ECE 271, PS/2 Keyboard Piano Design Project, Group 7

Caspian Hedlund, Matthew Hotchkiss, Dominic Hsiao, and Andrew Johnson

December 4th, 2020

Contents

1 Project Description

For the Design Project, our group chose to implement a Personal System/2 (PS/2) keyboard piano on a Field Programmable Gate Array (FPGA) that would play notes at the correct frequency on a speaker when a key was pressed on the keyboard. Before getting started on implementing the task, it was crucial that we got background information on how a PS/2 keyboard works. A lot of the information was provided to us through Canvas, such Wikipedia and burtonsys.com which was important for gaining information on the Pins used in PS/2. Moreover, understanding oscilloscope readings that were provided were crucial to completing this project. On top of everything that was given to us, we needed to do research on our own, this led us to websites like digitkey.com which provided the hexadecimal number that corresponds to every key on the PS/2 keyboard.

1.1 Inputs and Outpus

Inputs: This design accepts key presses from a PS/2 keyboard and passes them to the FPGA. The specific keys implemented are the 'a', 's', 'd', 'f', 'w', 'e', 'r', and 't' keys.

Outputs: A speaker creates a sound when one of the implemented keys is pressed. Each key creates a different tone similar to a piano. The speaker is connected to the FPGA using wires.

1.2 Research

What is a PS/2 Keyboard Connection? A PS/2 keyboard has two primary signals which are pertinent to the communication between it and its connections: the clock and data. A PS/2 clock is generated by the keyboard itself and is used primarily to send data from the keyboard to the host, or in this case the FPGA. With that being said, data is sent in 11 bit packages from the device to the host, each bit being sent one at a time. To that end, 11 clock cycles represents the complete sending of one data signal from device to host. More specifically, the PS/2 clock operates on a falling edge basis, meaning that data is written on the rising edge and should be read on the falling edge of the clock. Furthermore, data is passed in the order of least significant bit (LSB) to most significant bit (MSB). After data has been written on 11 clock cycles, both data and clock default to high while the keyboard is idle, i.e. there is no new data.

What does the data signal represent? Data, this 11 bit signal, is indicative of which key has been pressed on the PS/2 keyboard. Furthermore, only 8 bits of the 11 are used to signal the specific key at hand, which can be understood considering the "make code" for each key is a two digit hexadecimal value which requires only 8 bits of data. The other 3 bits of data are the start, stop, and parity bits. The start bit is always 0, the stop bit is always 1, and the parity bit is often used for error signaling by the keyboard. Thus, the sequential organization of the 11 bits of data from LSB to MSB reads as follows: start, data[0], data[1], data[2], data[3], data[4], data[5], data[6], data[7], parity, stop. This scheme is described in Table 1 below.

Key	Make Code (hex)
a	1C
w	1D
s	1B
d	23
e	24
f	2B
r	2D
t	2C

Table 1: Keyboard encoding scheme. The keyboard keys in the left column are represented by a make code in the right column. Each make code is composed of 8 bits, which are represented in hexadecimal in the table. The make code is transferred over the data signal from the PS/2 keyboard upon pressing one of the keys. The information in the table is referenced from DigiKey[1].

What is a musical note? Musical notes are merely sound waves recognizable to the human ear that operate at particular frequencies. To that end, a singular musical note can be identified by its frequency of oscillation. The musical notes used by the PS/2 keyboard piano and their frequencies are shown in Table 2 below.

Key	Note	Frequency (Hz)
a	A	220
w	B♭	233
s	B	247
d	C	262
e	D♭	277
f	D	294
r	E♭	311
t	E	330

Table 2: Each key in the leftmost column corresponds to a particular musical note in the center column. The frequency of the noise required for the musical note is in the rightmost column. The frequency is used as the frequency of the clock driving the speaker. The information in the table is referenced from Wikipedia[2].

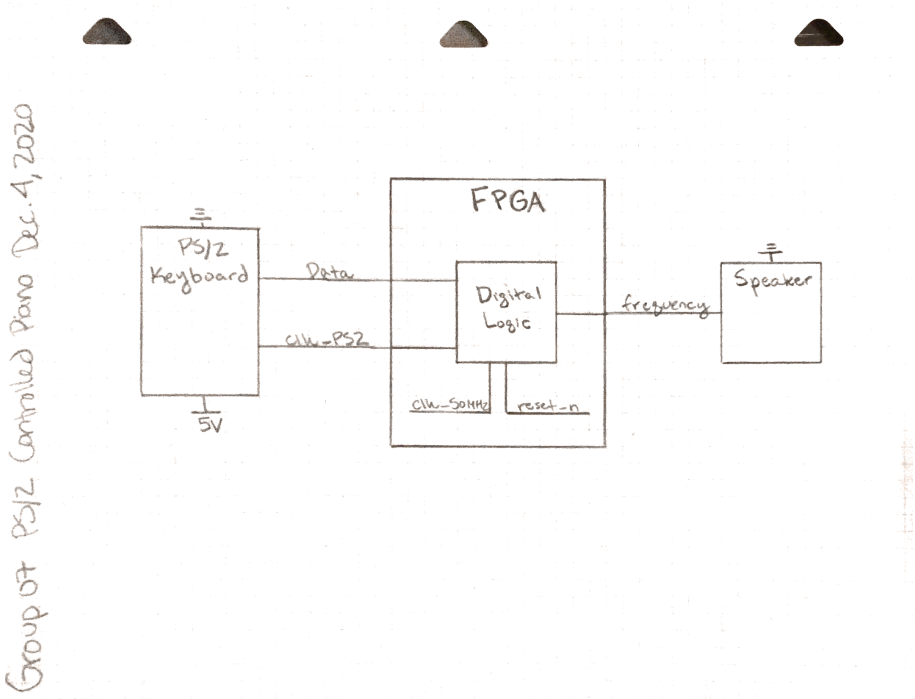

Figure 1: Picture of the top-level design.

After compiling our research, we had a general idea of what needed to be done in order to make a working PS/2 keyboard Piano. The first step towards completion was identifying the inputs and outputs of the entire project. For inputs, we knew that we would be taking in data from the PS/2 keyboard. Then we knew that there would have to be a reset_n on the FPGA to effectively turn the whole thing on. Finally, we knew that we would need to take input from both the onboard 50 Mhz clock on the FPGA, and the clock of the PS/2 keyboard. In terms of output, there would only be one as the only thing that needs to be output is the frequency to a speaker. All of these aspects of the project are shown in the block diagram in Figure 1.

1.3 Hardware Implementation

The hardware implementation of the keyboard was done mostly with parts that were on hand. Because of this, several of them were suboptimal. A PS2 port to pins adapter was made by cutting the male end off of a PS2 extension cord, and analyzing it to find which wires corresponded to which

pins. Using several resistors and a potentiometer, the voltage of the signals from the keyboard was reduced. Due to a lack of available resistors, we were only able to get them down to 3.7-4 volts rather than the 3.6 volts that the FPGA datasheet recommends for a high signal, but we tested it anyway and found it to be functional. The output audio signal was connected to a $\frac{1}{4}$ inch mono audio jack, which was then connected to a guitar amplifier. We found that while it was able to play the different tones, it was very quiet and played as a clicking noise rather than a constant tone. We believe that this is due to the poor signal quality that our system outputs.

2 High Level Description

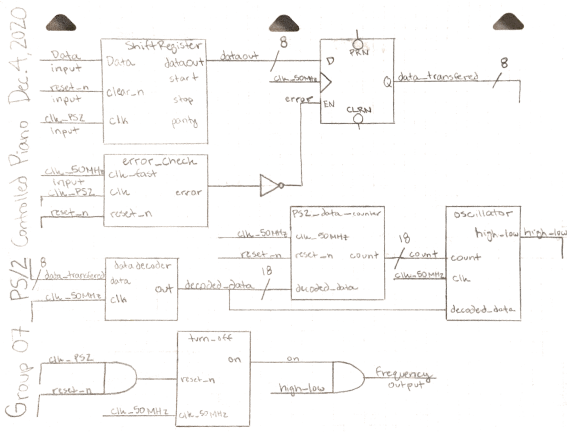

Figure 2: The top-level design of the entire project.

From here, we can piece together the entire design of the project. We know that the data from the keyboard gets inputted into a shift register that consists of 11 bits. Eight of the bits get taken and put into data input of an enabled D Flip-Flop. The D Flip-Flop is enabled by the error_check block, which ensures that all 11 bits of data from the PS/2 keyboard have been stored in the shift register. The output of the enabled D Flip-Flop goes into a data decoder that takes the 8 bit number, and turns into a ratio that gets inputted into a PS/2_data_counter; this block counts to that number, which corresponds to the specific note and its frequency. For the time period in which the counter is counting, an oscillator will make a comparison of values to create a frequency output that oscillates between high and low, thus producing a square wave of the correct frequency. Prior to outputting the frequency, the product of the output of the oscillator and the output of the turn_off block is computed to ensure that the sound does not play for too long. The output of said product is the output of the system: the frequency of the note. All of these aspects are shown in Figure 2.

Inputs: Reads a PS/2 keyboard for key presses.

Outputs: Plays a sound of a unique tone on a speaker depending on the key pressed.

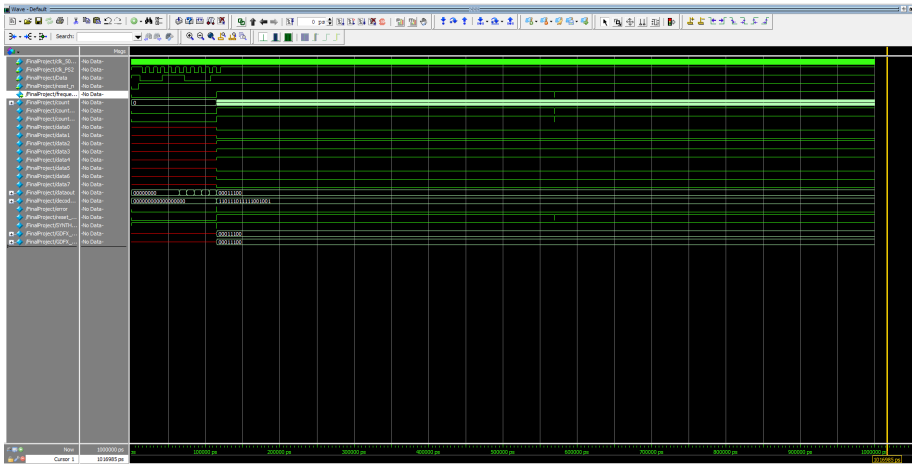

Figure 3: The simulation results for the top-level design.

Figure 3 shows the simulation results for the top-level design. In the simulation, the 'a' key is pressed. For the first part of the simulation the make code is transmitted. Afterwards, a clock signal reduced to 220Hz (the frequency of the musical note A) is outputted from the FPGA to the speaker. The simulation shows the design correctly interpreting the key pressed and outputting the correct frequency.

