ECE 271 Final Project

PS/2 Keyboard Controlled Music Box

Hugh Hazim
Amanda Jung
Raymond Lee

June 6, 2022

Table of Contents

1 Introduction

The purpose of this project is to design a system that would implement a Personal System/2 (PS/2) keyboard piano through the usage of concepts that were taught in digital logic design. The PS/2 port connection that was used in the 1960s to connect keyboards and mice to computers. This system would need to be able to read inputs from the PS/2 keyboard and output different frequency waveforms for different keys that are pressed. This output waveform would be connected to a speaker, to be able to hear the music notes that were assigned to the keyboard keys. The keys that are used in this project are A, S, D, F, W, E, R, and T, for a total of eight keys. The notes will be middle A, B, C, D, E, F, G, and high A, respectively, which ranges the frequencies from 440Hz to 880Hz.

The foundation for this system will be a field programmable gate array (FPGA), or more specifically, the DE10-Lite. In order to design this system, understanding of the PS/2 protocol and the fundamentals of digital logic design are essential. For the FPGA to interact with the PS/2 keyboard, a logic must be designed to take inputs from the serial data outputs of the PS/2 keyboard. This will be done through the usage of a register that functions similarly to a serial to parallel shift register, where one serial bit is read at a time before shifting to the next bit. The keys of the PS/2 keyboard each have their own unique scancode, and this will allow for a decoder to be designed to control the frequency that the key will result in outputting.

In order for the speaker to play the notes, an op-amp, which in this project was chosen to be the LM386, needs to be used. Components will need to be chosen so that the gain of the op-amp fits what is needed for the speaker. This will allow for the signal from the FPGA to be amplified enough to hear the speaker playing the note.

Figure 1: Image of PS/2 Connector

2 High Level Description

2.1 Top Level Hardware Diagram

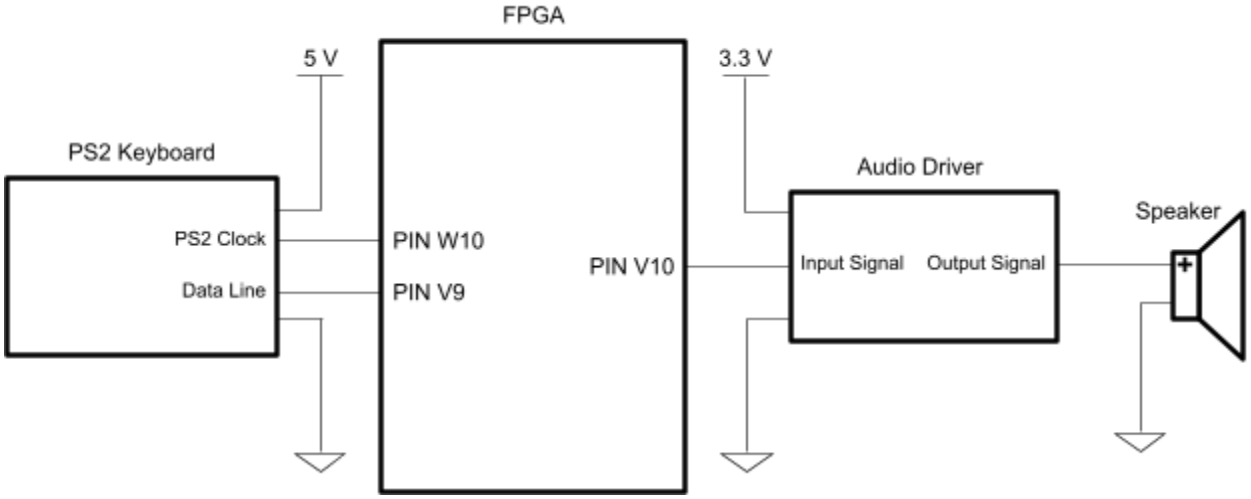


Figure 2: Top Level Hardware Diagram

Inputs: This takes a clock signal and data line input from a PS/2 keyboard and a 50 MHz clock signal and a reset input from the FPGA
Outputs: This outputs a frequency to a speaker corresponding to one of the eight possible notes

Description: The PS/2 keyboard and audio driver are both connected to the FPGA. The PS/2 clock signal and data line are connected pins W10 and V9 respectively. The audio driver is connected to pin V10. The output signal of the audio driver is inputted into the speaker. The FPGA and audio driver are both connected by VCC to 3.3V and ground to ground while the PS/2 keyboard is connected by VCC to 5V and ground to ground.

2.2 FPGA Usage Levels and Pins Used

Inputs/Outputs	Pins Used
50 MHz Clock	PIN_P11
Reset	PIN_F15
PS/2 Clock	PIN_W10
PS/2 Data	PIN_V9
Note Output	PIN_V10

Figure 3: Pins Used

Usage Levels: The FPGA used 264 / 49,760 logic elements, 156 registers, and 5 / 360 pins. In total the usage levels of the FPGA is only a little over 1%.

2.3 Top Level HDL Schematic

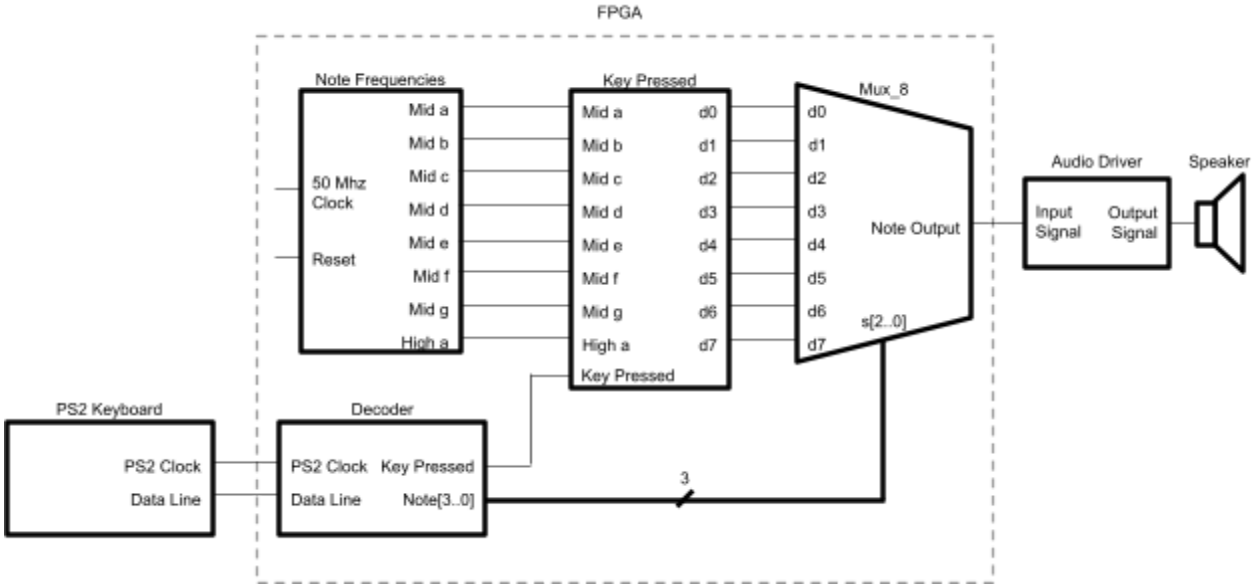


Figure 4: Top Level HDL Module Diagram

Inputs: This takes a clock signal and data line input from a PS/2 keyboard and a 50 MHz clock signal and a reset input from the FPGA
Outputs: This outputs a frequency to a speaker corresponding to one of the eight possible notes

Description: The 50 MHz clock signal from the FPGA is inputted into the note frequencies module. The clock signal is divided by eight different values which results in different frequencies that correspond to the eight notes in the middle octave (440 Hz to 880 Hz). The PS/2 clock and data line are inputted into the decoder module whenever a key is pressed on the PS/2 keyboard. The clock will increment a counter that determines when the data from the data line is read into the scancode. After the value for the 8 bit scancode is determined, it will be decoded into a 3 bit value that acts as the select input of an 8 input mux that determines which note will be played. If the scancode is not recognized as one of the chosen keys then the key pressed output will output a logic low value. This value is taken into the key pressed module along with the eight different note frequencies. If an unrecognized key is pressed then all of the outputs of this module will be a constant logic low output resulting in no sound from the speaker. If one of the recognized keys is pressed then the corresponding note will propagate through to the 8 input mux which will output a specific note based on the key that was pressed on the PS/2 keyboard.

3 Controller Descriptions

3.1 PS/2 Controller

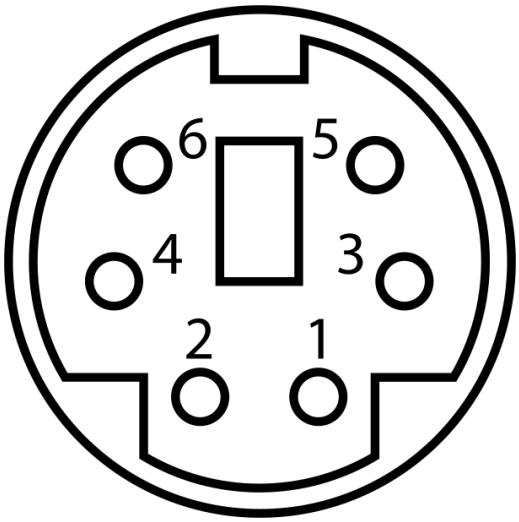


Figure 5: Female connector of PS/2

Pins of PS/2 Connector	Connection to FPGA
Pin 1: Data	Keyboard Decoder Pin
Pin 2: N/A	N/A
Pin 3: GND	Ground
Pin 4: Vcc	5V I/O of FPGA
Pin 5: CLK	Keyboard Decoder Pin
Pin 6: N/A	N/A

Figure 6: Description of the pins on the PS/2 connector and its connections to the FPGA

The keyboard being used is taking the inputs of the key presses through the six pins in the male PS/2 connector. The pins are each connected to the FPGA and are shown in the figure above, pin 1 is the data line that will give the information to the FPGA, pin 3 represents the ground line, pin 5 is connected to a VCC of 5 volts, and pin 4 is the clock signal generated by the keyboard. Pins 2 and 6 are not used for the inputs into the FPGA for this controller. As discussed earlier the keyboard buttons used to be decoded for the specific notes are A, S, D, F, W, E, R, and T.

The pins are all connected to the DE10-Lite GPIO 40-pin expansion headers. The GPIO has two VCC inputs of 5V and 3.3V that are both used and two ground pins. The 5V pin of the GPIO input is used for pin 5, and the ground is connected to pin 3. Pin 1 and pin 4 are connected to arbitrary GPIO pins that are not the four aforementioned pins for voltage and ground. The pins used will all be controlled using Quartus Prime Lite and assigned accordingly onto the FPGA.

The data input is read from the keyboard data line, which is a serial line, which needs to be taken into parallel to be used in this Keyboard Piano system. There are two lines from the keyboard, data and clock, which are both open collector connections. This means that the idle state of the bus lines is high. In this way, data is read on the falling edge of the clock signal. The data consists of 11 bits:

1. Start bit (always 0)
2-8. Data0 to Data7
9. Parity bit (XOR of all the data inputs)
11. Stop bit (always 1)

The waveforms for the keyboard outputs from the PS/2 connector are shown below.

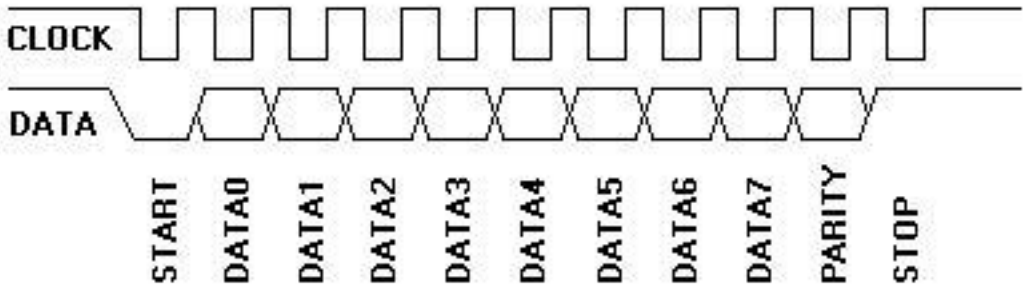


Figure 7: Waveform for the two Serial lines from PS/2 Keyboard: clock and data.

The least significant bit of the scancode is read first, with DATA0 being the least significant bit and DATA7 being the most significant bit. The 8-bits of data represent the key that is pushed. The start bit is always 0, and the stop bit is always 1. The parity is often used for error handling and catching, but was not critical in the implementation of the PS/2 Keyboard piano for this project.

For each key on the PS/2 keyboard, there is a unique scancode. This is the 8 bits of data that are read in the data line. Below shows the scancodes that are used for this project.

