displayed on the seven segment.

4.8 Board Mux



Inputs: This module intakes 9 bits representing a decimal frequency value from both the keyboard and the button board. It also intakes a select value.

Outputs: This module outputs 9 bits representing a decimal frequency value.

Description: The module is built from a case statement that determines which frequency value gets sent to the 7-segment display decoder, based on what the input type is active.

4.9 Signal Generator Top Level

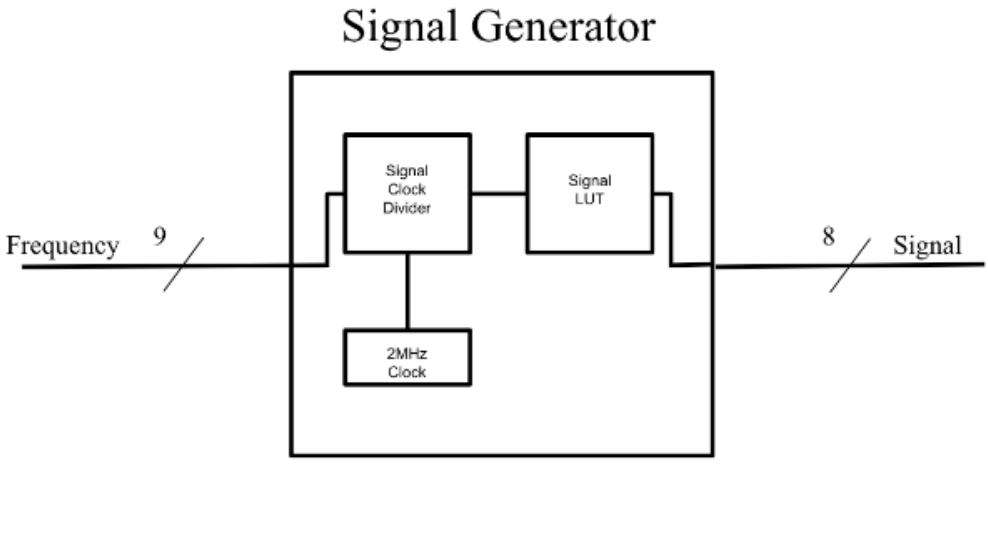


Figure 3: sigGenTopLevel

The signal generator is told which key is pressed by the decoder. It then outputs an 8 bit binary number to the DAC representing the height of the sound wave at each successive moment in time. A new 8 bit value is sent to the DAC a few hundred thousand times per second.