2.1 Shift Register

The shift register directly receives data from the PS/2 keyboard data line. This means that there needs to be 11 D Flip-Flops connected in series to store the data. With research on the PS/2 connection in mind, the data bits stored in the "middle" D Flip-Flops represent the make code and are bussed together and outputted. It is important to note that the start, stop, and parity bits were not utilized in the design. For inputs, there are the following: data coming from the keyboard, clear_n which is an asynchronous reset to the system of D Flip-Flops, and the PS2_clk that controls when the data is input into the D Flip-Flops. Per PS/2 protocol, data must be read on the falling edge of the PS/2 clock. To account for this in the design, the PS/2_clk input is negated.

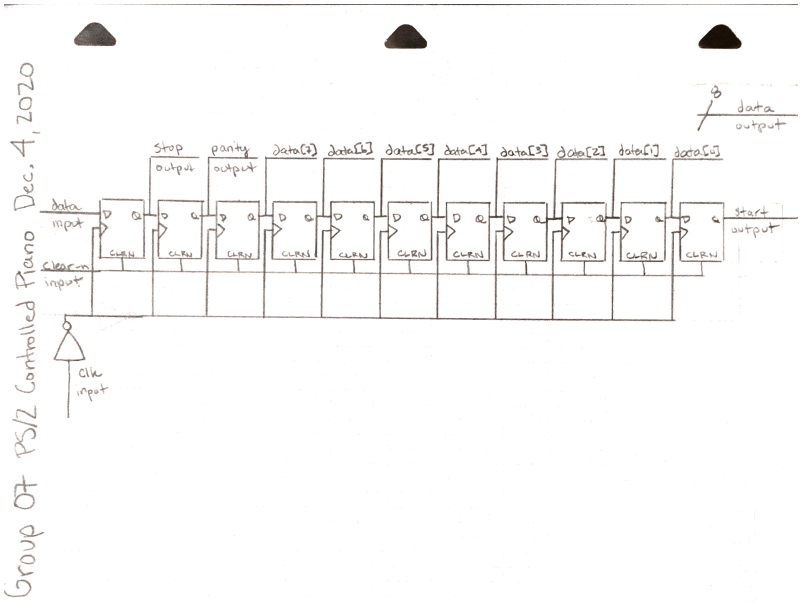

Figure 4: Expanded view of shift register in top-level design.

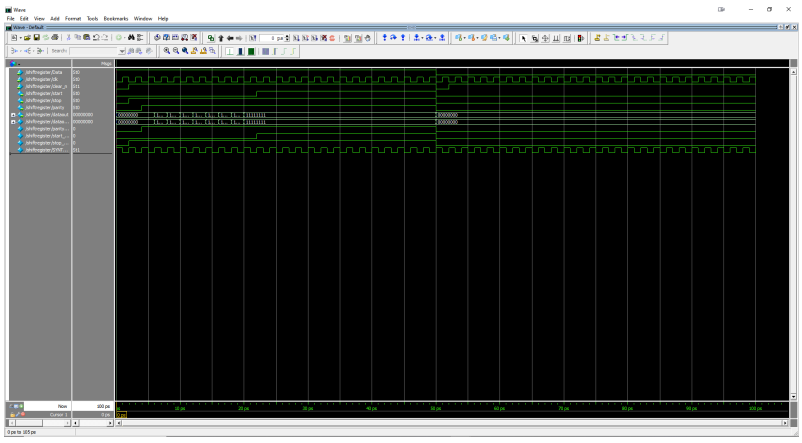

Figure 5: Simulation results from ModelSim after simulating an arbitrary set of bits going into the shift register.

2.1.1 D-Flip-Flop

A singular D Flip Flop is able to store one bit of data, and will only pass said bit from D to Q upon the rising edge of the clock signal. When D Flip-Flops are connected together in series, as depicted in the shift register, the ability is given to take in data bit by bit, and then output them as a whole bus of bits.

2.2 Enabled D-Flip-Flop

This block is very similar to that of a regular D Flip-Flop, except that this block has a built in enable so the data from the shift register isn't passed to the data decoder unless the enable is true. The inputs that are being used in this block are data, which comes from the output from the shift register, the clock, which comes from the 50 MHz FPGA clock, and the enable input which comes from the error check.

2.3 Error Check

It was mentioned above that the D Flip-Flops in series output the bits that have been stored on each rising edge of the clock. However, it is essential that the data does not leave the shift register and pass to the decoder until all 11 of the bits have been read. This unit checks to make sure that there are 11 bits of new data before the data is sent through the enabled D Flip-Flop. Upon verifying that 11 bits have been registered, the error check will reset its internal counter and enable the D Flip-Flop.

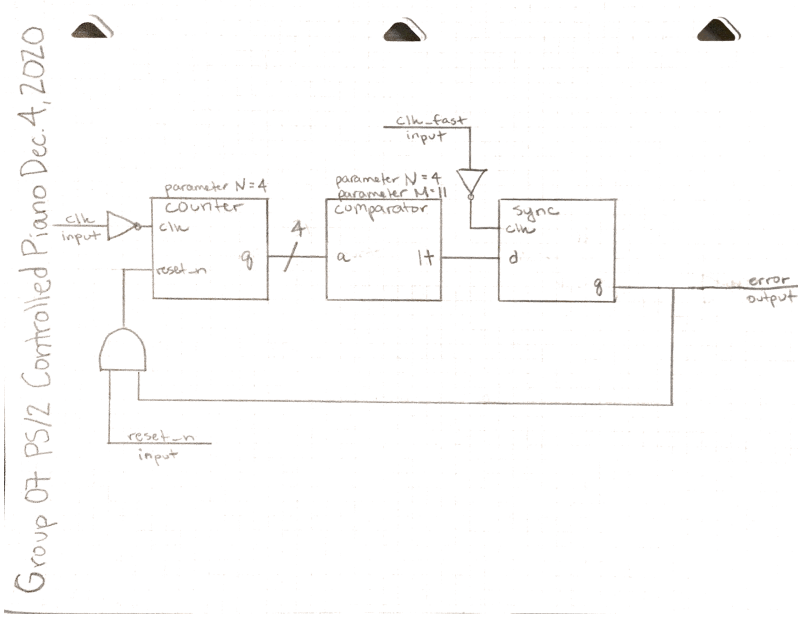

Figure 6: Expanded view of the error check unit in the top-level design.

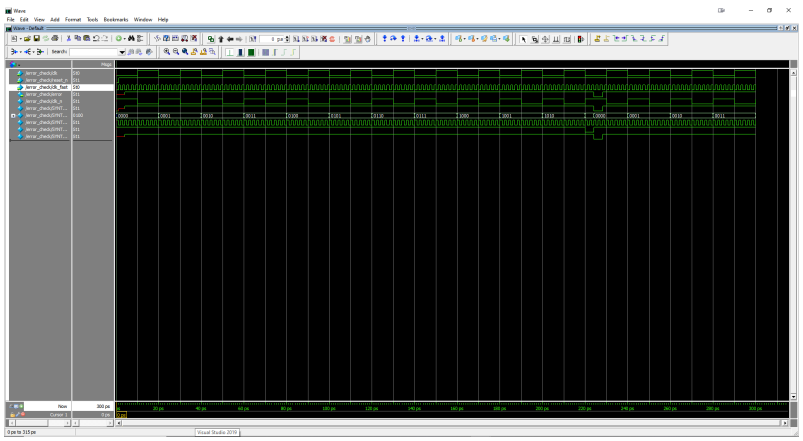

Figure 7: Simulation results from ModelSim after simulating possible errors.

2.3.1 Counter

The first block that is needed to complete this unit is a counter. In this case, the counter is essentially an incrementer on every falling edge of the PS/2 clock. This means that it is in sync with the shift register. Using this allows us to keep track of how many bits have been read in the shift register for each new set of data. On every falling edge of the clock cycle, the output is the number at which the incrementer is at. This counter is parameterized by 4 bits, as that is all that is needed to store a value of 11. In terms of inputs, we have the PS/2_clock and the product of the reset_n input and the output of the synchronizer, which are the external and internal resets respectively.

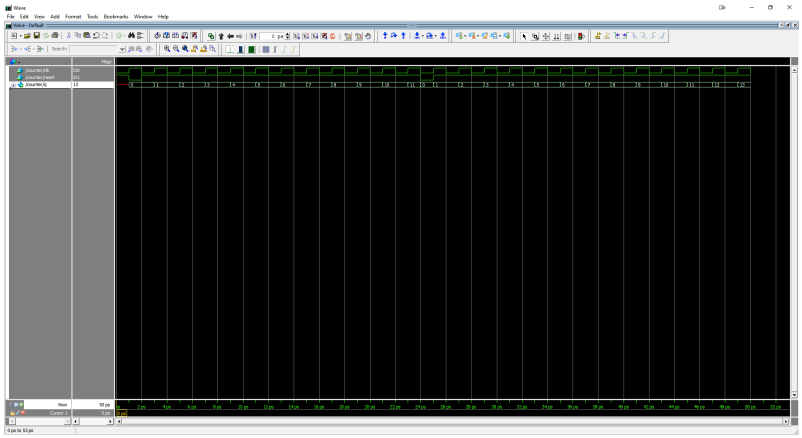

Figure 8: Simulation results from ModelSim after simulating possible errors.

2.3.2 Comparator

As the name suggests, this block compares two numbers. In this case, this comparator is taking in input from the output of the counter. The comparator then does a less than comparison with the input, and a predefined parameter M. In this case, M is equal to 11, because 11 signifies a full shift register. The output of the comparator is essentially a true or false of the less than comparison. If a is indeed less than M, the output would be a 1 or true, and anything else would output a 0 or false.

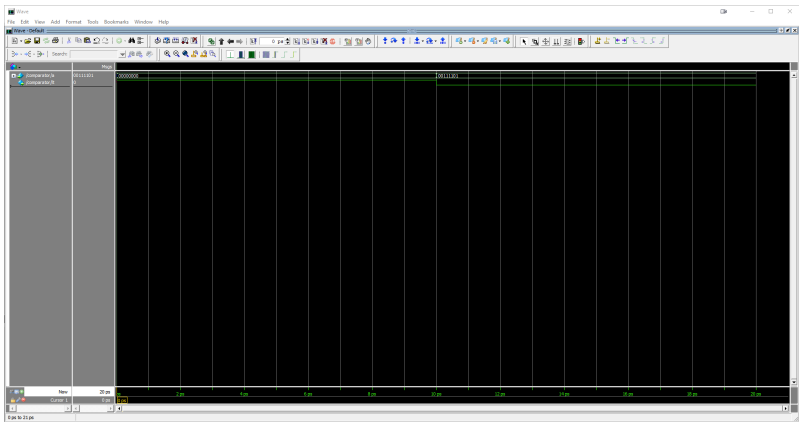

Figure 9: Simulation results from ModelSim after simulating possible comparisons.

2.3.3 Synchronizer

The synchronizer serves a similar purpose to that of a D Flip-Flop. To be more specific, in this case it is being used to sync the data with the clock. In this case, the inputs are a negated FPGA 50 Mhz clock. It is negated as we want the falling edge. The other input is the output from the

comparator. Then, on each falling edge of the clock, the data inputted from the comparator is then outputted to the overall output of the unit, and the "and" gate to signify a reset.

