Braille Calculator

E155 Final Project

December 10, 2021

Yoo-Jin Hwang and Udeema Shakya

Abstract

For our final project, we wanted to create a project that could potentially be beneficial to visually impaired individuals. Our goal for our project was to display the answer of a mathematical operation with braille characters. The mathematical expression consists of two single digit integers with an arithmetic operation. The system uses an I2S MAX98357 Amplifier, the STM32 Nucleo-64 microcontroller, and the FPGA MAX1000. The microcontroller solves the mathematical expression and then communicates with the FPGA to actuate the servos to the correct position for the overall result. The microcontroller also communicates with the audio amplifier using I2S to output the answer to the mathematical expression to the speakers.

Table of Contents

I. Introduction: Motivation, Block Diagram, Overview

The motivation of this project was to merge embedded systems with a social justice issue that we were both passionate about. We are both involved in the Living Learning Community at Harvey Mudd College and wanted to find a way to create a good impact for our project. After some research, we found that there was a gap in producing calculators that visually impaired individuals could use. We saw a couple conceptual designs such as the one below for Logitech in Figure 1.

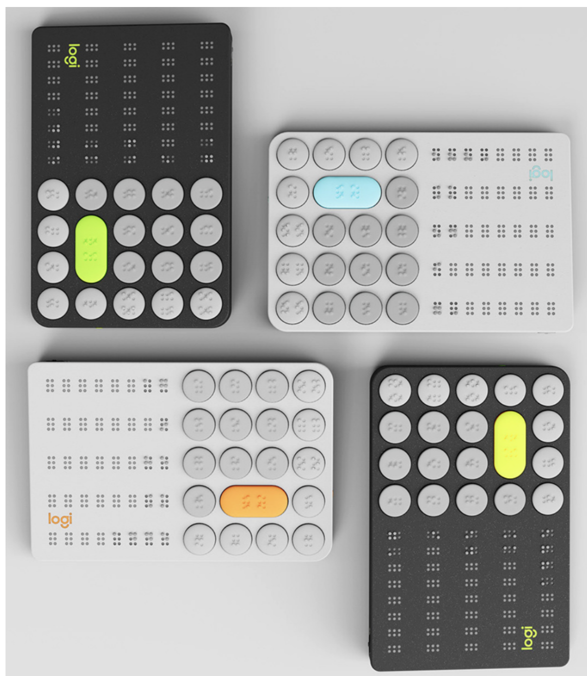

Figure 1: Conceptual Drawing of Braille Calculators

But there were no calculators that outputted the result on a braille format unless it is on a tactile graphics display which costs over a thousand dollars. From our market research, we settled on a simple, low-cost design to incorporate a technical challenge, building a calculator using inputs and a new serial communication line, and a social justice gap, an accessible calculator for visually impaired individuals. We aimed to incorporate our software, hardware knowledge to create a project that could help visually impaired individuals to use a calculator.

The overall structure of the result is that the user will enter the mathematical expression and then feel and listen to each character of the output. For example, if we have a mathematical expression “1+1=” then the speaker will output a high pitch sound for the number being positive, a long low tone for the zero value in the tens place, and then the two pulses in the ones place.

The project includes the use of the STM32 Microcontroller and the MAX1000 FPGA. The microcontroller acts as the SPI primary which communicates the mathematical expression output to the SPI secondary which is our FPGA. Then, we wanted to output the mathematical answer to a speaker using I2S. The MCU will then act as a I2S primary which then the decoded signal goes to the MAX98357A I2S Amplifier to the speakers. Connected to the MCU consists of an user button that tells the MCU that the user is ready for reading and hearing the next character in the output. The main functions that the MCU provides is the mathematical decoder with SPI, I2S communication with the amplifier, and the user switch that the user switches on and off when they are ready to feel the next value.

Next, for the FPGA, the value that the user inputs into the keypad for the mathematical expression is then transferred to the MCU through SPI. On the FPGA there is also a switch that will stop the servos from moving. Note that the FPGA handles the debouncing of the keys. The FPGA will then actuate the microservos to the output of the mathematical expression once the user hits the button to say that they are ready to read.

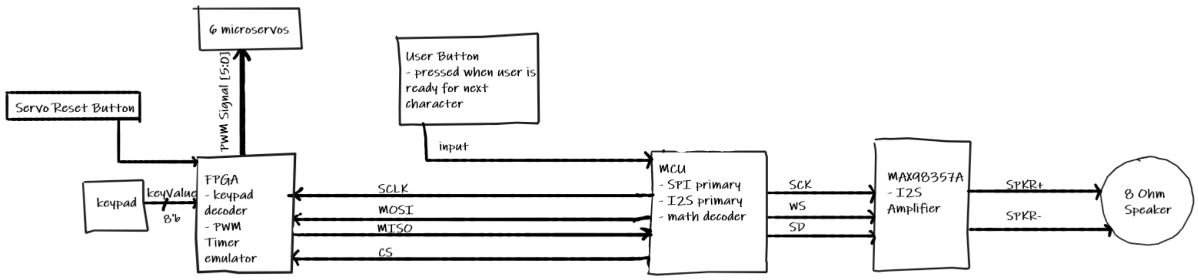

Figure 2: Overall Block Diagram of the Braille Calculator System

II. New Hardware

We used position micro servos and an I2S Audio Amplifier breakout board which were both two new pieces of hardware.

Micro Servos

Servos come in three different flavors: positional rotation, continuous rotation, and linear. Positional rotation servos typically can only rotate up to 180 degrees. These servos can be set to a certain degree by changing the PWM wave. Continuous rotation servos can rotate the full 360 degrees. Rather than the position being set on these servos, RPM and turning direction are set. Linear servos, like their name, uses linear motion rather than rotational.

For this project we decided to use positional servos to display the braille character. We had initially wanted to use solenoids at first, but due to concerns about current draw we switched over to servos. To display braille characters using servos we only need two position states. A high position (this was set to 90 degrees) and a low position (this was set to zero degrees). As only two discrete position states were required, we decided to use positional servos.

We needed to use a 5% duty cycle for the low position and a 10% duty cycle for the high position. The PWM period was 20 ms, equivalent to 50HZ and wave amplitude was 5V.

I2S Audio Amplifier MAX98357

The MAX98357 is a Class-D Mono low-cost amplifier that is manufactured by Adafruit. It interfaces with I2S and has left and right channel information. It is a great new hardware to add for our project as it takes two breakout boards (I2S DAC and amplifier) and combines them into one. Overall, only the SCK, data line, and word select. Specifically the WS is connected to the LRC on the I2S amplifier board, SCK is connected to the BCLK, and the DIN is connected to the serial data line on the MCU. The gain by tying it to the Vin or ground and SD/Mode which is the shutdown mode can be configured by tying it to ground or a voltage divider for a specific voltage output. The specific schematic configurations are included in the Schematics section.