4.10 Signal Clock Divider

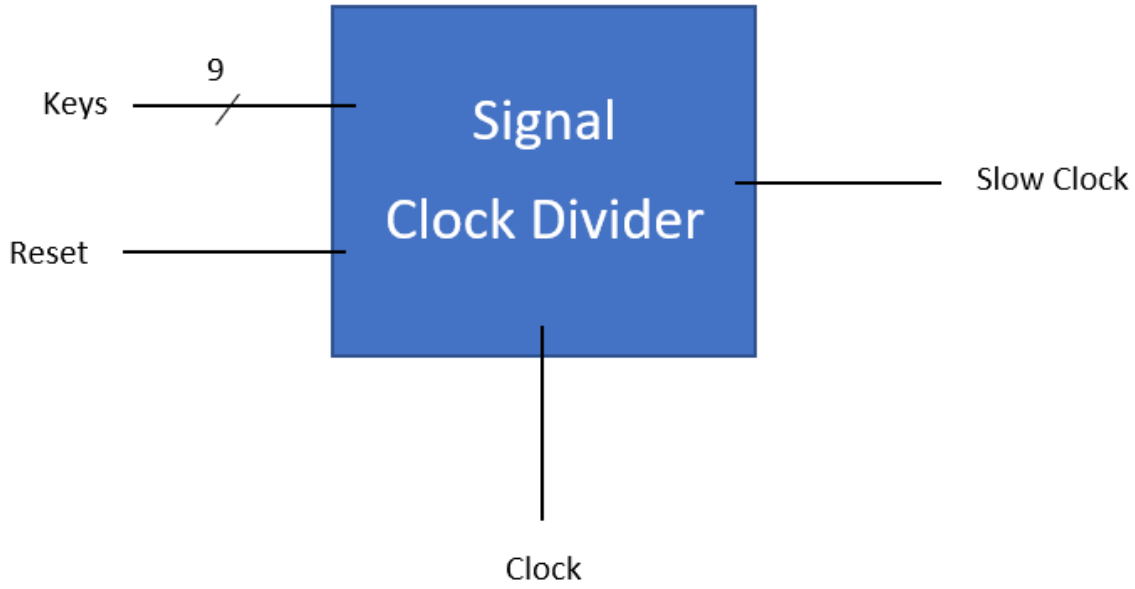


Figure 4: Signal Clock Divider

The signal clock divider sub-module takes the built-in 2.08 Mhz clock signal and slows it down by an amount dependent on which frequency was selected. It accomplishes this by counting a specific number of clock edges from the 2.08 Mhz clock, and flipping its own output clock signal when it reaches that specific number.

This specific number is determined with a look-up table: each button corresponds with a single entry in the lookup table.

4.11 Signal Look Up Table

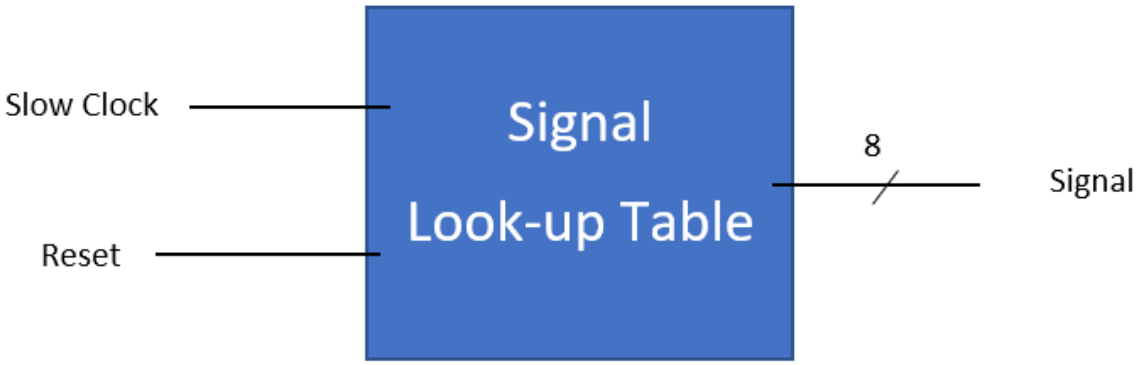


Figure 5: sigLUT

The signal look-up table submodule takes the slow clock as an input and outputs the 8-bit number representing the height of the sine wave. Filling the entries of the lookup table was one of the hardest parts of the project, because there is no way to directly compute sine using SystemVerilog.

Our solution was to write a program in C++ to generate the verilog code. The code from that file

was then copied and pasted into our program. The C++ code that generated said code is included below.

4.12 System Clock

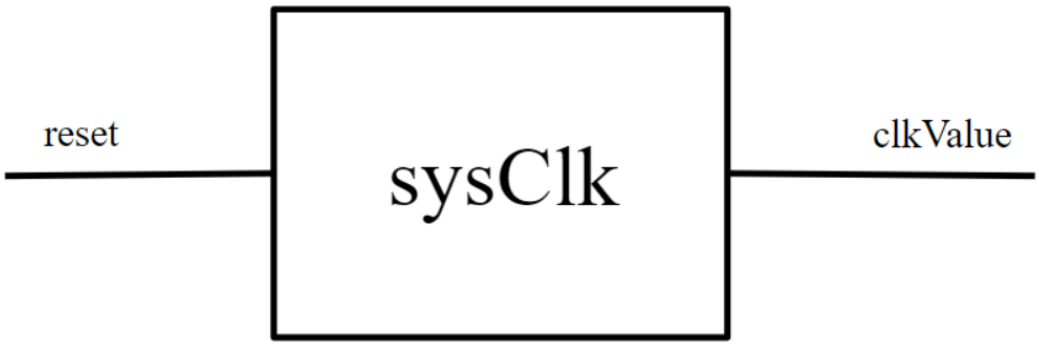


Figure 6: Sync

Inputs: reset;

Outputs: clkValue;

Description: This module accesses the FPGA's chip oscillator and generate the 2.08 MHz system clock output.