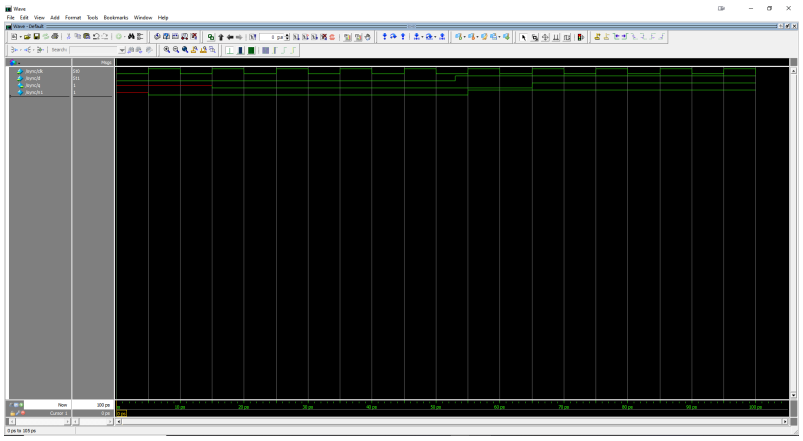

Figure 10: Simulation results from ModelSim after simulating possible synchronizations.

2.4 Data Decoder

The purpose of this block is to generate a number which must be counted to in order to generate a frequency. The input of this block is the 8 bit number that is outputted from the enabled D Flip-Flop. To implement this, a case statement was used. On the left side of the case statement is the 8 bit number that was inputted from the enabled D Flip-Flop. As we know, each key is a different 8 bit number, thus the case statement checks which key has been pressed via the 8 bit number, then sets the output to the frequency of the FPGA clock (50 Mhz) divided by the frequency of the note that has been coded to the key that has been pressed. The result is a ratio by which we must slow our FPGA clock. Please refer to the following table for the ratios as they relate to the keys, their make codes, and their frequencies.

Key	Make Code (hex)	Note	Frequency (Hz)	Ratio of 50MHz/Frequency
a	1C	A	220	227273
w	1D	B♭	233	214592
s	1B	B	247	202429
d	23	C	262	190840
e	24	D♭	277	180505
f	2B	D	294	170068
r	2D	E♭	311	160722
t	2C	E	330	151515

Table 3: The key column contains the letter of the key pressed on the keyboard. he make code column contains the bit sequence representing the key. The note column represents the musical note that is played upon pressing the key. The frequency column contains the frequency of the tone of the note. The information in the frequency column is referenced from Wikipedia[2]. The ratio column is the ratio of 50MHz over the frequency to determine the number of clock cycles to count before emitting an oscillation on the speaker.

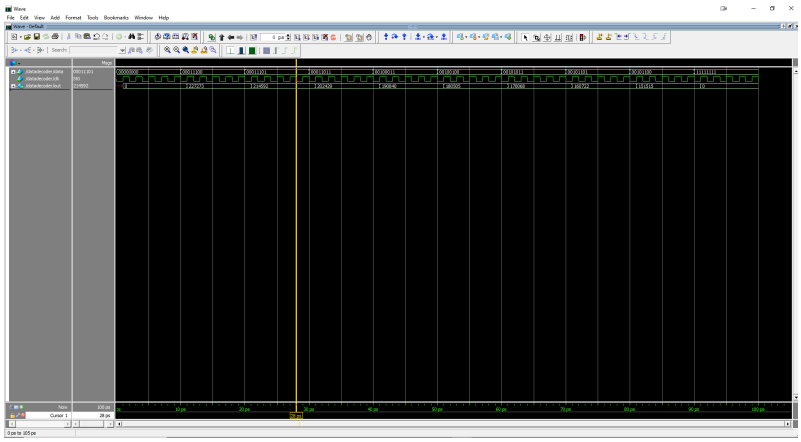

Figure 11: Simulation results from ModelSim of the data decoder after decoding arbitrary values.

2.5 PS2_data_counter

The purpose of the PS2_data_counter is to repeatedly count to the number outputted from the decoder. In order to achieve this, the inputs to this unit are the 50 MHz clock, reset_n, and the 18 bit number from the decoder. In order to count, there is a counter that increments upon the rising edge of the clock. Then, a comparator compares the number from the decoder to the current count, and the synchronizer synchronizes the output data on the 50 MHz clock. When the value of the synchronized output is zero, the counter resets back to zero and begins counting again.

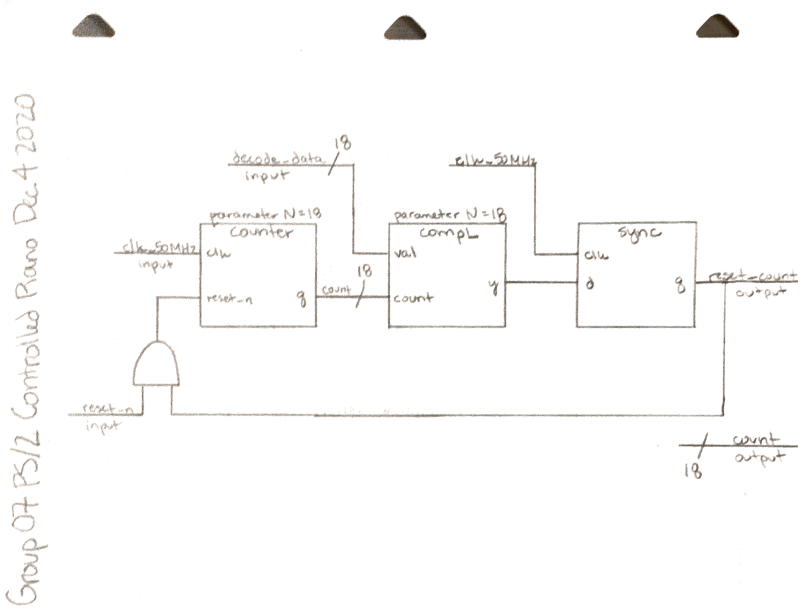

Figure 12: Expanded view of the data counter unit in the top-level design.

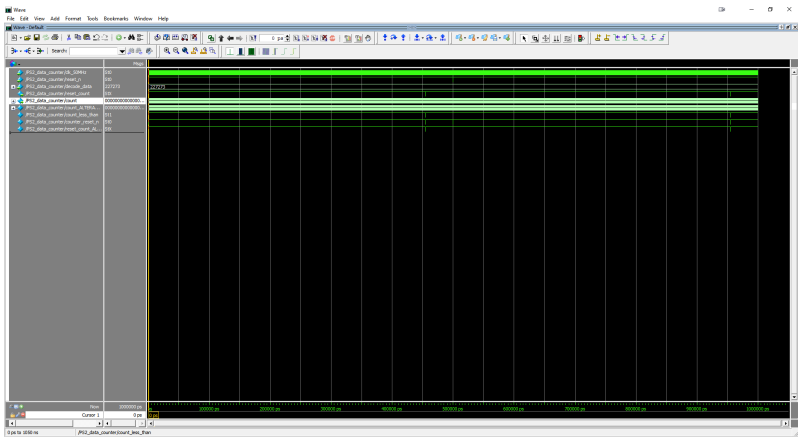

Figure 13: Simulation results from ModelSim of the data counter after counting to arbitrary values.

2.5.1 Counter

Please refer to previous instantiations of the counterfor more thorough details. The input for this counter is the 50 Mhz FPGA clock, and an active low reset which is based upon the product of the reset switch and the synchronizer. In this case, the counter has the ability to count to an 18 bit number, as specified by its N parameter. CompL - CompL is essentially a comparator, except in this block we are comparing two

2.5.2 CompL

CompL is essentially a comparator, except in this block we are comparing two inputted N-bit data values, as opposed to an inputted value and a parameter value. Hence, the two inputs for this are the value from the decoder, and the current number from the counter. This block checks if the number from the counter is less than the value from the decoder. The output y is based upon whether the inequality statement is true.

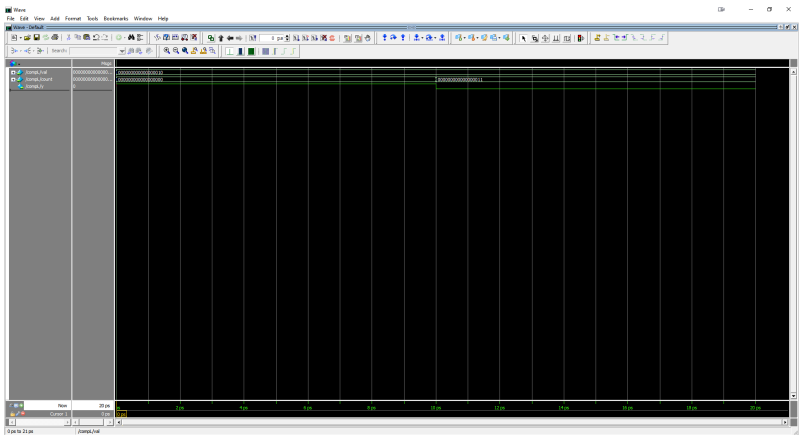

Figure 14: Simulation results from ModelSim of the CompL module after comparing arbitrary values.

2.5.3 Synchronizer

Please refer to previous instantiations of the synchronizer for more thorough details. In this case, the inputs are the 50 MHz FPGA clock, and the high or low from the CompL block, then the output is synchronized high or low.

2.6 Oscillator

The purpose of the oscillator is to manufacture a square wave oscillation. In order to do so, half of the count produced in the PS2_data_counter must equate to a high signal and the other half low. So as the counter progresses and resets continually, the signal oscillates between high and low. The inputs of this unit are the data from the decoder, the data from the PS/2 data counter, and the 50 Mhz FPGA clock. The output is a high or low, which represents whether the square wave is in a high or low phase.

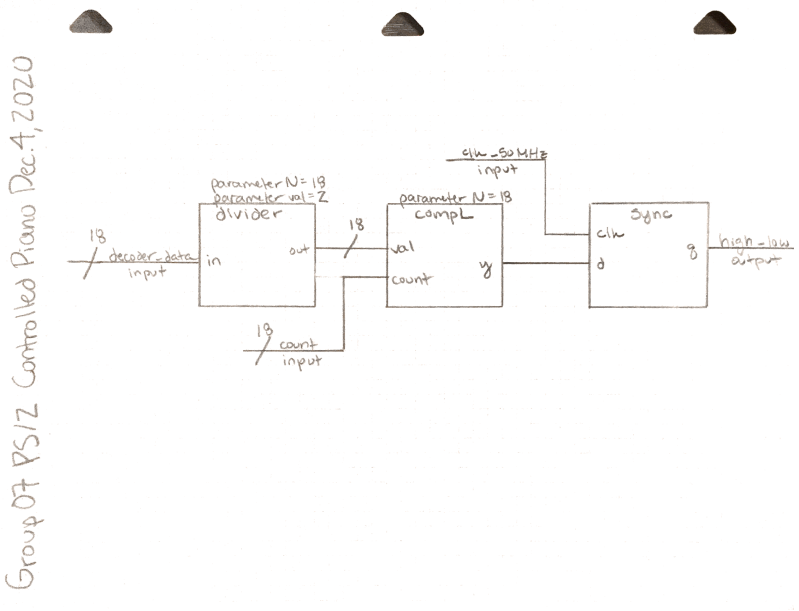

Figure 15: Expanded view of the oscillator unit in the top-level design.