III. Schematics

The schematic of our entire system is shown below:

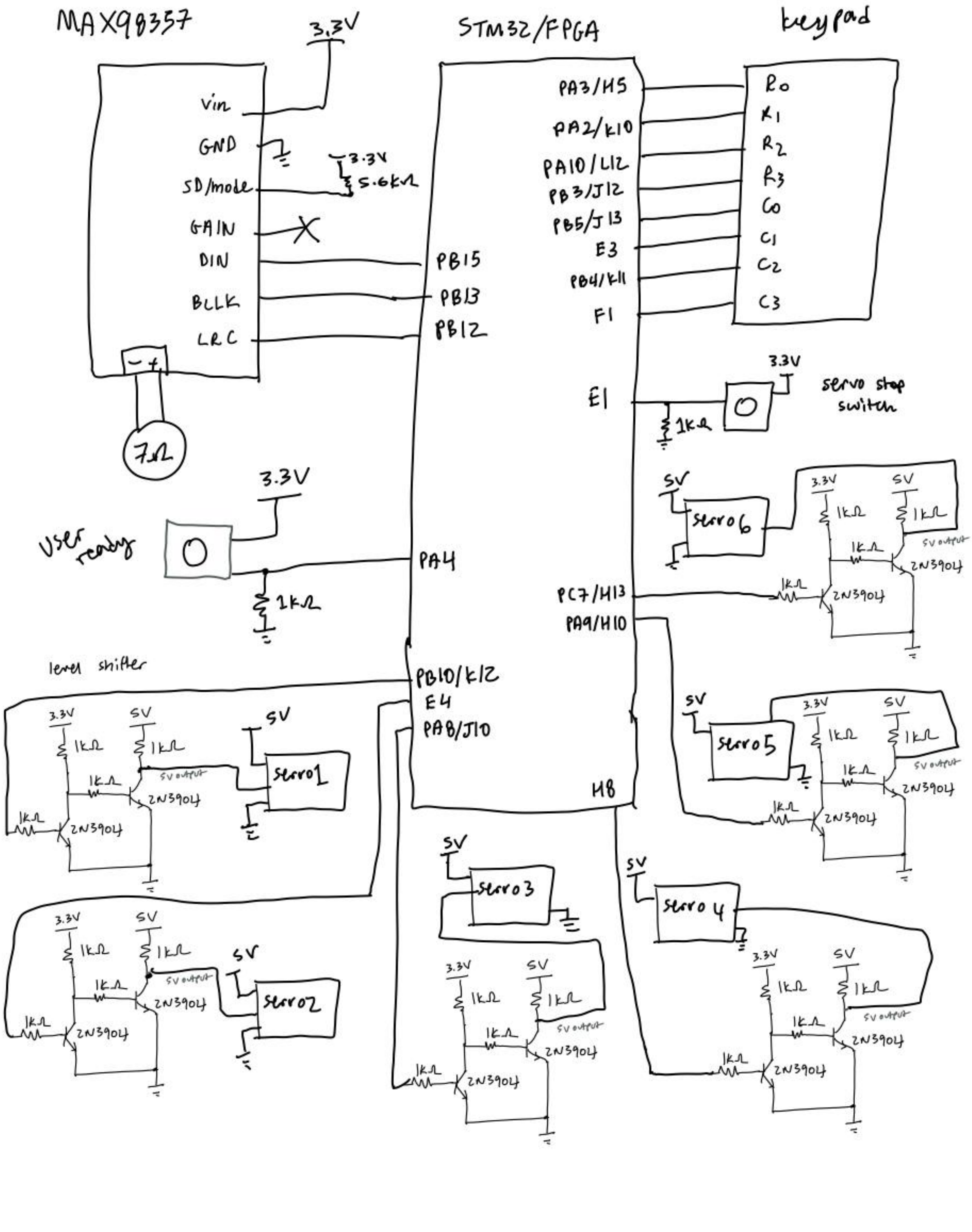

Figure 3: Schematic of Braille Calculator System

As stated above, the microcontroller and the FPGA are communicating through SPI. The microcontroller and the audio amplifier are connected through I2S. The servos are driven by the FPGA.

Schematic for I2S Audio Amplifier

The schematic diagram is between the connection between the STM32 or the MCU and the MAX98357 which is the I2S Audio Amplifier. Specifically, on the MAX98357, the LRC which stands for the left/right clock which tells the amplifier when the data is for the left and right channel. The LRC is then connected to the WS which is the word select. The BCLK which is the pin that tells the amplifier when they read data on the data pin, and DIN which is the actual data coming in which is then connected to the SD pin on the STM32 which is the data communicated. Note that this amplifier did not require a MCLK so that pin was left disconnected. Note that there are default settings on the audio amplifier. Specifically, note that the gain pin, when the pin is not connected to anything then the gain will default to 9dB. Then, for the SD/Mode pin, if the voltage on the SD is between 0.16V and 0.77V then the SD will take the average of the left and right channels. It is given that there is an internal 100kOhm pulldown resistor on the SD so we need to use a pull up resistor on the SD of around 560kOhm so using the voltage divider makes the voltage input to be 0.5V to the SD pin. The specifics of the MAX98357 is documented on Reference [1] included below.

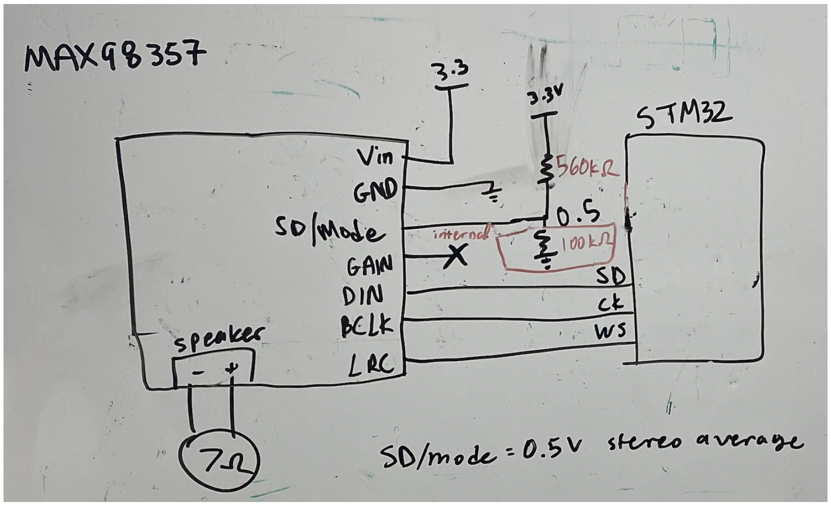

Figure 4: Schematic of the MAX98357 including the speakers and the STM32

Schematic for Micro Servos

At first, we proposed in our initial design to have solenoids for the braille output. However, as discussed with Prof Brake, solenoids require high amounts of current (up to a couple Amps), and neither our microcontroller nor our FPGA can supply that high amount of current. Thus, we transitioned to using microservos to actuate our servos. As referring to [2] in the reference, the servo runs on a 4.8-6V logic level and so 3.3 to 5V level shifters are needed for getting the 3.3V PWM signal from the microcontroller to actuate the servos at the correct 5V logic level.

