Real-Time Audio Spectrum Analyzer – Final Report

Weston Miller & Ingrid Tsang

Abstract

The goal of this project is to build a real-time audio spectrum analyzer. The system performs Fourier analysis on a real-time audio signal from a microphone using the FPGA and displays the spectrum of the audio signal on an LCD. The microcontroller is responsible for sampling data from the microphone, sending and receiving data from the FPGA, and updating the spectrum on the LCD.

Introduction

Growing up using applications such as Windows Media Player and QuickTime, there would be various graphs on the side providing information about the audio signal being captured by the application. Captivated by these curious moving lines as young children, we would play different sounds with the computer and make noises to see how the graphs changed. As we entered college and began taking engineering courses, we learnt that these curious moving lines were in fact outputs of Fourier analysis, which helps to determine the frequency composition of a given signal. For audio signals, this allows us to perform tasks such as visualizing different pitches present and their intensities, performing filtering to remove noise from a signal, and audio recognition. Ignited by this interest, we decided to recreate the frequency versus amplitude graphs we saw as children for this project, where we will build a real-time audio spectrum analyzer.

Audio data is first sampled from the electret microphone through the microcontroller's analog-to-digital converter (ADC), with the sampling frequency controlled by a timer on the microcontroller. The audio data is stored in the microcontroller's memory from the ADC using direct memory access (DMA) until all the samples are collected. The microcontroller then sends the audio data through SPI to the FPGA, which implements a FFT hardware accelerator. After the FPGA calculates the FFT, the output is sent through SPI back to the microcontroller. The microcontroller takes the FFT output and converts it to a pixel array, which is sent through SPI to the graphic LCD to display the frequency versus amplitude graph on the display. The system block diagram is shown in Figure 1.

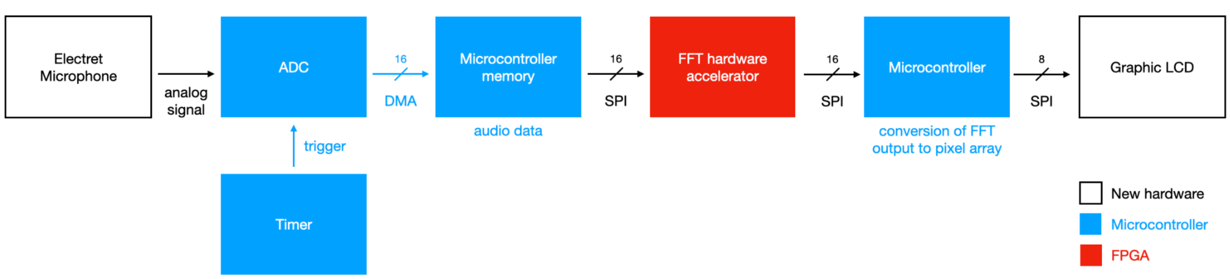


Figure 1: System block diagram

Schematics

The schematic diagram for the breadboarded circuit is shown in Figure 2, and the pin assignments for the FPGA are shown in Table 1.

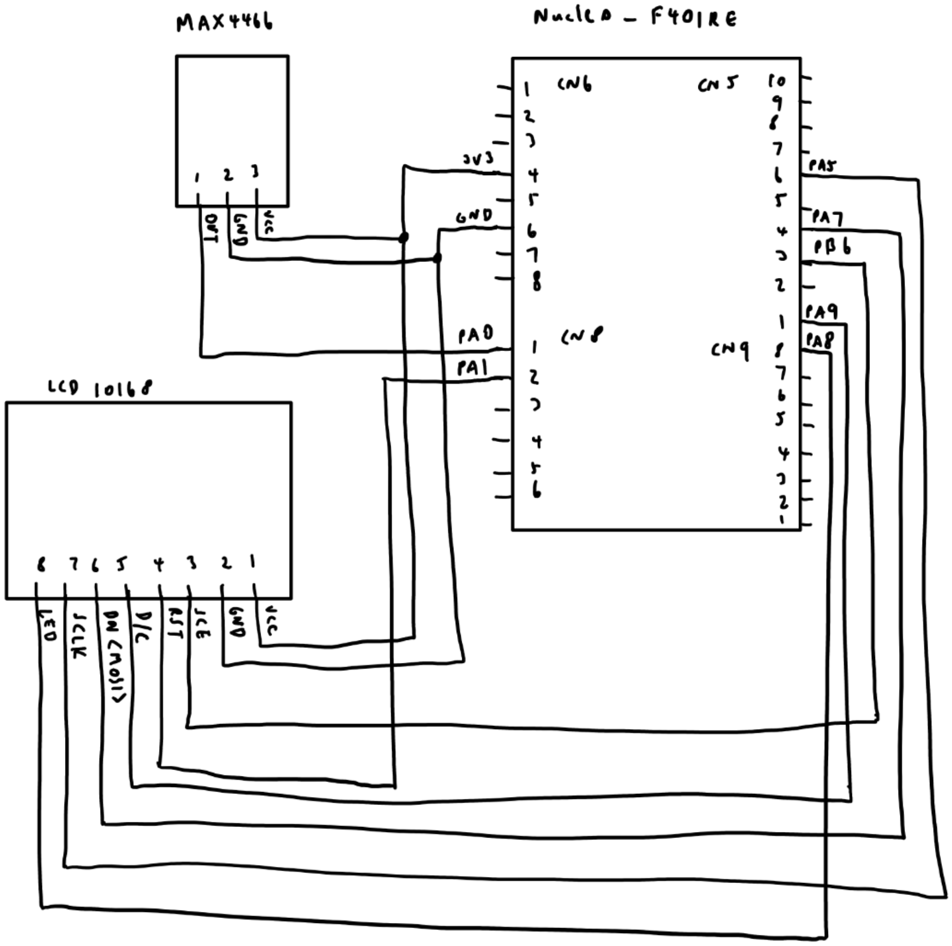


Figure 2: Schematic of breadboard circuit

Node Name	Direction	Location
CLK	Input	PIN_H6
DONE	Output	PIN_H5
LOAD	Input	PIN_K11
SCK	Input	PIN_H4
SDI	Input	PIN_J1
SDO	Output	PIN_J2

Table 1: Pin Assignments for the FPGA

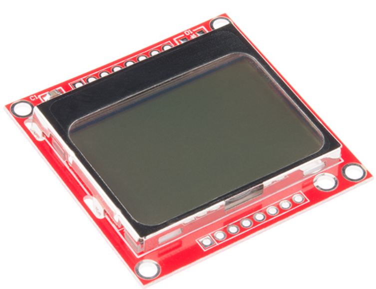
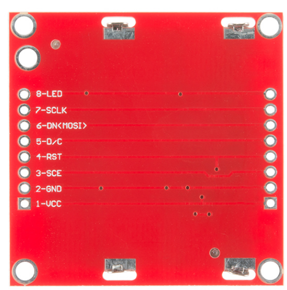
New Hardware
There are two pieces of new hardware used in the system: the electret microphone and the graphic LCD.

Electret Microphone
The breakout (as shown in Figure 3) consists of a MAX4466 amplifier with an electret microphone, and the gain can be adjusted using a small trimmer pot on the back of the breakout.

Figure 3: Electret microphone with amplifier (ADA 1063) [4]

The electret microphone is used to capture (analog) audio signals as the input to the system, which is sent to the microcontroller for further processing. Interfacing with the microphone was performed using the microcontroller's ADC to convert the analog signal from the microphone to a digital signal that can be processed by the rest of the system.

Graphic LCD
The 48 by 84 pixel graphic LCD, which comes from an old Nokia 5110, is mounted on a board and with a PCD8544 controller. Figures 4 and 5 show the graphic LCD and the pinouts.

Figures 4-5: Graphic LCD with pinouts on the back of the board (LCD 10168) [6]

The LCD is used to display the spectrum of the input audio signal as a frequency versus amplitude graph. The dominant frequency in the audio signal is also displayed on the side of the graph (as shown in Figures 16-19).

