Final Project

Gerri Fox

fox.ge@northeastern.edu

Submit date: 12/2/2020

Due Date: 12/03/2020

Abstract

This is the final project for EECE 2160: Embedded Design and Enabling Robotics and utilizes the DE1-SoC board and Quartus Verilog. It involves connecting the DE1-SoC board to a laptop and writing Verilog code in Quartus Verilog to create a switch debouncer that can debounce and read input from the 4x4 Keypad matrix attached to the DE1-SoC board. Based on the input from the button, the program then increments the value of the LEDs by the value of the button pressed as well as generates a tone to produce to the speaker connected to the board based on the button pressed. Lastly, the code also displays the letters of my full name plus "Fall 2020" on all 6 of the seven segment displays in a continuous rotating manner. The devices being used in this lab are the DE1-SoC board, the LEDs, the speaker and the seven segment display. It serves as practice to debouncing switches and learning about a 4x4 Keypad matrix system.

Introduction

This lab serves as an application to writing Verilog code using Quartus Verilog to design a switch debouncer. Specifically, it introduces the concept and process of switch debouncing applying it to debouncing a 4x4 Matrix Keypad. This lab practices the syntax of coding in Verilog including the concepts of modules and submodules. This lab also involves connecting the DE1-SoC board to Quartus Verilog by connecting inputs and outputs to actually be able to run and show the outcomes of the code. This lab assumes knowledge in programming of the DE1-SoC board such as using the seven segment displays and the speaker as well as with coding in Verilog at a basic level, such as case statements, modules and submodules. The project provides background knowledge and the data sheet for using the 4x4 Matrix Keypad.

Software and Hardware used

- DE1-SoC Board
- Windows laptop
- Quartus Verilog
- USB A to USB B cable
- DE1-SoC power cable
- DE1-SoC Board speaker
- DE1-SoC Board 4x4 Matrix Keypad
- 6 Seven Segment Displays
- Wires (to connect speaker)

Project Steps

1. Plug in the DE1-SoC board using the power cable and the USB A-to-USB B connection to my laptop and turn on the power
2. Ensure Quartus Schematic is installed
3. Create folder on desktop for finalproject
4. Run a new Project Wizard on Quartus using the location of the folder created above
5. Type the Name Filter as 5CSEMA5F31C6
6. Create a new Verilog file
7. Write code directly in the file
8. Ensure the code compiles
9. Go to pin planner to assign each input/output to a location on the DE1-SoC board
10. Recompile the program
11. Go to program device and select 5CSEMA5
12. Change the file to the output file created in the desktop folder from above and select program/configure

13. Press start to run your program using the DE1-SoC.

Project Discussion

Assignment 1:
The very first thing the program does when running is set the output of all ten LEDs to all zeros as well as ensure that no tones are being played on the speaker. This is programmed via the "initial begin" statement.

Assignment 2:
To debounce a 4x4 Matrix Keypad, which is connected to the DE1-SoC board via the GPIO JP1 Expansion port ports 10-24, the process was the same as debouncing one key, thus similar to the switchdebouncer module from lab 7, except, obtaining the switch value to debounce is more involved. Rather than just reading the value from the input of the switch, the 4x4 Matrix Keypad had to be read 4 times to scan each of the 4 columns separately. Taking a step back, the 4x4 Matrix Keypad is programmed to connect each of the 16 buttons via connecting a column and a row. Thus, the keypad is comprised of 4 columns and 4 rows, thus there is a unique column/row connection for each of the 16 buttons. The keypad takes in input for the columns to scan and produces output of the row values depending on the column input. Thus, the program outputs the columns and takes in the outputted row values from the keypad as inputs to interpret in the program. To scan a column, one column must be scanned at a time. Scanning happens by setting the value of the column to be scanned as 0 and the rest as 1. Setting a column to 1 effectively disables the buttons in the column as they will always return 0 for row values due to the built in high resistors in the 4x4 Matrix Keypad and assuming the row values have been inverted via the program code. So, in order to read the value of the button pushed, the program must scan each of the 4 columns separately, invert then record the outputted row values and then put them together to figure out which button has been pressed. The row value outputs a 4 bit binary string, and with inverters, outputs a high signal if a button has been pressed, thus the program interprets each of the 4 row values into a 16 bit binary string starting with the output value from scanning column 4, all the way on the left of the 16 bit binary string, and then filling in the row values for the next 3 columns. I have attached below the table of all of the 16 bit binary strings for each button. The code then interprets the number produced by this string to determine which button has been pressed. The row value output is in the order of R4-R3-R2-R1 where if one of the bits in the 4 bit string is 1, after going through an inverter, then that means that the button in that row was pressed. Then, the program knows which column was scanned to figure out the precise location of the pressed button. Now that it is determined which button was pressed, the program can debounce the button by reading the values twice, waiting 10 ms between each reading. Then, the program compares the values of the produced 16 bit binary string and if they are equal from both readings, then that means that that button was officially pressed. We must debounce buttons as there is usually some random, unstable oscillation when a button is first pressed, thus we can avoid reading this unstable value by reading twice,

with some time in between, to ensure a stable connection flow. Once the button has been established as pressed, then in order to determine when to continue reading the value of the next button pushed, first the program must determine that the button has been released. This is accomplished by reading the value of the button another time after waiting 10ms and determining if it is equal to 0, meaning it is not pushed. If it is, then the program will continue waiting for another button to be pushed, if not, it will continue waiting 10ms and reading the same button value until it is 0. This process is then repeated throughout the entirety of the program and the entire process utilizes and is triggered by the positive edge of the clock.

Button	C4 C3 C2 C1
1	0000 0000 0000 0001
2	0000 0000 0001 0000
3	0000 0001 0000 0000
4	0000 0000 0000 0010
5	0000 0000 0010 0000
6	0000 0010 0000 0000
7	0000 0000 0000 0100
8	0000 0000 0100 0000
9	0000 0100 0000 0000
0	0000 0000 1000 0000
A	0001 0000 0000 0000
B	0010 0000 0000 0000
C	0100 0000 0000 0000
D	1000 0000 0000 0000
*	0000 0000 0000 1000
#	0000 1000 0000 0000

0000 - R4 R3 R2 R1

0001 - means button is in row 1
0010 - means button is in row 2

0100 - means button is in row 3
1000 - means button is in row 4

Assignment 3:
To add the value of the button pushed to the LEDs, this is accomplished by the program knowing the correspondence between the the 16 bit binary string used to read which button is pressed and what that button's value equates to, with the * button being 14 and the # button being 15. Thus, in the debouncer submodule, once it has been established that a certain button has been pressed, the program then has the current value of the LEDs stored, and adds the desired value to the current value and feeds that output via the pin assignments to the LEDs on the DE1-SoC board.

Assignment 4:
The ensuring that the value of the button pushed is only added to the LEDs once is accomplished via the last half of what is explained above in assignment 2. The switch debouncer, after it is established that a button has been pressed, continues reading that specific button to ensure that the 16 bit binary string produced now is all zeros meaning the button is no longer pushed. Thus, the program will not continue to read the value of buttons and go into the code that adds values to the LEDs until it is known that the original button has been released, otherwise the code would loop over and read like the original button has been pressed again, even though it was never released.