For the level shifters, a common schematic is to use a MOSFET with a combination of two pull up resistors. Due to our resources in the electronics lab, we decided to use a combination of two 2N3904 transistors, pull up resistors to the 3.3V and 5V supply, and resistors in between the transistors. From an online resource [3], we used a similar schematic as shown in the one below where on the collector end of the Q2 we had the 5V output and on the base on a

Q1 transistor we had the 3.3V input of the PWM signal provided by the FPGA to actuate the servos.

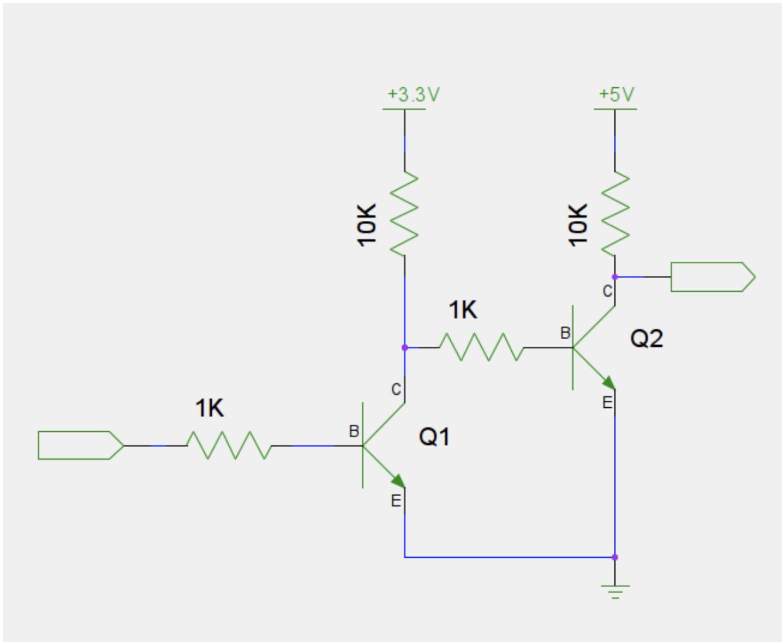

Figure 5: Schematic of a Level Shifter with 3.3V to 5V with two transistors

The only difference with this schematic provided above and our schematic (which is seen in Section III) is the resistor values. Since the servos are positional servos, we needed to provide a 50Hz PWM signal with 5% duty cycle to be at the zero angle position, 10% duty cycle to be at the 90 degree position, and a 15% duty cycle to be at the 180 degree position.

VI. FPGA Design

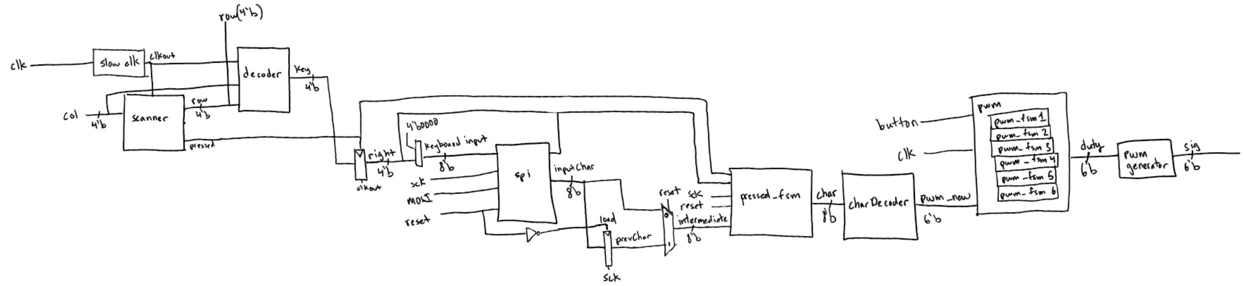

Figure 6: Overall Block Diagram of FPGA Design

The SystemVerilog code consists of 7 primary submodules: a module called slow_clk to output the correct 50Hz signal for the servos from the default clock signal at 12MHz. This included us to use a clock divide factor which was formed from the mathematical equation (12MHz * 2)/x = (100Hz)*2. Note that it is 100Hz not 50Hz here because we want to toggle this signal on and off twice in a period. Once we solve for x which is our clock divide factor, as noted above, we know the duty cycle so we know what percentage of the clock divide factor needs to be triggered on and off. In our case, we are trying to run this on a 10% duty cycle so we will need to trigger our signal at 10% of the clock divide factor.

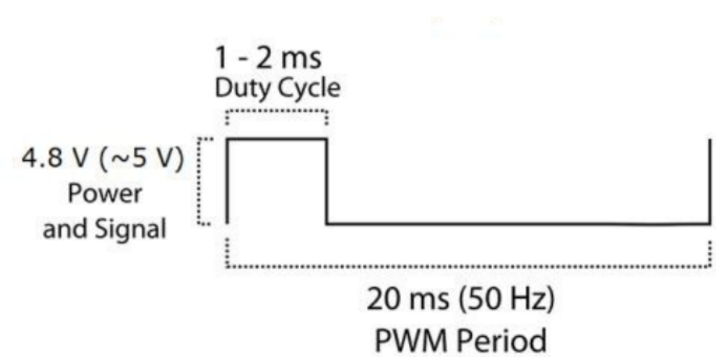

Figure 7: PWM Example with Duty Cycle provided by Reference [2]

The next couple of modules described includes the handling of the key press on the keypad, logic for the keypad, and the actual inputted value of the value pressed to send to the MCU. The second module is the scanner fsm which scans through the keypad to see what the value of the keypad might be. We will provide a high on each column and then move the rows to the next one. This module will continuously collect the keypad input and also handle the debouncing of the keys. The third module is the hex_decoder which holds the logic for converting the rows and columns to actual hexadecimal numbers. The next module is the charDecoder which translates the input character to a six bit signal for the correct orientation of the braille output. Then we have a flop enable module to keep track of values inputted to the keypad.

