An LED Array Audio Visualizer

Final Project Report
December 8th, 2017
E155

Evan Atchison and Zayra Lobo

Abstract:

The goal of this project was to create an audio visualizer using an off-the-shelf LED array, Raspberry Pi, FPGA, microphone, and MCP3002 ADC. By playing music or other audio into the microphone, the user is supposed to see a frequency spectrum displayed on the array. The Fast Fourier Transform (FFT) that converts the microphone data from the time domain to the frequency domain is performed on the FPGA. Though microphone data was displayed on the LED array and the FFT functioned properly, the post-FFT data was not properly sorted into logarithmically increasing frequency bins. Furthermore, our SPI master protocol between the FPGA and the ADC failed to properly acquire input data, thereby producing output that did not significantly correspond to the audio input data.

I. Introduction

The motivation of this project was to create an audio visualizer on an LED array. A microphone collects audio input from the environment and outputs a voltage proportional to the amplitude of the sound. The signal is then passed to an Analog-Digital Converter (ADC) which the FPGA read from via SPI. The FPGA accumulates signals over time and performs a Fourier transform on a 32-point data set. It then shifts this data out to the Raspberry Pi via SPI. The Raspberry Pi interprets the raw post-FFT data and displays it on the LED array to create a spectrum analyzer.

II. New Hardware

The Electret microphone + amplifier, 32 x 32 RGB LED array, and LED array Pi HAT from Adafruit were the three new pieces of hardware used by the team for this project. The microphone + amplifier board had three pins to interface with: V_{cc}, V_{out}, and ground. The team supplied 3.3V to V_{cc} (since the supply voltage range is 2.4 - 5.5V) and read the values from the V_{out} pin to CH0 on the MCP3002 ADC [1], [2]. The MAX4466 amplifier connected to the microphone had a gain that could be adjusted to be any value from 25 to 125. For this project, an intermediate gain value was used.

The 32 x 32 LED array was selected to provide as high of a resolution as possible for the output display while not going over the self-determined $100 budget of the project, $50 of which was reimbursed. The team designed the frequency spectrum such that every two columns of LEDs formed one frequency range “bin”. Thus, the 32 columns of LEDs were broken into 16 frequency ranges, which ideally would have been spaced on a logarithmic scale to model the logarithmic scale of octaves. Ideally, each column would also have a specified frequency range,

such as 20Hz - 40Hz, thus displaying the amplitude of signals falling in that frequency range for that audio sample.

The Pi HAT was used to connect the Pi control signals to the LED array via a ribbon cable, power the array via a barrel jack power cable plugged into the HAT, and shift between the 3.3V logic of the Pi to the 5V logic needed for the LED array [3]. Also, in order to use the Python library that Adafruit provided for the array, the Pi had to interface with the LED array through this HAT.

III. Schematics

A high-level block diagram of the entire system shows how each hardware component was connected:

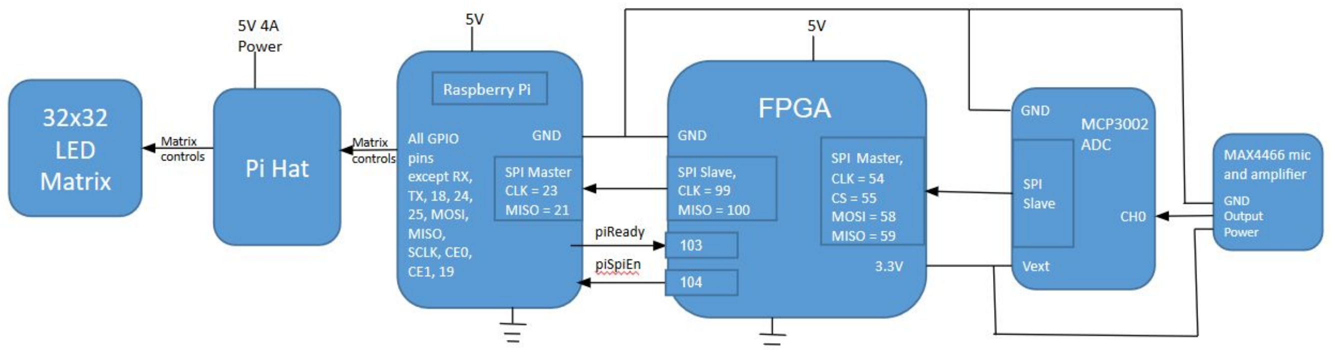

Fig. 1: The overall data flow for the project.

A block diagram for the overall FPGA logic demonstrates the function of the FPGA in this system:

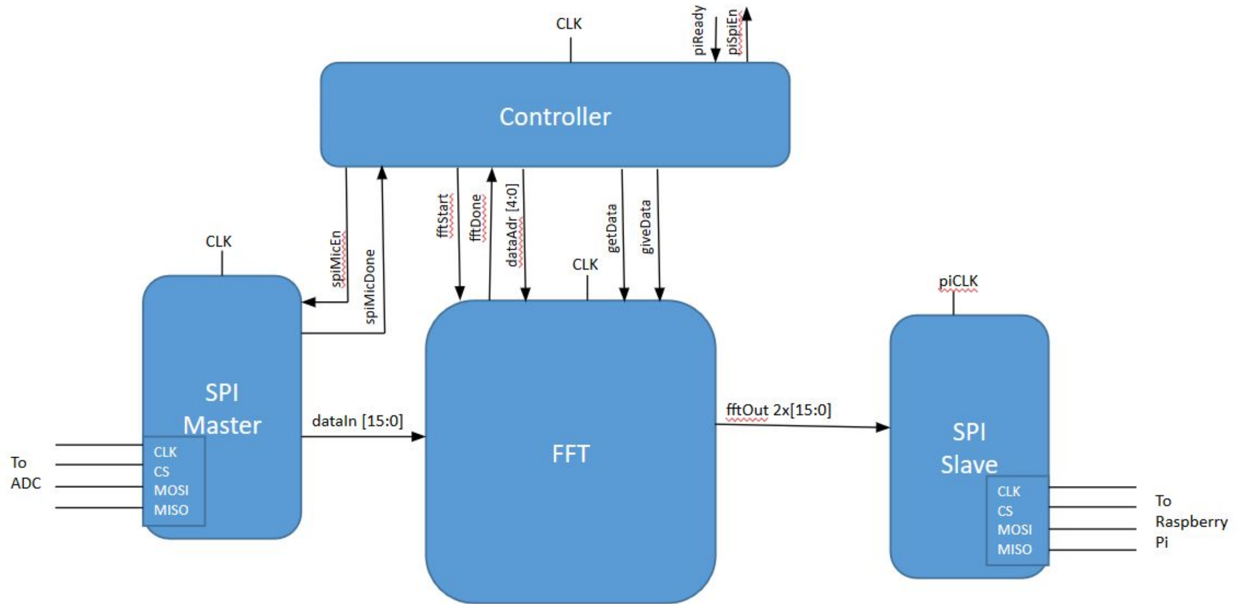

Fig. 2: High-level modules implemented on the FPGA.