Key	Binary Scancode	Hexadecimal	Decimal
a	00011100	1C	28
s	00011011	1B	27
d	00100011	23	35
f	00101011	2B	43
w	00011101	1D	29
e	00100100	24	36
r	00101101	2D	45
t	00101100	2C	44

Figure 8: PS/2 Keyboard Scancodes for the keys used in this project in binary, hexadecimal, and decimal.

3.2 Audio Driver Module: LM386 Amplifier

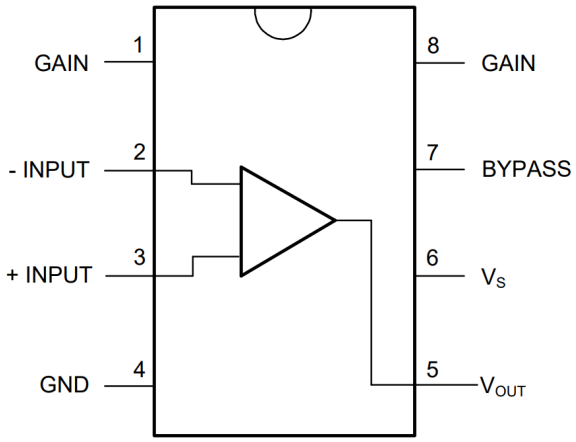


Figure 9: LM386 Pin Connections

For the speaker to have an audible signal for each note an amplifier must be used, in this case using the LM386 operational amplifier. For the LM386 the pins used are 2, 3, 4, 5, and 6. How an op-amp works is by setting the -input and +input nodes to the same voltage by changing Vout's voltage using a dependent voltage source. Using an inverted op-amp configuration as shown below the amount that Vout is incremented by is decided by R2/R1 known as the gain.

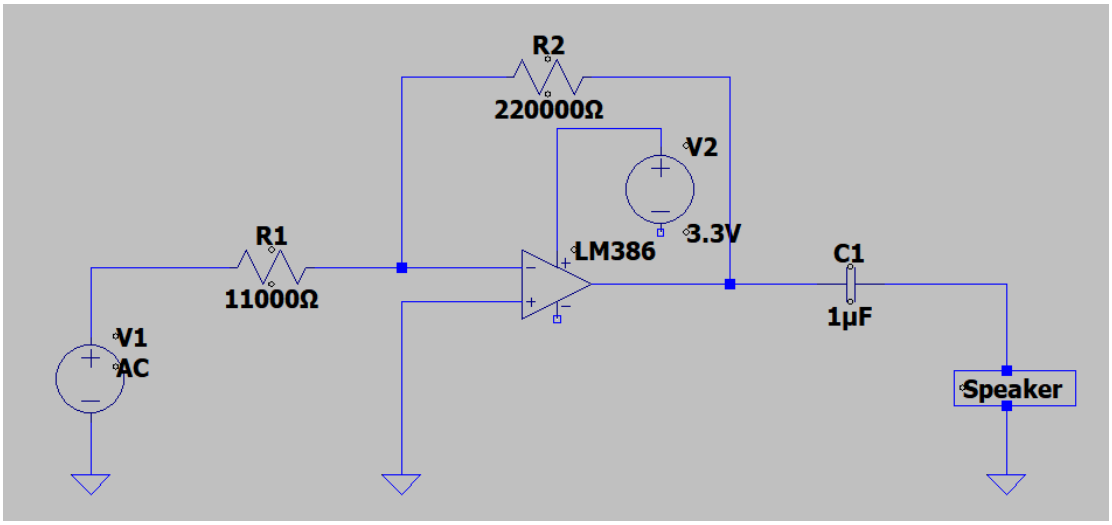


Figure 10: Audio Driver Circuit

A capacitor is added into the circuit between the op-amp and speaker to control the voltage going through it, this will stop the speaker from being blown out. The V1 shown in figure 10 is the audio signal that is interpreted as an alternating voltage. V2 is an added voltage source to the speaker that will allow it to supply the voltage it needs to make the two inputs of the op-amp equal.

4 HDL Components

4.1 Top Module Components

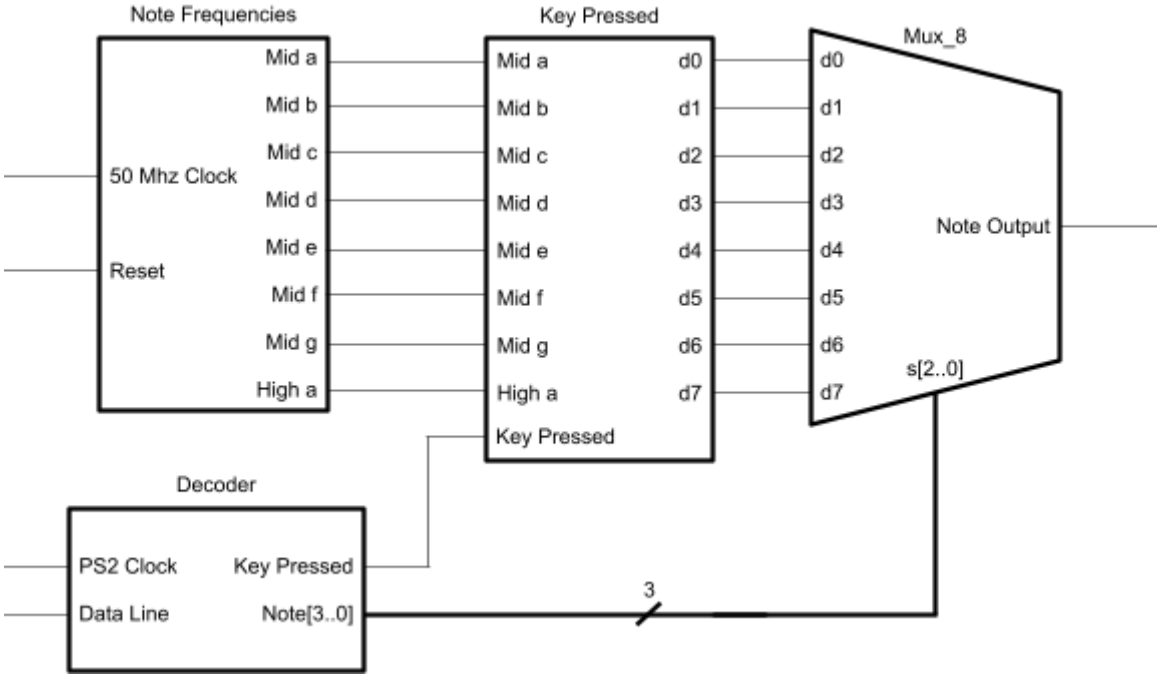


Figure 11: Top Module Diagram

Inputs: This takes a clock signal and data line input from a PS/2 keyboard and a 50 MHz clock signal and a reset input from the FPGA
Outputs: This outputs a frequency to a speaker corresponding to one of the eight possible notes

Simulation: The simulation for this module starts by forcing the active high reset to high and then low so that all values are known. The inputted clock signal is set to be 50 MHz and the PS2 clock and data line were forced to match the data given in the csv file for the key input “a”. All the timings were shifted by 0.001 so that all values were positive. The output of the scancode is shown to match the expected value for when “a” is pressed. The “note_out” signal is shown to take the signal of “middle_a” which also matches the expected note output based on the note each key was assigned to. Simulations for other keys s, d, and f can be found in the appendix.

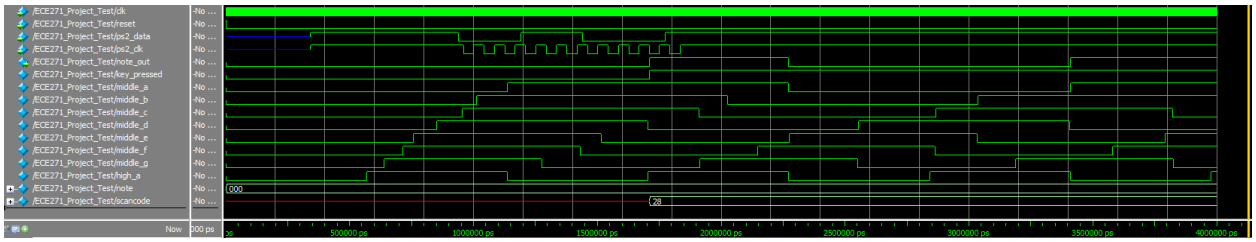


Figure 12: Top Module Simulation

4.2 Note Frequencies Top Module

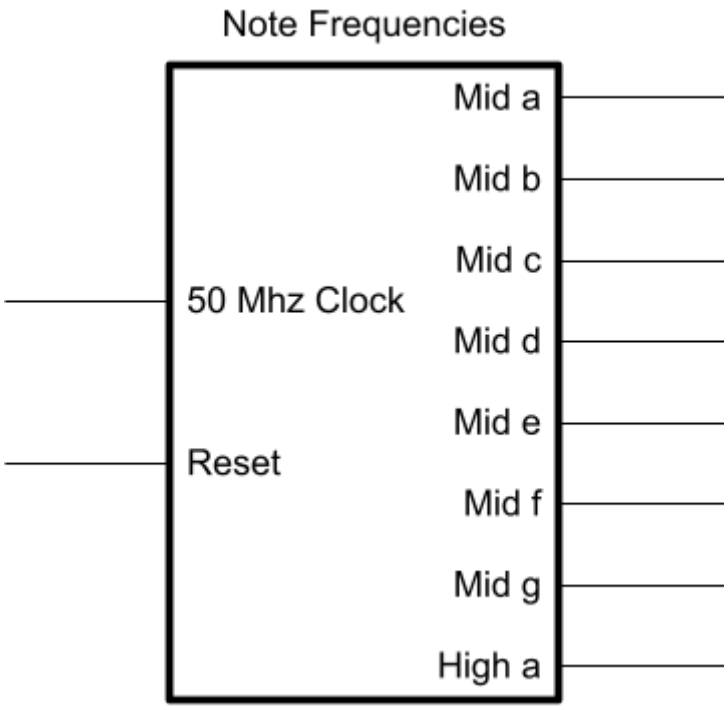


Figure 13: Note Frequencies Module Diagram

Inputs: A 50 MHz clock and an active high reset
Outputs: Eight divided clock signals corresponding to the notes in the 440 Hz to 880 Hz range

Description: This note frequencies module combines eight of the clock divider modules with specified parameter values such that the output clock signals match the frequencies of the eight desired notes. Parameter 2 was found by dividing the 50 MHz clock signal by the frequency of the desired note to find the value that the clock would have to be divided by. The value of parameter 1 is found simply by dividing parameter 2 by 2.

Note	Frequency	Parameter 1	Parameter 2
Middle A	440	56818	113636
Middle B	493.8833	50619	101238
Middle C	523.2511	47778	95556
Middle D	587.3295	42566	85132
Middle E	659.2551	37922	75844
Middle F	698.4565	35928	71586
Middle G	783.9909	31888	63776
High A	880	28409	56818

Figure 14: Note Frequencies Comparator Parameters

4.2.1 Clock Divider Module

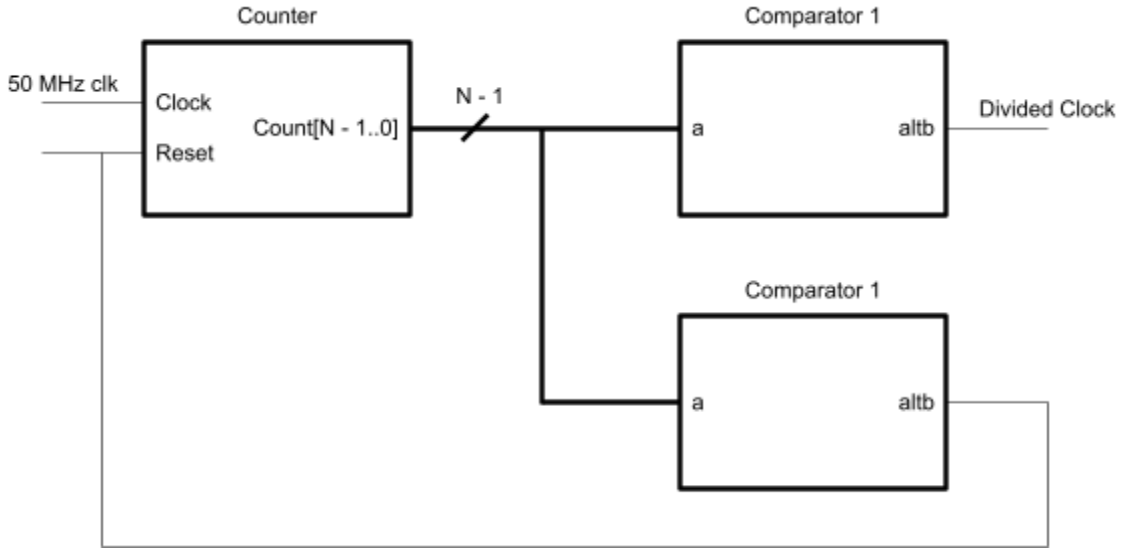


Figure 15: Clock Divider Module Diagram

Inputs: A 50 MHz clock and an active high reset
Outputs: A divided clock signal

Description: This clock divider takes in a 50 MHz clock signal and outputs a divided clock signal based on the parameters M for the comparators. The clock will be divided by the M parameter of the bottom comparator and the top comparator's M parameter will be half of the bottom to make the high and low logic portions of the signal even. This will result in an output clock that goes through one period for every M cycles of the 50 MHz clock.