The microcontroller (controller device) communicates with the LCD (peripheral device) through SPI. The D/C pin (pin 5 in Figure 5) allows the microcontroller to toggle between communicating data and commands. To begin interfacing with the LCD, the first step is to reset the LCD using the RST pin (pin 4 in Figure 5) and then configure the settings by sending commands to the LCD, including choosing the instruction set, the addressing mode and the display mode. For this system, vertical addressing was selected, as the main purpose of the LCD is to display the frequency versus amplitude graph as a series of bars, thus it is more logical to populate the pixel array by column. The microcontroller then sends data to the LCD 8 bits at a time, with each bit representing one pixel being turned on or off. It is possible to configure specific 8-bit blocks of pixels, however this was not needed for this system as for each new FFT output, the entire display is updated.

Microcontroller Design

The microcontroller in the system acts as a coordinating device — responsible for sampling data from the microphone, sending and receiving data from the FPGA, and updating the spectrum on the LCD — as shown in blue in Figure 1.

The first step for the microcontroller is to set up the initial configurations, including for the GPIO pins, the SPI, the direct memory access (DMA), the analog-to-digital converter (ADC) and the graphic LCD. After that, the microcontroller enters an infinite loop that repeatedly executes a series of functions to sample and store the audio data, send the audio data to the FPGA, receive the FFT output from the FPGA, convert the FFT output to the pixel array, and send the pixel array to the graphic LCD.

To sample the audio data from the microphone, the microcontroller's ADC is used. The sampling frequency is controlled by one of the peripheral timers. To collect the audio data samples, an ADC conversion is performed every time the timer's counter reaches the desired threshold. The ADC is configured to be in DMA mode, so once an ADC conversion is completed, the data in the ADC's data register is stored in the specified location (the `audioData` array). For this system, the sampling frequency was chosen to be 3200 Hz (but this can be easily changed) and 32 samples are collected, due to the design of the FFT (see the "FPGA Design" section). Therefore, the system can display audio signals up to 1600 Hz (the Nyquist frequency is $\frac{3200}{2} = 1600$ Hz) with a bin width of 100 Hz (each bin is $\frac{1}{32/3200} = 100$ Hz wide).

The next step in the infinite loop for the microcontroller is to send and receive data from the FPGA, which performs the FFT. The microcontroller and FPGA communicate via SPI, as per the transmission interface specified by the FFT SPI module (see the "FPGA Design" section). The LOAD pin is used to communicate when data for the FFT is being between the microcontroller and the FPGA, and the DONE pin is used to communicate when the FFT is complete and the data is ready to be sent from the FPGA to the microcontroller. The output from the FFT is a series of complex numbers in Q15 (further explained in the "FPGA Design" section). Therefore, before further processing the microcontroller converts the real and imaginary components of each complex number into doubles and then computes the magnitudes of each of these complex numbers to determine the amplitudes at each frequency.

Lastly, the microcontroller also creates the pixel array, which is sent through SPI to the graphic LCD. As discussed in the "New Hardware" section, vertical addressing is used for this system (i.e. the pixels are sent to the LCD by column). The function first converts the FFT output to the frequency versus amplitude bar graph. Thus for each frequency, the pixel array is populated based on the amplitude corresponding to the frequency, normalized by a fixed amplitude determined empirically (and can be easily changed). The second part of the function is responsible for displaying the dominant frequency as text on the right side of the bar graph. This is done by storing the digits 0-9 and characters 'H' and 'z' as arrays of pixels, identifying the dominant frequency (that with the highest amplitude) from the FFT output, and then for each digit or character, adding each column of pixels iteratively to the pixel array. Finally, the rest of the pixel array is populated with 0s to overwrite the pixels from the previous array loaded onto the LCD.

A bare-metal design was used for this system, performing the SPI communication with the FPGA and LCD in series (instead of in parallel). This resulted in reduced complexity, as interrupts do not have to be used. Through testing, we also found that even with this design, there was no perceivable lag between the microphone capturing the audio signal and the output on the display, thus further justifying this design choice. Additionally, in the implementation of the microcontroller operations, each function was kept modular, which allowed for smoother unit testing of the system to identify parts of the system that needed to be debugged. For example, to unit test the conversion of the FFT output to the pixel array and the SPI communication with the LCD (especially before the FFT hardware accelerator was complete), test FFT output was used and the operation of this module was verified by comparing the output on the display to the expected output generated by a Python program.

FPGA Design

The FFT hardware accelerator consists of two modules: the “FFT core” which is responsible for performing the FFT on loaded data and the “SPI module” which handles the SPI communication with the microcontroller. The top-level block diagram of the 32-point FFT module is shown in Figure 6.

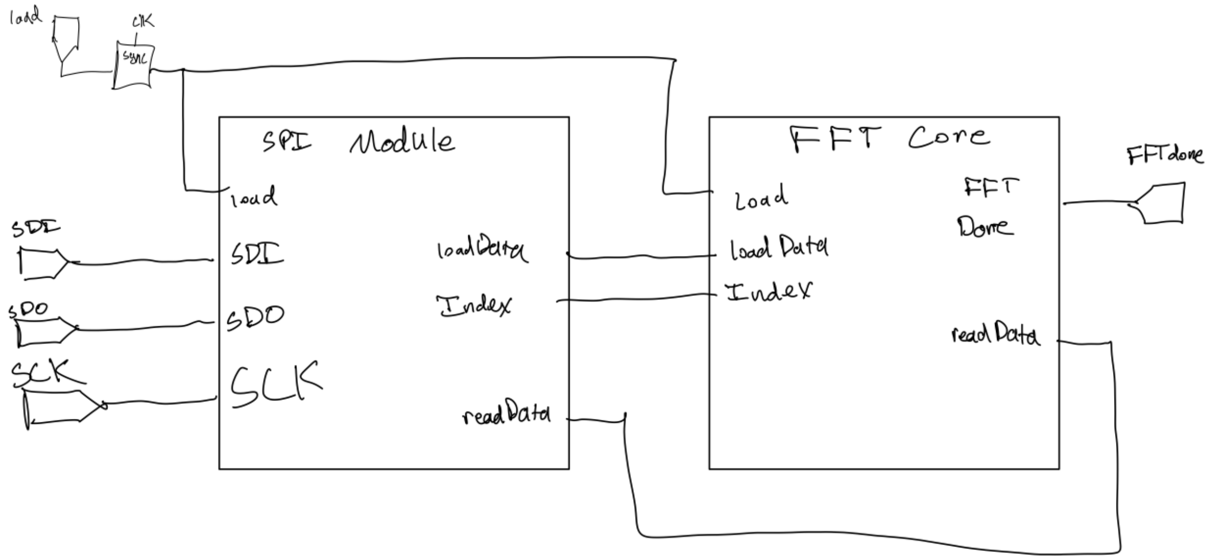


Figure 6: Top-level block diagram of FFT module

FFT Core

At its root, the FFT is a series of “butterfly” operations. The individual butterfly operation is described by the following pair of equations:

$$A' = A + \omega^{k} * B \qquad (1)$$

$$B' = A + \omega^{k} * B \qquad (2)$$

Where ω^{k} is the corresponding “root of unity”. A schematic of an 8-point FFT is shown in Figure 7. For an 8-point FFT, there are 12 butterfly operations that need to be performed ($\frac{N}{2}\ log_{2}N$). It is useful to identify the specific butterfly operation in terms of its *i* and *j* index as identified in Figure 7.[1] Importantly, each butterfly only depends on the results from the previous level. This means that one can compute each butterfly sequentially if the outputs are stored. Moreover, the order that one performs the butterfly operations at each level does not matter as long the results are stored and completed before the butterfly of the next level.

Given this insight, a relatively straightforward way to implement the FFT in hardware is to have a butterfly module that performs the butterfly operation, a read-write data structure to store the inputs/outputs to each butterfly operation, a read-only data structure that stores the “roots of

[1] For a more complete description and derivation of the FFT, see [2].

unity", and an "address generation unit" that generates the addresses to retrieve and store values from/in the data structures. This implementation of the FFT follows this general implementation, as shown in the top-level schematic in Figure 8.