A block diagram of the FFT shows how this modules is controlled and operates:

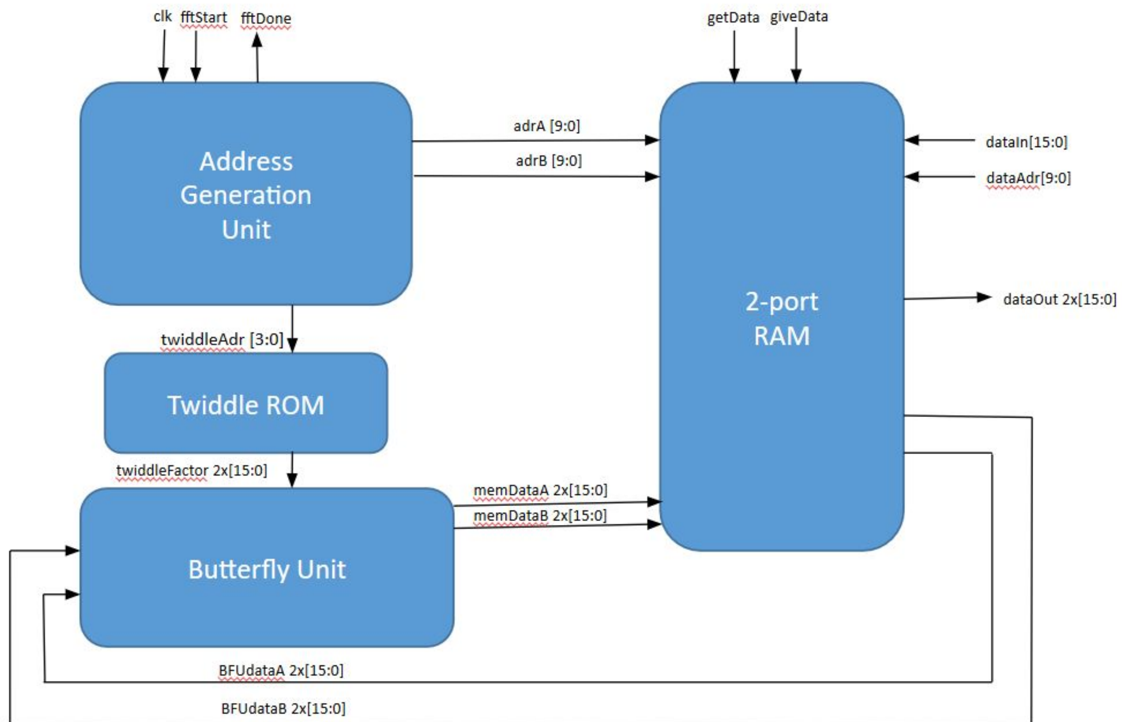

Fig. 3: Submodules implemented to make the FFT work.

IV. Microcontroller Design

The Raspberry Pi interfaces with the FPGA to receive output from the FFT and then drives the LED array to display that output. A HAT (Hardware Attached on Top) attaches to the Pi to allow for simpler interfacing between the Pi and the array. The HAT adds several features that make interfacing with the array easier, such as a ribbon cable output for the array signals, a barrel jack power input to power the array, and onboard level shifters to convert the Pi's 3.3V logic to 5V logic for the array.

Initially, the team planned to use Henner Zeller's C/C++ library for driving the LED matrix because it would allow for the use of EasyPIO.h, a C header file that included pin mapping and SPI interfaces necessary for the Pi to run in order to communicate with the FPGA [4], [5]. However, shortly before the project deadline, the team realized that this library would use almost all of the Pi's GPIO pins and all of the SPI pins to drive the array, leaving no quick method to implement an SPI interface between the Pi and the FPGA. Fortunately, Adafruit had forked this Github repository to create their own Python version of the same library that does not use the Pi SPI pins [6]. Using the classes and functions in this library, the team was able to write their own functions to produce spectrum patterns on the LED array that corresponded to the FFT output from the FPGA. Because EasyPIO.h is a C header file that remaps the Pi pins, it could not be used with the Python module which also had its own mapping of the pins. Thus, the SpiDev module was used to handle SPI signals from the Pi [7].

V. FPGA Design

A. General Implementation

The FPGA handles both signal acquisition from the microphone and signal processing via the FFT, and then passes the transformed signal to the Raspberry Pi. To do this, we implemented four high-level modules on the FPGA, as shown in Fig. 2.

To read signals from the microphone, the FPGA must communicate with the on-board MCP3002 as an SPI Master. To establish communication, we use a simple state machine to shift out the sixteen bits of the MCP3002's startup sequence on the MOSI line in sync with the clock signal provided to the MCP3002. With each bit shifted out, we capture the incoming bit on the MISO line and build a 16-bit value, which is masked to 10 bits (the data width of the ADC) and passed to the appropriate bit-reversed memory address in the FFT's memory block. The FSM for the Master SPI module is as shown:

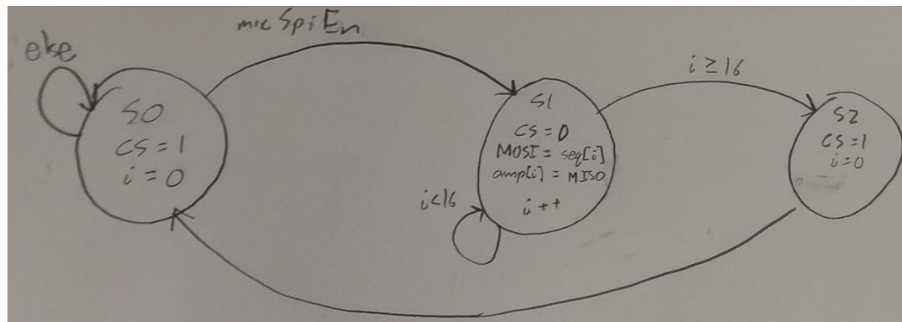

Fig. 4: Master SPI FSM

The most important data processing occurs within the FFT module. The design for our FFT was based on G. William Slade's *The Fast Fourier Transform in Hardware: A Tutorial Built on an FPGA Implementation* [8]. The implementation involves four key submodules: a two-port memory block, a butterfly unit which performs a two-point Fourier Transform, a lookup table for twiddle factors, and an address generation unit to specify the locations of inputs and

outputs to the butterfly unit within the memory.We modify Slade's implementation by eliminating much of the pipelining work that they do. In doing so we simplify the implementation significantly, but sacrifice the ability to write to the memory every clock cycle. Writing to the memory every other clock cycle did not significantly affect lag between matrix updates in our final implementation.

After performing the Fourier transform, we must shift out data as the Pi requests it, so we implement an SPI slave module. The procedure to do this is extremely simple, although not a pure implementation of an SPI interface. We use only the clock and MISO lines, shifting out a single bit of the 32-bit FFT output on each positive clock edge. Two GPIO lines between the FPGA and Pi act as "ready" signals for each, which help ensure that the SPI module receives data from the correct place in memory before the Pi sends clock signals.