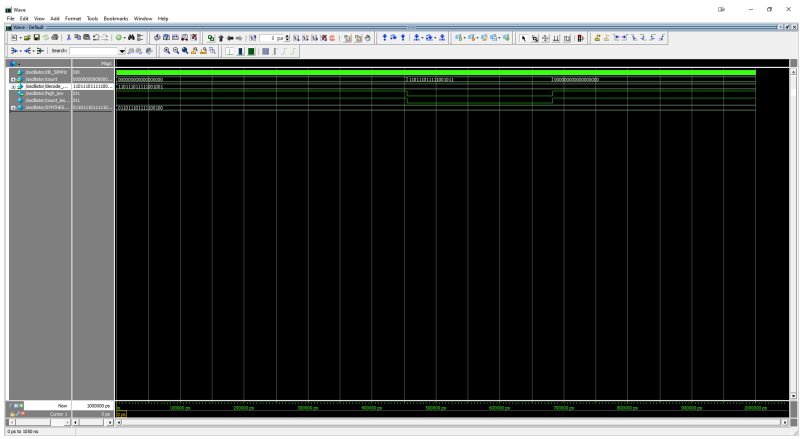

Figure 16: Simulation results from ModelSim of the oscillator after counting to arbitrary values.

2.6.1 Divider

The divider simply accepts two parameters N and val, and divides an N-bit input, in, by val. In this case, N is 18 and val is 2 to divide the number from the decoder by 2. The input is the number from the decoder, and the output is the number from the decoder divided by 2.

2.6.2 CompL

Please refer to previous instantiations of CompL for more thorough details. In this case, the CompL is checking if the input from the counter is less than the value from the divider and outputs

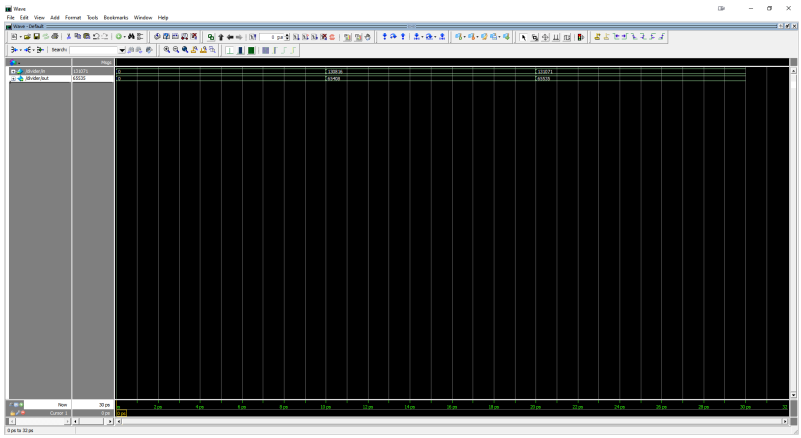

Figure 17: Simulation results from ModelSim of the divider after dividing arbitrary values.

the truth of the inequality. For the first half of the count, CompL will evaluate true, and for the latter half of the count, CompL will evaluate false. This is how the oscillation is created.

2.6.3 Synchronizer

Please refer to previous instantiations of the synchronizer for more thorough details. The output for this synchronizer is either a high or low value generated from the CompL block, which represents the wave of the note.

2.7 Turn Off

In the case that a key is pressed, a note will be played, however, said note mustn't be played without end. In order to prevent this, a unit needs to be implemented that turns off the note after a specified amount of time. The inputs of this unit are the 50 Mhz FPGA clock and the product of the reset switch and the PS/2 clock for reset_n. The output of this unit is a high or low (true or false) based upon if the note has passed its specified run time.

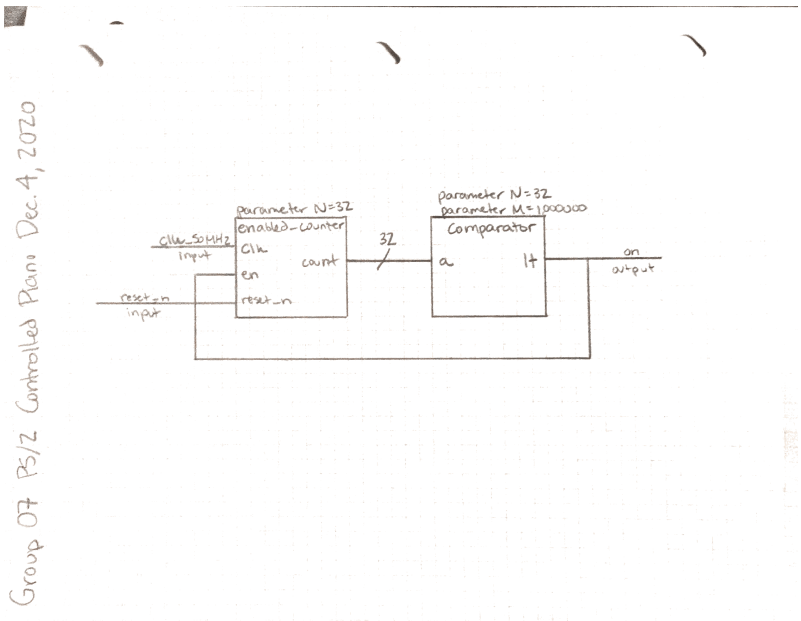

Figure 18: Expanded view of the turn_off unit in the top-level design.

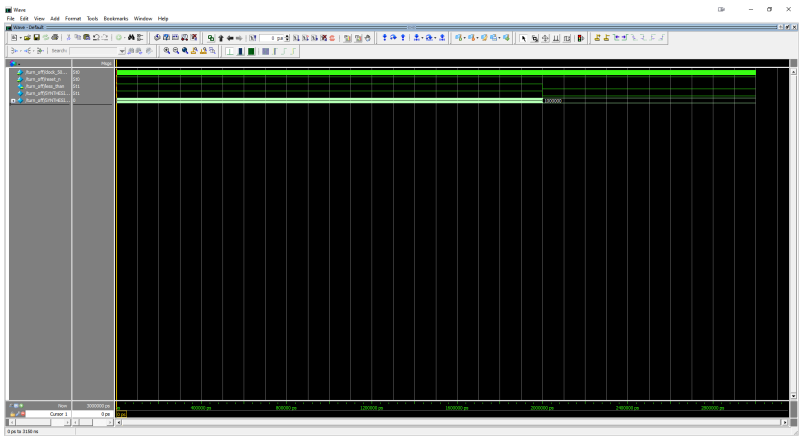

Figure 19: Simulation results from ModelSim of the turn_off unit after counting to desired value.

2.7.1 Enable counter

The purpose of this block is to count the time that the note has been played for. In order to do this, it is essentially a counter that increments on every clock cycle. However, the difference is that this block is enabled by its output, meaning when the comparison is false, the count will stop until the counter is reset. And, the reset_n of this block is attached to the PS/2 clock, in addition to the reset_n input of the whole system, because new data is being read if the PS/2 clock drives low, meaning the count should be reset so the note can play if the data is valid. The inputs to this block are the 50 Mhz FPGA clock, the PS/2 clock, and the reset switch. Finally, the output is the incremented number on every clock cycle.

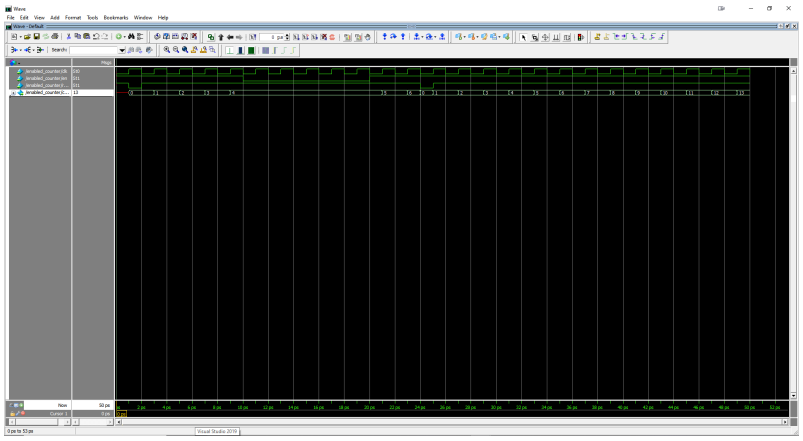

Figure 20: Simulation results from ModelSim of the enable counter after counting to desired value with and without the enable.

2.7.2 Comparator

Please refer to previous instantiations of the comparator for more thorough details. In this instance of the comparator, we are seeing if the number output by the enabled counter is less than 1000000. If the output is true, the note continues to play, if the output is false the note ceases to play and the counter is disabled. The number 1 million was chosen because it is roughly .02 seconds at 50 MHz.