Then we have a module called pwm_fsm and the pwm_generator. The pwm_fsm inputs the button, clk, and the character value. Internal signals include the start degree and end degree for the servo to move in the 0 position and the desired actuated position (ranging up to 180 degrees). The pwm_generator includes six 19 bit signals for each duty cycle value which is the value of the percentage of the clock divide number to be turned on for each servo.

Finally, a module for the spi_secondary is used and was adapted from Reference [4], which inputs the sck, mosi, chip select, and 8 bit data from the primary. This module then outputs another 8 bit data value and also a 1 bit miso signal. Inside this module, there includes a 3 bit counter for when a full byte is transferred, a loadable shift register, and another register to align the miso to the falling edge of sck.

V. Microcontroller Design

The MCU served as the SPI primary in the communication between the FPGA and the MCU. Additionally, it also communicated with the audio amplifier through I2S. SPI communication was set to 8 bits while I2S was set to 16 bits. For I2S, the Philips audio standard was used. Additionally, a GPIO pin was configured as an input. A switch was attached to this pin and was used as the user's next character button.

Other uses of the MCU included: storing keypad inputs, decoding inputs and performing math operation, separating the answer into three 8 bit signals to be transmitted back to the FPGA, and generating sine waves to be sent to the speaker. Specifically, we The high level pseudocode of the MCU is shown below. See Appendix B for full MCU C code.

Note that we had three separate sine waves made in the MCU code for the number data which outputted the number of pulses for the ones and tens place and two sine waves to produce two separate tones for positive and negative signal. Note that when producing this data for the sine waves, we had to make sure the signal was properly separate for the left and right signals.

MCU High Level Pseudocode:

```
main(

 Configure flash;
 Configure clock (set to 84MHz);
 Configure SPI;
 Configure I2S;
 Configure GPIO Pin as Input;

 Generate sine wave data (one sine wave for negative tone, positive tone, and numbers
 tone)
```

```
Declare internal variables

while(1){
   1) COLLECT KEYPAD INPUTS UNTIL EQUALS ENTER BUTTON ENTERED
   2) DECODES MATH OPERATIONS
   4) PERFORM MATH OPERATION
   5) SEPARATE ANSWER INTO CHUNKS (Sign, 10s place, 1s place)
   6) LOOP THROUGH ANSWER CHUNKS:
       SEND CHUNK TO FPGA
       PLAY CORRESPONDING SOUND ON SPEAKERS
       WAIT FOR NEXT BUTTON TO BE PRESSED
   7) GO BACK TO TOP OF WHILE LOOP
   }
}
```

VI. Results

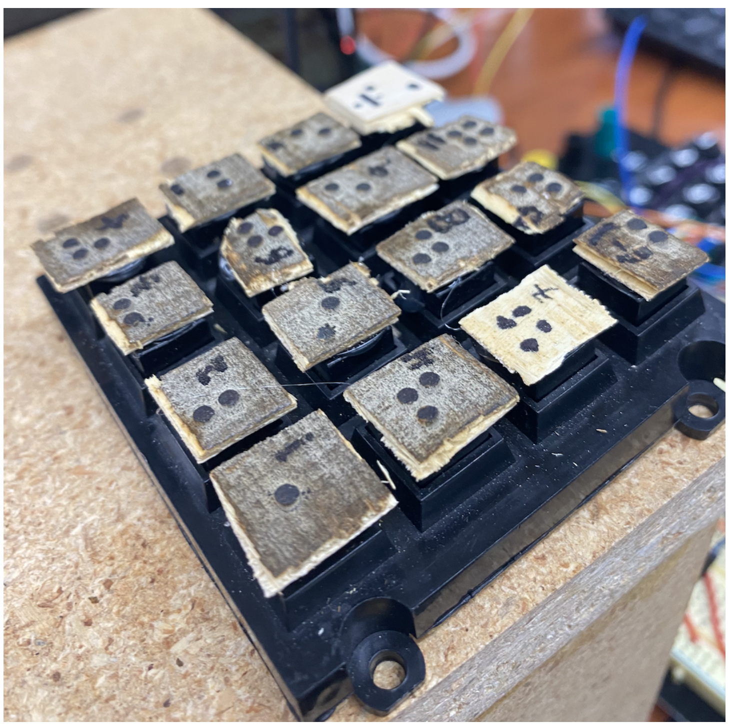

Figure 8: Lasercut Braille Keypad

Overall, our hard work paid off and our project was a success! We were able to correctly output the result of one digit mathematical expressions using addition, subtraction, multiplication, and division using the servos and have the speaker also output the correct values. There was not great documentation on the types of servos we ordered so using a function generator, we had to experiment to know that it was a position servo and dig through the internet to know what duty cycle and the frequency output the PWM signal is needed for specific servo positions. Additionally, another roadblock that we faced is realizing that the servo has a 4.8V-6V logic level output to 5V so a level shifter needed to be made for each servo.

Future work would include outputting recorded mp3 or wav files on the speaker that correspond to the character being displayed by the serovs. For example, outputting an audio recording saying “one” when the number one is displayed using the servos. This work would also include configuring SPI communication with an SD card since the MCU is limited in the amount of data it can hold and audio files are very large. An alternative may be using a DFPlayer Mini and using UART to output the correct mp3 files. We also had issues with the servos going haywire before the user starting inputting characters into the keypad. We developed a workaround which having a separate button to turn the servos off while the user was inputting their math expression then turn the servos back on once they were done.

VII. References

[1] MAX98357 I2S Audio Amplifier Datasheet.

https://learn.adafruit.com/adafruit-max98357-i2s-class-d-mono-amp/pinouts

[2] SG90 Microservo Datasheet.

http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf

[3] Simple Level Shifter

https://thecustomizewindows.com/2019/08/simple-level-shifter-with-transistors-3-3v-5v/

[4] Harris, David Money. “Digital Design and Computer Architecture: Chapter Nine I/O Systems” p. 530

[5] Logitech Inspired Conceptual Design

https://www.yankodesign.com/2020/01/23/a-logitech-inspired-braille-calculator-concept-for-the-visually-impaired/

VIII. Bill of Materials

Item	Quantity	Price per unit	Total	Notes
SG90 9g Micro Servos	2	10.99	21.98	Sold in packs of four so, total quantity is eight.
MAX98357A	1	8.99	8.99	I2S audio amplifier breakout board
Soft Bendable Aluminum Rods	1	13.45	13.45	Used to create braille characters
Sewing Pins	1	2.97	2.97	Used to create braille characters
Plywood		n/a	n/a	Spare wood available in student machine shop
Transistors	12	n/a	n/a	2N3904, two for each servo
8 Ohm Speaker	1	n/a	n/a	Used the one from digital lab
Hot glue		n/a	n/a	Available in student makerspace
1kOhm Resistors	24	n/a	n/a	For the 3V to 5V level shifters, 4 for each servo (available in digital lab)
			47.39	
		Total with tax	50.28	