Finally, we implemented a controller module which effectively segmented the sequence of operations performed by the FPGA, ensuring that data was captured, transformed, and shifted out in the correct order. The controller FSM is as shown:

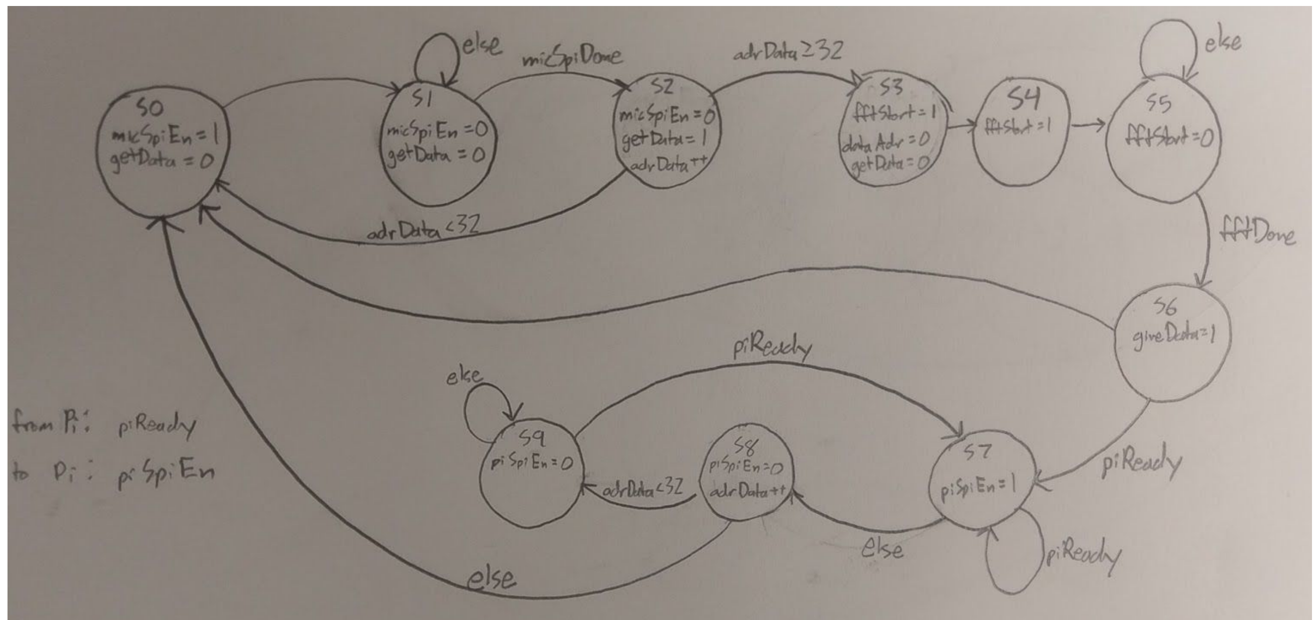

Fig. 5: FPGA controller state machine

B. Fast Fourier Transform in Hardware

Implementing the FFT module in hardware on the FPGA proved to be the most significant challenge of the project. In particular, we ran into issues with interpreting Slade's implementation into a functional non-pipelined FFT. We will describe the design process of each submodule (shown in Fig. 3) and outline the points of confusion we faced for the benefit of future groups hoping to implement FFT.

Memory: The two-port memory block we designed began as a very straightforward modification of the canonical form of a single-port RAM module in SystemVerilog. However, we chose to add another level of complexity to the memory block by providing it with a third address from the high-level controller module and single-bit *getData* and *giveData* signals. These signals allow the controller to override the FFT and read/write data to the memory during data acquisition and delivery. While this implementation was effective for our 32-point FFT, it prevented the module from being interpreted properly as a RAM block and therefore caused significant problems when we attempted to scale to 512-point FFT. We recommend future groups find another way to load values into and out of the two-port RAM without additional inputs and outputs.

Twiddle ROM: The twiddle factors are stored in a hard-coded lookup table. For higher-order FFT, it is preferable to encode this lookup table in a text file. Our 32-point values were provided in Slade's paper, while the values for our attempted 512-point FFT came from a MATLAB algorithm found in Curt Hillier and Maik Brett's *MPC5775K Twiddle Factor Generator User Guide*. [9]

Butterfly Unit: The Butterfly Unit (BFU) uses a complex multiplier to multiply a twiddle factor with a complex value and then performs an addition and subtraction with the result to generate output as described by Slade. To correctly implement the BFU, we interpret twiddle factors as signed fixed-point number with 15 decimal bits. We perform signed complex multiplication in the complex multiplier submodule (SystemVerilog requires that the values be specified as signed when performing the multiplication) and then left-shift the result by 15 bits to keep it the same order of magnitude as the input data before performing the addition and subtraction operations.

Controller: To implement the controller, we implemented in hardware the C algorithm shown by Slade using a state machine to contain a double-nested for loop. The loop is incremented every other clock cycle, and the second clock cycle of each loop iteration is used to activate *memwr,* where the output of the BFU is stored in memory. Our state machine is as shown:

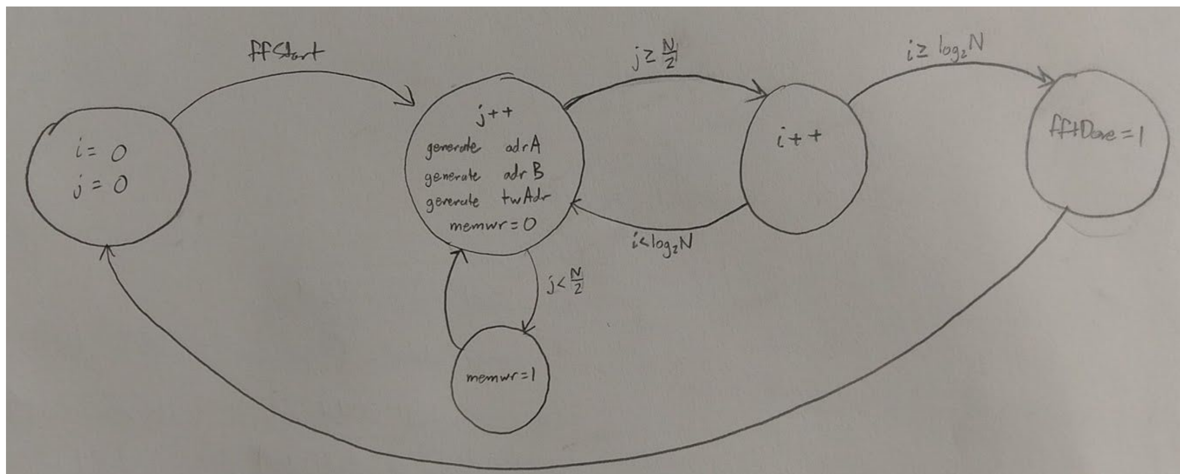

Fig. 6: State machine of FFT logic in implemented in SystemVerilog HDL

We performed our FFT on the square wave used in testing by Slade in ModelSim, and received the following results:

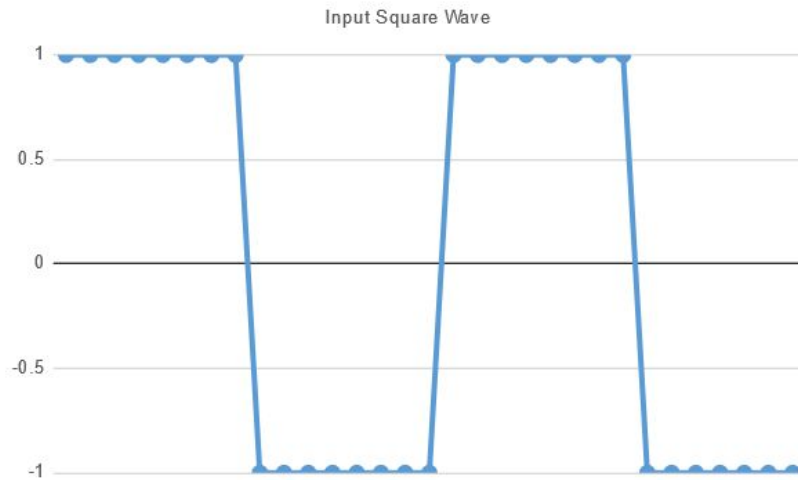

Fig. 7: Square wave test input to FFT

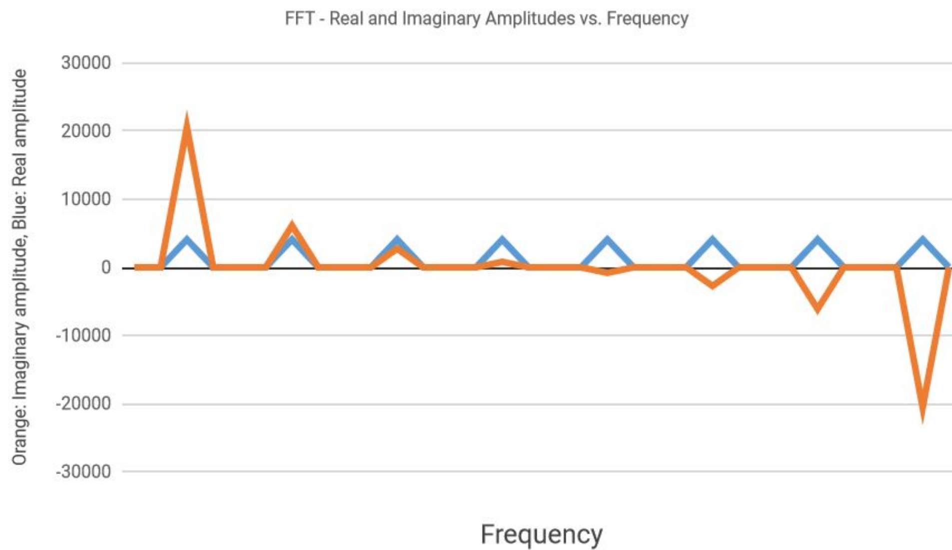

Fig. 8: Output of FFT given above square wave as input

The y-axis scaling of the second graph is as a fraction of 32768.We see the negative values are inverted, but our result otherwise matches Slade's. This value inversion is insignificant, as we take the total magnitude of the signal in the end.

VI. Results

The Pi received values from the FPGA via SPI, and after performing the magnitude calculation and scaling, the values did not exceed the LED array's upper limit of 31. When a

constant tone was played at a specified frequency, the LED array was expected to display a spike in the frequency "bin" that the specified frequency corresponded to. However, these values did not appear to correspond with the sound input to the microphone. Even when a constant tone was played about a centimeter away from the microphone, the output of the array did not change to show a spike in the respective frequency "bin" corresponding to the tone's frequency.

The failure to display output corresponding to the input from the microphone was due to two main problems. First, we failed to process the data correctly on the Pi. We allocated the "bins" (the bars of the graph) linearly with the frequency list output by the FFT, while we had intended to do so logarithmically. More importantly, we believe our SPI master module connected to the microphone failed to acquire data correctly. We saw all bins on the matrix fill to about halfway for most of the matrix's operation, which would suggest that all frequencies analyzed were present in roughly equal proportion, which is very unlikely. This strongly suggests that we read values from the microphone incorrectly. Upon reflection, our state-machine implementation of the SPI master interface was likely unnecessarily complicated when a simpler interface without a state machine would have sufficed. Furthermore, we sent a continuous clock signal to the ADC, rather than only sending it with our read signals, which likely confused the ADC and garbled the data we were receiving. Solving these issues with the SPI master would likely have greatly increased the functionality of our project, but unfortunately we only got the data display working very close to the final checkoff and did not have the time to thoroughly debug the system.

VII. References

[1] Maxim, "Low-Cost, Micropower, SC70/SOT23-8, Microphone Preamplifiers with Complete Shutdown," MAX4466 datasheet, 2001.

[2] Microchip, "2.7V Dual Channel 10-Bit A/D Converter with SPI Serial Interface," MCP3002 datasheet, 2011.

[3] Adafruit. "Adafruit RGB Matrix HAT + RTC for Raspberry Pi - Mini Kit." Adafruit.com, 21 Jan. 2015, www.adafruit.com/product/2345.

[4] Zeller, Henner. "rpi-rgb-led-matrix." GitHub, 1 Nov. 2017, github.com/hzeller/rpi-rgb-led-matrix.

[5] Lichtman, Sarah & Vasquez, Joshua, "EasyPIO.h," 8 Oct. 2013.

[6] Adafruit. "adafruit/rpi-rgb-led-matrix." GitHub, 27 Aug. 2017, https://github.com/adafruit/rpi-rgb-led-matrix.

[7] TightDev. "SpiDev Documentation." *Tightdev.net*, tightdev.net/SpiDev_Doc.pdf.

[8] Slade, George. (2013). The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation.

[9] Hillier, Curt and Brett, Maik (2015) MPC5775K Twiddle Factor Generator User Guide.