A SystemVerilog Files

```
module comparator #(parameter N = 8, M = 60)
                                        (input logic [(N-1):0] a, output logic lt);
        assign lt = (a < M);
endmodule
```

```
module compL #(parameter N = 18)(input logic [N-1:0] val, input logic [N-1:0] count, output logic y);
        always_comb
                y = (count<val);
endmodule
```

```
module compL2 #(parameter N = 18)(input logic [N-1:0] val, input logic [N-1:0] count, output logic y);
        always_comb
                y = (count<(val/2));
endmodule
```

```
module counter #(parameter N = 8)
                                (input logic clk, input logic reset, output logic [(N-1):0] q);
                always_ff@(posedge clk, negedge reset)
                        if (reset == 0) q <= 0;
                        else q <= q+1;
endmodule
```

```
module datadecoder(input logic [7:0] data, input logic clk, output logic [17:0] out);

        always_ff @(posedge clk)
                case(data)
                        8'b00011100: out <= 227273; //'a' key A note
                        8'b00011101: out <= 214592; //'w' key B sharp note
                        8'b00011011: out <= 202429; //'s' key B note
                        8'b00100011: out <= 190840; //'d' key C note
                        8'b00100100: out <= 180505; //'e' key D sharp note
                        8'b00101011: out <= 170068; //'f' key D note
                        8'b00101101: out <= 160722; //'r' key E sharp note
                        8'b00101100: out <= 151515; //'t' key E note
                        default: out <= 0;
                endcase
endmodule
```

```
module divider #(parameter N = 18, val = 2)(input logic [N-1:0] in, output logic [N-1:0] out);

always_comb
        out = in/val;

endmodule
```

```
module enabled_counter #(parameter N = 10)(input logic clk, en, reset_n, output logic [N-1:0] count);

        always_ff@(posedge clk, negedge reset_n)
                        if (reset_n == 0) count <= 0;
                        else
                                if (en == 1)
                                        count <= count+1;
endmodule
```

```
// Copyright (C) 2018  Intel Corporation. All rights reserved.
// Your use of Intel Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Intel Program License
// Subscription Agreement, the Intel Quartus Prime License Agreement,
// the Intel FPGA IP License Agreement, or other applicable license
// agreement, including, without limitation, that your use is for
// the sole purpose of programming logic devices manufactured by
// Intel and sold by Intel or its authorized distributors.  Please
// refer to the applicable agreement for further details.

// PROGRAM                "Quartus Prime"
// VERSION                "Version 18.0.0 Build 614 04/24/2018 SJ Lite Edition"
// CREATED                "Wed Dec 02 12:12:12 2020"

module error_check(
        clk,
        reset_n,
        clk_fast,
        error
);


input wire      clk;
input wire      reset_n;
input wire      clk_fast;
output wire     error;

wire    clk_n;
wire    SYNTHESIZED_WIRE_0;
wire    [3:0] SYNTHESIZED_WIRE_1;
wire    SYNTHESIZED_WIRE_2;
wire    SYNTHESIZED_WIRE_3;
wire    SYNTHESIZED_WIRE_4;

assign  error = SYNTHESIZED_WIRE_4;




counter b2v_inst(
        .clk(clk_n),
        .reset(SYNTHESIZED_WIRE_0),
        .q(SYNTHESIZED_WIRE_1));
        defparam        b2v_inst.N = 4;


comparator      b2v_inst2(
        .a(SYNTHESIZED_WIRE_1),
```

```
53          .lt(SYNTHESIZED_WIRE_3));
54          defparam        b2v_inst2.M = 11;
55          defparam        b2v_inst2.N = 4;
56
57
58  sync    b2v_inst3(
59          .clk(SYNTHESIZED_WIRE_2),
60          .d(SYNTHESIZED_WIRE_3),
61          .q(SYNTHESIZED_WIRE_4));
62
63  assign  SYNTHESIZED_WIRE_0 = reset_n & SYNTHESIZED_WIRE_4;
64
65  assign  clk_n =  ~clk;
66
67  assign  SYNTHESIZED_WIRE_2 =  ~clk_fast;
68
69
70  endmodule
```

```
1   // Copyright (C) 2018  Intel Corporation. All rights reserved.
2   // Your use of Intel Corporation's design tools, logic functions
3   // and other software and tools, and its AMPP partner logic
4   // functions, and any output files from any of the foregoing
5   // (including device programming or simulation files), and any
6   // associated documentation or information are expressly subject
7   // to the terms and conditions of the Intel Program License
8   // Subscription Agreement, the Intel Quartus Prime License Agreement,
9   // the Intel FPGA IP License Agreement, or other applicable license
10  // agreement, including, without limitation, that your use is for
11  // the sole purpose of programming logic devices manufactured by
12  // Intel and sold by Intel or its authorized distributors.  Please
13  // refer to the applicable agreement for further details.
14
15  // PROGRAM              "Quartus Prime"
16  // VERSION              "Version 18.0.0 Build 614 04/24/2018 SJ Lite Edition"
17  // CREATED              "Wed Dec 02 12:30:55 2020"
18
19  module FinalProject(
20          clk_50MHz,
21          clk_PS2,
22          Data,
23          reset_n,
24          frequency
25  );
26
27
28  input wire      clk_50MHz;
29  input wire      clk_PS2;
30  input wire      Data;
31  input wire      reset_n;
32  output wire     frequency;
33
34  wire    [17:0] count;
35  wire    data_exists;
36  reg     [7:0] data_transfered;
37  wire    [7:0] dataout;
38  wire    [17:0] decode_data;
39  wire    error;
40  wire    SYNTHESIZED_WIRE_0;
41  wire    SYNTHESIZED_WIRE_1;
42  wire    SYNTHESIZED_WIRE_2;
43
44
45
46
47
48  shiftregister   b2v_inst(
49          .Data(Data),
50          .clear_n(reset_n),
51          .clk(clk_PS2),
52
53
54
55          .dataout(dataout));
56
57
58  turn_off        b2v_inst1(
59          .clock_50MHz(clk_50MHz),
60          .reset_n(SYNTHESIZED_WIRE_0),
61          .less_than(data_exists));
62
63  assign  frequency = SYNTHESIZED_WIRE_1 & data_exists;
64
65  assign  error =  ~SYNTHESIZED_WIRE_2;
66
67
68  PS2_data_counter        b2v_inst2(
69          .clk_50MHz(clk_50MHz),
70          .reset_n(reset_n),
71          .decode_data(decode_data),
72
73          .count(count));
74
75
76  oscillator      b2v_inst25(
77          .clk_50MHz(clk_50MHz),
78          .count(count),
79          .decode_data(decode_data),
80          .high_low(SYNTHESIZED_WIRE_1));
81
82  assign  SYNTHESIZED_WIRE_0 = clk_PS2 & reset_n;
83
84
85  datadecoder     b2v_inst6(
86          .clk(clk_50MHz),
87          .data(data_transfered),
88          .out(decode_data));
89
90
91  always@(posedge clk_50MHz)
92  begin
93  if (error)
94          begin
95          data_transfered[7:0] <= dataout[7:0];
96          end
```

```
97  end
98
99
100 error_check      b2v_inst9(
101         .clk_fast(clk_50MHz),
102         .clk(clk_PS2),
103         .reset_n(reset_n),
104         .error(SYNTHESIZED_WIRE_2));
105
106
107 endmodule
```

```
1   // Copyright (C) 2018  Intel Corporation. All rights reserved.
2   // Your use of Intel Corporation's design tools, logic functions
3   // and other software and tools, and its AMPP partner logic
4   // functions, and any output files from any of the foregoing
5   // (including device programming or simulation files), and any
6   // associated documentation or information are expressly subject
7   // to the terms and conditions of the Intel Program License
8   // Subscription Agreement, the Intel Quartus Prime License Agreement,
9   // the Intel FPGA IP License Agreement, or other applicable license
10  // agreement, including, without limitation, that your use is for
11  // the sole purpose of programming logic devices manufactured by
12  // Intel and sold by Intel or its authorized distributors.  Please
13  // refer to the applicable agreement for further details.
14
15  // PROGRAM              "Quartus Prime"
16  // VERSION              "Version 18.0.0 Build 614 04/24/2018 SJ Lite Edition"
17  // CREATED              "Wed Dec 02 12:12:49 2020"
18
19  module oscillator(
20          clk_50MHz,
21          count,
22          decode_data,
23          high_low
24  );
25
26
27  input wire      clk_50MHz;
28  input wire      [17:0] count;
29  input wire      [17:0] decode_data;
30  output wire     high_low;
31
32  wire    count_less_than_half;
33  wire    [17:0] SYNTHESIZED_WIRE_0;
34
35
36
37
38
39  sync    b2v_inst13(
40          .clk(clk_50MHz),
41          .d(count_less_than_half),
42          .q(high_low));
43
44
45  compL   b2v_inst5(
46          .count(count),
47          .val(SYNTHESIZED_WIRE_0),
48          .y(count_less_than_half));
49          defparam        b2v_inst5.N = 18;
50
51
52  divider b2v_inst7(
53          .in(decode_data),
54          .out(SYNTHESIZED_WIRE_0));
55          defparam        b2v_inst7.N = 18;
56          defparam        b2v_inst7.val = 2;
57
58
59  endmodule
```

```
1   // Copyright (C) 2018  Intel Corporation. All rights reserved.
2   // Your use of Intel Corporation's design tools, logic functions
3   // and other software and tools, and its AMPP partner logic
4   // functions, and any output files from any of the foregoing
5   // (including device programming or simulation files), and any
6   // associated documentation or information are expressly subject
7   // to the terms and conditions of the Intel Program License
8   // Subscription Agreement, the Intel Quartus Prime License Agreement,
9   // the Intel FPGA IP License Agreement, or other applicable license
10  // agreement, including, without limitation, that your use is for
11  // the sole purpose of programming logic devices manufactured by
12  // Intel and sold by Intel or its authorized distributors.  Please
13  // refer to the applicable agreement for further details.
14
15  // PROGRAM              "Quartus Prime"
16  // VERSION              "Version 18.0.0 Build 614 04/24/2018 SJ Lite Edition"
17  // CREATED              "Wed Dec 02 12:12:22 2020"
18
19  module PS2_data_counter(
20          clk_50MHz,
21          reset_n,
22          decode_data,
23          reset_count,
24          count
25  );
26
27
28  input wire      clk_50MHz;
29  input wire      reset_n;
30  input wire      [17:0] decode_data;
31  output wire     reset_count;
32  output wire     [17:0] count;
33
34  wire    [17:0] count_ALTERA_SYNTHESIZED;
35  wire    count_less_than;
36  wire    counter_reset_n;
37  wire    reset_count_ALTERA_SYNTHESIZED;
38
39
40
41
42
```

```
counter b2v_inst2(
        .clk(clk_50MHz),
        .reset(counter_reset_n),
        .q(count_ALTERA_SYNTHESIZED));
        defparam        b2v_inst2.N = 18;


sync    b2v_inst4(
        .clk(clk_50MHz),
        .d(count_less_than),
        .q(reset_count_ALTERA_SYNTHESIZED));

assign  counter_reset_n = reset_n & reset_count_ALTERA_SYNTHESIZED;


compL   b2v_inst7(
        .count(count_ALTERA_SYNTHESIZED),
        .val(decode_data),
        .y(count_less_than));
        defparam        b2v_inst7.N = 18;

assign  reset_count = reset_count_ALTERA_SYNTHESIZED;
assign  count = count_ALTERA_SYNTHESIZED;

endmodule
```

```
// Copyright (C) 2018  Intel Corporation. All rights reserved.
// Your use of Intel Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Intel Program License
// Subscription Agreement, the Intel Quartus Prime License Agreement,
// the Intel FPGA IP License Agreement, or other applicable license
// agreement, including, without limitation, that your use is for
// the sole purpose of programming logic devices manufactured by
// Intel and sold by Intel or its authorized distributors.  Please
// refer to the applicable agreement for further details.

// PROGRAM                 "Quartus Prime"
// VERSION                 "Version 18.0.0 Build 614 04/24/2018 SJ Lite Edition"
// CREATED                 "Wed Dec 02 12:13:01 2020"

module shiftregister(
        Data,
        clk,
        clear_n,
        start,
        stop,
        parity,
        dataout
);


input wire      Data;
input wire      clk;
input wire      clear_n;
output wire     start;
output wire     stop;
output wire     parity;
output wire     [7:0] dataout;

reg     [7:0] dataout_ALTERA_SYNTHESIZED;
reg     parity_ALTERA_SYNTHESIZED;
reg     start_ALTERA_SYNTHESIZED;
reg     stop_ALTERA_SYNTHESIZED;
wire    SYNTHESIZED_WIRE_12;




always@(posedge SYNTHESIZED_WIRE_12 or negedge clear_n)
begin
if (!clear_n)
        begin
        dataout_ALTERA_SYNTHESIZED[6] <= 0;
        end
else
        begin
        dataout_ALTERA_SYNTHESIZED[6] <= dataout_ALTERA_SYNTHESIZED[7];
        end
end


always@(posedge SYNTHESIZED_WIRE_12 or negedge clear_n)
begin
if (!clear_n)
        begin
        dataout_ALTERA_SYNTHESIZED[5] <= 0;
        end
else
        begin
        dataout_ALTERA_SYNTHESIZED[5] <= dataout_ALTERA_SYNTHESIZED[6];
        end
end


always@(posedge SYNTHESIZED_WIRE_12 or negedge clear_n)
begin
if (!clear_n)
        begin
        stop_ALTERA_SYNTHESIZED <= 0;
        end
else
        begin
        stop_ALTERA_SYNTHESIZED <= Data;
        end
end


always@(posedge SYNTHESIZED_WIRE_12 or negedge clear_n)
begin
if (!clear_n)
```

```
90            begin
91            dataout_ALTERA_SYNTHESIZED[4] <= 0;
92            end
93    else
94            begin
95            dataout_ALTERA_SYNTHESIZED[4] <= dataout_ALTERA_SYNTHESIZED[5];
96            end
97    end
98
99    assign   SYNTHESIZED_WIRE_12 =  ~clk;
100
101
102
103   always@(posedge SYNTHESIZED_WIRE_12 or negedge clear_n)
104   begin
105   if (!clear_n)
106           begin
107           dataout_ALTERA_SYNTHESIZED[3] <= 0;
108           end
109   else
110           begin
111           dataout_ALTERA_SYNTHESIZED[3] <= dataout_ALTERA_SYNTHESIZED[4];
112           end
113   end
114
115
116   always@(posedge SYNTHESIZED_WIRE_12 or negedge clear_n)
117   begin
118   if (!clear_n)
119           begin
120           dataout_ALTERA_SYNTHESIZED[2] <= 0;
121           end
122   else
123           begin
124           dataout_ALTERA_SYNTHESIZED[2] <= dataout_ALTERA_SYNTHESIZED[3];
125           end
126   end
127
128
129   always@(posedge SYNTHESIZED_WIRE_12 or negedge clear_n)
130   begin
131   if (!clear_n)
132           begin
133           dataout_ALTERA_SYNTHESIZED[1] <= 0;
134           end
135   else
136           begin
137           dataout_ALTERA_SYNTHESIZED[1] <= dataout_ALTERA_SYNTHESIZED[2];
138           end
139   end
140
141
142   always@(posedge SYNTHESIZED_WIRE_12 or negedge clear_n)
143   begin
144   if (!clear_n)
145           begin
146           dataout_ALTERA_SYNTHESIZED[0] <= 0;
147           end
148   else
149           begin
150           dataout_ALTERA_SYNTHESIZED[0] <= dataout_ALTERA_SYNTHESIZED[1];
151           end
152   end
153
154
155   always@(posedge SYNTHESIZED_WIRE_12 or negedge clear_n)
156   begin
157   if (!clear_n)
158           begin
159           start_ALTERA_SYNTHESIZED <= 0;
160           end
161   else
162           begin
163           start_ALTERA_SYNTHESIZED <= dataout_ALTERA_SYNTHESIZED[0];
164           end
165   end
166
167
168   always@(posedge SYNTHESIZED_WIRE_12 or negedge clear_n)
169   begin
170   if (!clear_n)
171           begin
172           dataout_ALTERA_SYNTHESIZED[7] <= 0;
173           end
174   else
175           begin
176           dataout_ALTERA_SYNTHESIZED[7] <= parity_ALTERA_SYNTHESIZED;
177           end
178   end
179
180
181   always@(posedge SYNTHESIZED_WIRE_12 or negedge clear_n)
182   begin
183   if (!clear_n)
184           begin
185           parity_ALTERA_SYNTHESIZED <= 0;
186           end
187   else
188           begin
189           parity_ALTERA_SYNTHESIZED <= stop_ALTERA_SYNTHESIZED;
190           end
191   end
192
193   assign   start = start_ALTERA_SYNTHESIZED;
194   assign   stop = stop_ALTERA_SYNTHESIZED;
195   assign   parity = parity_ALTERA_SYNTHESIZED;
196   assign   dataout = dataout_ALTERA_SYNTHESIZED;
197
198   endmodule
```

```
1   module sync(input logic clk, input logic d, output logic q);
2           logic n1;
3           always_ff@(posedge clk)
4                   begin
5                           n1 <= d;
```

```
                        q <= n1;
                end
endmodule
```

```
// Copyright (C) 2018  Intel Corporation. All rights reserved.
// Your use of Intel Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Intel Program License
// Subscription Agreement, the Intel Quartus Prime License Agreement,
// the Intel FPGA IP License Agreement, or other applicable license
// agreement, including, without limitation, that your use is for
// the sole purpose of programming logic devices manufactured by
// Intel and sold by Intel or its authorized distributors.  Please
// refer to the applicable agreement for further details.

// PROGRAM              "Quartus Prime"
// VERSION              "Version 18.0.0 Build 614 04/24/2018 SJ Lite Edition"
// CREATED              "Wed Dec 02 12:30:26 2020"

module turn_off(
        clock_50MHz,
        reset_n,
        less_than
);


input wire      clock_50MHz;
input wire      reset_n;
output wire     less_than;

wire    SYNTHESIZED_WIRE_0;
wire    [31:0] SYNTHESIZED_WIRE_1;

assign  less_than = SYNTHESIZED_WIRE_0;



enabled_counter b2v_inst(
        .clk(clock_50MHz),
        .en(SYNTHESIZED_WIRE_0),
        .reset_n(reset_n),
        .count(SYNTHESIZED_WIRE_1));
        defparam        b2v_inst.N = 32;


comparator      b2v_inst3(
        .a(SYNTHESIZED_WIRE_1),
        .lt(SYNTHESIZED_WIRE_0));
        defparam        b2v_inst3.M = 1000000;
        defparam        b2v_inst3.N = 32;


endmodule
```