Simulation: The simulation for this module takes M parameters to be 5 and 10 for the first and second comparator respectively meaning the input clock signal will be divided by 10. The simulation starts by forcing the active high reset to high and then low so that all values are known. The clock signal then begins rising and falling with a period of 2 ps. The output clock signal is seen to have a period of 20 ps meaning the signal was divided by 10 as expected.

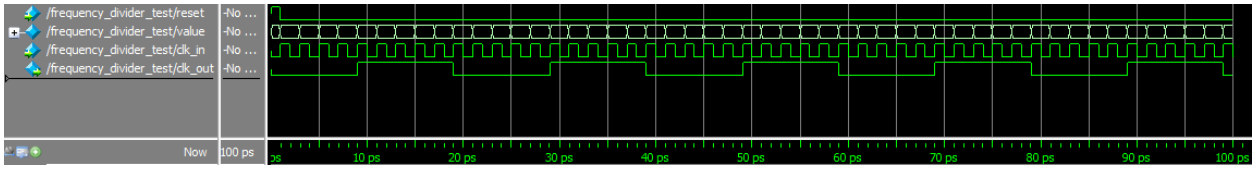


Figure 16: Clock Divider Module Simulation

4.2.2 Comparator Module

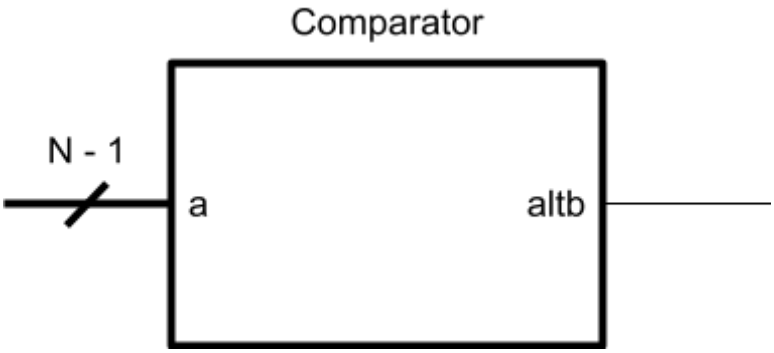


Figure 17: Comparator Module Diagram

Inputs: An N - 1 bit value
Outputs: A 1 bit logic high or logic low signal

Description: This comparator takes an N - 1 bit value determined by the parameter N. The actual value is generated from a clock and counter which is then compared to a value determined by the parameter M. If the input value is less than M then the output will produce a high logic level and if the input value is greater or equal to M then the output will produce a low logic level.

Simulation: The simulation for this module takes both the N and M parameters to be 8. Values for the input "a" were tested from 0 to 16 at increments of 2. The output "altb" is shown to be at high logic level when the value of a is less than 8 and low logic level when the value of a is greater than or equal to 8 which matches the expected result of a less than operator.

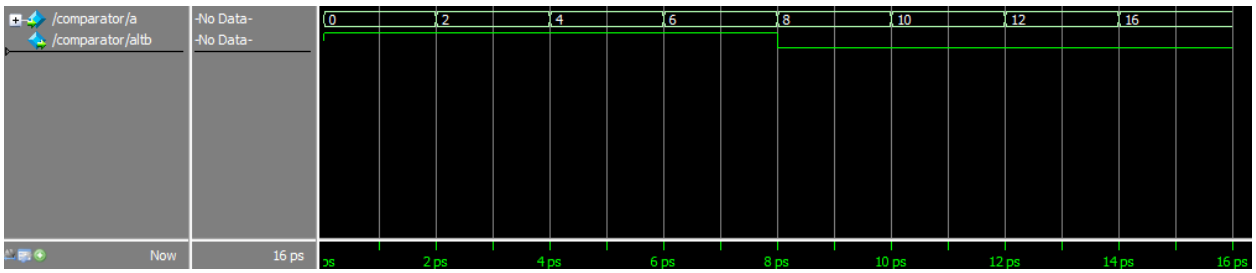


Figure 18: Comparator Module Simulation

4.2.3 Counter Module

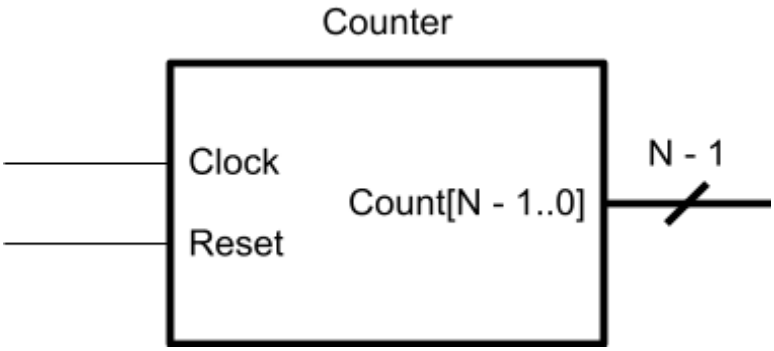


Figure 19: Counter Module Diagram

Inputs: A 50 MHz clock signal and an active high reset
Outputs: An N - 1 bit value

Description: This counter will continue incrementing every clock cycle until the reset input is put into a high logic level or it reaches the maximum value given by N - 1 bits determined by the parameter N. The timing intervals for reset will be determined by a comparator whose output connects to the reset input.

Simulation: The simulation for this module starts by forcing the active high reset to high and then low so that all values are known. The clock signal then begins rising and falling with a period of 2 ps. The output "q" of the counter is shown to increment on every rising edge of the clock signal.

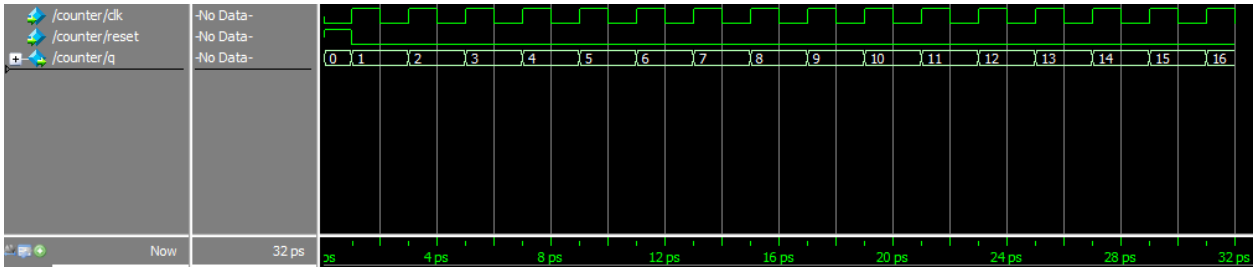


Figure 20: Counter Module Simulation

4.3 Eight-Input Mux Top Module

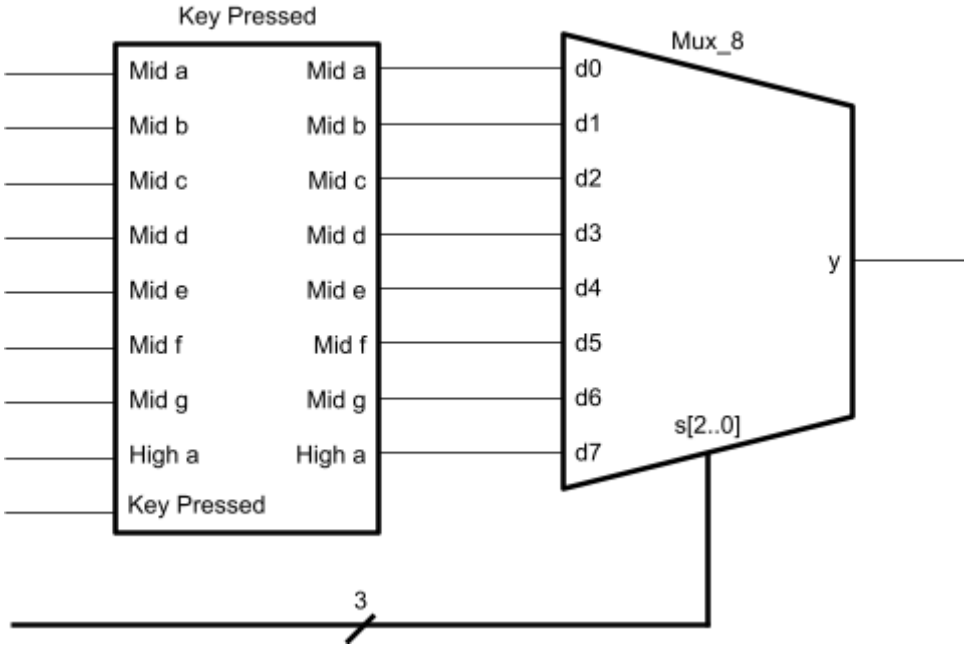


Figure 21: Eight-Input Mux Module Diagram

Inputs: A 3 bit select input, key pressed input, and eight different 1 bit signals
Outputs: A 1 bit signal

Description: This eight input mux takes in eight different 1 bit signals and a 3 bit select input that determines which signal propagates to the output y. Values are selected based on the corresponding unsigned value for the select input (e.g. 000 selects d0 and 111 selects d7). Each of the 1 bit signal inputs will take in a divide clock signal that has the frequency of the desired note.

Simulation: The simulation for this module gives each of the input signals a pulse at a specific time interval. As the select input is cycled between each of the input signals, the output of "y" is shown to take on the pulse of each respective input signal during the corresponding selecting input.

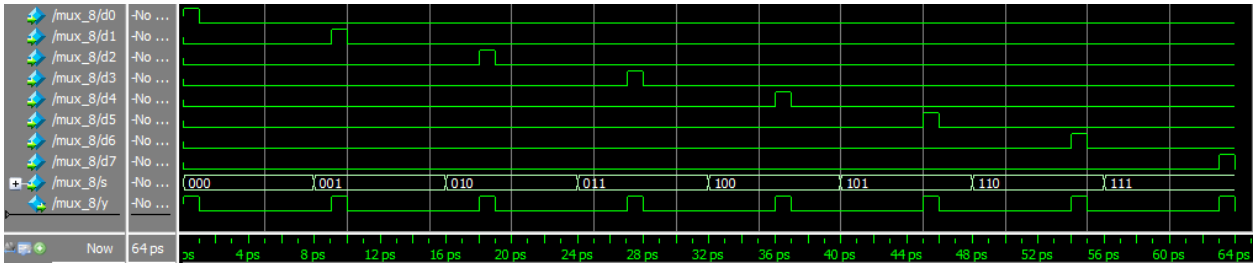


Figure 22: Eight-Input Mux Module Simulation

4.3.1 Two-Input Mux Module

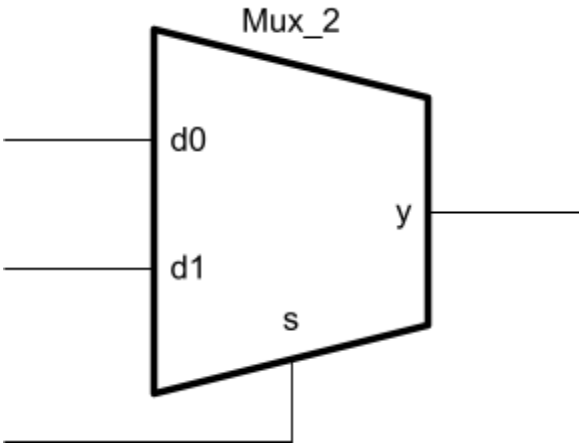


Figure 23: Two-Input Mux Module Diagram

Inputs: A 1 bit select input and two different 1 bit inputs
Outputs: A 1 bit output

Description: This two input mux takes in two different 1 bit inputs and a select input that determines the value that propagates to the output y. If the select input is logic low then the output will be given by d0 and if the select input is logic high then the output will be given by d1.

Simulation: The simulation for this module cycled through every combination for the "d1", "d0", and select input. The output "y" is shown to take on the value of d1 when select is at high logic level and takes on the value of d0 when select is at low logic level.