VIII. Parts List

Part	**Source**	**Vendor Part #**	**Price**
32x32 RGB LED Matrix Panel - 5mm Pitch	Adafruit	2026	$44.95
Electret Microphone Amplifier - MAX4466 with Adjustable Gain	Adafruit	1063	$6.95
Adafruit RGB Matrix HAT + RTC for Raspberry Pi - Mini Kit	Adafruit	2345	$24.95
5V 4A (4000mA) switching power supply - UL Listed	Adafruit	1466	$14.95

IX. Appendices

A. Pi Code

```
import spidev
import time
import RPi.GPIO as GPIO
import math
import Image
import ImageDraw
import time
from rgbmatrix import Adafruit_RGBmatrix

matrix = Adafruit_RGBmatrix(32, 1)

#Evan Atchison & Zayra Lobo
#December 5, 2017
#Using rpi-rgb-led-matrix-py library, draws a spectrum on a 32x32 array
#given a 32 bit array of values between 0 and 31

def drawSpectrum(amp):

      # Bitmap example w/graphics prims
      image = Image.new("RGB", (32, 32))
      draw  = ImageDraw.Draw(image)   # Declare Draw instance before prims

      count = 0
      i = 0
      matrix.Clear()

      #Use draw.line in library to draw the spectrum with amp values
      draw.line((0, 0, 0, amp[0]), fill = "#FF0000")
      draw.line((1, 0, 1, amp[0]), fill = "#FF0000")

      draw.line((2, 0, 2, amp[1]), fill = "#FF8000")
      draw.line((3, 0, 3, amp[1]), fill = "#FF8000")

      draw.line((4, 0, 4, amp[2]), fill = "#FFFF00")
      draw.line((5, 0, 5, amp[2]), fill = "#FFFF00")

      draw.line((6, 0, 6, amp[3]), fill = "#80FF00")
      draw.line((7, 0, 7, amp[3]), fill = "#80FF00")

      draw.line((8, 0, 8, amp[4]), fill = "#00FF00")
      draw.line((9, 0, 9, amp[4]), fill = "#00FF00")
```

```
        draw.line((10, 0, 10, amp[5]), fill = "#00FF80")
        draw.line((11, 0, 11, amp[5]), fill = "#00FF80")

        draw.line((12, 0, 12, amp[6]), fill = "#00FFFF")
        draw.line((13, 0, 13, amp[6]), fill = "#00FFFF")

        draw.line((14, 0, 14, amp[7]), fill = "#0080FF")
        draw.line((15, 0, 15, amp[7]), fill = "#0080FF")

        draw.line((16, 0, 16, amp[8]), fill = "#0000FF")
        draw.line((17, 0, 17, amp[8]), fill = "#0000FF")

        draw.line((18, 0, 18, amp[9]), fill = "#7F00FF")
        draw.line((19, 0, 19, amp[9]), fill = "#7F00FF")

        draw.line((20, 0, 20, amp[10]), fill = "#FF00FF")
        draw.line((21, 0, 21, amp[10]), fill = "#FF00FF")

        draw.line((22, 0, 22, amp[11]), fill = "#FF007F")
        draw.line((23, 0, 23, amp[11]), fill = "#FF007F")

        draw.line((24, 0, 24, amp[12]), fill = "#FF0000")
        draw.line((25, 0, 25, amp[12]), fill = "#FF0000")

        draw.line((26, 0, 26, amp[13]), fill = "#FF8000")
        draw.line((27, 0, 27, amp[13]), fill = "#FF8000")

        draw.line((28, 0, 28, amp[14]), fill = "#FFFF00")
        draw.line((29, 0, 29, amp[14]), fill = "#FFFF00")

        draw.line((30, 0, 30, amp[15]), fill = "#80FF00")
        draw.line((31, 0, 31, amp[15]), fill = "#80FF00")

        matrix.SetImage(image.im.id, 0, 0)

#Start SPI communication
spi = spidev.SpiDev()
spi.open(0,1)
spi.bits_per_word = 8
spi.max_speed_hz = 100000

#Initialize GPIO pins & variables
GPIO.setmode(GPIO.BCM)
GPIO.setup(24, GPIO.IN)
GPIO.setup(25, GPIO.OUT, initial = GPIO.HIGH)
```

```
j = 0
maxAmp = 0
amp = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

#SPI communication & calling drawSpectrum to display values
while(1):
      for k in range(0,16):
            while(j < 2):
                  piSpiEn = GPIO.input(24)
                  if(piSpiEn): #when the FPGA is done with FFT
                        realData1 = spi.xfer([0x00])
                        realData2 = spi.xfer([0x00])
                        imagData1 = spi.xfer([0x00])
                        imagData2 = spi.xfer([0x00])

                        realDataNum1 = realData1[0]
                        realDataNum2 = realData2[0]
                        imagDataNum1 = imagData1[0]
                        imagDataNum2 = imagData2[0]

                        #Convert data from signed to unsigned values

                        if(realDataNum2 > 127):
                              actualRealData2 = (realDataNum2 - 256) * 128
                        else:
                              actualRealData2 = realDataNum2 * 128

                        actualRealData = realDataNum1 + actualRealData2

                        if(imagDataNum2 > 127):
                              actualImagData2 = (imagDataNum2 - 256) * 128
                        else:
                              actualImagData2 = imagDataNum2 * 128

                        actualImagData = imagDataNum1 + actualImagData2

                        #Tell FPGA the Pi is busy
                        GPIO.output(25, GPIO.LOW)
                        #Compute amplitudes
                        temp = math.sqrt(actualRealData**2 +
                        actualImagData**2)
                        temp = temp/1024
                        if(maxAmp < temp):
                              maxAmp = temp
```

```
                #Tell FPGA the Pi can receive more data
                GPIO.output(25, GPIO.HIGH)
                j += 1

        j = 0
        #Cast to an int before writing into array
        amp[k] = int(maxAmp)
        maxAmp = 0
        print amp[k]
    #Draw spectrum with the current int data in amp
    drawSpectrum(amp)
spi.close()
```