B Simulation Files (Do scripts)

```
#############################################################################
# aHoldKeyTester.do
#############################################################################
add wave *
force reset_n 0 @ 0, 1 @ 400
force clk_50MHz 1 @ 0, 0 @ 1 -r 2

force clk_PS2 1 @ 000, 0 @ 509600, 1 @ 510100, 0 @ 510500, 1 @ 510900, 0 @ 511300, 1 @ 511700, 0 @ 512200, 1 @ 512600, 0 @ 513000, 1 @ 513400, 0 @ 513800, 1 @ 514200, 0 @ 514700, 1 @ 515100, 0 @ 515500, 1 @ 515900, 0 @ 516300, 1 @ 516800, 0 @ 517200, 1 @ 517600, 0 @ 518000, 1 @ 518400, 0 @ 5462400, 1 @ 5462900, 0 @ 5463300, 1 @ 5463700, 0 @ 5464100, 1 @ 5464500, 0 @ 5464900, 1 @ 5465400, 0 @ 5465800, 1 @ 5466200, 0 @ 5466600, 1 @ 5467000, 0 @ 5467500, 1 @ 5467900, 0 @ 5468300, 1 @ 5468700, 0 @ 5469100, 1 @ 5469500, 0 @ 5470000, 1 @ 5470400, 0 @ 5470800, 1 @ 5471200, 0 @ 6381600, 1 @ 6382000, 0 @ 6382400, 1 @ 6382800, 0 @ 6383300, 1 @ 6383700, 0 @ 6384100, 1 @ 6384500, 0 @ 6384900, 1 @ 6385400, 0 @ 6385800, 1 @ 6386200, 0 @ 6386600, 1 @ 6387000, 0 @ 6387400, 1 @ 6387900, 0 @ 6388300, 1 @ 6388700, 0 @ 6389100, 1 @ 6389500, 0 @ 6389900, 1 @ 6390400, 0 @ 7319200, 1 @ 7319600, 0 @ 7320000, 1 @ 7320500, 0 @ 7320900, 1 @ 7321300, 0 @ 7321700, 1 @ 7322100, 0 @ 7322500, 1 @ 7323000, 0 @ 7323400, 1 @ 7323800, 0 @ 7324200, 1 @ 7324600, 0 @ 7325100, 1 @ 7325500, 0 @ 7325900, 1 @ 7326300, 0 @ 7326700, 1 @ 7327100, 0 @ 7327600, 1 @ 7328000, 0 @ 8256200, 1 @ 8256600, 0 @ 8257000, 1 @ 8257400, 0 @ 8257900, 1 @ 8258300, 0 @ 8258700, 1 @ 8259100, 0 @ 8259500, 1 @ 8259900, 0 @ 8260400, 1 @ 8260800, 0 @ 8261200, 1 @ 8261600, 0 @ 8262000, 1 @ 8262500, 0 @ 8262900, 1 @ 8263300, 0 @ 8263700, 1 @ 8264100, 0 @ 8264500, 1 @ 8265000, 0 @ 9175600, 1 @ 9176000, 0 @ 9176400, 1 @ 9176900, 0 @ 9177300, 1 @ 9177700, 0 @ 9178100, 1 @ 9178500, 0 @ 9179000, 1 @ 9179400, 0 @ 9179800, 1 @ 9180200, 0 @ 9180600, 1 @ 9181000, 0 @ 9181500, 1 @ 9181900, 0 @ 9182300, 1 @ 9182700, 0 @ 9183100, 1 @ 9183500, 0 @ 9184000, 1 @ 9184400
force Data 1 @ 000, 0 @ 509400, 1 @ 512000, 0 @ 514500, 1 @ 517800, 0 @ 5462200, 1 @ 5464800, 0 @ 5467300, 1 @ 5470600, 0 @ 6381400, 1 @ 6383900, 0 @ 6386400, 1 @ 6389700, 0 @ 7319000, 1 @ 7321500, 0 @ 7324000, 1 @ 7327400, 0 @ 8256000, 1 @ 8258500, 0 @ 8261000, 1 @ 8264300, 0 @ 9175400, 1 @ 9177900, 0 @ 9180400, 1 @ 9183800

run 20000000
```

```
#############################################################################
# aKeyTester.do
#############################################################################
add wave *
force reset_n 0 @ 0, 1 @ 400
force clk_50MHz 1 @ 0, 0 @ 1 -r 2

force clk_PS2 1 @ 000, 0 @ 307500, 1 @ 328200, 0 @ 349100, 1 @ 369800, 0 @ 390900, 1 @ 411600, 0 @ 432700, 1 @ 453400, 0 @ 474500, 1 @ 495200, 0 @ 516400, 1 @ 537000, 0 @ 558200, 1 @ 578800, 0 @ 600000, 1 @ 620600, 0 @ 641800, 1 @ 662400, 0 @ 682800, 1 @ 703500, 0 @ 724600, 1 @ 745300
force Data 1 @ 000, 0 @ 297700, 1 @ 423200, 0 @ 548600, 1 @ 715100

run 5000000
```

```
#############################################################################
# comparatorTester.do
#############################################################################
add wave *
force N 4 @ 0
force M 11 @ 0
force a 1 @ 0, 12 @ 10
run 20
```

```
#############################################################################
# counterTester.do
#############################################################################
add wave *
force clk 0 @ 0, 1 @ 1 -r 2
force reset 1 @ 0, 0 @ 1, 1 @ 2, 0 @ 24, 1 @ 25
run 50
```

```
#############################################################################
# dataDecoderTester.do
#############################################################################
add wave *
force clk 0 @ 0, 1 @ 1 -r 2
force data 0 @ 0, 00011100 @ 10, 00011101 @ 20, 00011011 @ 30, 00100011 @ 40, 00100100 @ 50, 00101011 @ 60, 00101101 @ 70, 00101100 @ 80, 11111111 @ 90
run 100
```

```
#############################################################################
# dividerTester.do
#############################################################################
add wave *
force in 0000000000000000 @ 0, 1111111100000000 @ 10, 1111111111111111 @ 20
run 30
```

```
#############################################################################
# dKeyTester.do
#############################################################################
add wave *
force reset_n 0 @ 0, 1 @ 400
force clk_50MHz 1 @ 0, 0 @ 1 -r 2

force clk_PS2 1 @ 000, 0 @ 307400, 1 @ 328300, 0 @ 349000, 1 @ 369800, 0 @ 390900, 1 @ 411600, 0 @ 432800, 1 @ 453400, 0 @ 474600, 1 @ 495300, 0 @ 516400, 1 @ 537200, 0 @ 558300, 1 @ 579000, 0 @ 600100, 1 @ 620900, 0 @ 642000, 1 @ 662700, 0 @ 683100, 1 @ 703800, 0 @ 724900, 1 @ 745600
force Data 1 @ 000, 0 @ 297600, 1 @ 339500, 0 @ 423200, 1 @ 548800, 0 @ 590600, 1 @ 715400


run 5000000
```

```
#############################################################################
# eKeyTester.do
#############################################################################
add wave *
force reset_n 0 @ 0, 1 @ 400
force clk_50MHz 1 @ 0, 0 @ 1 -r 2

force clk_PS2 1 @ 000, 0 @ 307400, 1 @ 328600, 0 @ 349000, 1 @ 369900, 0 @ 390900, 1 @ 411600, 0 @ 432700, 1 @ 453400, 0 @ 474600, 1 @ 495200, 0 @ 516400, 1 @ 537100, 0 @ 558200, 1 @ 578900, 0 @ 600100, 1 @ 620700, 0 @ 641900, 1 @ 662600, 0 @ 683500, 1 @ 704100, 0 @ 725300, 1 @ 746000
force Data 1 @ 000, 0 @ 297600, 1 @ 423200, 0 @ 465000, 1 @ 548700, 0 @ 590500, 1 @ 673900

run 5000000
```

```
##################################################################################
# enableCounterTester.do
##################################################################################
add wave *
force clk 0 @ 0, 1 @ 1 -r 2
force reset_n 1 @ 0, 0 @ 1, 1 @ 2, 0 @ 24, 1 @ 25
force en 1 @ 0, 0 @ 10, 1 @ 20
run 50
```

```
##################################################################################
# errorCheckTester.do
##################################################################################
add wave *
force clk_fast 0 @ 0, 1 @ 1 -r 2
force clk 0 @ 0, 1 @ 10 -r 20
force reset_n 0 @ 0, 1 @ 1
run 300
```

```
##################################################################################
# fKeyTester.do
##################################################################################
add wave *
force reset_n 0 @ 0, 1 @ 400
force clk_50MHz 1 @ 0, 0 @ 1 -r 2

force clk_PS2 1 @ 000, 0 @ 307400, 1 @ 328500, 0 @ 349000, 1 @ 370000, 0 @ 390900, 1 @ 411600, 0 @
     432800, 1 @ 453500, 0 @ 474600, 1 @ 495300, 0 @ 516500, 1 @ 537100, 0 @ 561400, 1 @ 582400, 0 @
     600200, 1 @ 620800, 0 @ 642000, 1 @ 662700, 0 @ 683600, 1 @ 704300, 0 @ 725500, 1 @ 746100
force Data 1 @ 000, 0 @ 297600, 1 @ 339500, 0 @ 423200, 1 @ 465100, 0 @ 506900, 1 @ 548800, 0 @ 590600,
     1 @ 674100

run 5000000
```

```
##################################################################################
# oscilatorTester.do
##################################################################################
add wave *
force clk_50MHz 0 @ 0, 1 @ 1 -r 2
force decode_data 1101110111111001001 @ 0
force count 0 @ 0, 110111011111001011 @ 454546, 0 @ 681819
run 1000000
```

```
##################################################################################
# PS2DataCounterTester.do
##################################################################################
add wave *
force clk_50MHz 0 @ 0, 1 @ 1 -r 2
force reset_n 0 @ 0, 1 @ 1, 0 @ 500000, 1 @ 500002
force decode_data 1101110111111001001 @ 0, 110111011111001001 @ 500000
run 1000000
```

```
##################################################################################
# PS2KeyboardTester.do
##################################################################################
add wave *
force reset_n 0 @ 0, 1 @ 10000
force Data 1 @ 0, 0 @ 12000, 1 @ 42000, 0 @ 72000, 1 @ 107000
force clk_PS2 1 @ 0, 0 @ 15000, 1 @ 20000, 0 @ 25000, 1 @ 30000, 0 @ 35000, 1 @ 40000, 0 @ 45000, 1 @
     50000, 0 @ 55000, 1 @ 60000, 0 @ 65000, 1 @ 70000, 0 @ 75000, 1 @ 80000, 0 @ 85000, 1 @ 90000, 0 @
     95000, 1 @ 100000, 0 @ 105000, 1 @ 110000, 0 @ 115000, 1 @ 120000
force clk_50MHz 1 @ 0, 0 @ 1 -r 2
run 1000000
```

```
##################################################################################
# rKeyTester.do
##################################################################################
add wave *
force reset_n 0 @ 0, 1 @ 400
force clk_50MHz 1 @ 0, 0 @ 1 -r 2