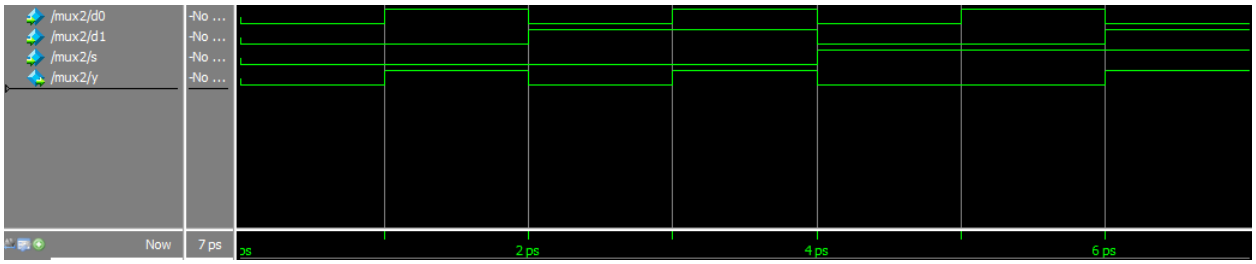


Figure 24: Two-Input Mux Module Simulation

4.3.2 Key Pressed Module

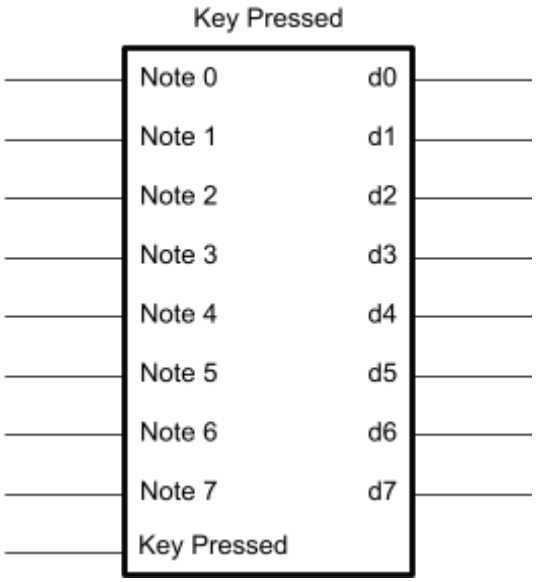


Figure 25: Key Pressed Module Diagram

Inputs: Eight different 1 bit signals and a key pressed input
Outputs: Eight different 1 bit signals

Description: The key pressed module takes AND operation of each of the eight 1 bit signals and the key pressed input. This results in the desired note propagating through to its respective output when the key pressed is recognized and low logic level for each of the eight 1 bit signal outputs when the key pressed is not recognized.

Simulation: The simulation for this module gives each of the input signals a pulse at a specific time interval. The key pressed input is set at low logic level for the first 64 ps of the simulation and at high logic level for the next 64 ps. During the first 64 ps, all the outputs remain at low logic level because the key pressed input is also at a low logic level. During the next 64 ps, all the outputs take on the pulse of their respective input because the key pressed is at a high logic level.

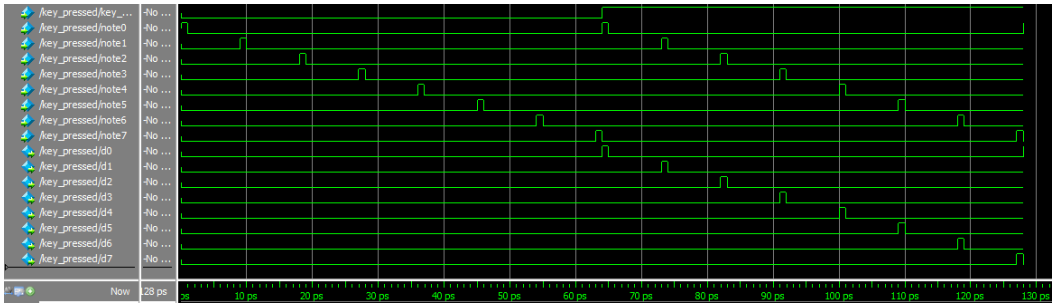


Figure 26: Key Pressed Module Simulation

4.4 Decoder

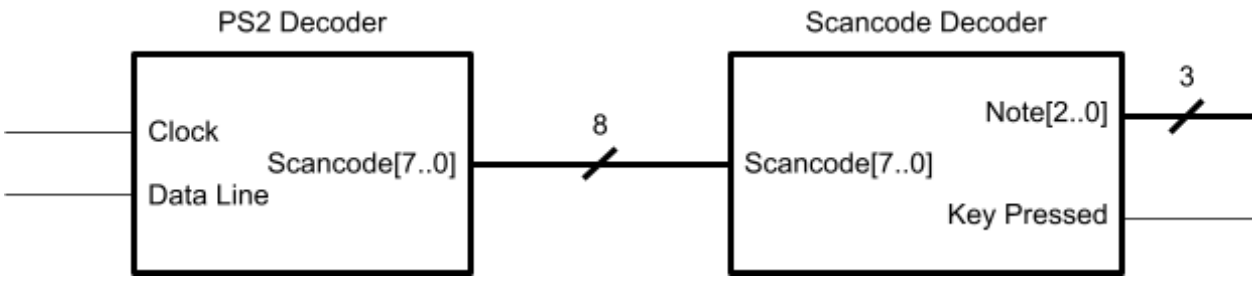


Figure 27: Decoder Module Diagram

Inputs: 10 - 16.7 kHz clock signal and data signal
Outputs: A 3 bit value and 1 bit output

Description: The decoder module combines the PS/2 decoder and scancode decoder so that the clock signal and data line are converted into a 3 bit value. This 3 bit value will be taken as the enable input of the eight input mux that determines which of the eight notes to play. The key pressed output will output a high logic level when a known key is pressed and a low logic level when an unknown key is pressed.

Simulation: The simulation for this module takes a 2 ps period clock signal as an input as well as all the corresponding data line inputs for each of the a, s, d, f, w, e, r, t keys. After the PS/2 decoder determines the scancode for each respective key input and the scancode decoder converts the scancodes into 3 bit values, the 3 bit output is shown to take on a value from 0 to 7 corresponding to the different keys pressed.

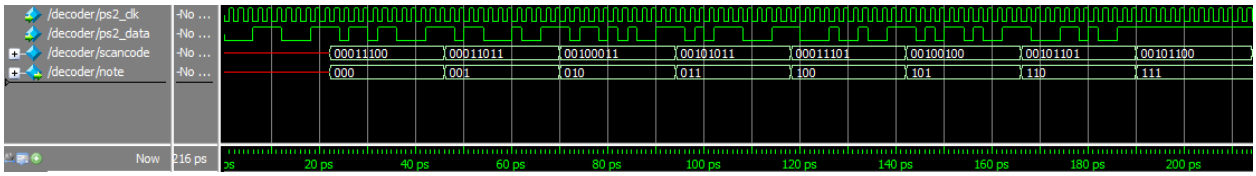


Figure 28: Decoder Module Simulation

4.4.1 PS/2 Decoder Module

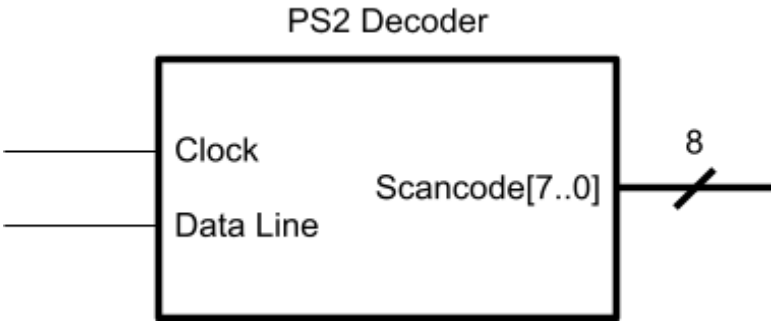


Figure 29: PS/2 Decoder Module Diagram

Inputs: 10 - 16.7 kHz clock signal and data signal
Outputs: An 8 bit value

Description: This PS/2 decoder module takes in a clock signal and data line given by the PS/2 keyboard whenever a key is pressed. The clock signal will increment a counter from 0 to 10 that determines when data from the data line is read into the scancode output. If the counter is greater than 0 and less than 9 then the value of the data line is read into the "counter - 1" index of a temporary variable before being copied to the output value of the scancode.

Simulation: The simulation for this module takes a 2 ps period clock signal as an input as well as all the corresponding data line inputs for each of the a, s, d, f, w, e, r, t keys. The "output" is shown to take on the scancode values on each negative edge of the clock when the counter is between greater than 0 and less than 9. The value stored in temp is then copied to the scancode output when all the necessary values of the data line are read. The scan code outputs are shown to match the values of each of the corresponding keys. The clock and data line csv data were simulated for the keys a, s, d, and f and can be found in the appendix.

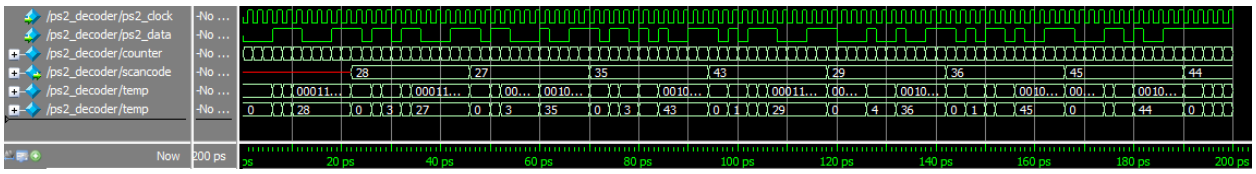


Figure 30: PS2 Decoder Module Simulation

4.4.2 Scancode Decoder Module

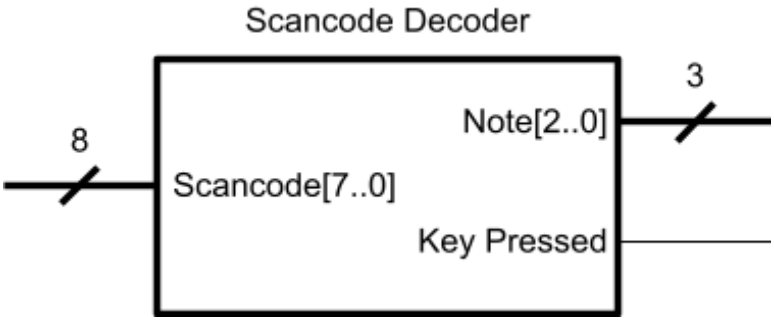


Figure 31: Scancode Decoder Module Diagram

Inputs: An 8 bit value
Outputs: A 3 bit value and 1 bit output

Description: This scancode decoder takes an 8 bit scancode value corresponding to the keys a, s, d, f, w, e, r, t and converts them into a 3 bit value so that the eight input mux can determine which of the eight notes to play. There will also be a 1 bit "key pressed" output that will be at high logic level when a known scancade is inputted and at low logic level when an unknown scancode is inputted.

Simulation: The simulation for this module inputted each of the scancode values for the a, s, d, f, w, e, r, t keys. The output is shown to be a 3 bit value with each scancode corresponding to a different value.

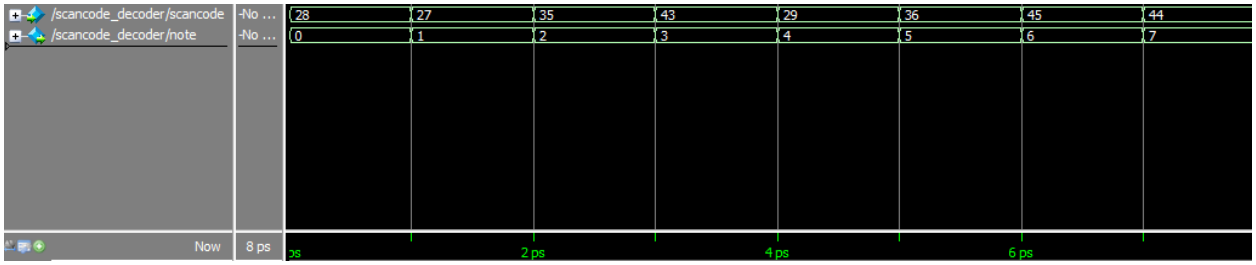


Figure 32: Scancode Decoder Module Simulation

5 Hardware Demo

Link to demo: https://youtu.be/_DiC5BU7cK4

6 References

[1] PS2 protocol:
A. Chapweske, “PS/2 Mouse/Keyboard Protocol,” *The PS/2 mouse/keyboard protocol*, 1999. [Online]. Available: https://www.burtonsys.com/ps2_chapweske.htm. [Accessed: 05-Jun-2022].