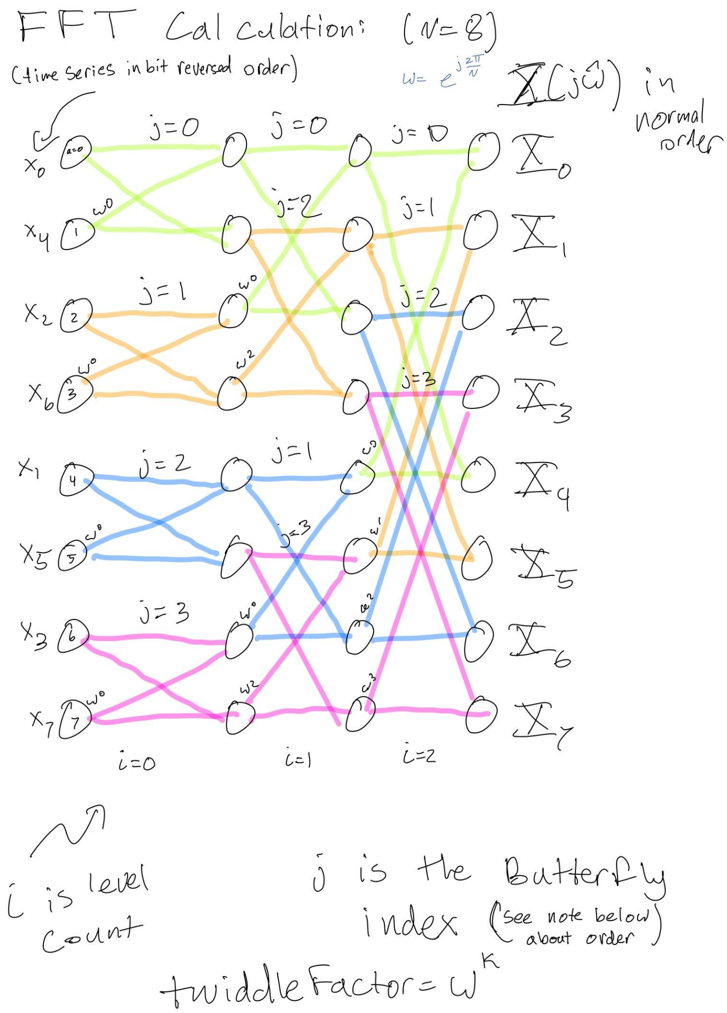


Figure 7: Schematic of the butterfly operation. Each cross represents a butterfly operation.

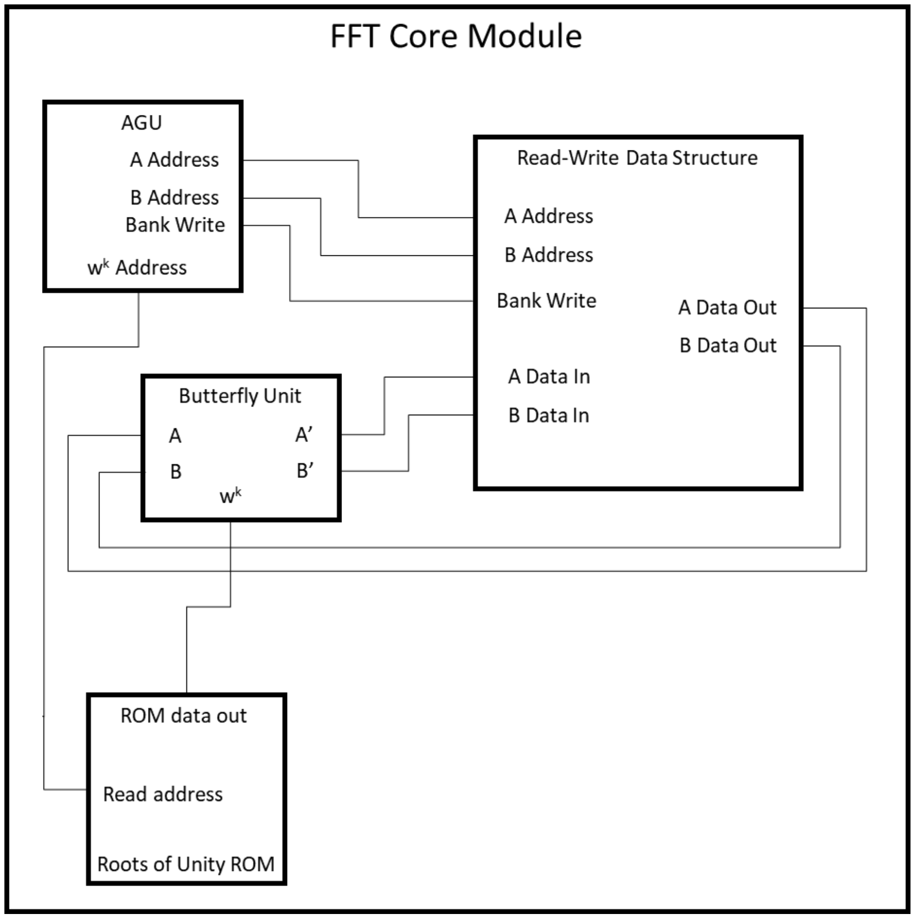


Figure 8: Top-level schematic of the FFT core

Butterfly Unit

The operation of the butterfly unit is specified by (1) and (2) above. This arithmetic was specified combinationally. The resulting block diagram is shown in Figure 9. There are a couple of subtleties worth noting regarding the implementation of the butterfly unit. Firstly, A, B, and w^k are generally complex. This means that each input must have a real and imaginary part, and multiplication requires four individual multipliers. Secondly, all of the signals are represented using Q15, similar to Slade in [1]. Q15 is a desirable number format because the product of two Q15 numbers is between -1 and 1. Furthermore, it is easy to represent the product as a Q15 number by simply shifting out the least significant bits of the product.

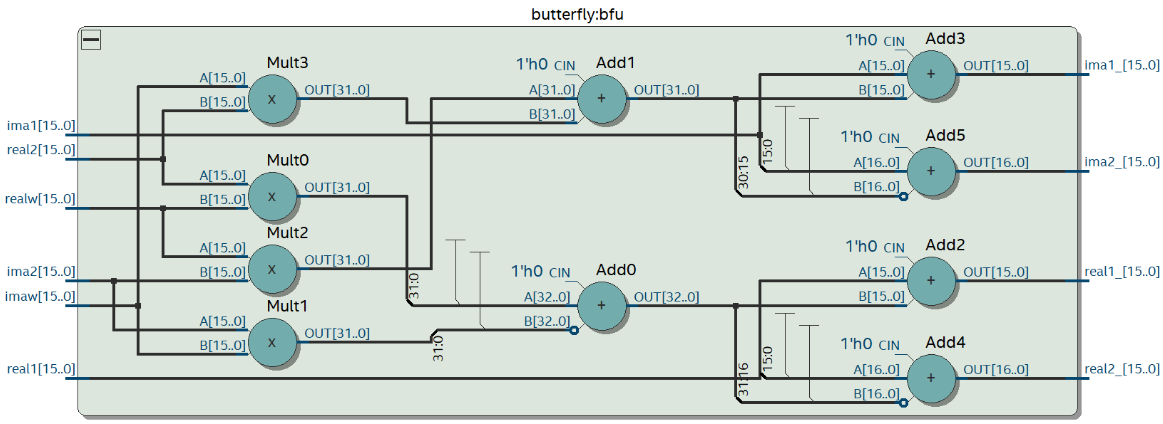


Figure 9: Schematic of butterfly unit

Read-Write Data Structure

Figure 10 shows a schematic of the memory structure. Because memory cannot be read from and written to on the same clock cycle, there are two memory banks and the module alternates reading from one and writing to the other at every FFT level. Theoretically, each memory bank consists of two (one for real/imaginary data each) dual-port RAM blocks. However, the current implementation uses registers instead of RAM, as the read address line of the RAM is registered, which would add a single clock cycle delay between the read addresses and the data out delay. Figure 11 shows the block diagram of a memory bank.

The time-series data is loaded into data bank 1 in bit reversed order, then the transformed data will end up in data bank zero in normal order (address 0 corresponds to X(0), etc.). Data is loaded into data bank 1 by asserting the “load” signal, and on the positive clock edge, the data on the Load Data lines will be loaded into the address at the bit reverse of the index line.

Roots of Unity ROM

The roots of unity ROM is a 16 x 32 bit read-only memory that contains all of the roots of unity (w^0 through w^{15}) in Q15 format. The most significant bits correspond to the real part and the least significant bits correspond to the imaginary value. The address is the power of the root of unity (e.g. w^0 will be stored in address zero).