force clk_PS2 1 @ 0ps, 0 @ 3075ps, 1 @ 3282ps, 0 @ 3491ps, 1 @ 3698ps, 0 @ 3910ps, 1 @ 4117ps, 0 @ 4328
     ps, 1 @ 4535ps, 0 @ 4747ps, 1 @ 4953ps, 0 @ 5165ps, 1 @ 5372ps, 0 @ 5584ps, 1 @ 5790ps, 0 @ 6002ps,
      1 @ 6209ps, 0 @ 6421ps, 1 @ 6627ps, 0 @ 6924ps, 1 @ 7043ps, 0 @ 7255ps, 1 @ 7462ps
force Data 1 @ 0ps, 0 @ 2977ps, 1 @ 3396ps, 0 @ 3814ps, 1 @ 4233ps, 0 @ 5069ps, 1 @ 5488ps, 0 @ 5906ps,
     1 @ 6741ps

run 5000000
```

```
##################################################################################
# shiftRegisterTester.do
##################################################################################
add wave *
force clk 0 @ 0, 1 @ 1 -r 2
force clear_n 0 @ 0, 1 @ 2, 0 @ 50, 1 @ 52
force Data 1 @ 0, 0 @ 50
run 100
```

```
##################################################################################
# sKeyTester.do
##################################################################################
add wave *
force reset_n 0 @ 0, 1 @ 400
force clk_50MHz 1 @ 0, 0 @ 1 -r 2

force clk_PS2 1 @ 000, 0 @ 307500, 1 @ 328300, 0 @ 349100, 1 @ 369800, 0 @ 390900, 1 @ 411700, 0 @
     432800, 1 @ 453400, 0 @ 474600, 1 @ 495300, 0 @ 516400, 1 @ 537100, 0 @ 558200, 1 @ 579000, 0 @
     600100, 1 @ 620700, 0 @ 641900, 1 @ 662600, 0 @ 683500, 1 @ 704100, 0 @ 725300, 1 @ 746000
force Data 1 @ 000, 0 @ 297700, 1 @ 339600, 0 @ 381400, 1 @ 423200, 0 @ 548700, 1 @ 673900

run 5000000
```

```
##################################################################################
# syncTester.do
##################################################################################
add wave *
force clk 0 @ 0, 1 @ 5 -r 10
force d 0 @ 0, 1 @ 53
run 100
```

```
#############################################################################
# turnOffTester.do
#############################################################################
add wave *
force reset_n 0 @ 0, 1 @ 1
force clock_50MHz 0 @ 0, 1 @ 1 -r 2
run 3000000
```

```
#############################################################################
# waterKeyTester.do
#############################################################################
add wave *
force reset_n 0 @ 0, 1 @ 400
force clk_50MHz 1 @ 0, 0 @ 1 -r 2

force clk_PS2 1 @ 000, 0 @ 509600, 1 @ 510000, 0 @ 510400, 1 @ 510900, 0 @ 511300, 1 @ 511700, 0 @
      512100, 1 @ 512500, 0 @ 513000, 1 @ 513400, 0 @ 513800, 1 @ 514200, 0 @ 514600, 1 @ 515100, 0 @
      515500, 1 @ 515900, 0 @ 516300, 1 @ 516700, 0 @ 517100, 1 @ 517600, 0 @ 518000, 1 @ 518400, 0 @
      1701200, 1 @ 1701600, 0 @ 1702000, 1 @ 1702500, 0 @ 1702900, 1 @ 1703300, 0 @ 1703700, 1 @ 1704100,
       0 @ 1704500, 1 @ 1705000, 0 @ 1705400, 1 @ 1705800, 0 @ 1706200, 1 @ 1706600, 0 @ 1707100, 1 @
      1707500, 0 @ 1707900, 1 @ 1708300, 0 @ 1708700, 1 @ 1709100, 0 @ 1709600, 1 @ 1710000, 0 @ 2105000,
       1 @ 2105400, 0 @ 2105800, 1 @ 2106300, 0 @ 2106700, 1 @ 2107100, 0 @ 2107500, 1 @ 2107900, 0 @
      2108300, 1 @ 2108800, 0 @ 2109200, 1 @ 2109600, 0 @ 2110000, 1 @ 2110400, 0 @ 2110900, 1 @ 2111300,
       0 @ 2111700, 1 @ 2112100, 0 @ 2112500, 1 @ 2112900, 0 @ 2113400, 1 @ 2113800, 0 @ 2136200, 1 @
      2136600, 0 @ 2137100, 1 @ 2137500, 0 @ 2137900, 1 @ 2138300, 0 @ 2138700, 1 @ 2139200, 0 @ 2139600,
       1 @ 2140000, 0 @ 2140400, 1 @ 2140800, 0 @ 2141300, 1 @ 2141700, 0 @ 2142100, 1 @ 2142500, 0 @
      2142900, 1 @ 2143300, 0 @ 2143800, 1 @ 2144200, 0 @ 2144600, 1 @ 2145000, 0 @ 3284900, 1 @ 3285300,
       0 @ 3285700, 1 @ 3286100, 0 @ 3286600, 1 @ 3287000, 0 @ 3287400, 1 @ 3287800, 0 @ 3288200, 1 @
      3288700, 0 @ 3289100, 1 @ 3289500, 0 @ 3289900, 1 @ 3290300, 0 @ 3290700, 1 @ 3291200, 0 @ 3291600,
       1 @ 3292000, 0 @ 3292400, 1 @ 3292800, 0 @ 3293200, 1 @ 3293700, 0 @ 4466500, 1 @ 4466900, 0 @
      4467300, 1 @ 4467700, 0 @ 4468100, 1 @ 4468600, 0 @ 4469000, 1 @ 4469400, 0 @ 4469800, 1 @ 4470200,
       0 @ 4470700, 1 @ 4471100, 0 @ 4471500, 1 @ 4471900, 0 @ 4472300, 1 @ 4472700, 0 @ 4473200, 1 @
      4473600, 0 @ 4474000, 1 @ 4474400, 0 @ 4474800, 1 @ 4475300, 0 @ 4497700, 1 @ 4498100, 0 @ 4498500,
       1 @ 4498900, 0 @ 4499300, 1 @ 4499800, 0 @ 4500200, 1 @ 4500600, 0 @ 4501000, 1 @ 4501400, 0 @
      4501800, 1 @ 4502300, 0 @ 4502700, 1 @ 4503100, 0 @ 4503500, 1 @ 4503900, 0 @ 4504400, 1 @ 4504800,
       0 @ 4505200, 1 @ 4505600, 0 @ 4506000, 1 @ 4506400, 0 @ 4868600, 1 @ 4869100, 0 @ 4869500, 1 @
      4869900, 0 @ 4870300, 1 @ 4870700, 0 @ 4871100, 1 @ 4871600, 0 @ 4872000, 1 @ 4872400, 0 @ 4872800,
       1 @ 4873200, 0 @ 4873700, 1 @ 4874100, 0 @ 4874500, 1 @ 4874900, 0 @ 4875300, 1 @ 4875800, 0 @
      4876200, 1 @ 4876600, 0 @ 4877000, 1 @ 4877400, 0 @ 5272400, 1 @ 5272900, 0 @ 5273300, 1 @ 5273700,
       0 @ 5274100, 1 @ 5274500, 0 @ 5275000, 1 @ 5275400, 0 @ 5275800, 1 @ 5276200, 0 @ 5276600, 1 @
      5277100, 0 @ 5277500, 1 @ 5277900, 0 @ 5278300, 1 @ 5278700, 0 @ 5279100, 1 @ 5279600, 0 @ 5280000,
       1 @ 5280400, 0 @ 5280800, 1 @ 5281200, 0 @ 5303700, 1 @ 5304100, 0 @ 5304500, 1 @ 5304900, 0 @
      5305300, 1 @ 5305700, 0 @ 5306200, 1 @ 5306600, 0 @ 5307000, 1 @ 5307400, 0 @ 5307800, 1 @ 5308300,
       0 @ 5308700, 1 @ 5309100, 0 @ 5309500, 1 @ 5309900, 0 @ 5310300, 1 @ 5310800, 0 @ 5311200, 1 @
      5311600, 0 @ 5312000, 1 @ 5312400, 0 @ 6452700, 1 @ 6453100, 0 @ 6453500, 1 @ 6453900, 0 @ 6454400,
       1 @ 6454800, 0 @ 6455200, 1 @ 6455600, 0 @ 6456000, 1 @ 6456500, 0 @ 6456900, 1 @ 6457300, 0 @
      6457700, 1 @ 6458100, 0 @ 6458600, 1 @ 6459000, 0 @ 6459400, 1 @ 6459800, 0 @ 6460200, 1 @ 6460600,
       0 @ 6461100, 1 @ 6461500, 0 @ 7245300, 1 @ 7245700, 0 @ 7246100, 1 @ 7246600, 0 @ 7247000, 1 @
      7247400, 0 @ 7247800, 1 @ 7248200, 0 @ 7248700, 1 @ 7249100, 0 @ 7249500, 1 @ 7249900, 0 @ 7250300,
       1 @ 7250800, 0 @ 7251200, 1 @ 7251600, 0 @ 7252000, 1 @ 7252400, 0 @ 7252800, 1 @ 7253300, 0 @
      7253700, 1 @ 7254100, 0 @ 7276500, 1 @ 7276900, 0 @ 7277300, 1 @ 7277800, 0 @ 7278200, 1 @ 7278600,
       0 @ 7279000, 1 @ 7279400, 0 @ 7279900, 1 @ 7280300, 0 @ 7280700, 1 @ 7281100, 0 @ 7281500, 1 @
      7282000, 0 @ 7282400, 1 @ 7282800, 0 @ 7283200, 1 @ 7283600, 0 @ 7284000, 1 @ 7284500, 0 @ 7284900,
       1 @ 7285300, 0 @ 8047600, 1 @ 8048100, 0 @ 8048500, 1 @ 8048900, 0 @ 8049300, 1 @ 8049700, 0 @
      8050200, 1 @ 8050600, 0 @ 8051000, 1 @ 8051400, 0 @ 8051800, 1 @ 8052300, 0 @ 8052700, 1 @ 8053100,
       0 @ 8053500, 1 @ 8053900, 0 @ 8054400, 1 @ 8054800, 0 @ 8055200, 1 @ 8055600, 0 @ 8056000, 1 @
      8056400, 0 @ 8078900, 1 @ 8079300, 0 @ 8079700, 1 @ 8080100, 0 @ 8080500, 1 @ 8080900, 0 @ 8081400,
       1 @ 8081800, 0 @ 8082200, 1 @ 8082600, 0 @ 8083000, 1 @ 8083500, 0 @ 8083900, 1 @ 8084300, 0 @
      8084700, 1 @ 8085100, 0 @ 8085600, 1 @ 8086000, 0 @ 8086400, 1 @ 8086800, 0 @ 8087200, 1 @ 8087600
force Data 1 @ 000, 0 @ 509400, 1 @ 510200, 0 @ 511100, 1 @ 511900, 0 @ 514400, 1 @ 517000, 0 @ 1701000,
      1 @ 1703500, 0 @ 1706000, 1 @ 1709400, 0 @ 2104800, 1 @ 2109000, 0 @ 2136000, 1 @ 2136900, 0 @
      2137700, 1 @ 2138500, 0 @ 2141100, 1 @ 2143600, 0 @ 3284700, 1 @ 3287200, 0 @ 3288900, 1 @ 3289700,
       0 @ 3290600, 1 @ 3293000, 0 @ 4466300, 1 @ 4470500, 0 @ 4497500, 1 @ 4500000, 0 @ 4502500, 1 @
      4505800, 0 @ 4868400, 1 @ 4871000, 0 @ 4871800, 1 @ 4873500, 0 @ 4874300, 1 @ 4876000, 0 @ 5272200,
       1 @ 5276500, 0 @ 5303400, 1 @ 5306000, 0 @ 5307600, 1 @ 5308500, 0 @ 5309300, 1 @ 5311800, 0 @
      6452500, 1 @ 6453300, 0 @ 6454200, 1 @ 6455000, 0 @ 6456700, 1 @ 6457500, 0 @ 6458400, 1 @ 6460000,
       0 @ 7245100, 1 @ 7249300, 0 @ 7276300, 1 @ 7278800, 0 @ 7279700, 1 @ 7281300, 0 @ 7282200, 1 @
      7283800, 0 @ 8047500, 1 @ 8051600, 0 @ 8078700, 1 @ 8079500, 0 @ 8080300, 1 @ 8081200, 0 @ 8082800,
       1 @ 8083700, 0 @ 8084500, 1 @ 8086200

run 20000000
```

```
#############################################################################
# wKeyTester.do
#############################################################################
add wave *
force reset_n 0 @ 0, 1 @ 400
force clk_50MHz 1 @ 0, 0 @ 1 -r 2

force clk_PS2 1 @ 000, 0 @ 307400, 1 @ 328200, 0 @ 349000, 1 @ 369700, 0 @ 390900, 1 @ 411500, 0 @
      432700, 1 @ 453400, 0 @ 474500, 1 @ 495200, 0 @ 516300, 1 @ 537000, 0 @ 558200, 1 @ 578800, 0 @
      600000, 1 @ 620700, 0 @ 641800, 1 @ 662500, 0 @ 683400, 1 @ 704100, 0 @ 725200, 1 @ 745900
force Data 1 @ 000, 0 @ 297700, 1 @ 339600, 0 @ 381400, 1 @ 423200, 0 @ 548700, 1 @ 674000

run 5000000
```