B. FPGA Code

```
module finalproject(input logic clk,
                    input logic piReady,
                    input logic sdi,
                    input logic micMISO,
                    input logic piclk,
                    output logic piSpiEn,
                    output logic sdo,
                    output logic micMOSI,
                    output logic micCS,
                    output logic sclk,
                    output logic [7:0] leds);
        //signals for the controller
        logic micSpiDone, fftDone, getData, giveData, fftStart, micSpiEn,
        fftDoneHold;
        logic [4:0] adrData;

        //signals for spiMic
        logic [15:0] micData;

        //signals for spiPi, fft
        logic [31:0] fftOut;

        //debugging
        logic [3:0] stateNumber;
        logic [1:0] spiStateNumber;

        slowclk final_slowclk(clk, sclk);
        controller final_controller(sclk, micSpiDone, fftDone, piReady,
        getData, giveData, fftStart, micSpiEn, piSpiEn, fftDoneHold,
        adrData, stateNumber);
```

```
            spiMic final_spiMic(micSpiEn, sclk, micMISO, micMOSI, micCS,
            micData, micSpiDone, spiStateNumber);
            spiPi final_spiPi(piclk, fftOut, sdo);
            minifft final_fft(sclk, fftStart, getData, giveData, adrData,
            micData, fftDone, fftOut);

            logic fftWasDone, piSpiEnWas, micSpiDoneWas;
            always_ff @(posedge clk)
                  begin
                        if(fftDone) fftWasDone <= fftDone;
                        else fftWasDone <= fftWasDone;

                        if(piSpiEn) piSpiEnWas <= piSpiEn;
                        else piSpiEnWas <= piSpiEnWas;

                        if(micSpiDone) micSpiDoneWas <= micSpiDone;
                        else micSpiDoneWas <= micSpiDoneWas;
                  end

             //debugging leds
            assign leds[1:0] = spiStateNumber;
            assign leds[2] = piReady;
            assign leds[3] = piSpiEn;
            assign leds[7:4] = stateNumber;

endmodule

//Make a slow clock to access from other modules
module slowclk(input logic clk,
            output logic sclk);

      logic [31:0] sclkCount = 32'b0;
      logic [31:0] sclkDelay = 32'd1000; //clock divider sets sampling rate


      always_ff @(posedge clk)
            if (sclkCount > sclkDelay)
                  begin
                        sclk <= ~sclk;
                        sclkCount <= 0;
                  end
            else sclkCount <= sclkCount + 1;
endmodule

//master controller for all FPGA logic
module controller(input logic clk,
                                    input logic micSpiDone,
                                    input logic fftDone,
```

```
                              input logic piReady,
                              output logic getData,
                              output logic giveData,
                              output logic fftStart,
                              output logic micSpiEn,
                              output logic piSpiEn,
                              output logic fftDoneHold,
                              output logic [4:0] adrData,
                              output logic [3:0] stateNumber);
logic nextGetData, nextGiveData, nextFftStart, nextMicSpiEn,
nextPiSpiEn, nextFftDoneHold;
logic [4:0] nextAdrData;
logic [3:0] nextStateNumber;

typedef enum logic [3:0] {S0, S1, S2, S3, S4, S5, S6, S7, S8, S9}
statetype;
statetype state, nextstate;

always_ff @(posedge clk)
      begin
            state <= nextstate;
            getData <= nextGetData;
            giveData <= nextGiveData;
            fftStart <= nextFftStart;
            micSpiEn <= nextMicSpiEn;
            piSpiEn <= nextPiSpiEn;
            fftDoneHold <= nextFftDoneHold;
            adrData <= nextAdrData;
            stateNumber <= nextStateNumber;
      end

always_comb
      begin
            nextAdrData = 0;
            nextstate = S0;
            case(state)
                  S0: //initialize mic SPI
                        begin
                              nextstate = S1;
                              nextGetData = 0;
                              nextGiveData = 0;
                              nextFftStart = 0;
                              nextMicSpiEn = 1;
                              nextPiSpiEn = 0;
                              nextFftDoneHold = 0;
                              nextAdrData = adrData;
                              nextStateNumber = 4'b0000;
                        end
```

```
S1: //wait for spi value to fill
      begin
            if(micSpiDone) nextstate = S2;
            else nextstate = S1;
            nextGetData = 0;
            nextGiveData = 0;
            nextFftStart = 0;
            nextMicSpiEn = 1;
            nextPiSpiEn = 0;
            nextFftDoneHold = 0;
            nextAdrData = adrData;
            nextStateNumber = 4'b0001;
      end
S2: //put spi value in memory and loop or finish
      begin
            if(adrData<31) nextstate = S0;
            else nextstate = S3;
            nextGetData = 1;
            nextGiveData = 0;
            nextFftStart = 0;
            nextMicSpiEn = 0;
            nextPiSpiEn = 0;
            nextFftDoneHold = 0;
            nextAdrData = adrData+5'b00001;
            nextStateNumber = 4'b0010;
      end
S3: //initialize fft
      begin
            nextstate = S4;
            nextGetData = 0;
            nextGiveData = 0;
            nextFftStart = 1;
            nextMicSpiEn = 0;
            nextPiSpiEn = 0;
            nextFftDoneHold = 0;
            nextAdrData = 0;
            nextStateNumber = 4'b0011;
      end
S4: //hold fftstart for one more clock cycle
      begin
            nextstate = S5;
            nextGetData = 0;
            nextGiveData = 0;
            nextFftStart = 1;
            nextMicSpiEn = 0;
            nextPiSpiEn = 0;
            nextFftDoneHold = 0;
            nextAdrData = 0;
```

```
                        nextStateNumber = 4'b0100;
                end
        S5: //wait for fft to be done
                begin
                        if(fftDone) nextstate = S6;
                        else nextstate = S5;
                        nextGetData = 0;
                        nextGiveData = 0;
                        nextFftStart = 0;
                        nextMicSpiEn = 0;
                        nextPiSpiEn = 0;
                        nextFftDoneHold = 0;
                        nextAdrData = 0;
                        nextStateNumber = 4'b0101;
                end
                S6: //configure memory to give data, check to
                    //make sure pi is ready
                begin
                        if(piReady) nextstate = S7;
                        else nextstate = S0;
                        nextGetData = 0;
                        nextGiveData = 1;
                        nextFftStart = 0;
                        nextMicSpiEn = 0;
                        nextPiSpiEn = 0;
                        nextFftDoneHold = 1;
                        nextAdrData = 0;
                        nextStateNumber = 4'b0110;
                end
        S7: //enable spi, wait for pi to get data
                begin
                        if(piReady) nextstate = S7;
                        else nextstate = S8;
                        nextGetData = 0;
                        nextGiveData = giveData;
                        nextFftStart = 0;
                        nextMicSpiEn = 0;
                        nextPiSpiEn = 1;
                        nextFftDoneHold = fftDoneHold;
                        nextAdrData = adrData;
                        nextStateNumber = 4'b0111;
                end
        S8: //increment address counter
                begin
                        if(adrData<31) nextstate = S9;
                        else nextstate = S0;
                        nextGetData = 0;
                        nextGiveData = giveData;
```

```