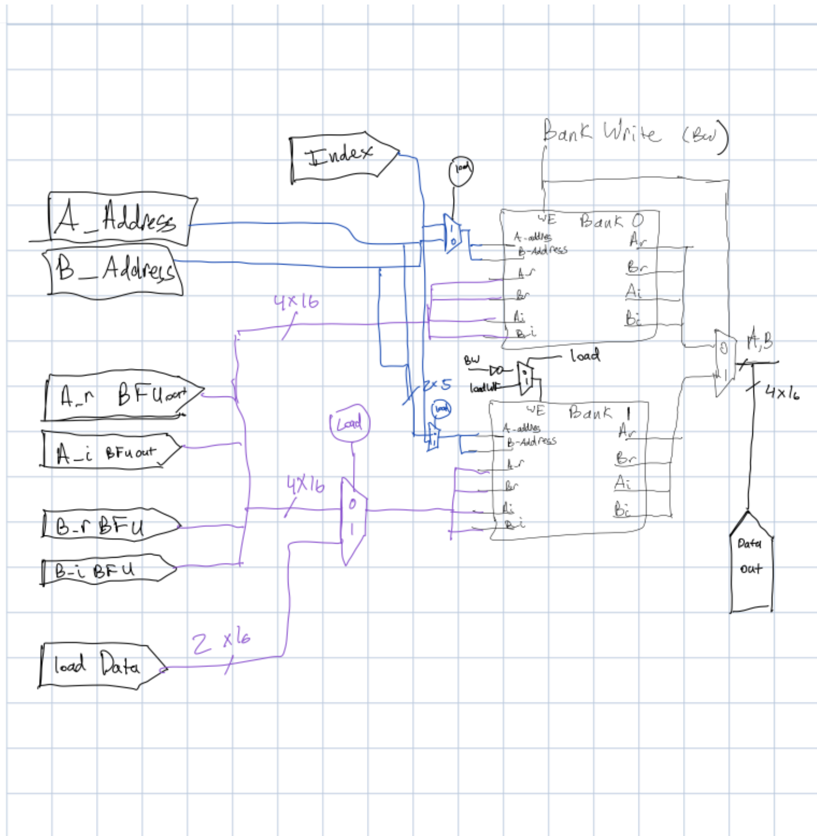


Figure 10: Schematic of the memory structure

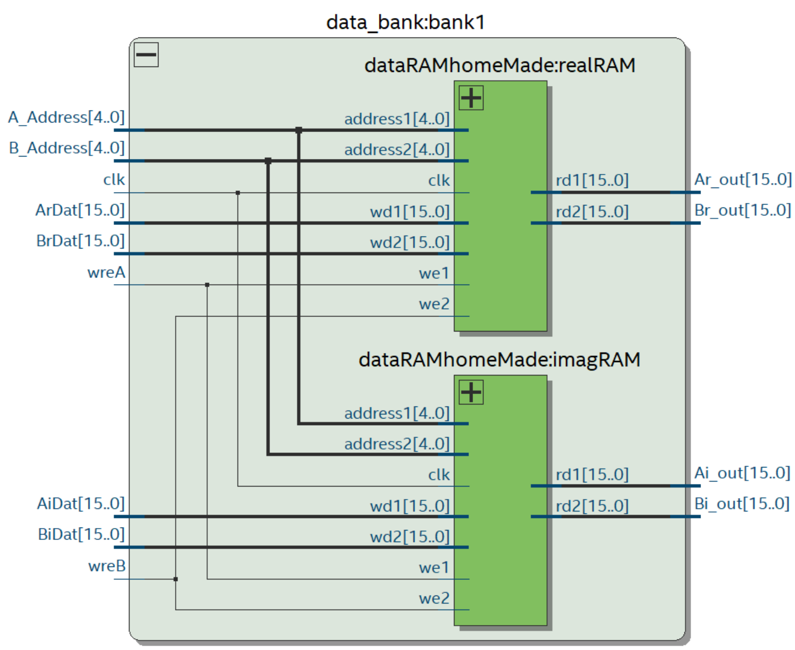


Figure 11: Schematic of a data bank

Address Generation Unit

The general strategy for address generation is outlined by Cohen in [3]. In short, if we do the butterfly operations in the order described by Cohen, there is a relatively simple relationship between the level (i) and *j* index and the addresses for A and B:

$$\text{A\_address} = \text{Rotate}_5(2*j,\ i)$$
$$\text{B\_address} = \text{Rotate}_5(2*j+1,\ i)$$

Furthermore, the correct root of unity address for butterfly (i,j) is found by zeroing out the lowest (4 - i) bits of j :

$$K = j\ \&\ [5b11111 << (5 - i - 1)]$$

We can keep track of *i* and *j* for the 32-point FFT with a single 7-bit counter. The lowest 4 bits are the *j* counter and the top 3 bits are the *i* counter. Figure 12 shows a schematic of the address generation unit.

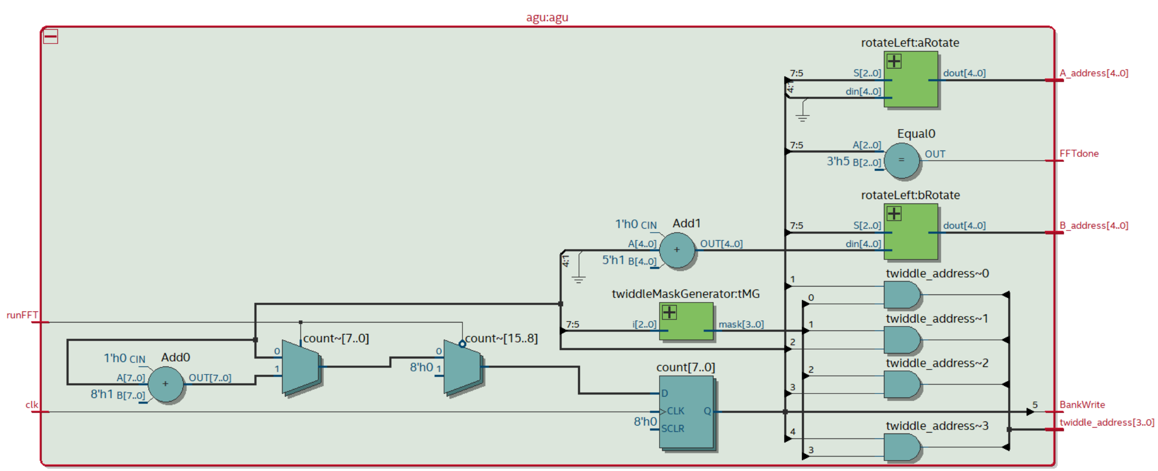


Figure 12: Schematic of the address generation unit. The twiddleMaskGenerator and the RotateLeft modules are combinational logic blocks.

FFT Core Function

To validate the functionality of the FFT core, a 32-point FFT was performed on the same signal (an impulse at n = 3) using Matlab's FFT function and using a ModelSim simulation of the FFT Core module. Figure 13 shows the results of these simulations. There is good agreement between Matlab FFT and the transform generated by the FFT Core Module.