4.13 Free State Machine

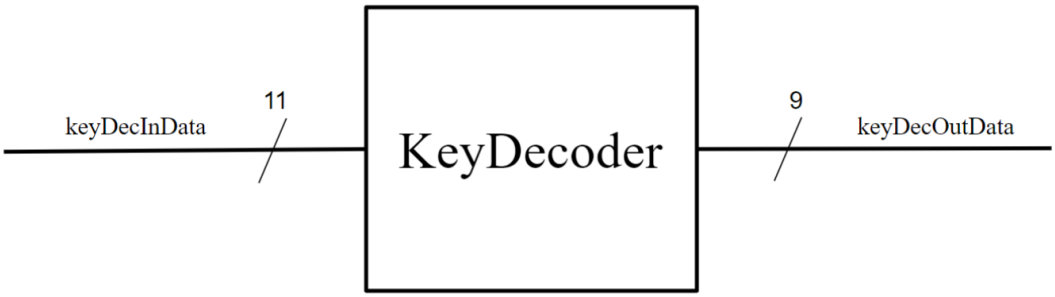


Figure 7: Sync

nputs: This module intakes a clock value, in the design the clock is set to the FPGA's oscillating frequency of 2.08MHz.

Outputs: This module outputs a 3-bit state value.

Description: This module cycles through four different states, one for each digit that can be displayed on the 7-segment display, with the initial state set to the first digit. On the rising edge of the clock value, the state is changed which changes the digit which is illuminated on the display.

5 Appendix

5.1 Code

1. Top Module

```
module topModule (
        input logic ps2data, // 1 byte input from PS2, one at a time
        input logic keyClk,
        input logic [7:0] buttonBoard,
        output logic [6:0] segments,
        output logic [7:0] signal
);

/*****************************
All internal variables:
*****************************/
        //sync
        logic keyClk2;
        logic regReset;
        //sysClk
        logic sysClkReset;      //use this for signalgenerater as well
        logic clkValue;
        //register
        logic [10:0] regOutData;
        //keyDecoder
        logic [8:0] keyDecOutData;
        //bLUT1
        logic [8:0] buttons;
        //freqMux
        logic whichFreqOut;
        logic [8:0] dataOut_fm;
        //signalgenerator
        logic signalGeneratorClk;
        //keyLUT
        logic [8:0] keys;
        logic [8:0] keyFreq;
        //bLUT2
        logic [8:0] buttonFreq;
        //boardMux
        logic whichBoard;
        logic [8:0] dataOut_bm;
        //digitalseparator
        logic [3:0] thousands, hundreds, tens, ones;
        //state_machines
        logic [2:0] state;
        //mainMux
        logic [3:0] displayDigit;
```

```
//----------------------------------------------------------------------------//
        //logic keyClk2;
        //logic regReset;
        sync clkSync (
                .regReset(regReset), //output
                .keyClk2(keyClk2), //output
                .keyClk1(keyClk),
                .sysClk(clkValue),
                .reset(sysClkReset)
        );
//----------------------------------------------------------------------------//
        //logic sysClkReset
        //logic clkValue;
        sysClk sysClkMod (
                .clkValue(clkValue), //output
                .reset(sysClkReset)
        );

//----------------------------------------------------------------------------//
        //logic [10:0] regOutData;
        register r (
                .regInData(ps2Data),
                .clk(keyClk2),
                .sysReset(regReset),
                .regOutData(regOutData) //output
        );
//----------------------------------------------------------------------------//
        //logic [8:0] keyDecOutData;
        KeyDecoder KD (
                .keyDecInData(regOutData),
                .keyDecOutData(keyDecOutData) //output
        );

//----------------------------------------------------------------------------//
        //logic [8:0] buttons;
        bLUT1 b1(
                .buttonBoard(buttonBoard),
                .buttons(buttons) //output
        );

//----------------------------------------------------------------------------//
        //logic whichFreqOut;
        //logic [8:0] dataOut_fm;
        freqMux fm (
                .keyFreqIn_fm(keyDecOutData),
                .bbFreqIn_fm(buttons),
                .dataOut_fm(dataOut_fm),
                .whichFreqOut(whichFreqOut)
        );

//----------------------------------------------------------------------------//
        //logic signalGeneratorReset;
        signalgenerator sg(
                .keys(dataOut_fm),
                .reset(sysClkReset),
                .clk(clkValue), //2.08 MHz CLK
                .signal(signal)
        );

//----------------------------------------------------------------------------//
    //logic [8:0] keys;
        //logic [8:0] keyFreq;
        keyLUT kl(
                .keys(dataOut_fm),
                .keyFreq(keyFreq)
        );

//----------------------------------------------------------------------------//
        //logic [8:0] buttonFreq;
        bLUT2 b2(
                .buttons(buttonBoard),
                .buttonFreq(buttonFreq)
        );
```

```
//------------------------------------------------------------------------//
        //logic whichBoard;
        //logic [8:0] dataOut_bm;
        boardMux bm (
                .keyFreq_bm(keyFreq),
                .bbFreq_bm(buttonFreq),
                .dataOut_bm(dataOut_bm),
                .whichBoard(whichBoard)
        );

//------------------------------------------------------------------------//
        //logic [3:0] thousands, hundreds, tens, ones;
        DigitSeparator ds(
                .displayValue(dataOut_bm),
                .thousands(thousands),
                .hundreds(hundreds),
                .tens(tens),
                .ones(ones)
        );
//------------------------------------------------------------------------//
        //logic [2:0] state;
        state_machine sm(
                .reset_n(sysClkReset),
                .clk_i(clkValue),
                .state(state)
        );
//------------------------------------------------------------------------//
        //logic [3:0] displayDigit;
        mainMux mm(
                .state(state),
                .digit(displayDigit),
                .thousands(thousands),
                .hundreds(hundreds),
                .tens(tens),
                .ones(ones)
        );
//------------------------------------------------------------------------//
        segDecoder sd(
                .displayIn(displayDigit),
                .segs(segments)
        );

endmodule
```