Assignment 5:
To generate tones only when pressing a button, this is accomplished similar to how adding to the LEDs only once when a button is pushed is done. This feature is added to the switchdebouncer submodule explained in previous assignments as the code knows which tone frequency is connected with each of the 16 bit binary string outputs, thus when one of them is established as pressed, the submodule outputs the subsequent countlow and counthigh values that are then fed into the toneprocessor submodule from a previous lab to play the tone on the speaker, which is connected to the GPIO JP1 Expansion port port 1. A brief overview of how a tone is generated is that the countlow value represents the count for when the signal to the speaker should be low and the counthigh value is the same but with outputting a high signal. These values are determined based on the frequency of the desired tone and divided by the frequency of the clock being used, which in this case is 50 MHz. The speaker works as a 50% duty cycle speaker in order to achieve sound. Then, as explained above when checking to ensure a button has been released before moving on, in order to have the tones only be generated when a button is pressed, once it has been established that a button has been released, the program outputs countlow and counthigh values that equate to no tones being generated, thus the tonegenerator submodule does not generate any tones and the process repeats. Since there are 16 buttons, the tones generated are from C5 to C7 with the 0 button equating to no tones.

Assignment 6:
This module rotates through my name added with “fall 2020” in a continuous manner, thus printing “gerri fox fall 2020” continuously over all 6 of the seven segment displays. This module works by initially setting each of the seven segment displays to all 1s to effectively print nothing. Then, it has a counter that counts one second at a time and at each second, it increments the rotation by 1. Thus, the first letter “g” is printed on the rightmost sevensegment display. Then, after another second, the seven segment displays are rotated so the g moves to the second most right display and an “e” is printed on the rightmost. This continues until the entire phrase is displayed, then it starts over, thus the only time a seven segment display is blank is at the very start. This is achieved by setting the outputs to the seven segment displays as local variables that update each one second interval. And they are updated by being set to the value of the preceding seven segment display, thus effectively moving each letter forward. The looping value is achieved as values are continuously set to the next one every one second and the first value always goes back to the end. Only 6 are displayed at a time as only 6 of the variables are linked to the output pins for the seven segment displays, although the rotating is one big loop. To display the letters, the output given to the seven segment displays is backwards and 1 denotes that bit being off and 0 denotes the bit being on (visible). Note that the letter x is impossible to display using the 7 bits of the display thus the letter “H” is printed instead as it is the closest resemblance to an “X” that can be made using the 7 bits of the seven segment display. The following chart shows the values outputted to the seven segment display to display each letter:

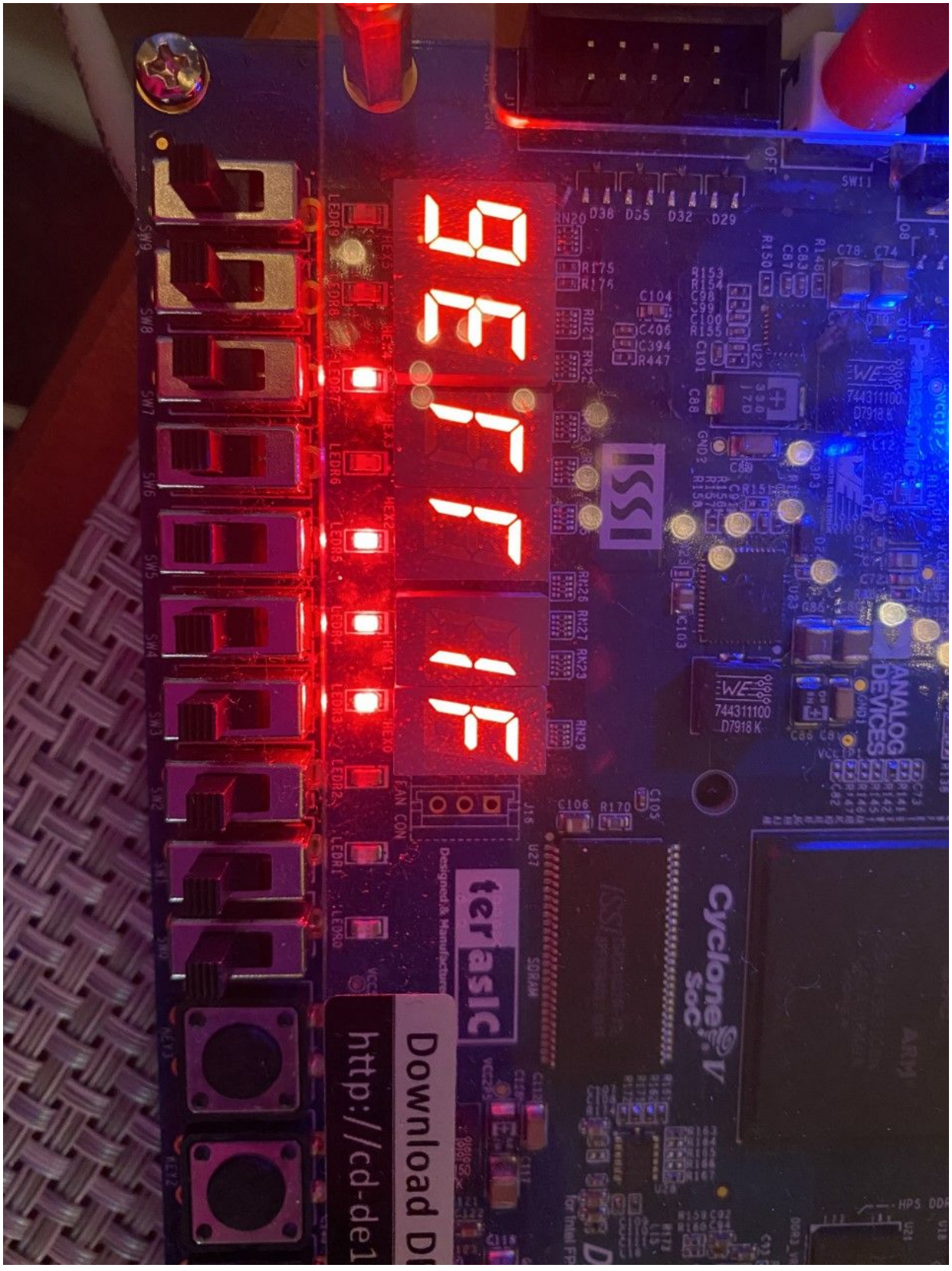
Letter	Seven Segment Display Inputs(a-g)
g	0010000
E	0000110
r	1001110
r	1001110
I	1111001
F	0001110
O	1000000
H	0001001
F	0001110
A	0001000
L	1000111
L	1000111