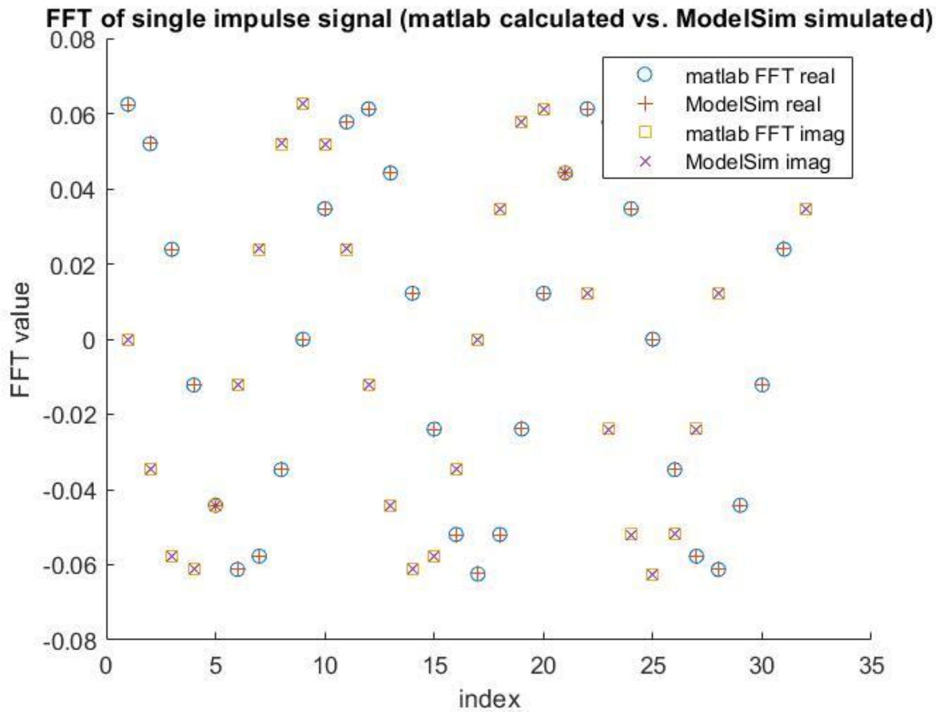


Figure 13: Comparison of FFT of test signal using Matlab and ModelSim

FFT Core Module Interface

The inputs and outputs of the FFT core module are:

- Inputs:
    - Clk - system clock that drives all synchronous modules
    - 2x loadData (16 bit) busses (1_r, 1_i)
    - 1x Index (5'bit)
    - Load - control signal driven high to load/read data from FFTcore
    - runFFT - control signal driven high to run FFT after load/read is completed
- Outputs:
    - 2x readData (16 bit) busses (1_r, 1_i,)
    - FFTdone - signal goes high when FFT has finished

The FFT can either be in a done state or a running state. When the FFT is in a done state, the results of the previous transform can be read out simultaneously with the writing in of new data values. To do this, load must be asserted. When load is asserted, data is loaded into the data bank on the positive clock edge; the data on the Load Data lines will be loaded into the addresses on the Load Addresses lines. Simultaneously, the data at the address on the read line will appear on the data lines to be readData lines. Importantly, there are two read/load addresses that are loaded/read simultaneously. This means that the data must be loaded in and read out in pairs. To have the FFT core run after loading is complete, drive load low and runFFT high. RunFFT must be driven high continuously until the FFTdone signal is asserted.

SPI Module

To load data into the FFT Core module, the SPI module must drive the index and loadData lines. To read data from the FFT core, the SPI module must be able to take in the read data. Figures 14 and 15 show block diagrams of the SPI module, which is built around a 32-bit shift register. The shift register shifts on the negative clock edge, and the SDI is strobed on the rising edge. This corresponds to a clock polarity of 0 and a clock phase of 0. The control signal generator is responsible for capturing the index, loading the read data into the shift register, and then capturing the load data after the read data is shifted out, as shown in Figure 15. The control signal generator is built around a counter, which keeps track of the number of negative clock edges so that the SPI module can correctly interpret the SPI communication.

The SPI module specifies the following 40-bit transmission protocol:

- Microcontroller sends: {5'b index, 3'b0, MSB real data, LSB real data, MSB imaginary data, LSB imaginary data}
- Microcontroller receives: {8'b X, MSB real data, LSB real data, MSB imaginary data, LSB imaginary data}

Sending the index first allows time for the data being read out of the FFT core to arrive on the Read Data line before it needs to be loaded into the shift register. Care was taken to ensure that the index and load data signals are properly synchronized so they do not cause metastability problems. For the index, this was achieved by having an “async” capture register that is clocked by sck and a “sync” index register that is clocked by clk. The enable for the synchronized clk index register only goes high after the “async” register has been settled.

To achieve synchronization for the Load Data line, the enable for the Load Data capture register is only high for one clock cycle when a synchronized count signal reads 40. However, this strategy only works if the frequency of sck is less than ½ of the frequency of clk. Because we used the 12 MHz oscillator on the FPGA as the system clock for the FFT module, this means the maximum sck frequency would be 6MHz. In practice, the sck frequency is set significantly lower at ~1.3 MHz.