2. Register

```
module register(
        input logic regInData,                  //data line from ps2 keyboard
    input logic clk,                        //clock line from ps2 keyboard
    input logic sysReset,                   //set to 1 every 11 ticks
    output logic [10:0] regOutData              //bus to decoder
        );

        logic reset;
    logic sysCount;                             //tracks clock ticks
    reg [10:0] rawData;                                     //stores register data

    always_ff @ (posedge clk) begin         //keyboard clock line drives the module

                if(sysReset) begin
                        reset <= sysReset;
                end

                if(!reset) begin                        //if reset is set
                        sysCount <= sysCount + 1;       //count += 1
                end

        if(sysCount == 11)  begin           //writes to decoder every 11 clock ticks
            regOutData <= rawData;          //pass the now full register along bus to decoder
            reset <= 1;                         //set the reset
        end

        if(rawData == 'b0XXXXXXXX1) begin       //check for start and stop bits
            rawData <= rawData >> 1;        //shift data right
            rawData[0] <= regInData;        //pass data to first bit of register
        end

        if(reset) begin                         //if reset is set
            sysCount <= 0;                  //set count to 0
            reset <= 0;                         //reset the reset
        end
    end

endmodule
```

3. Key Decoder

```
module KeyDecoder(
                input logic [10:0] keyDecInData, //raw keyboard data
                output logic [8:0] keyDecOutData //display value for parser
        );

        always_comb
                case (keyDecInData)
                        'b00001011001: keyDecOutData = 'b000000001; //Key1
                        'b00001111011: keyDecOutData = 'b000000010; //Key2
                        'b00010011001: keyDecOutData = 'b000000100; //Key3
                        'b00010010101: keyDecOutData = 'b000001000; //Key4
                        'b00010111011: keyDecOutData = 'b000010000; //Key5
                        'b00011011011: keyDecOutData = 'b000100000; //Key6
                        'b00011110101: keyDecOutData = 'b001000000; //Key7
                        'b00011111001: keyDecOutData = 'b010000000; //Key8
                        'b00100011001: keyDecOutData = 'b100000000; //Key9
                endcase

endmodule
```