                              nextFftStart = 0;
                              nextMicSpiEn = 0;
                              nextPiSpiEn = 0;
                              nextFftDoneHold = fftDoneHold;
                              nextAdrData = adrData+5'b00001;
                              nextStateNumber = 4'b1000;
                        end
                  S9: //wait for pi to be ready again
                        begin
                              if(piReady) nextstate = S7;
                              else nextstate = S9;
                              nextGetData = 0;
                              nextGiveData = giveData;
                              nextFftStart = 0;
                              nextMicSpiEn = 0;
                              nextPiSpiEn = 0;
                              nextFftDoneHold = fftDoneHold;
                              nextAdrData = adrData;
                              nextStateNumber = 4'b1001;
                        end
            endcase
      end
endmodule

//Enables SPI communication between the ADC and FPGA
module spiMic(input logic micSpiEn,
                              input logic clk,
                              input micMISO,
                              output logic micMOSI,
                              output logic micCS,
                              output logic [15:0] micData,
                              output logic micSpiDone,
                              output logic [1:0] spiStateNumber);


      //instantiate variables for timing and decoder logic
      logic [31:0] count = 32'b0;
      logic [31:0] nextCount = 32'b0;
      logic [15:0] startSequence = 16'b1101000000000000;
      logic [15:0] amask = 16'b0000001111111111;

      logic [15:0] amp, nextAmp;

      logic [1:0] nextSpiStateNumber;

      //state definitions
      typedef enum logic [1:0] {S0, S1, S2} statetype;
```

```
statetype state, nextstate;

//state advancing logic
always_ff @(negedge clk)
	begin
		state <= nextstate;
		amp <= nextAmp;
		count <= nextCount;
		if(micSpiDone) micData <= (amp&amask);//-16'd512;
		else micData <= micData;
		spiStateNumber <= nextSpiStateNumber;
	end

//combinational logic for states
always_comb
	begin
		nextstate = S0;
		case(state)
			S0: //starter state, wait for enable
				begin
					nextSpiStateNumber = 2'b00;
					micCS = 1;
					micSpiDone = 0;
					micMOSI = 1'b0;
					nextAmp = amp;
					nextCount = 0;
					if(micSpiEn) nextstate = S1;
					else nextstate = S0;
				end
			S1: //send start sequence
				begin
					nextSpiStateNumber = 2'b01;
					micCS = 0;
					micSpiDone = 0;
					micMOSI = startSequence[15-count];
					nextAmp = amp;
					nextAmp[15-count] = micMISO;
					nextCount = count + 1;
					if (count>32'd14) nextstate = S2;
					else nextstate = S1;
				end
			S2: //finished state, assert done
				begin
					nextSpiStateNumber = 2'b11;
					micCS = 1;
					micSpiDone = 1;
					micMOSI = 1'b0;
					nextAmp = amp;
```

```
                                    nextCount = 0;
                                    nextstate = S0;
                              end
                  endcase
            end
endmodule

//simply shifts out data on clock edges - GPIO pins piReady and piSpiEn handle
when this occurs
module spiPi(input logic clk,
                              input logic [31:0] fftOut,
                              output logic sdo);

      logic [4:0] count = 5'b0;

      always_ff @(posedge clk)
            begin
                  sdo <= fftOut[count];
                  if(count == 5'd31) count <= 5'b0;
                  else count <= count + 5'b00001;
            end
endmodule

module minifft(input logic clk,
                        input logic fftStart,
                        input logic getData,
                        input logic giveData,
                        input logic [4:0] adrData,
                        input logic [15:0] micData,
                        output logic fftDone,
                        output logic [31:0] fftOut);

      //inputs/outputs for the controller
      logic [4:0] adrA, adrB;
      logic memwr, sclk;
      logic [3:0] twiddleAdr;

      //inputs/outputs for twiddleROM
      logic [15:0] twiddleFactor1, twiddleFactor2;

      //inputs/outputs for BFU
      logic [15:0] BFUdataAreal, BFUdataBreal, BFUdataAimag, BFUdataBimag;
      logic [15:0] memDataAreal, memDataBreal, memDataAimag, memDataBimag;

      minicontroller minifft_controller(clk, fftStart, fftDone, adrA, adrB,
memwr, twiddleAdr);
```

```
	minitwiddleROM minifft_twiddleROM(clk, twiddleAdr, twiddleFactor1, 
twiddleFactor2);

	minibutterfly minifft_butterfly(clk, twiddleFactor1, twiddleFactor2, 
	BFUdataAreal,BFUdataBreal, BFUdataAimag, BFUdataBimag, memDataAreal, 
memDataBreal, memDataAimag, memDataBimag);

	minimem minifft_mem(clk, memwr, getData, giveData, micData, adrData, 
adrA, adrB, memDataAreal, memDataBreal, memDataAimag, memDataBimag, 
	BFUdataAreal, BFUdataBreal, BFUdataAimag, BFUdataBimag, fftOut);
endmodule

 //need to generate control signals for the fft
module minicontroller(input logic clk,
					input logic fftStart,
					output logic fftDone,
					output logic [4:0] adrA,
					output logic [4:0] adrB,
					output logic memwr,
					output logic [3:0] twiddleAdr);

	logic [31:0] ja, jb, nextja, nextjb;
	logic [31:0] twiddle, nextTwiddle;
	logic nextmemwr, nextfftDone;
	logic [31:0] jcount, icount, nextjcount, nexticount;
	logic [31:0] N = 16'd16; //for 10 bit address: N = 16'd512
	logic [31:0] levels = 16'd5; //for 10 bit address: levels = 16'd10

	assign adrA = ja[4:0];
	assign adrB = jb[4:0];
	assign twiddleAdr = twiddle[3:0];

	typedef enum logic [2:0] {S0, S1, S2, S3, S4} statetype;
	statetype state, nextstate;

	//state advancing logic
	always_ff @(posedge clk)
		begin
			state <= nextstate;
			ja <= nextja;
			jb <= nextjb;
			twiddle <= nextTwiddle;
			jcount <= nextjcount;
			icount <= nexticount;
			memwr <= nextmemwr;
			fftDone <= nextfftDone;
		end
```

```
	always_comb
		begin
			nextstate = S0;
			case(state)
				S0: //wait for fftStart signal
					begin
						if(fftStart) nextstate = S1;
						else nextstate = S0;
						nextja = 32'b0;
						nextjb = 32'b0;
						nextTwiddle = 32'b0;
						nexticount = 32'b0;
						nextjcount = 32'b0;
						nextmemwr = 1'b0;
						nextfftDone = 1'b0;
					end
				S1: //j-incrementing for loop to generate addresses
					begin
						nextstate = S2;
						nextja = jcount<<1;
						nextjb = nextja+1;
						nextja =
((nextja<<icount)|(nextja>>(levels-icount)))&32'h1f; //10 bit mask: 3ff
						nextjb =
((nextjb<<icount)|(nextjb>>(levels-icount)))&32'h1f; //10 bit mask: 3ff
						nextTwiddle =
((32'hfffffff0>>icount)&32'hf)&jcount;
						nexticount = icount;
						nextjcount = jcount+32'b1;
						nextmemwr = 1'b0;
						nextfftDone = 1'b0;
					end
				S2: //write to mem from newly generated addresses
					begin
						if(jcount <(N)) nextstate = S1;
						else nextstate = S3;
						nextja = ja;
						nextjb = jb;
						nextTwiddle = twiddle;
						nexticount = icount;
						nextjcount = jcount;
						nextmemwr = 1'b1;
						nextfftDone = 1'b0;
					end
				S3: //i incrementing for loop
					begin
						if (icount<(levels-1)) nextstate = S1;
						else nextstate = S4;
```

```
                        nextja = ja;
                        nextjb = jb;
                        nextTwiddle = twiddle;
                        nexticount = icount+32'b1;
                        nextjcount = 0;
                        nextmemwr = 1'b0;
                        nextfftDone = 1'b0;
                    end
                S4: //finished state to assert fftDone
                    begin
                        nextstate = S0;
                        nextja = 32'b0;
                        nextjb = 32'b0;
                        nextTwiddle = 32'b0;
                        nexticount = 32'b0;
                        nextjcount = 32'b0;
                        nextmemwr = 1'b0;
                        nextfftDone = 1'b1;