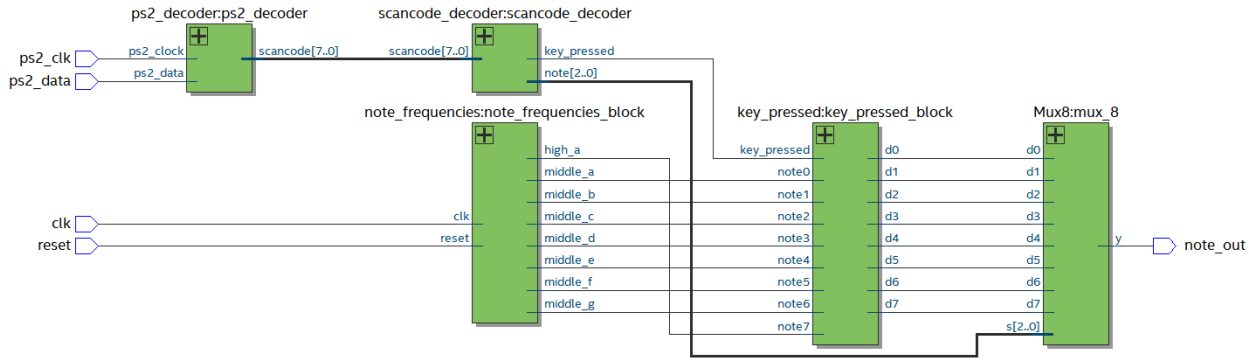
[2] LM386 datasheet:
Texas Instruments, “LM386 low voltage audio power amplifier datasheet (rev. C),” *Texas Instruments*, May-2017. [Online]. Available: https://www.ti.com/lit/ds/symlink/lm386.pdf. [Accessed: 07-Jun-2022].

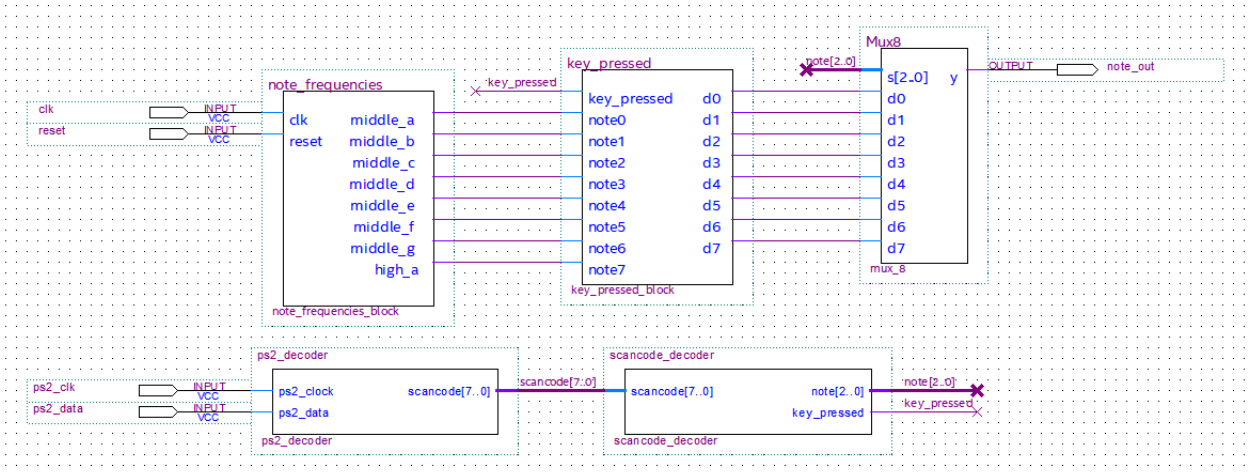
[3] Frequency chart:
Sammy, “Music note frequency chart - music frequency chart,” *MixButton*, 24-Oct-2021. [Online]. Available: https://mixbutton.com/mixing-articles/music-note-to-frequency-chart/. [Accessed: 06-Jun-2022].

7 Appendix

7.1 Design Synthesis and Analysis

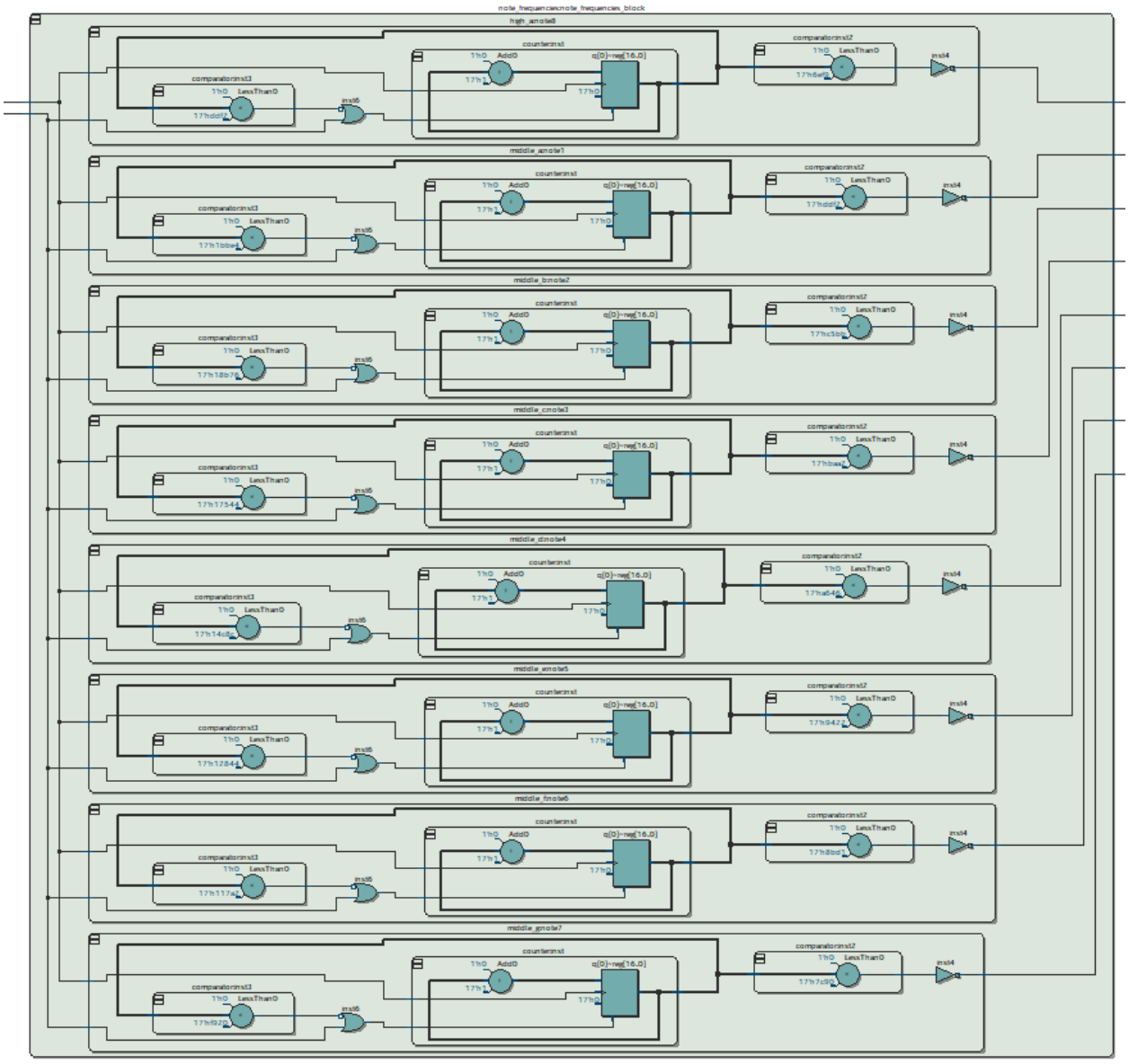
7.1.1 Top Module Synthesis and Analysis



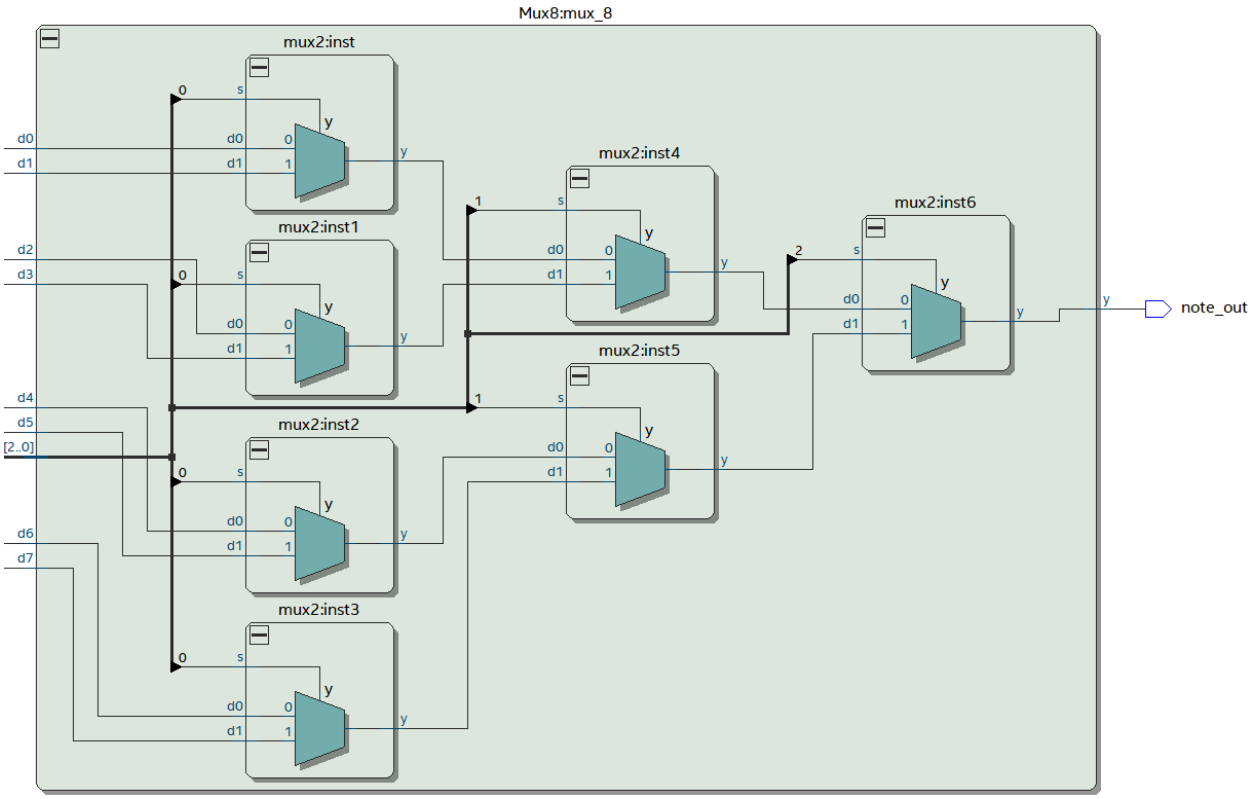


Total logic elements	271 / 49,760 (< 1 %)
Total registers	156
Total pins	5 / 360 (1 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