4. Key LUT

```
module keyLUT (
    input logic [8:0] keys,
    output logic [8:0] keyFreq
    );

    always_comb
        case (keys)
            'b000000001: keyFreq = 'b011011100; //Key1
                        'b000000010: keyFreq = 'b011110111; //Key2
                        'b000000100: keyFreq = 'b100000110; //Key3
                        'b000001000: keyFreq = 'b100100110; //Key4
                        'b000010000: keyFreq = 'b101001010; //Key5
                        'b000100000: keyFreq = 'b101011101; //Key6
                        'b001000000: keyFreq = 'b110001000; //Key7
                        'b010000000: keyFreq = 'b110111000; //Key8
                        'b100000000: keyFreq = 'b000000000; //Key9
        endcase

endmodule
```

5. Button LUT 1

```
module bLUT1(
    input logic [7:0] buttonBoard,
    output logic [8:0] buttons
    );

    always_comb
        case (buttonBoard)
                        'b00000001: buttons = 'b000000001; //Key1
                        'b00000010: buttons = 'b000000010; //Key2
                        'b00000011: buttons = 'b000000011; //Key3
                        'b00000100: buttons = 'b000000100; //Key4
                        'b00000101: buttons = 'b000000101; //Key5
                        'b00000110: buttons = 'b000000110; //Key6
                        'b00000111: buttons = 'b000000111; //Key7
                        'b00001000: buttons = 'b000001000; //Key8
                        'b10000000: buttons = 'b000001001; //Key9
                endcase

endmodule
```

6. Button LUT 2

```
module bLUT2(
    input logic [7:0] buttons,
    output logic [8:0] buttonFreq
    );

    always_comb
        case (buttons)
                        'b00000001: buttonFreq = 'b011011100; //Key1
                        'b00000010: buttonFreq = 'b011110111; //Key2
                        'b00000011: buttonFreq = 'b100000110; //Key3
                        'b00000100: buttonFreq = 'b100100110; //Key4
                        'b00000101: buttonFreq = 'b101001010; //Key5
                        'b00000110: buttonFreq = 'b101011101; //Key6
                        'b00000111: buttonFreq = 'b110001000; //Key7
                        'b00001000: buttonFreq = 'b110111000; //Key8
                        'b00001001: buttonFreq = 'b000000000; //Key9 (pause)
                endcase

endmodule
```

7. Button Frequency Mux

```
module freqMux (
        input logic [8:0] keyFreqIn_fm,
        input logic [8:0] bbFreqIn_fm,
        input logic whichFreqOut,
        output logic [8:0] dataOut_fm
        );

        always_comb
                case (whichFreqOut)
                        0: dataOut_fm = keyFreqIn_fm;
                        1: dataOut_fm = bbFreqIn_fm;
                endcase

endmodule
```

8. Board Mux

```
module boardMux (
        input logic [8:0] keyFreq_bm,
        input logic [8:0] bbFreq_bm,
        output logic [8:0] dataOut_bm,
        input logic whichBoard
        );

        always_comb
                case (whichBoard)
                        0: dataOut_bm = keyFreq_bm;
                        1: dataOut_bm = bbFreq_bm;
                endcase

endmodule
```

9. Signal Top Level

```
module signalgenerator( input logic [8:0] keys,
                                                input logic reset, clk,
                                                output logic [7:0] signal);

                //logic clk;
                logic LUTclk;

        //This is one of those magical built in modules that SystemVerilog gives us
//      OSCH #("2.08") osc_int (        //"2.08" specifies the operating frequency, 2.08 MHz.
//                                                                              //Other clock frequencies can be found in the MachX02's
//                      .STDBY(1'b0),                   //Specifies active state
//                      .OSC(clk),                              //Outputs clock signal to 'clk' net
//                      .SEDSTDBY());                   //Leaves SEDSTDBY pin unconnected

        SignalClockDivider divider(
                        .keys(keys),
                        .clk(clk),
                        .reset(reset),
                        .signalLUTclk(LUTclk));

        SignalLUT lut(
                        .clk(LUTclk),
                        .reset(reset),
                        .signal(signal));

endmodule
```