                    end
            endcase
        end
endmodule

//lookup table for twiddle addresses
module minitwiddleROM(input logic clk,
                        input logic [3:0] twiddleAdr,
                        output logic [15:0] twiddleFactor1,
                        output logic [15:0] twiddleFactor2);
    always_comb
        case(twiddleAdr)
            4'b0000:
                begin
                    twiddleFactor1 = 16'h7fff;
                    twiddleFactor2 = 16'h0000;
                end
            4'b0001:
                begin
                    twiddleFactor1 = 16'h7d89;
                    twiddleFactor2 = 16'h18f9;
                end
            4'b0010:
                begin
                    twiddleFactor1 = 16'h7641;
                    twiddleFactor2 = 16'h30fb;
                end
            4'b0011:
                begin
                    twiddleFactor1 = 16'h6a6d;
```

```
                twiddleFactor2 = 16'h471c;
            end
        4'b0100:
            begin
                twiddleFactor1 = 16'h5a82;
                twiddleFactor2 = 16'h5a82;
            end
        4'b0101:
            begin
                twiddleFactor1 = 16'h471c;
                twiddleFactor2 = 16'h6a6d;
            end
        4'b0110:
            begin
                twiddleFactor1 = 16'h30fb;
                twiddleFactor2 = 16'h7641;
            end
        4'b0111:
            begin
                twiddleFactor1 = 16'h18f9;
                twiddleFactor2 = 16'h7d89;
            end
        4'b1000:
            begin
                twiddleFactor1 = 16'h0000;
                twiddleFactor2 = 16'h7fff;
            end
        4'b1001:
            begin
                twiddleFactor1 = 16'he707;
                twiddleFactor2 = 16'h7d89;
            end
        4'b1010:
            begin
                twiddleFactor1 = 16'hcf05;
                twiddleFactor2 = 16'h7641;
            end
        4'b1011:
            begin
                twiddleFactor1 = 16'hb8e4;
                twiddleFactor2 = 16'h6a6d;
            end
        4'b1100:
            begin
                twiddleFactor1 = 16'ha57e;
                twiddleFactor2 = 16'h5a82;
            end
        4'b1101:
```

```
                        begin
                              twiddleFactor1 = 16'h9593;
                              twiddleFactor2 = 16'h471c;
                        end
                  4'b1110:
                        begin
                              twiddleFactor1 = 16'h89bf;
                              twiddleFactor2 = 16'h30fb;
                        end
                  4'b1111:
                        begin
                              twiddleFactor1 = 16'h8277;
                              twiddleFactor2 = 16'h18f9;
                        end
            endcase
endmodule

//does all of the butterfly operation
module minibutterfly(input logic clk,
                                    input logic [15:0] twiddleFactor1,
                                    input logic [15:0] twiddleFactor2,
                                    input logic [15:0] BFUdataAreal,
                                    input logic [15:0] BFUdataBreal,
                                    input logic [15:0] BFUdataAimag,
                                    input logic [15:0] BFUdataBimag,
                                    output logic [15:0] memDataAreal,
                                    output logic [15:0] memDataBreal,
                                    output logic [15:0] memDataAimag,
                                    output logic [15:0] memDataBimag);
      logic [31:0] BrealPostMult, BimagPostMult;

      complexMult bfu_cmult(twiddleFactor1, twiddleFactor2, BFUdataBreal,
                                                       BFUdataBimag,
BrealPostMult, BimagPostMult);

      assign memDataAreal = (BFUdataAreal + BrealPostMult[30:15]); //bitshift
by 15 for twiddle floating point
      assign memDataAimag = (BFUdataAimag + BimagPostMult[30:15]);
      assign memDataBreal = (BFUdataAreal - BrealPostMult[30:15]);
      assign memDataBimag = (BFUdataAimag - BimagPostMult[30:15]);

endmodule


//multiplying two complex numbers
module complexMult(input logic [15:0] twiddleFactor1,
                                    input logic [15:0] twiddleFactor2,
                                    input logic [15:0] BFUdataBreal,
```

```
                                      input logic [15:0] BFUdataBimag,
                                      output logic [31:0] BrealPostMult,
                                      output logic [31:0] BimagPostMult);
      logic signed [15:0] stwiddleFactor1, stwiddleFactor2; //MUST DO SIGNED
MULTIPLICATION!!
      logic signed [15:0] sBFUdataBreal, sBFUdataBimag;
      logic signed [31:0] sBrealPostMult, sBimagPostMult;
      assign stwiddleFactor1 = twiddleFactor1;
      assign stwiddleFactor2 = twiddleFactor2;
      assign sBFUdataBreal = BFUdataBreal;
      assign sBFUdataBimag = BFUdataBimag;
      assign sBrealPostMult = sBFUdataBreal*stwiddleFactor1 -
sBFUdataBimag*stwiddleFactor2; //real part
      assign sBimagPostMult = sBFUdataBreal*stwiddleFactor2 +
sBFUdataBimag*stwiddleFactor1; //imaginary part
      assign BrealPostMult = sBrealPostMult;
      assign BimagPostMult = sBimagPostMult;
endmodule

//double-addressed memory block
module minimem(input logic clk,
                        input logic memwr,
                        input logic getData,
                        input logic giveData,
                        input logic [15:0] micData,
                        input logic [4:0] adrData,
                        input logic [4:0] adrA,
                        input logic [4:0] adrB,
                        input logic [15:0] memDataAreal,
                        input logic [15:0] memDataBreal,
                        input logic [15:0] memDataAimag,
                        input logic [15:0] memDataBimag,
                        output logic [15:0] BFUdataAreal,
                        output logic [15:0] BFUdataBreal,
                        output logic [15:0] BFUdataAimag,
                        output logic [15:0] BFUdataBimag,
                        output logic [31:0] fftOut);
      logic [31:0] mem[31:0]; //bits 0-15 are real, 16-31 are imaginary
      logic [4:0] adrDatarev;
      assign adrDatarev = {adrData[0], adrData[1], adrData[2], adrData[3],
adrData[4]};
      always @(posedge clk)
            begin
                  if(getData) //loading values into memory
                        begin
                              mem[adrDatarev][15:0] <= micData;
                              mem[adrDatarev][31:16] <= 16'b0;
                              fftOut <= 31'b0;
```

```
                    BFUdataAreal <= 16'b0;
                    BFUdataAimag <= 16'b0;
                    BFUdataBreal <= 16'b0;
                    BFUdataBimag <= 16'b0;
                end
            else if(giveData) //loading values out of memory
                begin
                    fftOut <= mem[adrData];
                    BFUdataAreal <= 16'b0;
                    BFUdataAimag <= 16'b0;
                    BFUdataBreal <= 16'b0;
                    BFUdataBimag <= 16'b0;
                end
            else //using memory for fft
                begin
                    fftOut <= 31'b0;
                    BFUdataAreal <= mem[adrA][15:0];
                    BFUdataAimag <= mem[adrA][31:16];
                    BFUdataBreal <= mem[adrB][15:0];
                    BFUdataBimag <= mem[adrB][31:16];
                    if(memwr)
                        begin
                            mem[adrA][15:0] <= memDataAreal;
                            mem[adrA][31:16] <= memDataAimag;

                            mem[adrB][15:0] <= memDataBreal;
                            mem[adrB][31:16] <= memDataBimag;
                        end
                end
        end
endmodule
```