IX. Appendix: Verilog

```
module braille_calc(input logic clk,
                    input logic sck,
                    input logic button,
                    input logic [3:0] col,
                    input logic mosi, // MOSI
                    output logic miso, // MISO
                    input logic reset, // CS
                    output logic [3:0] row,
                    output logic [5:0] sig);
    // internal logic
    logic [18:0] duty0;
    logic [18:0] duty1;
    logic [18:0] duty2;
    logic [18:0] duty3;
    logic [18:0] duty4;
    logic [18:0] duty5;

    logic [5:0] pwmNew;

    logic clkout;
    logic pressed;
    logic [3:0] key;

    logic [3:0] right;
    logic [7:0] inputChar;
    logic [7:0] char, intermediate;
    logic [7:0] keyboardInput;
    logic load;
    logic a;

    /*
    Braille
    0 3
    1 4
    2 6
```

```
	*/

	assign load = !reset;
	assign intermediate = (reset)? prevChar:inputChar;

	slow_clk slow(clk, clkout);
	scanner_fsm scanner(clkout, col, row, pressed);
	hex_decoder decoder(clkout, row, col, key);
	flopen f1(clkout, pressed, key, right);
	flopen f2(clk, load, inputChar, prevChar);

	pressed_fsm pp(pressed, sck, reset, intermediate, right, char);

	charDecoder c(char[3:0], pwmNew);

	pwm_fsm fsm0(button, clk, pwmNew[0], duty0);
	pwm_fsm fsm1(button, clk, pwmNew[1], duty1);
	pwm_fsm fsm2(button, clk, pwmNew[2], duty2);
	pwm_fsm fsm3(button, clk, pwmNew[3], duty3);
	pwm_fsm fsm4(button, clk, pwmNew[4], duty4);
	pwm_fsm fsm5(button, clk, pwmNew[5], duty5);

	pwm_generator pwm(clk, duty0, duty1, duty2, duty3, duty4, duty5, sig);

	assign keyboardInput = {4'b0, right};

	spi_secondary spi(sck, mosi, miso, reset, keyboardInput, inputChar);

endmodule

module pressed_fsm (input logic pressed,
				input logic clkout, reset,
				input logic [7:0] intermediate,
				input logic [3:0] right,
				output logic [7:0] char);

	// state logic
	typedef enum logic [2:0] {S0, S1, S2} statetype;
```

```
	statetype state, nextstate;

	always_ff @(posedge clkout)
		state <= nextstate;

	// nextstate logic
	always_comb
		case(state)
			S0:	if (!pressed) begin
							nextstate = S0;
							end
						else nextstate = S1;
			S1:	if (right != 4'b1111) begin // when right is not pressed as 0xF aka
equal sign
							nextstate = S1;
							end
						else // if right = 0xF
							nextstate = S2;
			S2:	if (pressed == 0) begin
							nextstate = S2;
							end
						else
							nextstate = S1;
			default: nextstate = S0;
		endcase

		assign char = (state == S2 && !reset) ? intermediate:8'b0;

endmodule


/////////////////////////////////////////////
//// spi
/////////////////////////////////////////////
module spi_secondary(input  logic sck, // from master
                     input  logic mosi, // from master
                     output logic miso, // to master
                     input  logic reset, // system reset
```

```
                                        input  logic [7:0] d, // data to send
                                        output          logic [7:0] q);

      logic [2:0] cnt;
      logic qdelayed;

      // 3 bit counter when full byte is transferred
      always_ff @(negedge sck, posedge reset)
            if (reset) cnt =0;
            else cnt= cnt + 3'b1;

      // loadable shift register
      always_ff @(posedge sck)
            q <= (cnt == 0) ? {d[6:0], mosi} : {q[6:0], mosi};

      // align miso to falling edge of sck
      always_ff @(negedge sck)
            qdelayed = q[7];


      assign miso = (cnt == 0) ? d[7] : qdelayed;

endmodule


module pwm_generator(input logic clk,
                                              input logic [18:0] duty0,
                                              input logic [18:0] duty1,
                                              input logic [18:0] duty2,
                                              input logic [18:0] duty3,
                                              input logic [18:0] duty4,
                                              input logic [18:0] duty5,
                                              output logic [5:0] sig);
  // (Looking at Lab 2) Note that the default clock signal is at 12MHz
  // So for our clock divide factor, (12MHz * 2)/x = 50Hz/2
      // Note that this is because we want to toggle on and off twice in a period
  // then our clock divide would need to be x = 240,000
```

```
        logic [18:0] clk_divide; // 2^19 = 524288 so it is around 48Hz
  always @(posedge clk) begin// use always_ff triggered at positive edge of clock
                if (clk_divide < 19'd240000) clk_divide <= clk_divide+1;
    else clk_divide <= 0;
        end

  assign sig[0] = (clk_divide < duty0) ? 1:0;
        assign sig[1] = (clk_divide < duty1) ? 1:0;
        assign sig[2] = (clk_divide < duty2) ? 1:0;
        assign sig[3] = (clk_divide < duty3) ? 1:0;
        assign sig[4] = (clk_divide < duty4) ? 1:0;
        assign sig[5] = (clk_divide < duty5) ? 1:0;

endmodule

module pwm_fsm (input logic button,
                        input logic clk,
                        input logic char_pwm,
                        output logic [18:0] duty);

    logic [18:0] startDeg;
    logic [18:0] endDeg;

    // state logic
    typedef enum logic [2:0] {S0, S1, S2} statetype;
    statetype state, nextstate;

    always_ff @(posedge clk)
        state <= nextstate;

    // nextstate logic
    always_comb
        case(state)
            S0:     if (button)
                            nextstate = S1;
                        else nextstate = S0;
            S1:     if (!button)
                            nextstate = S0;
```

```
                    else if (char_pwm)
                        nextstate = S2;
                    else nextstate = S1;
            S2:     if (!button)
                        nextstate = S0;
                    else if (!char_pwm)
                        nextstate = S1;
                    else nextstate = S2;
            default: nextstate = S0;
        endcase

        assign startDeg = 19'd12000; // 0 angle position
//      assign endDeg = 19'd18000; // 45 angle position
        assign endDeg = 19'd24000; // 90 angle position