10. Signal Clock Divider

```
module signalgenerator( input logic [8:0] keys,
                                            input logic reset, clk,
                                            output logic [7:0] signal);

                //logic clk;
                logic LUTclk;

        //This is one of those magical built in modules that SystemVerilog gives us
//      OSCH #("2.08") osc_int (        //"2.08" specifies the operating frequency, 2.08 MHz.
//                                                                      //Other clock frequencies can be found in the MachX02's
//                      .STDBY(1'b0),                   //Specifies active state
//                      .OSC(clk),                              //Outputs clock signal to 'clk' net
//                      .SEDSTDBY());                   //Leaves SEDSTDBY pin unconnected

        SignalClockDivider divider(
                        .keys(keys),
                        .clk(clk),
                        .reset(reset),
                        .signalLUTclk(LUTclk));

        SignalLUT lut(
                        .clk(LUTclk),
                        .reset(reset),
                        .signal(signal));

endmodule
```

11. Signal LUT

```
module SignalLUT(       input logic clk, reset,
                                        output logic [7:0] signal
                                        );
        int counter;
        //Counter that generates the index for the lookup table
        always_ff @(posedge clk, posedge reset)
                        if (reset) counter <= 0;
                        else if (counter == 524) counter <= 0;
                        else counter <= counter + 1;

//lookup table. Values represent the height of a sine wave at a given
//moment in time. Total cycle is 525 units of time long.
        always_comb
                case(counter)
                        0: signal <= 8'b10000000;
                        1: signal <= 8'b10000001;
                        2: signal <= 8'b10000011;
                        3: signal <= 8'b10000100;
                        4: signal <= 8'b10000110;
                        5: signal <= 8'b10000111;
                        6: signal <= 8'b10001001;
                        7: signal <= 8'b10001010;
                        8: signal <= 8'b10001100;
                        9: signal <= 8'b10001101;
                        10: signal <= 8'b10001111;
                        11: signal <= 8'b10010000;
                        12: signal <= 8'b10010010;
                        13: signal <= 8'b10010011;
                        14: signal <= 8'b10010101;
                        15: signal <= 8'b10010110;
                        16: signal <= 8'b10011000;
                        17: signal <= 8'b10011001;
                        18: signal <= 8'b10011011;
                        19: signal <= 8'b10011100;
                        20: signal <= 8'b10011110;
                        21: signal <= 8'b10011111;
                        22: signal <= 8'b10100001;
                        23: signal <= 8'b10100010;
                        24: signal <= 8'b10100100;
                        25: signal <= 8'b10100101;
```

12. System Clock

```
module sysClk(
        output logic clkValue,
    );

    logic clk;

    assign clkValue = clk;

    OSCH #("2.08") osc_int (
        .STDBY(1'b0),
        .OSC(clk),
        .SEDSTDBY());

endmodule
```

13. State Machine

```
module state_machine( //example of a Moore type state machine
        input logic clk_i,
        input logic reset_n,

        output logic [2:0] state //The state outputted by this state machine
                );

                //next state register
                logic [2:0] state_n;

                //each possible value of the state register is given a unique name for easier use later
                parameter S0 = 3'b000; //First digit
                parameter S1 = 3'b001; //Second digit
                parameter S2 = 3'b011; //Third digit
                parameter S3 = 3'b100; //Fourth digit

                //asynchronous reset will set the state to the start, S0, otherwise, the state is changed
                //on the positive edge of the clock signal
                always_ff @ (posedge clk_i, negedge reset_n)
                        begin
                                if(!reset_n)
                                        state = S0;
                                else
                                        state = state_n;

                        end

                //this section defines what the next state should be for each possible state. in this
                //implementation, it simply rotates through each state automatically
                always_ff @ (*)
                        begin
                                case(state)
                                        S0: state_n = S1;
                                        S1: state_n = S2;
                                        S2: state_n = S3;
                                        S3:     state_n = S0;



                                        default: state_n = S0;
                                endcase
                        end
endmodule
```