2	0100100
0	1000000
2	0100100
0	1000000

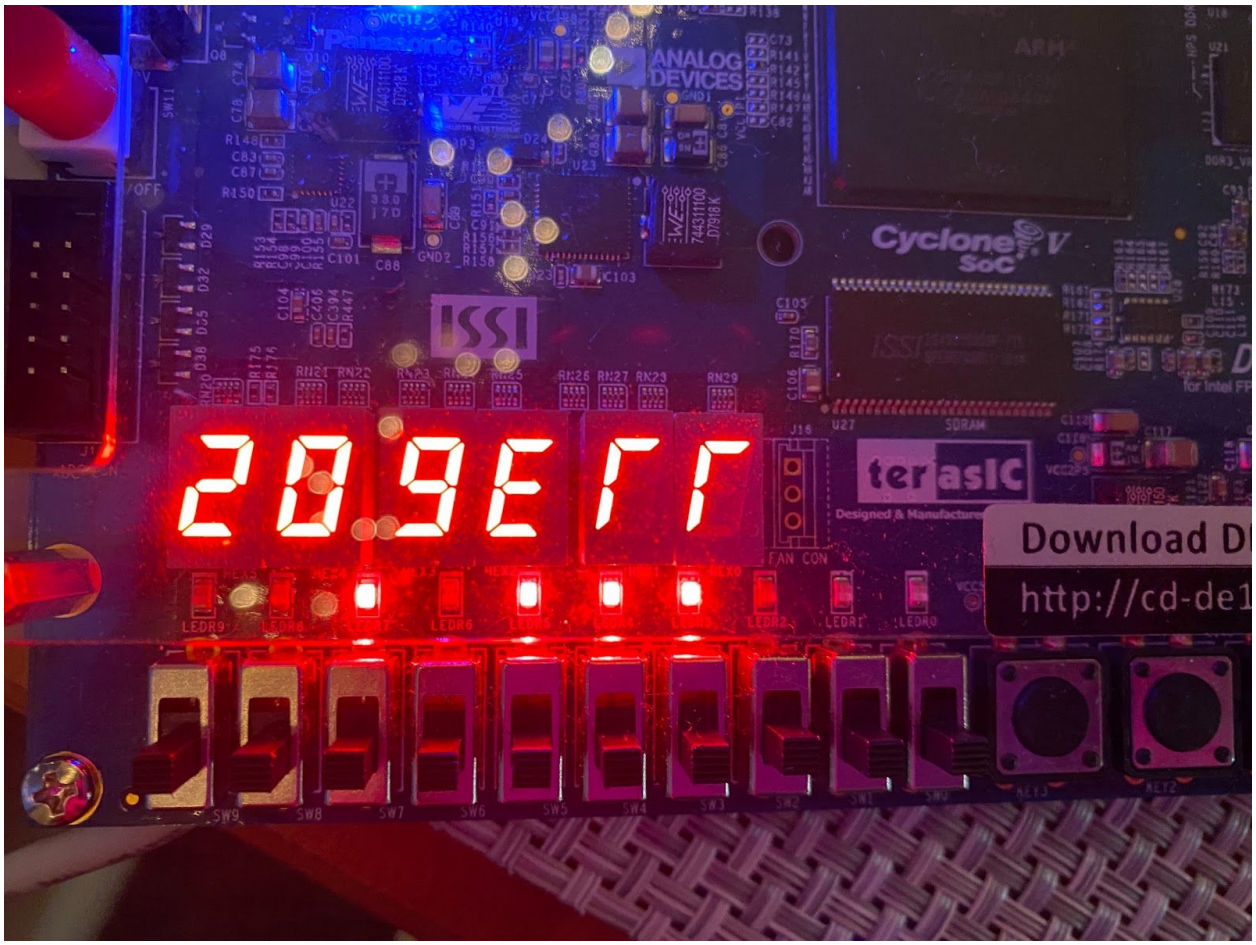
*lowercase g
*uppercase e
*two lowercase r
*i uses the rightmost bits (b & c)
*1 represents on and 0 represents off and the bits go in order starting from a on the left to g on the right
*as coded exactly to seven segment display outputs

Assignment 7:
All of the submodules described above are connected via the top module called finalproject. This module has the inputs of the 50 MHz clock and the rows value read from the 4x4 Matrix Keypad, and has outputs for the columns value of the 4x4 Matrix Keypad, the LEDs, all 6 Seven Segment displays and the signal to the speaker. The topmodule first calls the switchdebouncer submodule to read the value of the button pressed, then it sends output to the LEDs, as well as outputs the countlow and counthigh values for the tone which are then fed into the tonegenerator submodule to actually send the output signal to the speaker. Then, the rotatingdisplay submodule is also called here to start the continuous rotation and this is where the seven segment display gets its output.