C Python Scripts

```
import bpy
import os

#-------------------------------------------------------------------
# This function takes a list of floats from an oscilloscope
# representing voltage and generates a force statement
# replicating the voltages.
# Params:
#    list - float array holding voltages
#    name - string holding name of variable to force
#    vl   - the minimum voltage allowed
#    vh   - the maximum voltage allowed
#-------------------------------------------------------------------
def gen_force(list, name, vl, vh):
    force = "force " + name + " "
    prev = -20  # Previous HIGH/LOW voltage
    i = 0
    for curr in list:
        if curr - prev >= vh - vl:    # Changes to LOW signal
            force = force + "1 @ " + "{0}".format(i) + "ps, "
            prev = vh
        elif curr - prev <= vl - vh: # Changes to HIGH signal
            force = force + "0 @ " + "{0}".format(i) + "ps, "
            prev = vl
        i = i + 1
    force = force[:-2] # Remove last ', ' using slicing
    return force

#-------------------------------------------------------------------
# This function converts a given CSV file into a string representing
# the contents of a DO file ready for simulation in ModelSim
# Params:
#    name - the filename of the CSV file to be read
#-------------------------------------------------------------------
def csv_to_do(name):
    # Place all values into these lists
    time = []
    ch1 = []
    ch2 = []

    # Find the filepath for the file to open
    target_file = os.path.join(directory, 'PS2Keyboard')
    target_file = os.path.join(target_file, name)

    # Open the file for reading and store each line separately
    f = open(target_file, "r")
    Lines = f.readlines()

    # Parse each line in CSV and store floats in proper lists
    for line in Lines[14:-1]: # Ignore first 14 and last 1 lines
        l = line.strip().split(',')
        t = float(l[0].split("e")[0]) * (10 ** float(l[0].split("e")[1]))
        time.append(t)
        ch1.append(float(l[1]))
        ch2.append(float(l[2]))

    f.close() # We are done reading

    # Generate the force statements with LOW=0.6 and HIGH=3.3
    force_ch1 = gen_force(ch1, name + "_ch1", 0.6, 3.3)
    force_ch2 = gen_force(ch2, name + "_ch2", 0.6, 3.3)

    return force_ch1 + "\n" + force_ch2 # Return file contents

#-------------------------------Function Calls----------------------#
# List holding filenames of all CSV files
files = ["a", "a_hold","d","e","f","g","r","s","t","w","water"]

# Get filepath to this Blender file and find the file to write to
blend_file_path = bpy.data.filepath
directory = os.path.dirname(blend_file_path)

# Find the file we will be writing the do file to
target_file = os.path.join(directory, 'PS2Keyboard')
target_file = os.path.join(target_file, 'force.do')

k = open(target_file, "w") # Open the combined DO file

for file in files: # Create DO file for each CSV file
    # Find the file we will be writing the do file to
    target_file = os.path.join(directory, 'PS2Keyboard')
    target_file = os.path.join(target_file, file + '.do')

    data = csv_to_do(file + ".csv") # Get file data

    f = open(target_file, "w") # Open the DO file for writing
    f.write(data)              # Write data to specific file
    k.write(data + "\n")       # Write data to combined file
    f.close()                  # Close file

k.close() # Close combined file, we are done
```

References

[1] S. Larson, "Ps/2 keyboard interface." www.digikey.com/eewiki/pages/viewpage.action?pageId=28278929, April 2020. Accessed: 20 November 2020.

[2] Wikipedia, "Piano key frequencies." en.wikipedia.org/wiki/Piano_key_frequencies, October 2020. Accessed: 20 November 2020.