14. Parser

```
module DigitSeparator(
  input logic [8:0] displayValue, //
  output logic [3:0] thousands,     //the MSB digit
  output logic [3:0] hundreds,      //the 100's digit
  output logic [3:0] tens,          //the 10's digit
  output logic [3:0] ones          //the LSB digit
 );

   assign thousands = (displayValue / 1000) % 10; //MSB Display
   assign hundreds = (displayValue / 100) % 10;
   assign tens = (displayValue / 10) % 10;
   assign ones = displayValue % 10;

endmodule
```

15. Main Mux

```
module mainMux(
        input logic [3:0] thousands, hundreds, tens, ones,
        input logic [2:0] state,
        output logic [3:0] digit
);
        parameter S0 = 3'b000; //First digit
        parameter S1 = 3'b001; //Second digit
        parameter S2 = 3'b011; //Third digit
        parameter S3 = 3'b100; //Fourth digit
        always
        begin
                case(state)
                        S0: digit = thousands;
                        S1: digit = hundreds;
                        S2: digit = tens;
                        S3: digit = ones;
                endcase
        end
endmodule
```

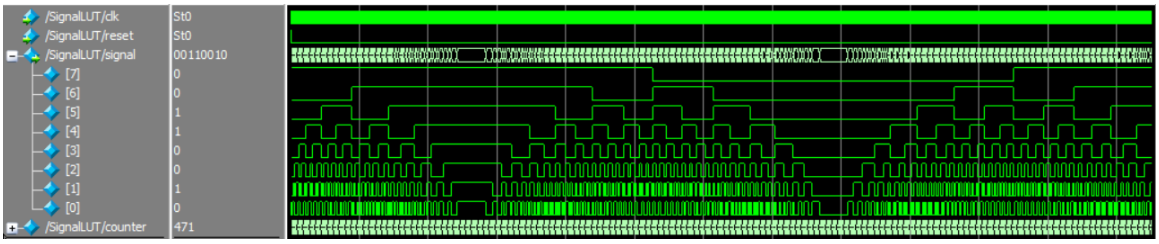
16. Segment Decoder

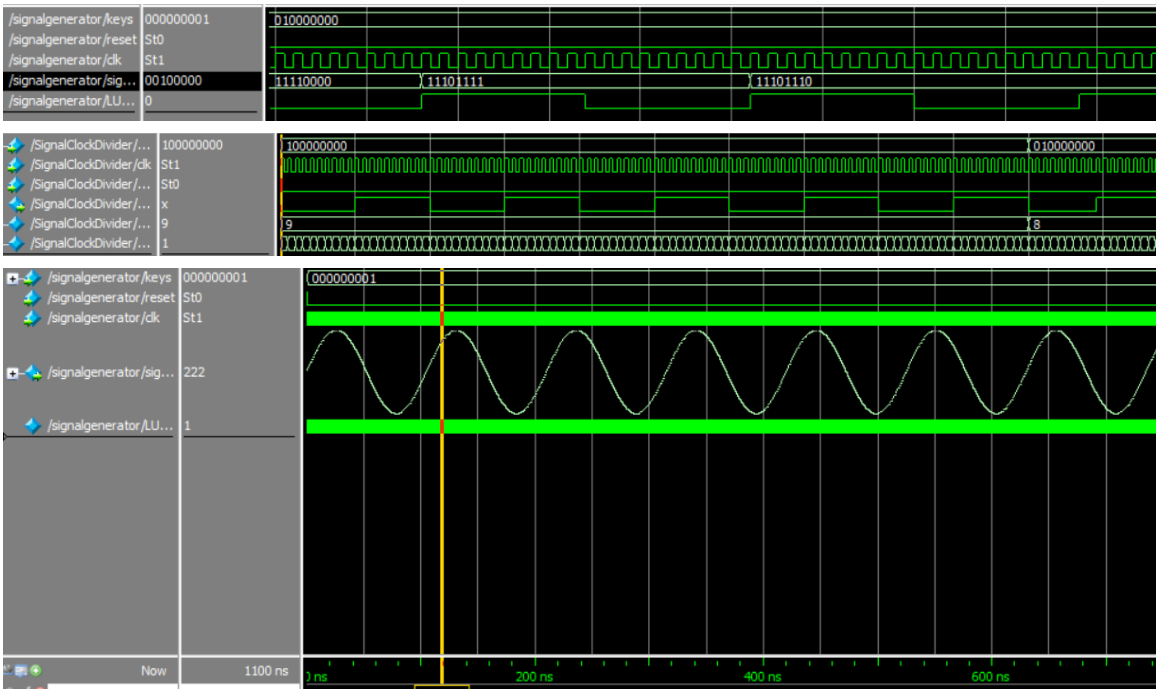
```
module segDecoder(
        input logic [3:0] displayIn, //inputs
        output logic [6:0] segs //display value from parser
        );

        always_comb
                case (displayIn)
                        0: segs = ~'b0111111;
                        1: segs = ~'b0000110;
                        2: segs = ~'b1011011;
                        3: segs = ~'b1001111;
                        4: segs = ~'b1100110;
                        5: segs = ~'b1101101;
                        6: segs = ~'b1111101;
                        7: segs = ~'b0000111;
                        8: segs = ~'b1111111;
                        9: segs = ~'b1110011;
                endcase
endmodule
```