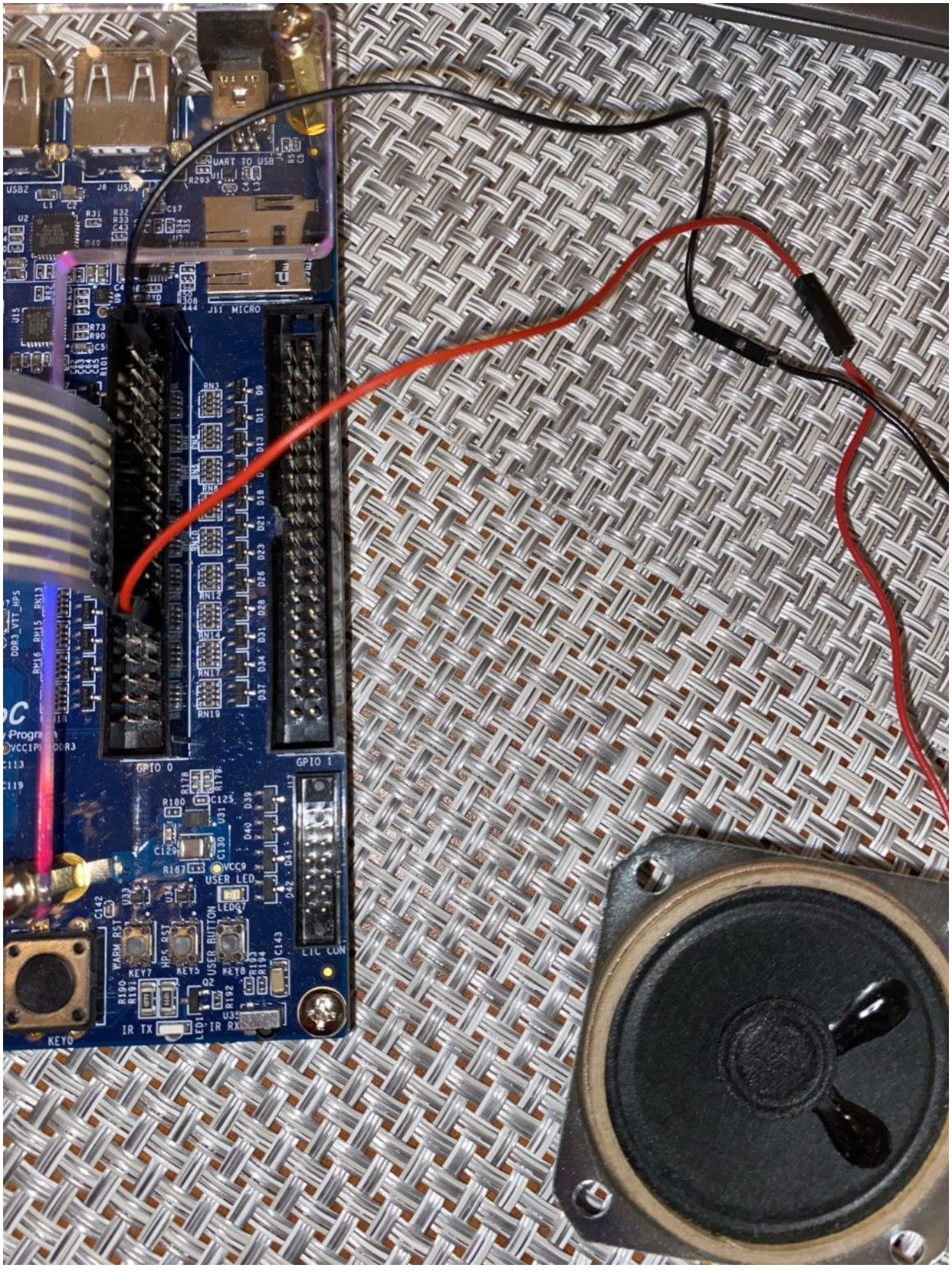
Results


Cyclone V
SoC
terasic
Download D
http://cd-de1
ANALOG DEVICES
ISSI

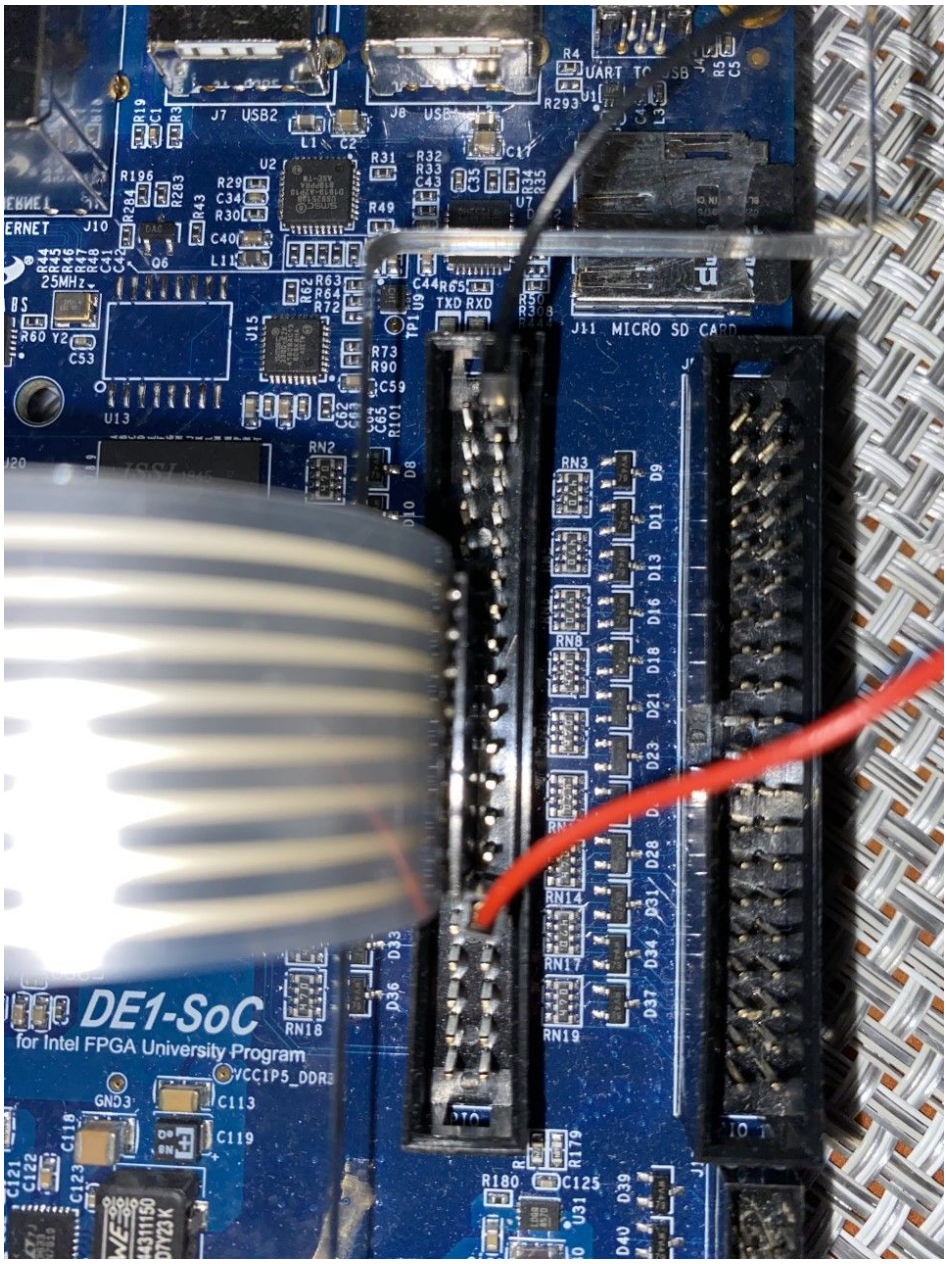
This photo shows the seven segment display starting. It also shows that some buttons have been pressed as the LEDs read a value greater than 0.



This photo shows the seven segment display rotating continuously as when it reaches the end it circles back to the beginning of the phrase and starts over.


UART TO USB
USB2
J11 MICRO
GPIO 0
GPIO 1
USER LED
USER BUTTON
WARM RST
HPS RST
IR TX
IR RX
I2C CON
KEY0

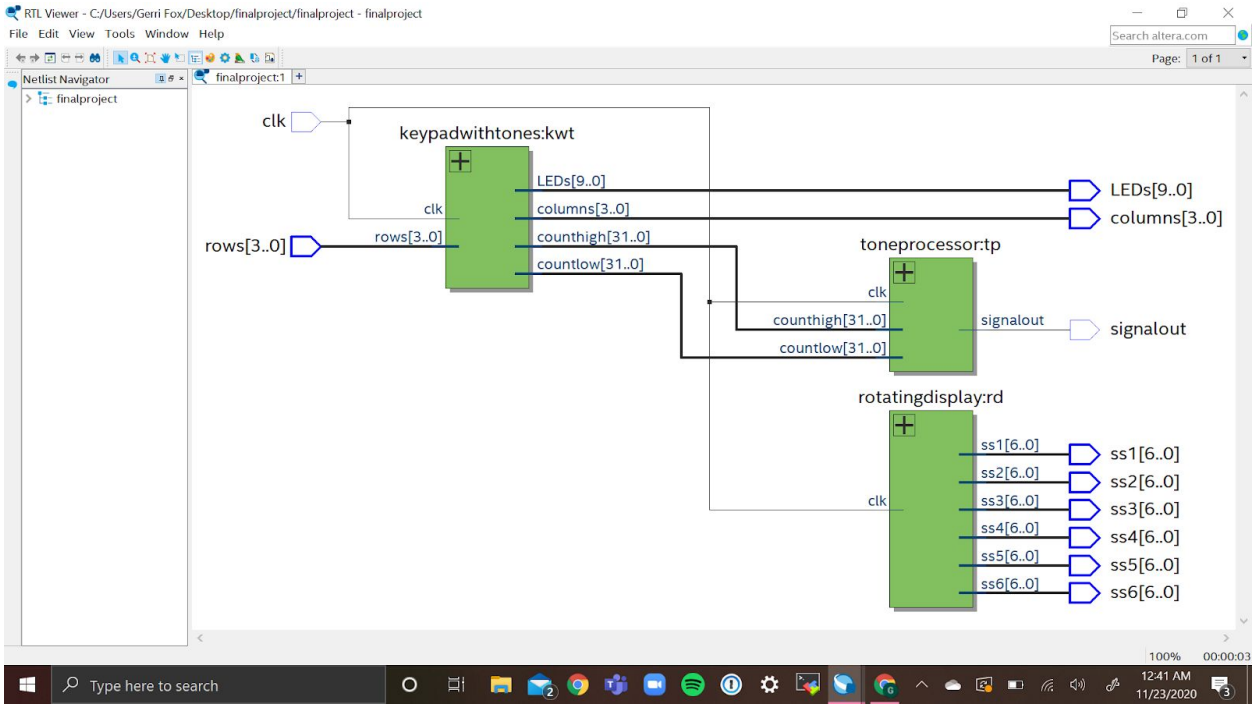
This photo shows the connection of the speaker to the DE1-SoC board via the GPIO JP1 Expansion port port 1 and the ground port.



This photo shows the speaker connection up close but also the connection of the 4x4 Matrix Keypad. The keypad is also connected via the GPIO JP1 expansion port and uses ports 10 through 24. These are the ports that are programmed to the pin planner in Quartus Verilog to read inputs and send outputs.