        assign duty = (state == S2) ? endDeg:startDeg;

endmodule

// Summary: Provides a slow clock for when sampling if the value pressed
// on the key pad is stable. If the value is the same on two clock edges then the
// clock is stabilized.
module slow_clk (input logic clk,
                    output logic clkout);
  // (Looking at Lab 2) Note that the default clock signal is at 12MHz
  // So for our clock divide factor, (12MHz * 2)/x = (200Hz/2)*2
      // Note that this is because we want to toggle on and off twice in a period
  // then our clock divide would need to be x = 120,000

        logic [16:0] clk_divide;

        // we don't need an else statement because it resets once it goes above 32 bits
        always @(posedge clk) // use always_ff triggered at positive edge of clock
            begin
                if (clk_divide < 17'd120000) clk_divide <= clk_divide + 1;
                else clk_divide <= 0;
            end

        // assigning the values that the LED should show
```

```
		assign clkout = clk_divide[16]; // switching enable off and on


endmodule
// Name: Yoo-Jin Hwang
// Email address: yhwang@hmc.edu
// Date: 10/5/21
// Summary: In order to scan and see what the value of the keypad might be
// we will provide a high on a column and then move the rows to the next one
module scanner_fsm (input logic clk,
						input logic [3:0] col,
						output logic [3:0] row,
						output logic pressed);


	typedef enum logic [2:0] {R0, R1, R2, R3, R4, R5} statetype;
	statetype state, nextstate;
	always_ff @(posedge clk)
		state <= nextstate;

	// logic for all the rows
	always_comb
		case(state)
			R0:
				if (col == 0) nextstate = R1;
				else nextstate = R4;
			R1:
				if (col == 0) nextstate = R2;
				else nextstate = R4;
			R2:
				if (col == 0) nextstate = R3;
				else nextstate = R4;
			R3:
				if (col == 0) nextstate = R0;
				else nextstate = R4;
			R4: // if key is pressed
				nextstate = R5;
			R5:
```

```
                    if (col[3:0]) nextstate = R5;
                    else nextstate = R0;
                default: nextstate = R0;
            endcase

  //case if key is pressed, all the rows turn on to search for next column value
        assign row[0] = (state == R0);
        assign row[1] = (state == R1);
        assign row[2] = (state == R2);
        assign row[3] = (state == R3);

  //only pressed when you first get into the state which is why we have the nextstate = R4
        assign pressed = (state == R0 | state == R1 | state == R2 | state == R3) & (nextstate ==
R4);

endmodule
// Name: Yoo-Jin Hwang
// Email address: yhwang@hmc.edu
// Date: 10/5/21
// Summary: Holds the logic for converting the row and col to actual
// hexadecimal numbers and also detects if there is even a keypad being
// pressed.
module hex_decoder (input logic clk,
                        input logic [3:0] row, col,
                        output logic [3:0] key); //value to put into the seven
decoder
        logic [7:0] value;
        assign value = {row, col};

        always_comb
            case(value) // row[3], row[2], row[1], row[0], col[3], col[2], col[1], col[0]
                8'b00010001: key = 4'h1;
                8'b00010010: key = 4'h2;
                8'b00010100: key = 4'h3;
                8'b00100001: key = 4'h4;
                8'b00100010: key = 4'h5;
                8'b00100100: key = 4'h6;
                8'b01000001: key = 4'h7;
```

```
				8'b01000010: key = 4'h8;
				8'b01000100: key = 4'h9;
				8'b10000010: key = 4'h0;
				8'b00011000: key = 4'hA;
				8'b00101000: key = 4'hB;
				8'b01001000: key = 4'hC;
				8'b10001000: key = 4'hD;
				8'b10000001: key = 4'hE;
				8'b10000100: key = 4'hF;
				default: key = 4'h0; //default case
			endcase
endmodule
/*
Braille
0 3
1 4
2 5
*/
module charDecoder (input logic [3:0] inputChar,
					output logic [5:0] pwmNew);
		always_comb
			case(inputChar)
				4'b0000: pwmNew = 6'b011010; // 0
				4'b0001: pwmNew = 6'b000001; // 1
				4'b0010: pwmNew = 6'b000011; // 2
				4'b0011: pwmNew = 6'b001001; // 3
				4'b0100: pwmNew = 6'b011001; // 4
				4'b0101: pwmNew = 6'b010001; // 5
				4'b0110: pwmNew = 6'b001011; // 6
				4'b0111: pwmNew = 6'b011011; // 7
				4'b1000: pwmNew = 6'b010011; // 8
				4'b1001: pwmNew = 6'b001010; // 9
				4'b1010: pwmNew = 6'b010110; // + A
				4'b1011: pwmNew = 6'b100100; // - B
				4'b1100: pwmNew = 6'b010100; // * C
				4'b1101: pwmNew = 6'b001100; // / D
				4'b1110: pwmNew = 6'b100100; // (-) E
				4'b1111: pwmNew = 6'b110110; // = F
```

```
                default: pwmNew = 6'b000000;
            endcase
endmodule
/*
  Revised:
  Brief:
        Flop with enable signal.

*/
module flopen (
        input logic clk, en,
        input logic [3:0] a,
        output logic [3:0] b);

        always_ff @(posedge clk)
            if (en) b <= a;

endmodule
module flop(input logic clk,
                input logic [3:0] d,
                output logic [3:0] q);
        always_ff@(posedge clk)
            q <= d;
endmodule
/////////////////////////////////////////////////////////////////////////////////
module testbench();
  logic         clk;
  logic         button;
  logic         reset;
  logic [3:0] col;
  logic [3:0] row;
  logic  [5:0]   sig;
  logic [3:0] testvectors[100000:0];
  logic [3:0] vectornum;

  assign button = 1;
```

```
  // instantiate device under test
  braille_calc dut(clk, button, col, row, sig);

  // generate clock
  always
    begin
      clk=1; #5; clk=0; #5;
    end
  // at start of test, load vectors and pulse reset
  initial
          begin
                  $readmemb("braille_calc.tv", testvectors);
                  vectornum = 0; reset = 1; #22; reset = 0;
          end
  // apply test vectors on rising edge of clk
          always @(posedge clk)
                  begin
                          #1; {col} = testvectors[vectornum];
          end
  // check results on falling edge of clk
          always @(negedge clk)
                  if (~reset) begin    // skip during reset
                          vectornum = vectornum + 1;
                  end
endmodule
```