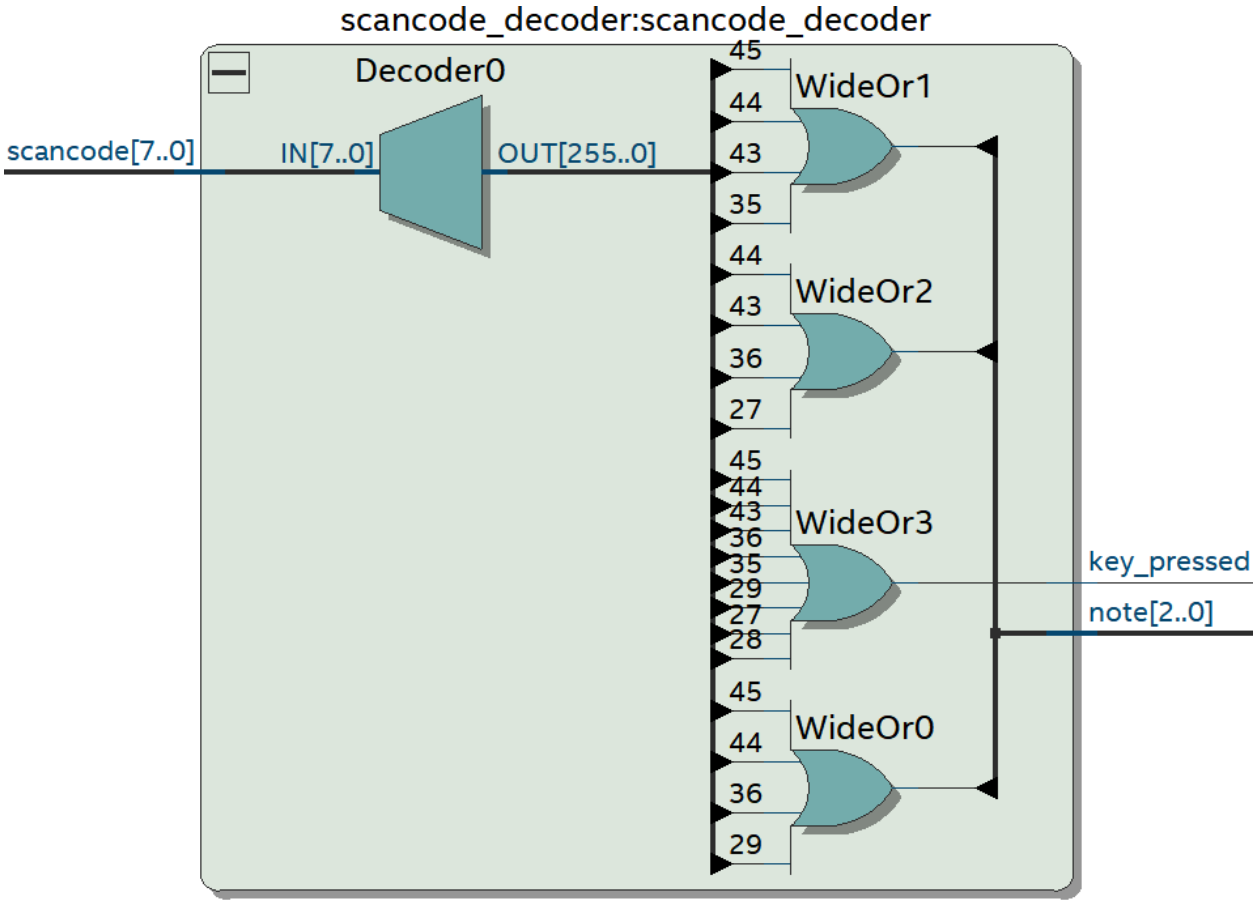
7.1.2 Note Frequencies Synthesis



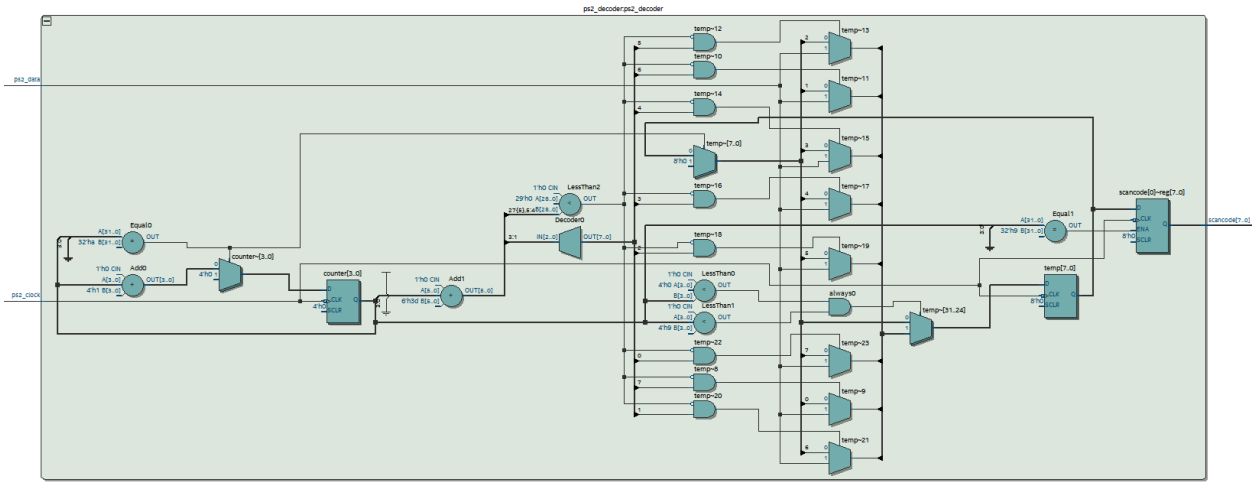
7.1.3 Eight-Input Mux Synthesis



7.1.4 PS/2 Decoder Synthesis



7.1.5 Scancode Decoder Synthesis



7.2 Source Code

7.2.1 Comparator Module Code

```
module comparator#(parameter N = 8, M = 8)
                  (input logic[N-1:0] a,
                  output logic altb);

   assign altb = (a < M);

endmodule
```

7.2.2 Counter Module Code

```
module counter#(parameter N = 8)
               (input logic clk,
               input logic reset,
               output logic [N-1:0] q);

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

7.2.3 Eight-Input Mux Top Module Code

```
module mux_8(
    input logic d0, d1, d2, d3, d4, d5, d6, d7,
    input logic [2:0] s,
    output logic y
);

    logic low, high, low_low, low_high, high_low, high_high;

    mux2 final_mux(low, high, s[2], y);

    //2 mux that connect final mux
    mux2 low_mux(low_low, low_high, s[1], low);
    mux2 high_mux(high_low, high_high, s[1], high);

    //4 mux that connect to low or high mux
    mux2 low_low_mux(d0, d1, s[0], low_low);
    mux2 low_high_mux(d2, d3, s[0], low_high);
    mux2 high_low_mux(d4, d5, s[0], high_low);
    mux2 high_high_mux(d6, d7, s[0], high_high);

endmodule
```

7.2.4 Two-Input Mux Module Code

```
module mux2(input logic d0, d1, s,
            output logic y);

   assign y = s ? d1: d0;
endmodule
```

7.2.5 PS/2 Decoder Module Code

```
module ps2_decoder(
    input logic ps2_clock,
    input logic ps2_data,

    output logic [7:0] scancode);

  logic [3:0] counter = 0;
  logic [7:0] temp = 0;
  //logic parity = 1;


  always_ff@(negedge ps2_clock)
    begin
       if(counter == 10)
          begin
             counter <= 0;
             temp <= 0;
    //       parity <= 1;
          end
       else
          begin
             counter <= counter + 1;
          end

       if(counter > 0 && counter < 9)
          begin
             temp[counter-1] <= ps2_data;
    //       parity <= parity ^ ps2_data;
          end
       if(counter == 9)
          begin
             scancode <= temp;
          end
    end
  endmodule
```

7.2.6 Scancode Decoder Module Code

```
module scancode_decoder(input logic[7:0] scancode,
                        output logic[2:0] note,
                        output logic key_pressed);

    always_comb
       case(scancode)
          28:          note = 0;    //a
          27:          note = 1;    //s
          35:          note = 2;    //d
          43:          note = 3;    //f
          29:          note = 4;    //w
          36:          note = 5;    //e
          45:          note = 6;    //r
          44:          note = 7;    //t
          default:     note = 0;
       endcase

    always_comb
       case(scancode)
          28:          key_pressed = 1;
          27:          key_pressed = 1;
          35:          key_pressed = 1;
          43:          key_pressed = 1;
          29:          key_pressed = 1;
          36:          key_pressed = 1;
          45:          key_pressed = 1;
          44:          key_pressed = 1;
          default:     key_pressed = 0;
       endcase
  endmodule
```

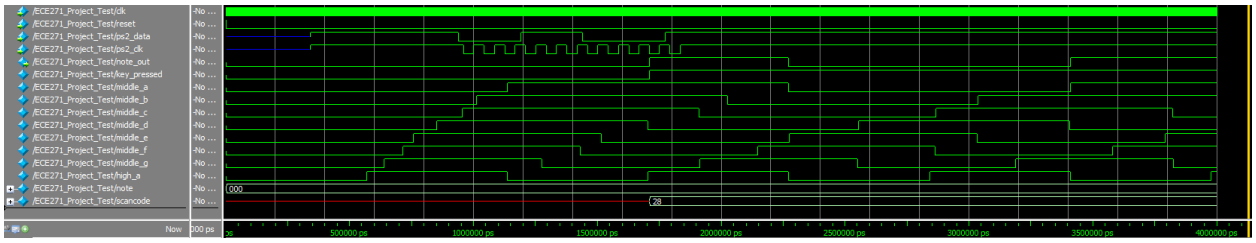
7.3 Simulation Results

7.3.1 Top Module Components Simulation

The timing values used for clock and data line of each key was based on the values from the given .csv files and can be found in section 7.4 of the appendix

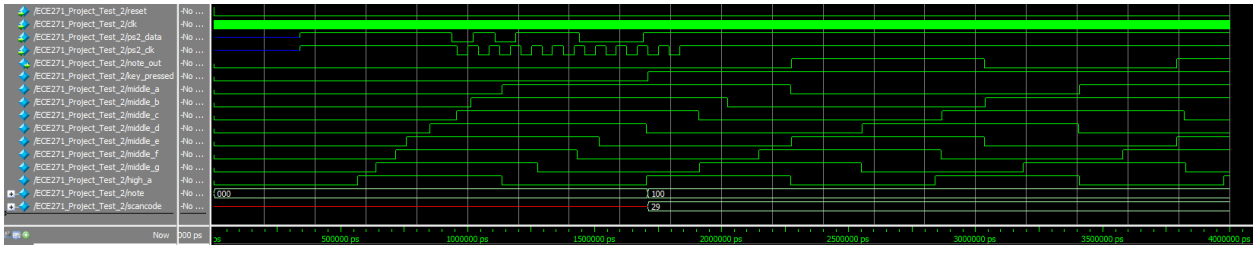
A

```
add wave reset
add wave clk
add wave ps2_data
add wave ps2_clk
add wave note_out
add wave key_pressed
add wave middle_a
add wave middle_b
add wave middle_c
add wave middle_d
add wave middle_e
add wave middle_f
add wave middle_g
add wave high_a
add wave note
add wave scancode

force clk 0 0, 1 0.01ns -r 0.02ns
force reset 1 0, 0 0.01ns
force ps2_clk 1 340ns, 0 959ns, 1 1000.4ns, 0 1042.2ns, 1 1083.6ns, 0 1126ns, 1 1167ns, 0 1209ns, 1 1251ns, 0 1293ns, 1 1334ns, 0 1377ns, 1 1418ns, 0 1460ns, 1 1502ns, 0 1544ns, 1 1585ns, 0 1627ns, 1 1669ns, 0 1710ns, 1 1751ns, 0 1793ns, 1 1835ns
force ps2_data 1 340ns, 0 939ns, 1 1190ns, 0 1441ns, 1 1774ns

run 4000ns
```



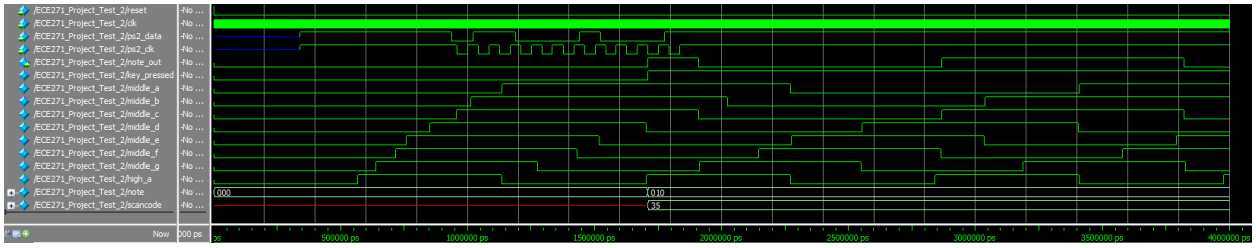
S

```
add wave reset
add wave clk
add wave ps2_data
add wave ps2_clk
add wave note_out
add wave key_pressed
add wave middle_a
add wave middle_b
add wave middle_c
add wave middle_d
add wave middle_e
add wave middle_f
add wave middle_g
add wave high_a
add wave note
add wave scancode

force clk 0 0, 1 0.01ns -r 0.02ns
force reset 1 0, 0 0.01ns
force ps2_clk 1 340ns, 0 959ns, 1 1000.4ns, 0 1042.2ns, 1 1083.6ns, 0 1126ns, 1 1167ns, 0
1209ns, 1 1251ns, 0 1293ns, 1 1334ns, 0 1377ns, 1 1418ns, 0 1460ns, 1 1502ns, 0 1544ns, 1
1585ns, 0 1627ns, 1 1669ns, 0 1710ns, 1 1751ns, 0 1793ns, 1 1835ns
force ps2_data 1 340ns, 0 939ns, 1 1023ns, 0 1107ns, 1 1190ns, 0 1441ns, 1 1692ns

run 4000ns
```



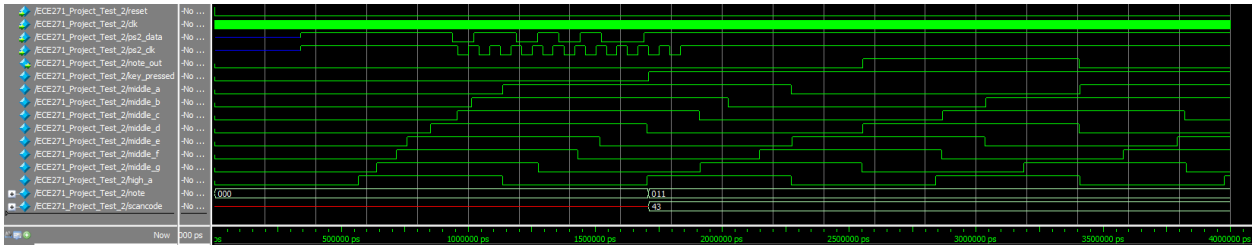
D

```
add wave reset
add wave clk
add wave ps2_data
add wave ps2_clk
add wave note_out
add wave key_pressed
add wave middle_a
add wave middle_b
add wave middle_c
add wave middle_d
add wave middle_e
add wave middle_f
add wave middle_g
add wave high_a
add wave note
add wave scancode