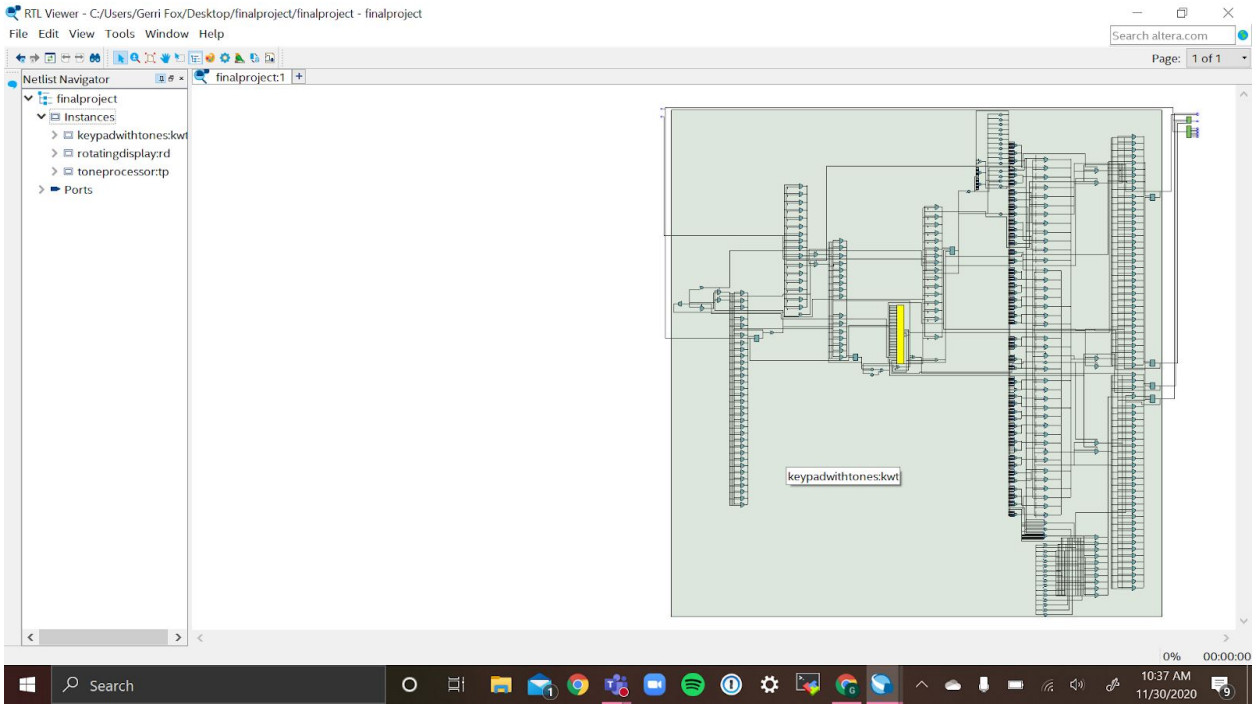
Analysis

Some of the benefits of programming this assignment in Verilog rather than using an object oriented language like C++ is that hardware programming gives the program more control to execute processes simultaneously. In this assignment specifically, we want the functionality of reading from the 4x4 Matrix Keypad as well as updating the LEDs at the same time as generating a tone, all while the seven segment display is rotating through a phrase continuously. Another benefit of programming this assignment in Verilog is that is allows for more flexibility of keeping time which is necessary in this project for determining when to read the values of the buttons for debouncing, for how long to play each tone, when to rotate the display, and any place a counter is needed, etc. Lastly, another benefit is the fact that we can design using circuits and logic such that we can use inverters and other logic to determine the behavior of the program inputs and outputs.

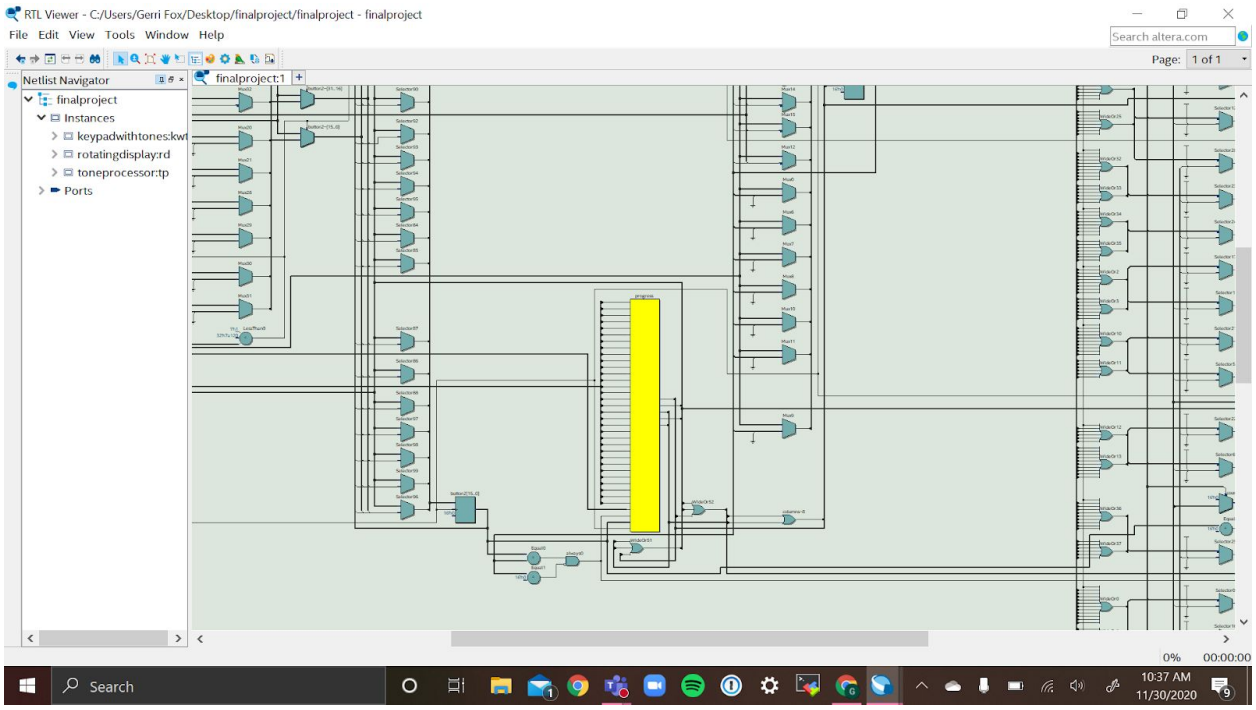


This logic is demonstrated as shown by the generated RTL Quartus Schematic as shown above. This schematic shows the inputs of the clock and the rows from the 4x4 Matrix Keypad. Both of these are then inputs to the keypadwithtones submodule which then outputs directly to the LEDs, as well as directly the column to scan to the 4x4 Matrix Keypad, then it also outputs countlow and counthigh to the toneprocessor submodule which then gives the proper output to the speaker. Lastly, the clock also serves as the only input to the rotating display submodule which then based on the time and local variables, it sets the sevensegment displays directly. This schematic clearly shows the inputs and outputs to the various submodules as well as the wires between submodules. As this clearly shows, the program has 2 inputs, the 50 Mega hertz clock and the row value from the 4x4 matrix keypad. The row value is then put through an inverter circuit in order to more easily interpret the data into a 16 bit binary string that represents the data of each button on the 4x4 matrix keypad. In order to produce this 16 bit binary string, a left shift is used to build the string along with a state machine to go through each state, in this case one example is