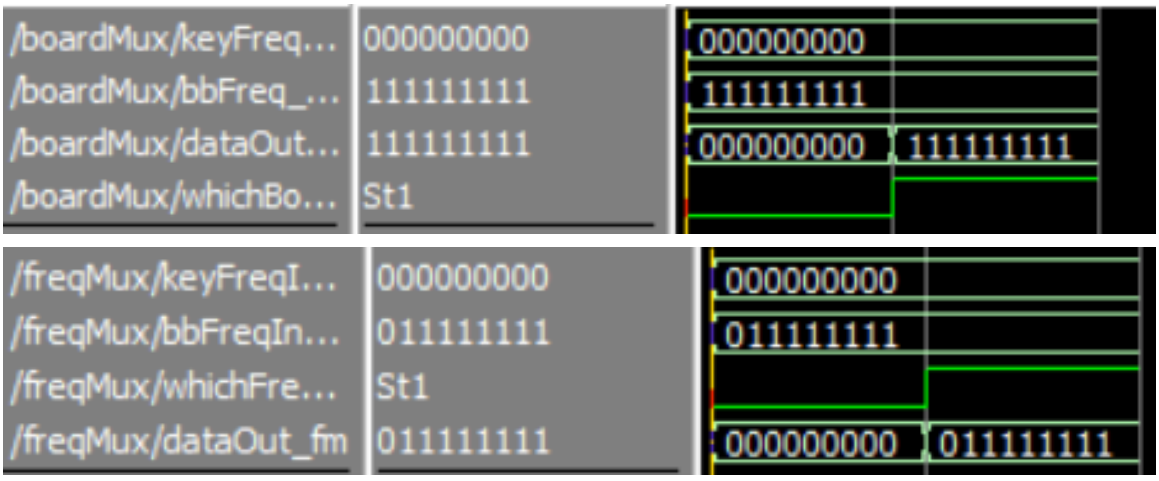
5.2 Testing

1. Signal Generator





2. Segment Decoder




```
VSIM 36> force bbFreqIn_fm 11111111 @ 0
VSIM 37> force keyFreqIn_fm 00000000 @ 0
VSIM 38> force whichFreqOut 0 @ 0, 1 @ 100
VSIM 39> run 200
```

```
VSIM 42> add wave *
# ** Warning: (vsim-WLF-5000) WLF file c
#           File in use by: Chase's Lapt
#           Attempting to use alternate
# ** Warning: (vsim-WLF-5001) Could not
#           Using alternate file: ./wlft
VSIM 43> force keyFreq_bm 00000000 @ 0
VSIM 44> force keyFreq_bm 000000000 @ 0
VSIM 45> force bbFreq_bm 111111111 @ 0
VSIM 46> force whichBoard 0 @ 0, 1 @ 100
VSIM 47> run 200
```