force clk 0 0, 1 0.01ns -r 0.02ns
force reset 1 0, 0 0.01ns
force ps2_clk 1 340ns, 0 959ns, 1 1000.4ns, 0 1042.2ns, 1 1083.6ns, 0 1126ns, 1 1167ns, 0
1209ns, 1 1251ns, 0 1293ns, 1 1334ns, 0 1377ns, 1 1418ns, 0 1460ns, 1 1502ns, 0 1544ns, 1
1585ns, 0 1627ns, 1 1669ns, 0 1710ns, 1 1751ns, 0 1793ns, 1 1835ns
force ps2_data 1 340ns, 0 939ns, 1 1022.8ns, 0 1190ns, 1 1441ns, 0 1525ns, 1 1775ns

run 4000ns
```



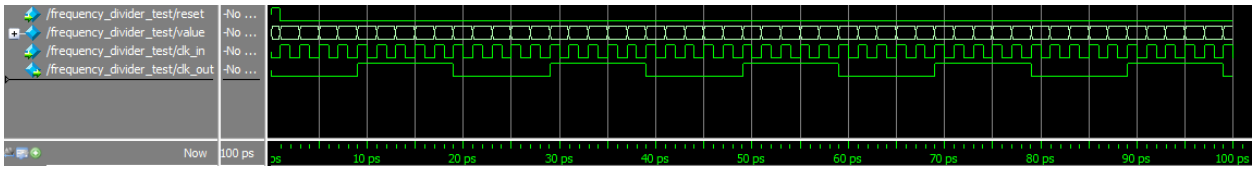
F

```
add wave reset
add wave clk
add wave ps2_data
add wave ps2_clk
add wave note_out
add wave key_pressed
add wave middle_a
add wave middle_b
add wave middle_c
add wave middle_d
add wave middle_e
add wave middle_f
add wave middle_g
add wave high_a
add wave note
add wave scancode

force clk 0 0, 1 0.01ns -r 0.02ns
force reset 1 0, 0 0.01ns
force ps2_clk 1 340ns, 0 959ns, 1 1000.4ns, 0 1042.2ns, 1 1083.6ns, 0 1126ns, 1 1167ns, 0
1209ns, 1 1251ns, 0 1293ns, 1 1334ns, 0 1377ns, 1 1418ns, 0 1460ns, 1 1502ns, 0 1544ns, 1
1585ns, 0 1627ns, 1 1669ns, 0 1710ns, 1 1751ns, 0 1793ns, 1 1835ns
force ps2_data 1 340ns, 0 939ns, 1 1023ns, 0 1190ns, 1 1274ns, 0 1358ns, 1 1441ns, 0 1525ns, 1
1692ns

run 4000ns
```



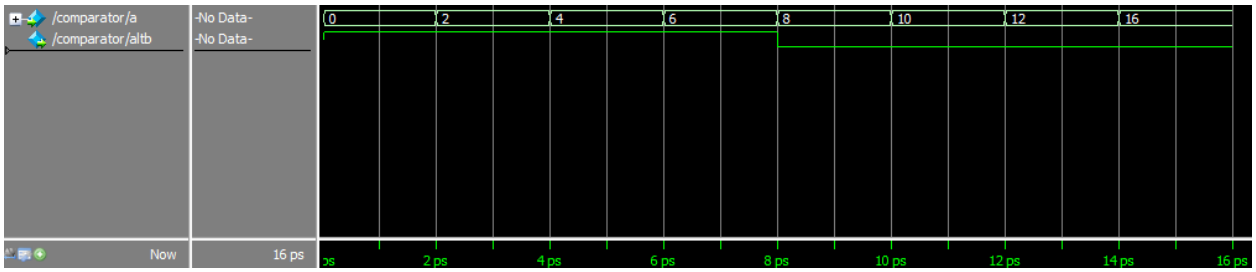
7.3.2 Clock Divider Top Module Simulation

```
1  add wave reset
2  add wave value
3  add wave clk_in
4  add wave clk_out
5
6  force reset 1 0, 0 1
7  force clk_in 0 0, 1 1 -r 2
8
9  run 100
```



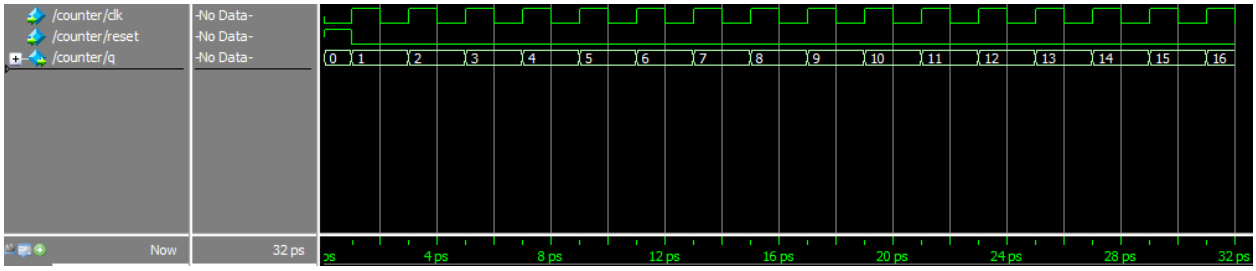
7.3.3 Comparator Module Simulation

```
1  add wave a
2  add wave altb
3
4  force a 00000000 0, 00000010 2, 00000100 4, 00000110 6
5  force a 00001000 8, 00001010 10, 00001100 12, 00010000 14
6
7  run 16
```



7.3.4 Counter Module Simulation

```
1  add wave clk
2  add wave reset
3  add wave q
4
5  force reset 1 0, 0 1
6  force clk 0 0, 1 1 -r 2
7
8  run 32
```

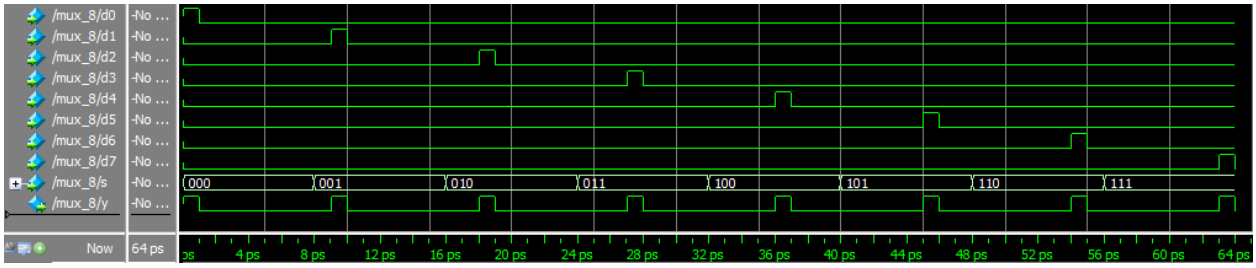


7.3.5 Eight-Input Mux Top Module Simulation

```
1   add wave d0
2   add wave d1
3   add wave d2
4   add wave d3
5   add wave d4
6   add wave d5
7   add wave d6
8   add wave d7
9   add wave s
10  add wave y
```

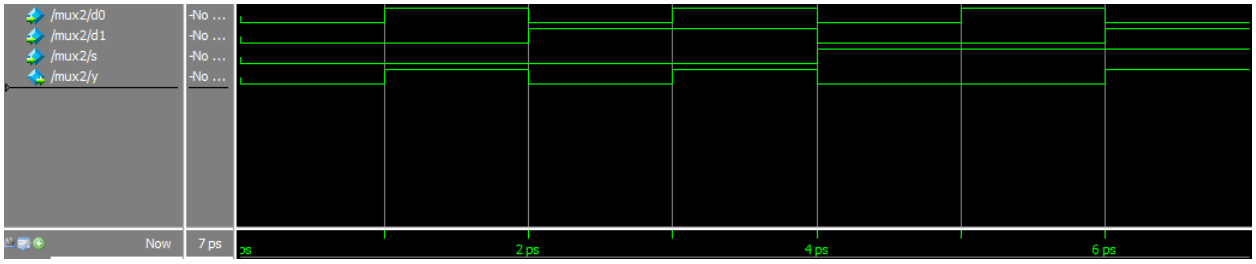
```
11
12  force d0 1 0, 0 1, 0 2
13  force d1 0 0, 1 9, 0 10
14  force d2 0 0, 1 18, 0 19
15  force d3 0 0, 1 27, 0 28
16  force d4 0 0, 1 36, 0 37
17  force d5 0 0, 1 45, 0 46
18  force d6 0 0, 1 54, 0 55
19  force d7 0 0, 1 63, 0 64
20
```

```
21  force s 000 0
22  force s 001 8
23  force s 010 16
24  force s 011 24
25  force s 100 32
26  force s 101 40
27  force s 110 48
28  force s 111 56
29
30  run 64
```



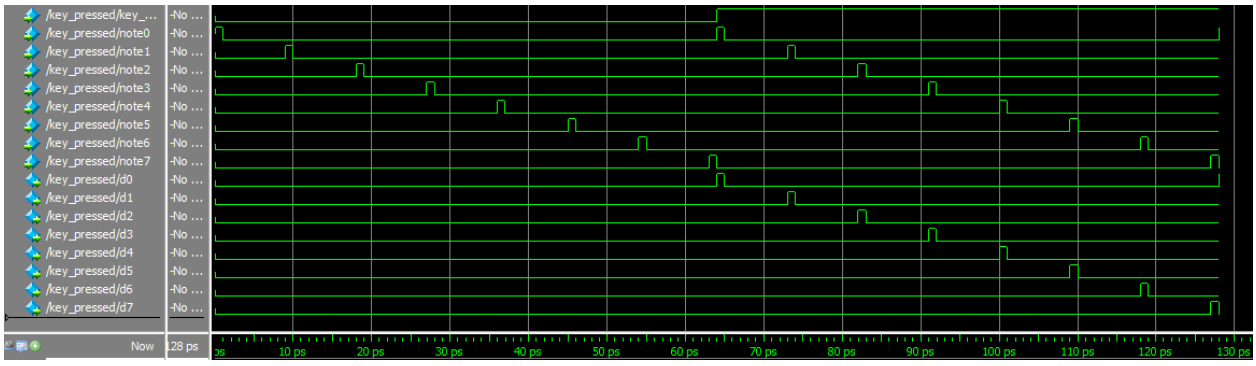
7.3.6 Two-Input Mux Module Simulation

```
1   add wave d0
2   add wave d1
3   add wave s
4   add wave y
5
6   force s 0 0, 1 4
7   force d1 0 0, 1 2 -r 4
8   force d0 0 0, 1 1 -r 2
9
10  run 7
```



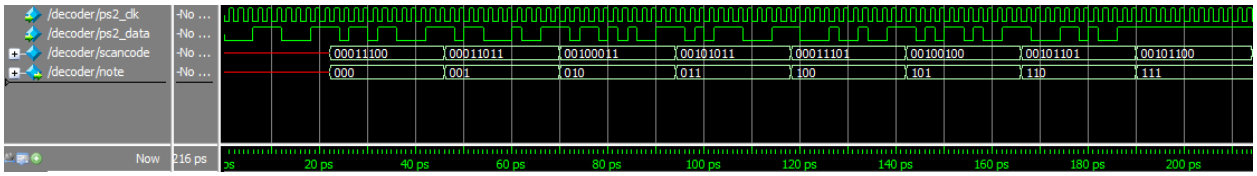
7.3.7 Key Pressed Module Simulation