X. Appendix: C Code

```
#include <stdio.h>
#include <time.h> // time library for sound delay (I don't think we end up using it)
#include "STM32F401RE.h"
#include "math.h"

// Labs to look at: lab5 (PWM led), lab6 (IoT temp.) lab4 for the keypad FSM

/*  HIGH LEVEL OF MCU OPERATIONS
    (GENERATE SINE WAVE DATA, NUMBERS, POSITIVE, NEGATIVE) - done
    1) MCU READS OPERATION FROM FPGA - done
    2) WAIT FOR ENTER BUTTON HIT (EQUIVALENT OF = ) - done
```

```
   3) DECODES OPERATIONS (8 BITS EACH) - done
   4) MATH OPERATION --- done
   5) SEPARATE INTO CHUNKS (THREE DIGITS) ---- done
   6) LOOP:
      SEND CHAR TO FPGA
      PLAY CORRESPONDING SOUND ON SPEAKERS
      WAIT FOR NEXT BUTTON TO PRESS
   7) RESTART MAIN WHILE LOOP

   THINGS TO DO:
   - CALCULATOR MODULE - done
   - DECODER MODULE - done
   - OUTPUTTING CHARACTER MODULE
   - SPEAKER SOUND MODULE (FIGURE OUT WHAT TO ASSIGN FOR EACH VALUE)
-- done

   OTHER THINGS THAT NEED TO HAPPEN:
   - INITIALIZE SPI AND I2S
   - INITIALIZE GPIO PIN FOR USER BUTTON
*/


////////////////////////////////////////////////
// PIN ASSIGNMENT SUMMARY
////////////////////////////////////////////////
//
// SPI communication between MCU and FPGA
// SPI1_NSS: PB6_G12
// SPI1_MOSI: PA7_J2
// SPI1_MISO: PA6_J1
// SPI1_SCK: PA5_H4
//
// I2S2 between MAX and STM
// I2S2_WS: PB12
// I2S2_CK: PB13
// I2S2_SD: PB15

////////////////////////////////////////////////
```

```
// KEYPAD BUTTONS OPERATION/CHARACTER ASSIGNMENTS
//////////////////////////////////////////////
// 1 --> HEX VAL 0001 --> Assigned number 1
// 2 --> HEX VAL 0010 --> Assigned number 2
// 3 --> HEX VAL 0011 --> Assigned number 3
// 4 --> HEX VAL 0100 --> Assigned number 4
// 5 --> HEX VAL 0101 --> Assigned number 5
// 6 --> HEX VAL 0110 --> Assigned number 6
// 7 --> HEX VAL 0111 --> Assigned number 7
// 8 --> HEX VAL 1000 --> Assigned number 8
// 9 --> HEX VAL 1001 --> Assigned number 9
// 0 --> HEX VAL 0000 --> Assigned number 0
// A --> HEX VAL 1010 --> Assigned operation + (or positive)
// B --> HEX VAL 1011 --> Assigned operation - (or negative)
// C --> HEX VAL 1100 --> Assigned operation * (multiply)
// D --> HEX VAL 1101 --> Assigned operation / (divide)
// E --> HEX VAL 1110 --> Assigned operation - (negative)
// F --> HEX VAL 1111 --> Assigned operation = (equals)

//////////////////////////////////////////////
// Constants
//////////////////////////////////////////////

//////////////////////////////////////////////
// Function Prototypes
//////////////////////////////////////////////
void output(uint8_t*, int16_t*, int16_t*, int16_t*);
void decoder(uint8_t*, char*, int*, int*, int*, int*);
char operationKey(unsigned int);
void calculator(char, int, int, int, int, uint8_t*);
void speakerOutput(int16_t*, int16_t*, int16_t*, uint8_t);

//////////////////////////////////////////////
// OUTPUT
// Inputs: Answer sign, 10s digit, 1s digit
// Funtion will send hex value to FPGA and play corresponding sound
// ans form --> [sign, 10s, 1s]
//////////////////////////////////////////////
```

```
void output(uint8_t * ans, int16_t* Neg_Data, int16_t* Pos_Data, int16_t* Number_Data) {
  // loop through ans to display character one at a time
  for (int i = 0; i < 3; i = i + 1) {
    digitalWrite(GPIOB, 6, 1); // Set CS High
    spiSendReceive(ans[i]);
    digitalWrite(GPIOB, 6, 0); // Set CS Low

    uint8_t a = spiSendReceive(ans[i]);
    while(SPI1->SR.BSY); // Confirm all SPI transactions are completed

    speakerOutput(Neg_Data, Pos_Data, Number_Data, ans[i]); // play corresponding character
on speakers
    if (i < 3) {
      while(digitalRead(GPIOA, 4)); // wait for user to reset switch
      while(!digitalRead(GPIOA, 4)); // wait for user to set switch
    }
  }
  return;
}

/////////////////////////////////////////////
// DECODER
// Inputs: an array of 8'b messages from FPGA, pointers to sign1, sign2, val1, val2, operation
are the extra four)
// Possible combos:
// number, operation, number (combo 1)
// number, operation, sign, number (combo 2)
// sign, number, operation, number (combo 3)
// sign, number, operation, sign, number (combo 4)
/////////////////////////////////////////////
void decoder(uint8_t* message, char* operation, int* sign1, int* sign2, int* val1, int* val2) {
  unsigned int first = message[0];
  unsigned int second = message[1];
  unsigned int third = message[2];
  unsigned int fourth = message[3];
  unsigned int fifth = message[4];

  // go through possible combos and set things
```

```
  if (first == 0xB || first == 0xE) { // first input is a negative sign --> combos 3 and 4
    *sign1 = -1;
    *val1 = second;
    *operation = operationKey(third);

    if (fourth == 0xB || fourth == 0xE) { // fourth input is a negative sign --> combo 4
      *sign2 = -1;
      *val2 = fifth;
    } else {
      *sign2 = 1;
      *val2 = fourth;
    }
  } else { // first input is positive --> combos 1 and 2
    *sign1 = 1;
    *val1 = first;
    *operation = operationKey(second);

    if (third == 0xB || third == 0xE) { // third input is a negative sign --> combo 2
      *sign2 = -1;
      *val2 = third;
    } else {
      *sign2 = 1;
      *val2 = third;
    }
  }

}

// Outputs operation key to be used

char operationKey(unsigned int hex) {
  char operation;
  switch(hex) {
    case 0xA:
      operation = 'A'; // add
      break;
    case 0xB:
      operation = 'S'; // subtract
```

```
            break;
        case 0xC:
            operation = 'M'; // multiply
            break;
        case 0xD:
            operation = 'D'; // divide
            break;
        case 0xE:
            operation = 'S'; // subtract
            break;
        case 0xF:
            operation = 'E'; // this is equals, could potentially not need
            break;
        default:
            operation = 'F'; // set to bogus key
            break;
    }

    return operation;
}

/////////////////////////////////////////////////
// Calculator Function
// Created 12/7/2021
// Executes math operation and returns array of hex numbers corresponding to sign, 10s, and 1s