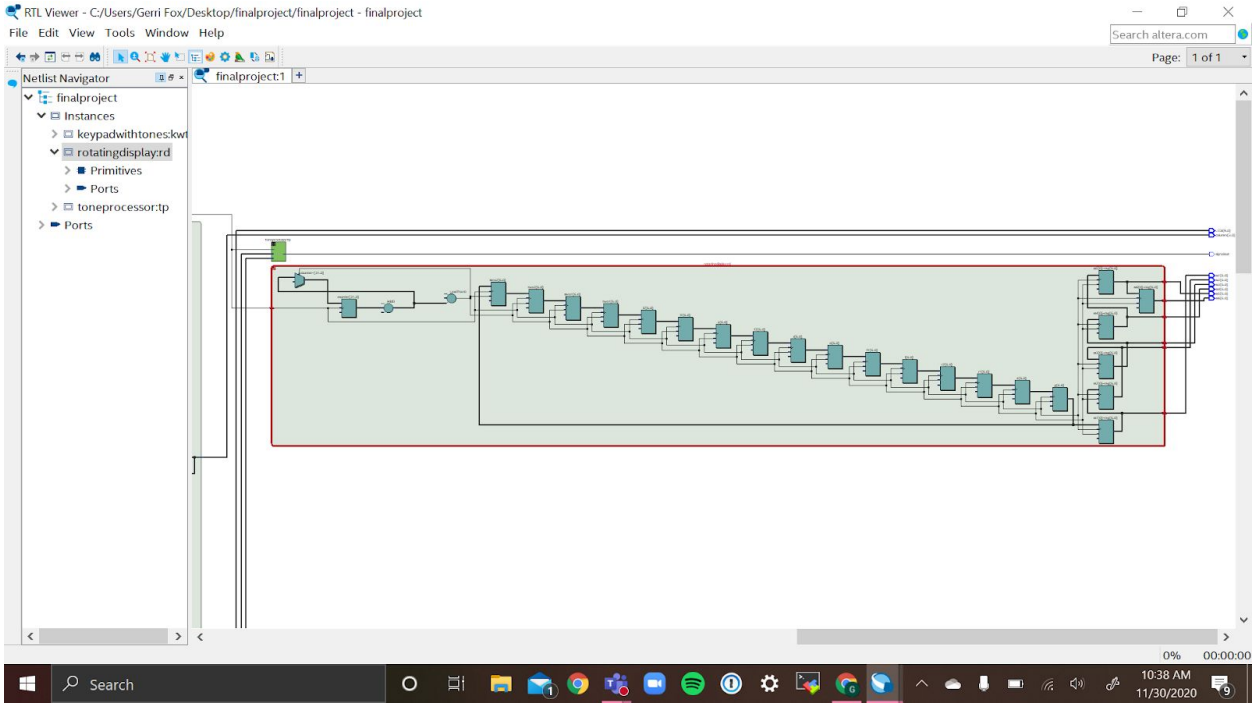
the states representing the switching of the columns being scanned. Then, based on the generated 16 bit binary string the LEDs value are sent to the output and and countlow and counthigh values are generated to be sent as inputs to the tone processor module. And finally, the rotatingdisplay submodule just takes in the clock and outputs to the 6 seven segment displays. A left shift is constructed using a series of multiplexers whose inputs are the bits to shift and the outputs are the shifted bits and takes in a selector which tells it which way to shift, 0 means shift right and 1 means shift left. In this case, we are doing a left shift thus the selector is 1. A state machine is constructed using internal D-Flip Flops. Going into more detail, a multiplexer works by having n selectors for 2^n inputs, as with n selectors there are 2^n possibilities that can be made, and a multiplexer works by the output being the input that matches the selectors. Thus, if we had a 4x1 MUX for example, there would be 2 selectors and 4 inputs, say 00, 01, 10, 11. Say the selectors were 01, then the output would be the data at 01. A D-Flip Flop works by being closely related to the positive edge of the clock (could also be the negative edge but this project utilizes the positive edge). The output is only affected when the clock signal transitions to positive and holds the input but it is only utilized on the positive edge of the clock. Thus, for example, if the clock switches to a positive transition, and the input is 1 at that time, then the output is 1, until the next positive transition where it evaluates the input, if the input is still 1, then the output remains 1, but if the input is now 0, then the output is 0. Some more smaller explanations of hardware used is for comparators, those are implemented through XNOR gates that take in the input(s), and then if there are more than 1 bit being compared, each corresponding bit is compared via a NAND gate, and then each of those outputs are then put into an AND gate. The project code also uses addition which is achieved via a full adder where each bit is put through a full adder with Cin and its corresponding bit where Cout and Sum represent the outputs and Cout is carried through each bit as the Cin value.



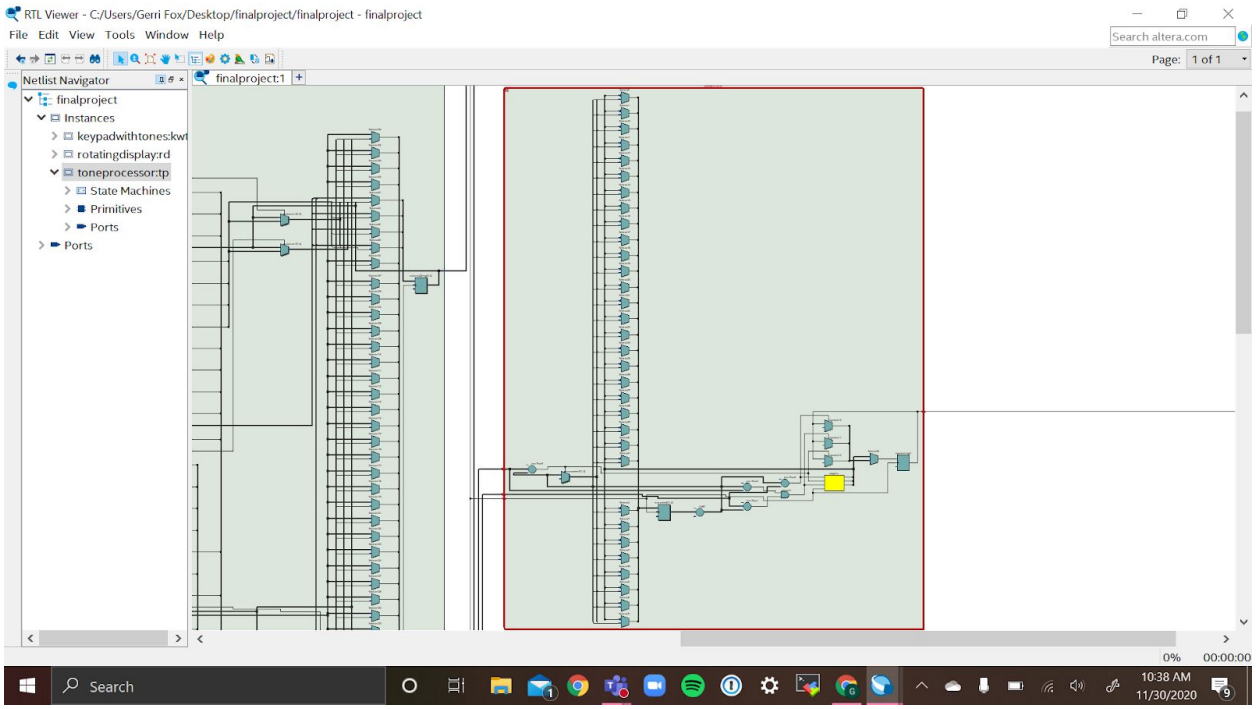
This is going within the keypadwithtones submodule and as I explained earlier, this verifies that it is a combination of 2-bit multiplexers as well as some basic AND and OR gates, inverters, comparators and a few D-Flip Flops as well as state machines, all explained above. The state machine directly links to my code that goes through each button pressed possibility in the switch debouncer. There is an army of multiplexers that is what determines what the column output is as well as even more 2x1 multiplexers to craft the 16 bit binary strings constructed by reading the data from the 4x4 matrix keypad. After the state machine, there are 2 more armies of multiplexers also with a D-Flip Flop that directly correspond to determining the countlow and counthigh values



This is a more zoomed in view of the state machine, and also seen are comparators, a D-Flip Flop, OR gates and multiplexers from the keypadwithtones submodule.