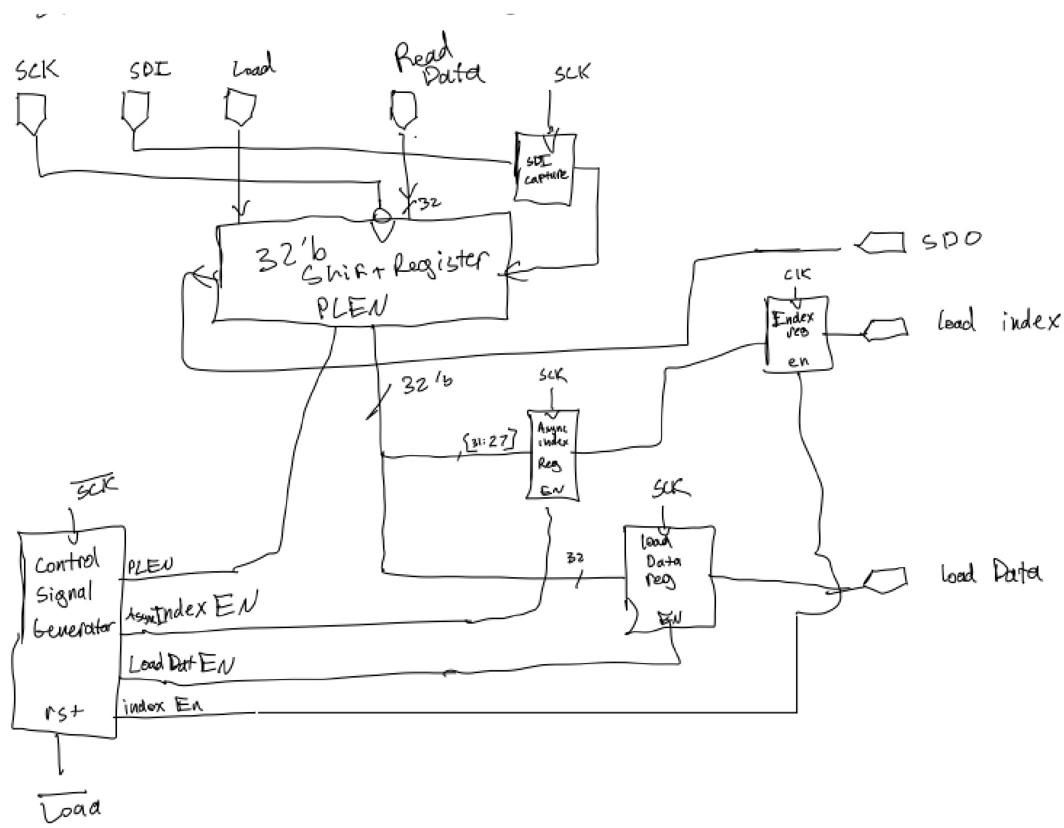


Figure 14: Block diagram of FFT SPI module

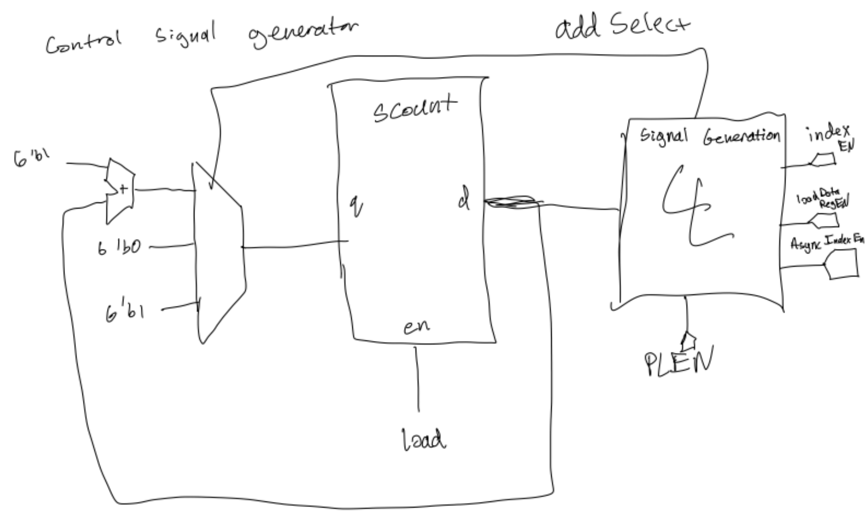
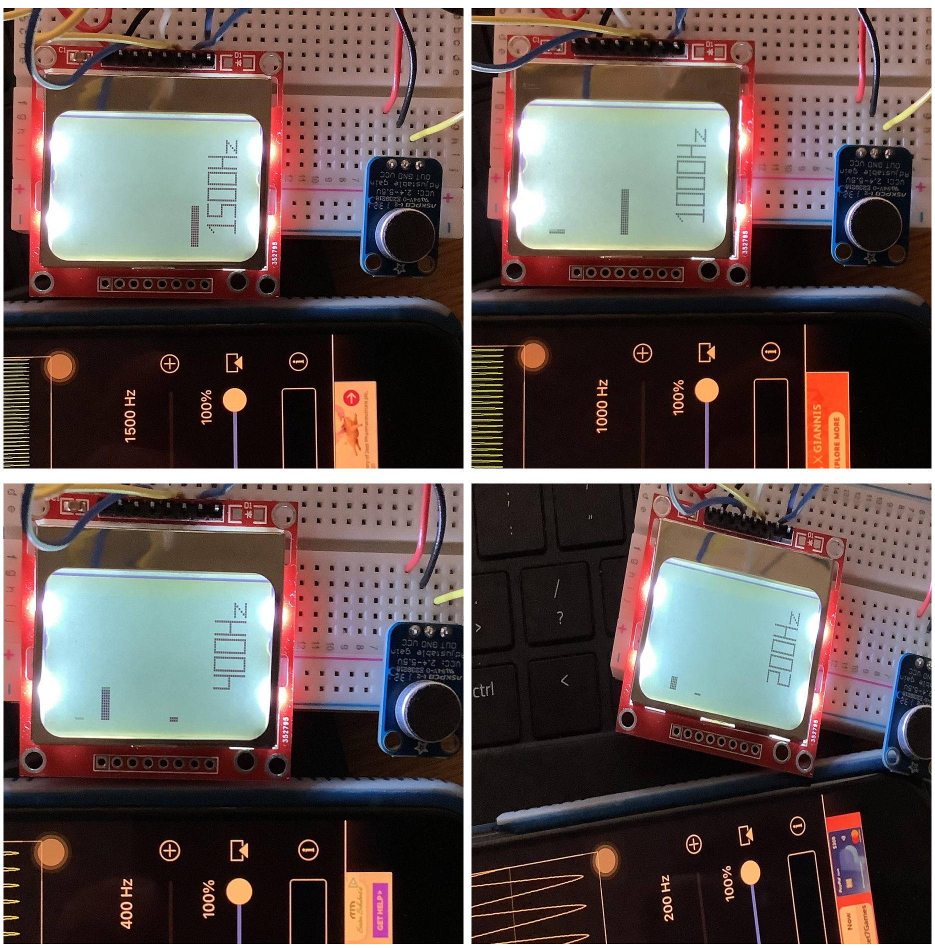


Figure 15: Block diagram of FFT SPI module control signal generator

Results
The system was able to successfully meet all the specifications outlined in the project proposal:

- Sample audio from a microphone
- Perform Fourier analysis on the audio signal using the FPGA
- Display the spectrum of the audio signal on a graphic LCD

This behavior was verified through extensive testing, as shown in Figures 16-19 and videos linked below. We played different notes in succession, and observed that the frequency versus amplitude graph and dominant frequency displayed changed with the music, with no perceptible lag between the audio signal and the display. The graph also clearly reflects different sound volumes, such that when the music is played more softly, the amplitudes are correspondingly smaller on the graph. Additionally, we also played various pure tones, and observed that there was a clear dominant frequency (more than one in the case where the frequency did not fit exactly into one of the 100 Hz bins) that corresponded to the frequency of the tone.



Figures 16-19: The spectrum analyzer in operation

Video Links

- Playing “Drop the Game” (link)
- Playing a series of pure tones (link)

References

[1] The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation (https://web.mit.edu/6.111/www/f2017/handouts/FFTtutorial121102.pdf)

[2] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, “Fast Fourier Transform,” in Numerical Recipes in C, Cambridge Univ. Pr., 1992.

[3] D. Cohen, "Simplified control of FFT hardware," in IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 24, no. 6, pp. 577-579, December 1976, doi: 10.1109/TASSP.1976.1162854.

[4] Electret Microphone Amplifier - MAX4466 with Adjustable Gain (https://www.adafruit.com/product/1063)

[5] MAX4466 Low-Noise Microphone Amp Datasheet (https://cdn-shop.adafruit.com/datasheets/MAX4465-MAX4469.pdf)

[6] Graphic LCD 84x48 - Nokia 5110 (https://www.sparkfun.com/products/10168)

[7] Graphic LCD 84x48 - Nokia 5110 Datasheet (https://www.sparkfun.com/datasheets/LCD/Monochrome/Nokia5110.pdf)

Bill of Materials

Item	**Vendor**	**Part Number**	**Quantity**	**Unit Price**	**Total Price**
MAX1000	Trenz Electronic	TEI0001-03-08-C8	1	$26.66	$26.66
Nucleo-F401RE	Mouser		1	$13.83	$13.83
Electret Microphone Amplifier - MAX4466 with Adjustable Gain	Adafruit	ADA1063	2[2]	(1) $6.95 (1) $9.68	$16.63
Graphic LCD 84x48 - Nokia 5110	SparkFun	LCD-10168	1	$7.95	$7.95
				Total	$65.07 + shipping

[2] Two microphones were purchased due to extenuating circumstances. Only one microphone is used in the system.