// Inputs:
//   - math operation, sign of value 1, value 1, sign of value 2, value 2, pointer to array
// Output:
//   - no explicit output but it edits the array passed into the function
//
// Note: sign1 and sign2 will either be positive one or negative one
/////////////////////////////////////////////////
void calculator(char operation, int sign1, int val1, int sign2, int val2, uint8_t *output) {
    int ans; // math operation answer

    // do operation
    switch(operation) {
        case 'A': // add
```

```
            ans = sign1*val1 + sign2*val2;
            break;
        case 'S': // subtract
            ans = sign1*val1 - sign2*val2;
            break;
        case 'M': // multiply
            ans = sign1*val1 * sign2*val2;
            break;
        case 'D': // divide (do not care about remainder)
            ans = (sign1*val1) / (sign2*val2);
            break;
        default:
            ans = 0;
    }

    // set output [sign, 10s, 1s]
    if (ans < 0) {
        output[0] = 11; // negative
        ans = ans * (-1);
    } else {
        output[0] = 10; // positive
    }

    // set tens and ones value
    output[1] = ans / 10;
    output[2] = ans % 10;
    return;
}

////////////////////////////////////////////////
// Speaker Output Function
// Created 12/7/2021
// Plays pulses and tones corresponding to inputted value
// 0-9 correspond to said numbers
// Hex val B (11 in decimal) corresponds to negative
// Hex val A (10 in decimal) corresponds to positive
// For positive sign --> play long high note
// For negative sign --> play long low tone
```

```
// For zero -----------> play long number tone
// For rest of numbers, play number of pulses corresponding to that number
// Inputs:
//   - NumberData(sine wave for numbers), NegData(sine wave for negative), PosData(sine wave
for positive), Val (unsigned hex value)
//
// Pins for I2S:
// PB5 for I2S3_SD on DS 46
// PB3 for I2S3_CK on DS 46
// PA4 for I2S3_WS on DS 45
////////////////////////////////////////////////
void speakerOutput(int16_t* Neg_Data, int16_t* Pos_Data, int16_t* Number_Data, uint8_t val)
{
   int16_t* sine_wave = Number_Data; // set default sine wave data to be for numbers
   int pulse = 50;                   // set default for loop pulse width to be for short pulses
   int pulseNum = 1;                 // set default for number of pulses to be 1
   int sineLength = 1000;

   if (val == 11) {        // play long tone, use neg sine wave
      sine_wave = Neg_Data;
      pulse = 150;
      sineLength = 2000;
   } else if (val == 10) { // play long tone, use pos sine wave
      sine_wave = Pos_Data;
      pulse = 150;
      sineLength = 800;
   } else if (val == 0) {
      pulse = 150;
   } else {
      pulseNum = val;
   }

   // play sound on speakers
   for(int a = 0; a < pulseNum; a = a + 1){
      for(int sound = 0; sound < pulse; sound = sound + 1){
         for (int i = 0; i < sineLength; i = i+2){
            i2sTransmission(sine_wave[i], sine_wave[i+1]);
         }
```

```
        }
        for(int sound = 0; sound < pulse; sound = sound + 1){
            for (int i = 0; i < sineLength; i = i+2){
                i2sTransmission(0, 0);
            }
        }
    }
}

////////////////////////////////////////////////
// Main Function
////////////////////////////////////////////////

// load pin is the PB6 aka CS

int main(void) {
    // Configure flash latency and set clock to run at 84 MHz
    configureFlash();
    configureClock();

    i2sInit();
    spiInit(1, 0, 0);

    // Enable PLLI2S
    RCC->CR.PLLI2SON = 1;

    // Enable GPIO clocks
    RCC->AHB1ENR.GPIOAEN = 1;
    RCC->AHB1ENR.GPIOBEN = 1;

    pinMode(GPIOA, 4, GPIO_INPUT);

    // Create sine waves for I2S
    int nu_data = 1000;
    int p_data = 800;
    int n_data = 2000;
    int16_t Number_Data[nu_data * 2];    // n_data is defined as 1000
    int16_t Pos_Data[p_data * 2];
```

```
  int16_t Neg_Data[n_data * 2];

        for (int i = 0; i < nu_data; i++) {
                Number_Data[i * 2] = (int16_t) (sin(2. * 3.14 * 9. * i / 1000.) * 9250); // L-ch (x
500 is amplitude)
                Number_Data[i * 2 + 1] =
                                (int16_t) (sin(2. * 3.14 * 11. * i / 1000.) * 9250); // R-ch
        }
  for (int i = 0; i < p_data; i++) {
    Pos_Data[i * 2] = (int16_t) (sin(2. * 3.14 * i / 800.) * 9250); // L-ch (x 500 is amplitude)
                Pos_Data[i * 2 + 1] =
                                (int16_t) (sin(2. * 3.14 * 3. * i / 800.) * 9250); // R-ch
  }
  for (int i = 0; i < n_data; i++) {
    Neg_Data[i * 2] = (int16_t) (sin(2. * 3.14 * 1. * i / 2000.) * 9250); // L-ch (x 500 is
amplitude)
                Neg_Data[i * 2 + 1] =
                                (int16_t) (sin(2. * 3.14 * 3. * i / 2000.) * 9250); // R-ch
  }

  // input from keypad
  uint8_t keypad_input[5] = {0x01, 0xA, 0x2, 0xFF, 0xFF};

  // for calculator
  char operation = 'A';
  int sign1;
  int val1;
  int sign2;
  int val2;

  // to output on FPGA and speakers
  uint8_t ans[3];

  while(1) {

    // collect key presses, loop through until enter is pressed
    int index = 0;
    while(1) {
```

```
            digitalWrite(GPIOB, 6, 1);
            spiSendReceive(0x00);
            digitalWrite(GPIOB, 6, 0);

            uint8_t key = spiSendReceive(0x05);
            while(SPI1->SR.BSY); // Confirm all SPI transactions are completed

            if ((key == 0xF) && (index > 0)) { // equals was pressed
                break;
            }

            if ((index == 0) && (key != 0xF)) {
                keypad_input[index] = key;
                index = index + 1;
            }

            // if new key is pressed, add it into array
            if ((index != 0) && (keypad_input[index - 1] != key)) {
                keypad_input[index] = key;
                index = index + 1;
            }
        }
        decoder(keypad_input, &operation, &sign1, &sign2, &val1, &val2);
        calculator(operation, sign1, val1, sign2, val2, ans);
        output(ans, Neg_Data, Pos_Data, Number_Data);
    }
}
```