This is the rotatingdisplay submodule which shows a series of D Flip-Flops as well as some addition and comparators, all of which are explained how they function logically above. The D-Flip Flops directly correspond to the rotating through each letter/number to display and determine when and which to set.



This is the RTL schematic for the toneprocessor submodule which as shown is a series of 2x1 multiplexers with some comparators, a couple D-Flip Flops, a basic AND gate and a state

machine. Here, the state machine corresponds to which region we are in, based on the provided countlow and counthigh values which are shown by the multiplexers and a D-Flip Flop and this condition checking is shown by the comparators. And finally, the last D-Flip Flop corresponds to my code that determines what the signalout value (either 0 or 1) should be.

Overall all of the results from this lab are correct, as expected and make sense. This lab provided a great example of understanding how to incorporate the DE1-SoC board's 50 MHz clock and was an example of using the positive edge, when the input goes from low to high of the clock, to determine the outputs as well as using submodules as almost function-like within the top module as well as great practice for switch debouncing and utilizing the 4x4 Matrix Keypad, the speaker, the LEDs and the seven segment displays. Possible sources of error exists in the code from not accounting for the fact that the two button values read can be equal but equal to 0 meaning it has not been pressed, thus nothing should happen. Another source of error is not inverting the outputs for the seven segment display, otherwise the display will be exactly opposite as expected. This inversion error could also happen if the row values outputted are not inverted, although the program could be programmed to match the non-inverted values, the inverted values are more intuitive. Another source of error is my program originally was not allowing the outputs of the columns to be sent and was attempting to read them directly after setting in the same step, which does not account for that this is hardware rather than software. Another source of error was when creating the 16 bit binary string output and shifting over the row values, the number of shifts was getting confused with the addition of the next inputs thus the code now breaks it into two clear steps. Lastly, another source of error is not accounting for when the button has been released or if the user is holding it down. This would allow for the program to never stop increasing the LEDs/seven segment display as well as allow for it to increase more than once if the user has kept the button pressed down. Another difficulty I faced at the very end was after traveling home with the DE1-SoC board, the speaker all of sudden only played tones very quietly. It worked perfectly at school, but after travel, without changing the code at all, the speaker's sound output became very quiet. I attempted to reconnect the speaker but it still did not change. This code can be applied to future, useful projects when needed to show a rotating display of some sort, maybe when programming a billboard. The speaker and LEDs are always useful to be familiar with as these attributes are used in almost every modern technology. Further, the idea of switch debouncing and using a 4x4 Matrix Keypad is very useful when working with any technology with any buttons.

Conclusion

The results from this lab prove to be an introduction to Quartus Verilog as well as linking the seven segment display, the LEDs, the speaker and the 4x4 Matrix Keypad by writing code for each of these topics using Quartus Verilog. In the future, having knowledge with embedded programming and design, such as Quartus Verilog is very important in creating well designed and interactive programs and advanced technology that incorporates these aspects to achieve a bigger product.

References

[1] Prof. Julius Marpaung, “Lab Report Guide”, Northeastern University, January 6 2020.

[2] Terasic, “DE1-SoC User Manual”, Terasic.com, January 28, 2019.

[3] Parallax, “4x4 Matrix Keypad”, Parallax.com, December 16, 2011.