Appendix A: FPGA Code
FFTcore and top level module:

```
/////////////////////////////////////////////
// spectrum_analyzer.sv
// HMC E155
/////////////////////////////////////////////

`timescale 1ns/1ns

////////////// SPECTRUM ANALYZER
////////////////////////////////////////////////////////

/////////////////////////////////////////////
// FFTcore testbench
/////////////////////////////////////////////
module FFTcoreTestbench();
       logic clk;
       logic runFFT;
       logic FFTdone;
       logic rst;
       logic load;
       logic [4:0] loadAddress, readAddress;
       logic signed [15:0] loadData_r, loadData_i;
       logic signed [15:0] readData_r, readData_i;
       logic [5:0] i;
       logic signed [15:0] realData [31:0];
       logic signed [15:0] imaginaryData [31:0];
       logic finished; // Variable to keep track of if the FFTdone has gone high.

   // Device under test
   FFTcore dut(clk, runFFT, load, loadAddress, readAddress, loadData_r, loadData_i,
FFTdone, readData_r, readData_i);

   // Initialize signals
   initial
       begin
           rst = 1'b1;
           runFFT = 1'b0;
           clk = 1'b0; #5;
           clk = 1'b1; #5;
           runFFT = 1'b1;
           rst = 1'b0;
```

```
            load = 1'b1;
            clk = 1'b0; #5;
            clk = 1'b1; #5;
            runFFT = 1'b0;
        end

    // clk generator
    initial
        forever begin
            clk = 1'b0; #5;
            clk = 1'b1; #5;
        end

    // Set up test case (right now it is an impulse)
    initial
        begin
            for(logic [5:0] j = 0; j < 32; j++) begin
                // realData[j] = {10'b0, j[0], j[1], j[2], j[3], j[4]} ;
                // imaginaryData[j] = {10'b0, j[0], j[1], j[2], j[3], j[4]};
                if( j==3) begin
                    realData[j] <= 16'h07ff;
                    imaginaryData[j] <= 16'h0000;
                    end
                else begin
                    realData[j] <= 16'h0000;        // Should be hfa00 for square wave
                    imaginaryData[j] <= 16'h0000;
                    end
            end
            i = 1'b0;                               // Initialize i to zero
            finished = 1'b0;                        // Assign finished to 0 to start
        end

    // Start running AGU
    always @(posedge clk) begin
        if (i < 32 && ~FFTdone) begin
                readAddress = i[4:0];
                loadAddress = {readAddress[0],readAddress[1], readAddress[2],
readAddress[3], readAddress[4]};
                loadData_r = realData[readAddress];
                loadData_i = imaginaryData[readAddress];
                i = i + 1;
            end
```

```
      else if (i == 32 && ~FFTdone) begin
              runFFT = 1'b1;
              load = 1'b0;
              i = i + 1;
          end
     else if (FFTdone & ~finished) begin
              runFFT = 1'b0;
              load = 1'b1;
              i = 1'b0;
              finished = 1'b1;
              readAddress = i[4:0];
              realData[readAddress] = readData_r;
              imaginaryData[readAddress] = readData_i;
              i = i+1;
          end
      else if (finished && i < 32) begin
              readAddress = i[4:0];
              #2;
              realData[readAddress] = readData_r;
              imaginaryData[readAddress] = readData_i;
              i = i + 1;
          end
      else if (finished && i > 31) begin
          $stop();
      end
   end
endmodule

/////////////////////////////////////////////
// spectrum_analyzer
//   Top level module of spectrum analyzer
//
//   As of now this is just a top level module
//   that contains all of the components to enable
//   using the netlist viewer.
//
//  PINS:
//          clk: (internal 12 MHz oscilator) H6
//          sck: (Connect to SPI1 sck of the MCU) PA5_H4
//          sdi: (Connect to SPI1 sdo of the MCU) PA7_J2
//          sdo: (Connect to SPI1 sdi of the MCU) PA6_J1
//          load: GPIOB pin 5 (K11)
```

```
//          FFTdone: GPIOA pin 3 (H5)
//
//
/////////////////////////////////////////////
module spectrum_analyzer(input  logic clk,
                        input  logic sck,
                        input  logic sdi,
                        output logic sdo,
                        input  logic load,
                        output logic FFTdone);
   logic loadDone;
   logic [4:0] loadAddress, readAddress;
   logic signed [15:0] loadData_r, loadData_i;
   logic signed [15:0] readData_r, readData_i;
   logic [31:0] loadData, readData;
   logic loadHalfSync, loadSync;

   // Synchronize the load signal that gets passed into the spi and core modules
   always_ff @(posedge clk) begin
           loadHalfSync <= load;
           loadSync <= loadHalfSync;
       end

   FFTcore core(clk, loadSync, loadAddress, readAddress, loadData_r, loadData_i,
FFTdone, readData_r, readData_i);
   onePoint_spi spi(clk, sck, sdi, loadSync, readData, sdo, readAddress, loadData);

   // Assign load/read data
   assign loadAddress = {readAddress[0], readAddress[1], readAddress[2],
readAddress[3], readAddress[4]};
   assign loadData_r = loadData[31:16];
   assign loadData_i = loadData[15:0];
   assign readData = {readData_r, readData_i};

endmodule

/////////////////////////////////////////////////////////////////////////////////////
/////
// FFT core
// Written by Weston Miller (wmiller@hmc.edu) and Ingrid Tsang (itsang@hmc.edu)
// In order to load data in, assert load
```

```
// On the next clock cycle, the data on the loadData lines will be loaded into the
loadAddress location
// Data should be provided in bit reversed order.
// Also: when load is asserted, the data from the previous run of the FFT will appear
on the readData lines
//
// start FFT, assert runFFT
////////////////////////////////////////////////////////////////////////////////
/////////////////////
module FFTcore(input  logic               clk,
               input  logic               load,
               input  logic        [4:0]  loadAddress, readAddress,
               input  logic signed [15:0] loadData_r, loadData_i,
               output logic               FFTdone,
               output logic signed [15:0] readData_r, readData_i);
   logic BankWrite;
   logic [4:0]  A_address, B_address;
   logic [3:0]  twiddle_address;
   logic wreB1, wreB0;

   logic signed [15:0] realw, imaw;
   logic signed [15:0] real1, imag1, real2, imag2, real1_, imag1_, real2_, imag2_;
   logic signed [15:0] real1bank1, imag1bank1, real2bank1, imag2bank1, real1bank0,
imag1bank0, real2bank0, imag2bank0;
   logic signed [15:0] bank1data1_r, bank1data1_i, bank1data2_r, bank1data2_i;

   twiddle_rom trom(twiddle_address, realw, imaw);
   butterfly bfu(real1, imag1, real2, imag2, realw, imaw, real1_, imag1_, real2_,
imag2_);
   agu agu(clk, runFFT, A_address, B_address, twiddle_address, FFTdone, BankWrite);

   // write enables for bank 0 and bank 1
   assign wreB1A = BankWrite || load;
   assign wreB1B = BankWrite && ~load;
   assign wreB0A = (~BankWrite && ~load);
   assign wreB0B = wreB0A;

   // Bank addresses for bank 0 and bank 1 (mux to select between load and AGU
addresses)
   logic [4:0] bank0Address_1, bank0Address_2, bank1Address_1, bank1Address_2;

   assign bank0Address_1 = load ? readAddress : A_address;
```

```
   assign bank0Address_2 = B_address;
   assign bank1Address_1 = load ? loadAddress : A_address;
   assign bank1Address_2 = B_address;

   // Bank1 write data muxes to support loading in data
   assign bank1data1_r = load ? loadData_r : real1_;
   assign bank1data1_i = load ? loadData_i : imag1_;
   assign bank1data2_r = real2_;
   assign bank1data2_i = imag2_;

   // Muxes to select between outputs of the data banks
   assign real1 = BankWrite ? real1bank0 : real1bank1;
   assign real2 = BankWrite ? real2bank0 : real2bank1;
   assign imag1 = BankWrite ? imag1bank0 : imag1bank1;
   assign imag2 = BankWrite ? imag2bank0 : imag2bank1;

   // Data structure definition
   data_bank bank0(clk, wreB0A, wreB0B, bank0Address_1, bank0Address_2, real1_,
imag1_, real2_, imag2_, real1bank0, imag1bank0, real2bank0, imag2bank0);
   data_bank bank1(clk, wreB1A, wreB1B, bank1Address_1, bank1Address_2, bank1data1_r,
bank1data1_i, bank1data2_r, bank1data2_i, real1bank1, imag1bank1, real2bank1,
imag2bank1);

   // Assign data output
   assign readData_r = real1bank0;
   assign readData_i = imag1bank0;

   // Controls to stop running once FFT is done
   logic isDone;
   always_ff @ (posedge clk) begin
       if (load) isDone <= 1'b0;
       else if(FFTdone) isDone <= 1;
       end

   assign runFFT = (~isDone && ~load);
endmodule

/////////////////////////////////////////////
// twiddle_rom
//   Twiddle factor ROM of spectrum analyzer
//   (Twiddle factors given in Table II of Slade paper)
/////////////////////////////////////////////
```

```
module twiddle_rom(input  logic        [3:0]  address,
                  output logic signed [15:0] realw, imaw);

 logic [31:0] rom[15:0];

 initial $readmemh("twiddle_rom.txt", rom);

 assign realw = rom[address][31:16];   // TODO: can this be in hex?
 assign imaw = rom[address][15:0];