```
add wave key_pressed
add wave note0
add wave note1
add wave note2
add wave note3
add wave note4
add wave note5
add wave note6
add wave note7
add wave d0
add wave d1
add wave d2
add wave d3
add wave d4
add wave d5
add wave d6
add wave d7

force note0 1 0, 0 1, 0 2 -r 64
force note1 0 0, 1 9, 0 10 -r 64
force note2 0 0, 1 18, 0 19 -r 64
force note3 0 0, 1 27, 0 28 -r 64
force note4 0 0, 1 36, 0 37 -r 64
force note5 0 0, 1 45, 0 46 -r 64
force note6 0 0, 1 54, 0 55 -r 64
force note7 0 0, 1 63, 0 64 -r 64
force key_pressed 0 0, 1 64

run 128
```



7.3.8 Decoder Module Simulation

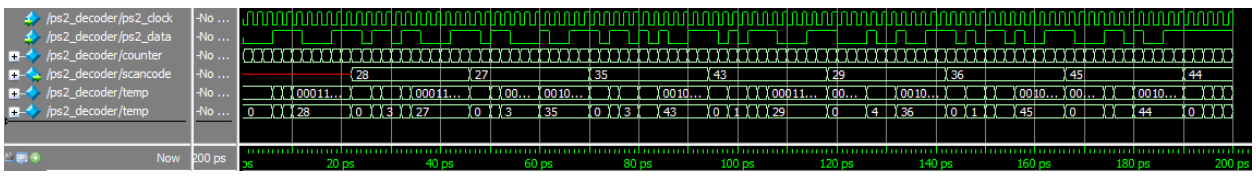
```
add wave ps2_clk
add wave ps2_data
add wave scancode
add wave note

force ps2_clk 0 0, 1 1 -r 2
force ps2_data 0 0, 0 2, 0 4, 1 6, 1 8, 1 10, 0 12, 0 14, 0 16, 1 18
force ps2_data 0 24, 1 26, 1 28, 0 30, 1 32, 1 34, 0 36, 0 38, 0 40, 1 42
force ps2_data 0 48, 1 50, 1 52, 0 54, 0 56, 0 58, 1 60, 0 62, 0 64, 1 66
force ps2_data 0 72, 1 74, 1 76, 0 78, 1 80, 0 82, 1 84, 0 86, 0 88, 1 90
force ps2_data 0 96, 1 98, 0 100, 1 102, 1 104, 1 106, 0 108, 0 110, 0 112, 1 114
force ps2_data 0 120, 0 122, 0 124, 1 126, 0 128, 0 130, 1 132, 0 134, 0 136, 1 138
force ps2_data 0 144, 1 146, 0 148, 1 150, 1 152, 0 154, 1 156, 0 158, 0 160, 1 162
force ps2_data 0 168, 0 170, 0 172, 1 174, 1 176, 0 178, 1 180, 0 182, 0 184, 1 186

run 210
```



7.3.9 PS2 Decoder Module Simulation

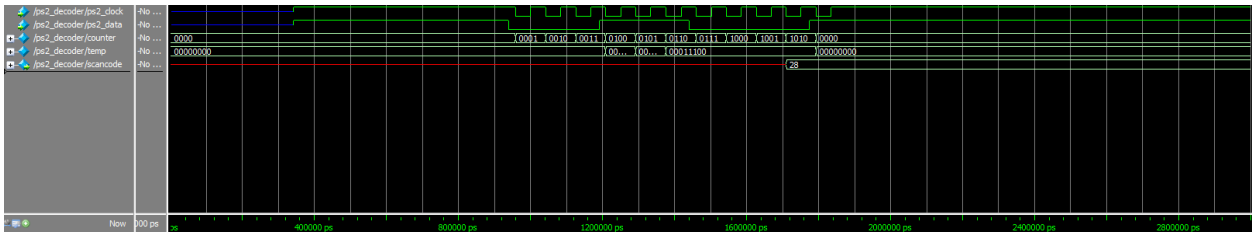
Ideal Conditions (All Keys)

```
add wave ps2_clock
add wave ps2_data
add wave counter
add wave temp
add wave scancode

force ps2_clock 0 0, 1 1 -r 2
force ps2_data 0 0, 0 2, 0 4, 1 6, 1 8, 1 10, 0 12, 0 14, 0 16, 1 18
force ps2_data 0 24, 1 26, 1 28, 0 30, 1 32, 1 34, 0 36, 0 38, 0 40, 1 42
force ps2_data 0 48, 1 50, 1 52, 0 54, 0 56, 0 58, 1 60, 0 62, 0 64, 1 66
force ps2_data 0 72, 1 74, 1 76, 0 78, 1 80, 0 82, 1 84, 0 86, 0 88, 1 90
force ps2_data 0 96, 1 98, 0 100, 1 102, 1 104, 1 106, 0 108, 0 110, 0 112, 1 114
force ps2_data 0 120, 0 122, 0 124, 1 126, 0 128, 0 130, 1 132, 0 134, 0 136, 1 138
force ps2_data 0 144, 1 146, 0 148, 1 150, 1 152, 0 154, 1 156, 0 158, 0 160, 1 162
force ps2_data 0 168, 0 170, 0 172, 1 174, 1 176, 0 178, 1 180, 0 182, 0 184, 1 186

run 200
```



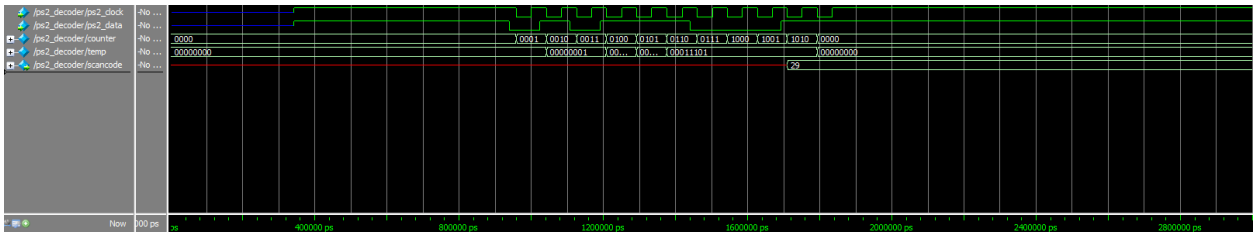
CSV Data A

add wave ps2_clock
add wave ps2_data
add wave counter
add wave temp
add wave scancode
force ps2_clock 1 340ns, 0 959ns, 1 1000.4ns, 0 1042.2ns, 1 1083.6ns, 0 1126ns, 1 1167ns, 0 1209ns, 1 1251ns, 0 1293ns, 1 1334ns, 0 1377ns, 1 1418ns, 0 1460ns, 1 1502ns, 0 1544ns, 1 1585ns, 0 1627ns, 1 1669ns, 0 1710ns, 1 1751ns, 0 1793ns, 1 1835ns
force ps2_data 1 340ns, 0 939ns, 1 1190ns, 0 1441ns, 1 1774ns
run 3000ns



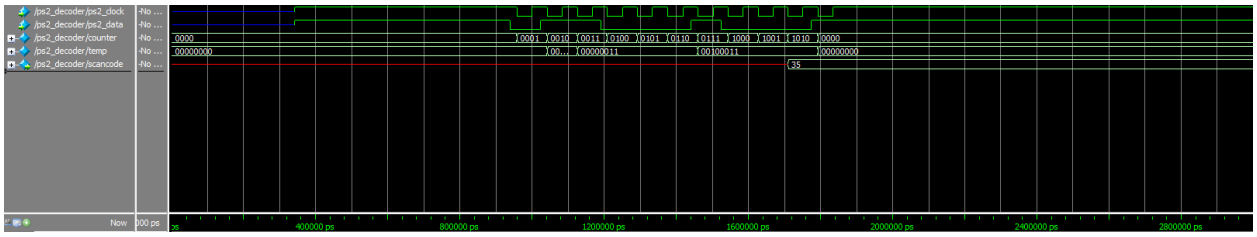
CSV Data S

add wave ps2_clock
add wave ps2_data
add wave counter
add wave temp
add wave scancode
force ps2_clock 1 340ns, 0 959ns, 1 1000.4ns, 0 1042.2ns, 1 1083.6ns, 0 1126ns, 1 1167ns, 0 1209ns, 1 1251ns, 0 1293ns, 1 1334ns, 0 1377ns, 1 1418ns, 0 1460ns, 1 1502ns, 0 1544ns, 1 1585ns, 0 1627ns, 1 1669ns, 0 1710ns, 1 1751ns, 0 1793ns, 1 1835ns
force ps2_data 1 340ns, 0 939ns, 1 1023ns, 0 1107ns, 1 1190ns, 0 1441ns, 1 1692ns
run 3000ns



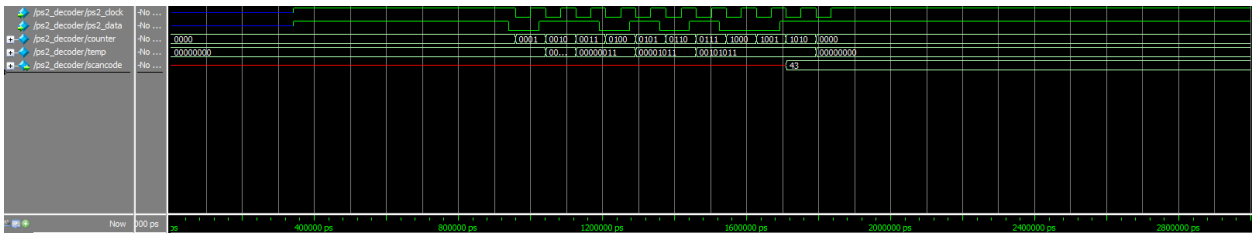
CSV Data D

add wave ps2_clock
add wave ps2_data
add wave counter
add wave temp
add wave scancode
force ps2_clock 1 340ns, 0 959ns, 1 1000.4ns, 0 1042.2ns, 1 1083.6ns, 0 1126ns, 1 1167ns, 0 1209ns, 1 1251ns, 0 1293ns, 1 1334ns, 0 1377ns, 1 1418ns, 0 1460ns, 1 1502ns, 0 1544ns, 1 1585ns, 0 1627ns, 1 1669ns, 0 1710ns, 1 1751ns, 0 1793ns, 1 1835ns
force ps2_data 1 340ns, 0 939ns, 1 1022.8ns, 0 1190ns, 1 1441ns, 0 1525ns, 1 1775ns
run 3000ns



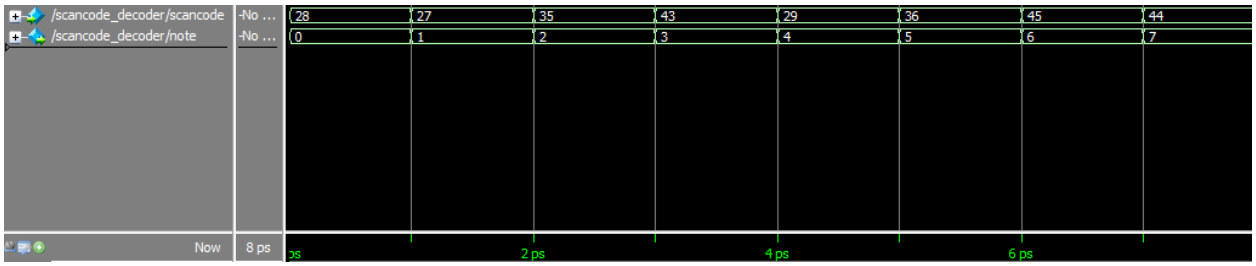
CSV Data F

add wave ps2_clock
add wave ps2_data
add wave counter
add wave temp
add wave scancode
force ps2_clock 1 340ns, 0 959ns, 1 1000.4ns, 0 1042.2ns, 1 1083.6ns, 0 1126ns, 1 1167ns, 0 1209ns, 1 1251ns, 0 1293ns, 1 1334ns, 0 1377ns, 1 1418ns, 0 1460ns, 1 1502ns, 0 1544ns, 1 1585ns, 0 1627ns, 1 1669ns, 0 1710ns, 1 1751ns, 0 1793ns, 1 1835ns
force ps2_data 1 340ns, 0 939ns, 1 1023ns, 0 1190ns, 1 1274ns, 0 1358ns, 1 1441ns, 0 1525ns, 1 1692ns
run 3000ns



7.3.10 Scancode Decoder Module Simulation

```
1   add wave scancode
2   add wave note
3
4   force scancode 00011100 0
5   force scancode 00011011 1
6   force scancode 00100011 2
7   force scancode 00101011 3
8   force scancode 00011101 4
9   force scancode 00100100 5
10  force scancode 00101101 6
11  force scancode 00101100 7
12
13  run 8
```



7.4 PS/2 Keyboard Timing from .csv Data

Logic	Clock (µs)	Data "A" (µs)	Data "S" (µs)	Data "D" (µs)	Data "F" (µs)	Data "W" (µs)	Data "E" (µs)	Data "R" (µs)	Data "T" (µs)
High	340	340	340	340	340	340	340	340	340
Low	959	939	939	939	939	939	939	939	939
High	1000.4	1190	1023	1022.8	1023	1023	1190	1023	1023
Low	1042.2	1441	1107	1190	1190	1107	1274	1107	1107
High	1083.6	1774	1190	1441	1274	1190	1441	1190	1190
Low	1126	-	1441	1525	1358	1441	1525	1358	1441
High	1167	-	1692	1775	1441	1692	1962	1442	1692
Low	1209	-	-	-	1525	-	-	1525	-
High	1251	-	-	-	1692	-	-	1692	-
Low	1293	-	-	-	-	-	-	-	-
High	1334	-	-	-	-	-	-	-	-
Low	1377	-	-	-	-	-	-	-	-
High	1418	-	-	-	-	-	-	-	-
Low	1460	-	-	-	-	-	-	-	-
High	1502	-	-	-	-	-	-	-	-
Low	1544	-	-	-	-	-	-	-	-
High	1585	-	-	-	-	-	-	-	-
Low	1627	-	-	-	-	-	-	-	-
High	1669	-	-	-	-	-	-	-	-
Low	1710	-	-	-	-	-	-	-	-
High	1751	-	-	-	-	-	-	-	-
Low	1793	-	-	-	-	-	-	-	-
High	1835	-	-	-	-	-	-	-	-