Appendix

```
module finalproject(clk, rows, LEDs, columns, signalout, ss1, ss2, ss3, ss4, ss5, ss6);

input [3:0] rows; //rows of keypad
input clk; //50 MHz clock
output [9:0] LEDs; //10 LEDs
output [3:0] columns; //columns of keypad
output signalout; //signalout to speaker
output [6:0] ss1, ss2, ss3, ss4, ss5, ss6; //6 seven segment displays
wire [31:0] countlow, counthigh; //countlow and counthigh for sound frequencies

keypadwithtones kwt(.rows(rows), .clk(clk), .LEDs(LEDs), .columns(columns), .countlow(countlow), .counthigh(counthigh));
rotatingdisplay rd(.clk(clk), .ss1(ss1), .ss2(ss2), .ss3(ss3), .ss4(ss4), .ss5(ss5), .ss6(ss6));
toneprocessor tp(.clk(clk), .countlow(countlow), .counthigh(counthigh), .signalout(signalout));

endmodule

module keypadwithtones(rows, clk, LEDs, columns, countlow, counthigh);

input [3:0] rows;
input clk;
output [9:0] LEDs;
reg [9:0] LEDs;
output [3:0] columns;
reg [3:0] columns;
reg[2:0] progress, colprogress;  //progress for overall program and what column is being scanned
reg [3:0] val1, val2; //represents the inverted value of rows
reg [15:0] button1, button2; //represents the 16 bit string of the keypad output
reg [31:0] counter; //counter for switch debouncing
```

```
output [31:0] countlow, counthigh;
reg [31:0] countlow, counthigh;

initial begin
LEDs = 10'b00_0000_0000;  //bits to output to leds
progress = 3'b000; //progress of where in program
colprogress = 3'b000; //progress for column scanner
counter = 0; //counter for time
columns = 4'b1110; //initially start by scanning column 4
end

always @(posedge clk) begin //at positive edge of the clock
case(progress)

3'b000: begin //read value of button first time

case(columns) //scan columns
4'b1110: begin
val1 = ~rows; //scan column 4
button1 = val1;
columns = 4'b1101; //set to scan column 3
end
4'b1101: begin
val1 = ~rows; //scan column 3
button1 = button1 << 4; //shift over column 4 values by 4
button1 = button1 + val1; //add column 3 value
columns = 4'b1011; //set to scan column 2
end
4'b1011: begin
val1 = ~rows; //scan column 2
button1 = button1 << 4; //shift over column 4 and 3 values by 4
button1 = button1 + val1; //add column 2
columns = 4'b0111; //set to scan column 1
end
4'b0111: begin
val1 = ~rows; //scan column 1
button1 = button1 << 4; //shift over column 4, 3 and 2 values by 4
button1 = button1 + val1; //add column 1
columns = 4'b1110; //reset columns back to scan column 4 for next time
progress = 3'b001; //move to next case
```

```
end
endcase
end

3'b001: begin  //wait 10 ms then scan button for second time
counter = counter + 1'b1;
if (counter >= 500000) begin  //wait 10 ms
case(columns) //start to scan button for a second time
4'b1110: begin
val2 = ~rows;
button2 = val2;
columns = 4'b1101;
end
4'b1101: begin
val2 = ~rows;
button2 = button2 << 4;
button2 = button2 + val2;
columns = 4'b1011;
end
4'b1011: begin
val2 = ~rows;
button2 = button2 << 4;
button2 = button2 + val2;
columns = 4'b0111;
end
4'b0111: begin
val2 = ~rows;
button2 = button2 << 4;
button2 = button2 + val2;
columns = 4'b1110;
counter = 0;
progress = 3'b010;
end
endcase
end
end

3'b010: begin  //see if button has been pushed
if ((button1 == button2) && (button1 != 16'h0000)) begin //if button has been pushed
```

```
case(button1) //figure out which button has been pushed then update LEDs and countlow and counthigh accordingly
        16'h80: begin //0
        //do nothing to LED
        //no sound
        countlow = 1136;
  counthigh = 2272;
        end

        16'h1: begin //1
        LEDs = LEDs + 1'b1;
        //C5, 523.25 Hz roughly
        countlow = 47755;
        counthigh = 95510;
        end

        16'h10: begin //2
        LEDs = LEDs + 2'b10;
        //D5, 587.33 Hz roughly
                countlow = 42565;
                counthigh = 85131;
        end

        16'h100: begin //3
        LEDs = LEDs + 2'b11;
        //E5, 659.26 Hz roughly
                countlow = 37921;
                counthigh = 75842;
        end

        16'h2: begin //4
        LEDs = LEDs + 3'b100;
        //F5, 698.46 Hz roughly
        countlow = 35803;
  counthigh = 71606;
        end

        16'h20: begin //5
        LEDs = LEDs + 3'b101;
        //G5, 783.99 Hz roughly
```

```
            countlow = 31888;
            counthigh = 63776;
    end

    16'h200: begin //6
    LEDs = LEDs + 3'b110;
    //A5, 880.00 Hz roughly
            countlow = 28409;
            counthigh = 56818;
    end

    16'h4: begin //7
    LEDs = LEDs + 3'b111;
    //B5, 987.77 Hz roughly
            countlow = 25390;
            counthigh = 50619;
    end

    16'h40: begin //8
    LEDs = LEDs + 4'b1000;
    //C6, 1046.5 Hz roughly
            countlow = 23889;
            counthigh = 47778;
    end

    16'h400: begin //9
    LEDs = LEDs + 4'b1001;
    //D6, 1174.7 Hz roughly
            countlow = 21282;
            counthigh = 42564;
    end

    16'h1000: begin //10(A)
    LEDs = LEDs + 4'b1010;
    //E6, 1318.5 Hz roughly
            countlow = 18960;
            counthigh = 37921;
    end

    16'h2000: begin //11(B)
```

```
        LEDs = LEDs + 4'b1011;
        //F6, 1396.9 Hz roughly
                countlow = 17896;
                counthigh = 35793;
        end

        16'h4000: begin //12(C)
        LEDs = LEDs + 4'b1100;
        //G6, 1568.0 Hz roughly
                countlow = 15943;
                counthigh = 31887;
        end

        16'h8000: begin //13(D)
        LEDs = LEDs + 4'b1101;
        //A6, 1760 Hz roughly
                countlow = 14204;
                counthigh = 28409;
        end

        16'h8: begin //14(*)
        LEDs = LEDs + 4'b1110;
        //B6, 1976 Hz rougly
                countlow = 12651;
                counthigh = 25303;
        end

        16'h800: begin //15(#)
        LEDs = LEDs + 4'b1111;
        //C7, 2093 Hz rougly
                countlow = 11944;
                counthigh = 23889;
        end

        endcase
progress = 3'b011; //if button has been pushed, go to next case to wait until it is released
end else progress = 3'b000; //if button has not been pushed, go back to beginning
end

3'b011: begin
```

```
counter = counter + 1'b1;
if (counter >= 5000000) begin //wait another 10 ms before reading again

case(columns) //read button again
4'b1110: begin
val2 = ~rows;
button2 = val2;
columns = 4'b1101;
end
4'b1101: begin
val2 = ~rows;
button2 = button2 << 4;
button2 = button2 + val2;
columns = 4'b1011;
end
4'b1011: begin
val2 = ~rows;
button2 = button2 << 4;
button2 = button2 + val2;
columns = 4'b0111;
end
4'b0111: begin
val2 = ~rows;
button2 = button2 << 4;
button2 = button2 + val2;
columns = 4'b1110;
counter = 0;
progress = 3'b100;
end
endcase
end
end

3'b100: begin
if (button2 == 16'h0000) begin //if button now is 0 (not pushed)
progress = 3'b000; //go back to beginning
countlow = 1136; //stop playing a tone
counthigh = 2272;
end else progress = 3'b011; //if button has not been released, go back to previous step to reread value
```

```
end

endcase
end


endmodule

module toneprocessor(clk, countlow, counthigh, signalout);

output signalout;
reg signalout;
input clk;
input [31:0] countlow, counthigh;
reg [31:0] onesecond, mycounter;
reg [3:0] progress;

initial begin
signalout=0; //initialize signalout to 0
onesecond = 100000000; //represents 1 second
mycounter = 0; //counter for which region
progress = 4'b0000; //progress of which region we are in
end

always@(posedge clk) begin
case(progress)

default: begin
                        end

4'b0000: begin
// Region 1
        mycounter = mycounter + 1'b1; //increase counter for region by 1
        if (mycounter < countlow) begin //if counter is less than countlow
        signalout = 0; //set signal to 0
        end
        else begin
        progress = 4'b0001; //set progress to region 2
        end
end
```

```
4'b0001: begin
//Region 2
        mycounter = mycounter + 1'b1; //increase counter for region by 1
        //if counter is equal to or greater than countlow and less than counthigh
        if ((mycounter >= countlow) && (mycounter < counthigh)) begin
        signalout = 1; //set signal to 1
        end
        else begin
        progress = 4'b0010; //set progress to region 3
        end
end

4'b0010: begin
//Region 3
        mycounter = mycounter + 1'b1; //increase counter for region by 1
        if (mycounter >= counthigh) begin //if counter is greater than or equal to counthigh
        signalout = 0; //set signal to 0
        mycounter = 0; //restart counter
        end
        else begin
        progress = 4'b0000; //set progress to region 1
        end

end

endcase
end
endmodule

module rotatingdisplay(clk, ss1, ss2, ss3, ss4, ss5, ss6);

input clk; //50 MHz clock
output reg [6:0] ss1, ss2, ss3, ss4, ss5, ss6; //6 seven segment displays
reg [31:0] counter; //counter for time
reg [6:0] g,e,r1,r2,i,f1,o,x,f2,a,l1,l2,two1,zero1,two2,zero2,start; //represent bit values for each letter

initial begin
//initially set all seven segment displays to blank
```

```
ss1 = 7'hFF;
ss2 = 7'hFF;
ss3 = 7'hFF;
ss4 = 7'hFF;
ss5 = 7'hFF;
ss6 = 7'hFF;
//represents the bit strings for all needed letters
g = 7'b0010000;
e = 7'b0000110;
r1 = 7'b1001110;
r2 = 7'b1001110;
i = 7'b1111001;
f1 = 7'b0001110;
o = 7'b1000000;
x = 7'b0001001;
f2 = 7'b0001110;
a = 7'b0001000;
l1 = 7'b1000111;
l2 = 7'b1000111;
two1 = 7'b0100100;
zero1 = 7'b1000000;
two2 = 7'b0100100;
zero2 = 7'b1000000;
end

always @(posedge clk) begin //at positive edge of clock
counter = counter + 1'b1;
if (counter >= 50000000) begin //wait 1 second
counter = 0; //reset counter

//move each previous seven segment display bit string up by one spot
//initially all empty
ss6 = ss5;
ss5 = ss4;
ss4 = ss3;
ss3 = ss2;
ss2 = ss1;
ss1 = g; //sets first seven segment display

//resets each letter in a circular pattern
```

```
start = g;
g = e;
e = r1;
r1 = r2;
r2 = i;
i = f1;
f1 = o;
o = x;
x = f2;
f2 = a;
a = l1;
l1 = l2;
l2 = two1;
two1 = zero1;
zero1 = two2;
two2 = zero2;
zero2 = start;

end
end

endmodule
```