endmodule

////////////// OTHER MODULES /////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//  FFT dataBank
//  written by Weston Miller (wmiller@hmc.edu) and Ingrid Tsang (itsang@hmc.edu)
//  Uses ram blocks that are really just registers.
//
//////////////////////////////////////////////////////////////////////////////////
module data_bank(input  logic        clk, wreA, wreB,
                input  logic [4:0]  A_Address, B_Address,
                input  logic [15:0] ArDat, AiDat, BrDat, BiDat,
                output logic [15:0] Ar_out, Ai_out, Br_out, Bi_out);
   logic clkEn = 1'b1;
   dataRAMhomeMade realRAM(A_Address, B_Address, clk, ArDat, BrDat, wreA, wreB,
Ar_out, Br_out);
   dataRAMhomeMade imagRAM(A_Address, B_Address, clk, AiDat, BiDat, wreA, wreB,
Ai_out, Bi_out);
endmodule

module dataRAMhomeMade(input  logic [4:0]  address1, address2,
                      input  logic        clk,
                      input  logic [15:0] wd1, wd2,
                      input  logic        we1, we2,
                      output logic [15:0] rd1, rd2);
       logic [15:0] RAM[31:0];

       assign rd1 = RAM[address1];
       assign rd2 = RAM[address2];

       always_ff @(posedge clk)
           begin
```

```
            if (we1) RAM[address1] <= wd1;
            if (we2) RAM[address2] <= wd2;
        end
endmodule
```

```
/////////////////////////////////////////////
// butterfly.sv
// HMC E155
/////////////////////////////////////////////

//////////////////////////////////////////////////////
// butterfly testbench
//////////////////////////////////////////////////////
module butterflyTestBench();
   logic signed [15:0] real1, ima1, real2, ima2, realw, imaw, real1_, ima1_, real2_,
ima2_;
   butterfly dut( real1, ima1, real2, ima2, realw, imaw, real1_, ima1_, real2_,
ima2_);

   initial
       begin
           real1 =16'b0;
           real2 =16'h7fff;
           ima1 =16'b0;
           ima2 =16'b0;
           imaw = 16'b0;
           realw =16'h7fff;
           #10;

           real1 =16'b0000000101000111;
           ima1 =16'b0000001010001111;
           real2 =16'h03d7;
           ima2 =16'hfeb8;
           imaw = 16'h471c;
           realw =16'h6a6d;
           #10;
       end
endmodule

/////////////////////////////////////////////
// butterfly
```

```
//   Butterfly unit of spectrum analyzer
/////////////////////////////////////////////
module butterfly(input  logic signed [15:0] real1, ima1, real2, ima2,          // 11
bits in 16-bit container (5 bits for bit growth)
                input  logic signed [15:0] realw, imaw,
                output logic signed [15:0] real1_, ima1_, real2_, ima2_);
   // (a + bi) * (c + di) = (ac - bd) + (ad + bc)i
   logic signed [31:0] mult1, mult2, mult3, mult4;
   assign mult1 = real2*realw;    //real
   assign mult2 = ima2*imaw;      //real
   assign mult3 = realw*ima2;     //imag
   assign mult4 = imaw*real2;     //imag

   logic signed [31:0] mult_real, mult_ima;

   assign mult_real = (mult1[31:0] - mult2[31:0]);
   assign mult_ima = (mult3[31:0] + mult4[31:0]);

   // A' = A + product
   assign real1_ = real1 + mult_real[30:15];
   assign ima1_ = ima1 + mult_ima[30:15];

   // B' = A - product
   assign real2_ = real1 - mult_real[30:15];
   assign ima2_ = ima1 - mult_ima[30:15];
endmodule
```

```
/////////////////////////////////////////////////////////////////////////////
// Adress Generation Unit (AGU) Testbench
// written by Weston Miller (wmiller@hmc.edu) and Ingrid Tsang (itsang@hmc.edu)
/////////////////////////////////////////////////////////////////////////////


module AGUtestbench();
    logic       clk, rst, runFFT, FFTdone, BankWrite;
    logic [4:0] A_address, B_address;
    logic [3:0] twiddle_address;


    // device under test
    agu dut(clk, runFFT, A_address, B_address, twiddle_address, FFTdone, BankWrite);

    // set reset high and runFFT low
    initial begin
```

```
        rst = 1'b1; #5;
        runFFT = 1'b0;
        #5;
    end

    // generate clock and load signals
    initial
        forever begin
            clk = 1'b0; #5;
            clk = 1'b1; #5;
        end

    // start running AGU


    always @(posedge clk) begin
        if(~runFFT)
            begin
                if(rst)
                    begin
                    rst = 1'b0;
                    runFFT = 1'b1;
                    end
                else
                    rst = 1'b1;
                end

      if(FFTdone)
        $stop();
      end

endmodule

////////////////////////////////////////////////////////////////////////////////
// Adress Generation Unit (agu)
// written by Weston Miller (wmiller@hmc.edu) and Ingrid Tsang (itsang@hmc.edu)
// generates addresses for the butterfly operations
// driging runFFT low resets the AGU.
////////////////////////////////////////////////////////////////////////////////
module agu(input  logic       clk, runFFT,
           output logic [4:0] A_address, B_address,
           output logic [3:0] twiddle_address,
           output logic       FFTdone, BankWrite);

    // counter
    logic [7:0] count;
    always_ff @(posedge clk)
```

```
        if (!runFFT)            count <= 8'b0;
        else if (runFFT)    count <= count+8'b1;

    // set up i and j signals
    logic [2:0] i;              // iteration level
    logic [3:0] j;              // butterly address pairs
    assign i = count[7:5];
    assign j = count[4:1];  // every j is around for 2 clock cycles

    // set up input signals to rotation blocks for A and B
    logic [4:0] arotIn, brotIn;
    assign arotIn = {j, 1'b0};      // 2j
    assign brotIn = arotIn + 5'b1;      // 2j + 1

    rotateLeft aRotate(arotIn, i, A_address);
    rotateLeft bRotate(brotIn, i, B_address);

    // generate twiddle mask
    logic [3:0] twiddleMask;
    twiddleMaskGenerator tMG(i, twiddleMask);
    assign twiddle_address = twiddleMask[3:0] & j[3:0];

    assign BankWrite = i[0]; // write on the second clock cycle
    // Need to add some complexity here: FFTdone should stay high until next run
starts
    // to do this: have rising edge detection register
    logic justStarted;
    logic wasRunFFT;
    logic finished;

    // detect rising edge of runFFT
    always_ff @(posedge clk) begin
        wasRunFFT <= runFFT;
        justStarted <= runFFT & ~wasRunFFT;
        end

    // finished register
    always_ff @(posedge clk) begin
        if (justStarted) finished <= 1'b0;
        else if(i==3'b101) finished <= 1'b1;
    end
```

```
    assign FFTdone = finished;

endmodule

///////////////////////////////////////////////////////////////////////////////
// Rotate left of N-bit word din by S module
// written by Weston Miller (wmiller@hmc.edu) and Ingrid Tsang (itsang@hmc.edu)
// Synchronous rotate left by S
////////////////////////////////////////////////////////////////////////////////
module rotateLeft(input  logic [4:0] din,
                  input  logic [2:0] S,         // number of bits to shift by
                  output logic [4:0] dout);

    always_comb
    begin
        case (S)
            // 3'b000, 3'b101 : dout <= din;
            // 3'b001, 3'b110 : dout <= {din[3:0], din[4]};
            // 3'b010, 3'b111 : dout <= {din[2:0], din[4:3]};
            // 3'b011         : dout <= {din[1:0], din[4:2]};
            // 3'b100         : dout <= {din[0], din[4:1]};
            // default        : dout <= din;
            3'b000 : dout <= din;
            3'b001 : dout <= {din[3:0], din[4]};
            3'b010 : dout <= {din[2:0], din[4:3]};
            3'b011 : dout <= {din[1:0], din[4:2]};
            3'b100 : dout <= {din[0], din[4:1]};
            3'b101 : dout <= din;
            3'b110 : dout <= {din[3:0], din[4]};
            3'b111 : dout <= {din[2:0], din[4:3]};
            default: dout <= din;
        endcase
    end
    // assign dout = rotated;

endmodule

///////////////////////////////////////////////////////////////////////////////
// Masking out module for twiddle factor address generation
// written by Weston Miller (wmiller@hmc.edu) and Ingrid Tsang (itsang@hmc.edu)
// takes in the i and j counter values and outputs the memory address for the twiddle
factor
///////////////////////////////////////////////////////////////////////////////
module twiddleMaskGenerator(input  logic [2:0] i,
                            output logic [3:0] mask);

    always_comb
```

```
    case(i)
        3'b000 : mask <= 4'b0000;
        3'b001 : mask <= 4'b1000;
        3'b010 : mask <= 4'b1100;
        3'b011 : mask <= 4'b1110;
        3'b100 : mask <= 4'b1111;
        default: mask <= 4'b0000;
    endcase

endmodule
```

SPI module:

```
/////////////////////////////////////////////
// simple_spi.sv
// HMC E155
/////////////////////////////////////////////
`timescale 1ns/1ns

///////////////////////////////////////////////////////
// spi module testbench

//////////////////////////////////////////////////////
module onePoint_spi_TestBench();

   logic clk, sck, cs, MISO, MOSI, MISOcapture;
   logic [4:0] index;
   logic [31:0] readData, loadData;
   logic [31:0] memBank0[31:0], memBank1[31:0];
   logic [39:0] master_buffer;
   logic [31:0] misoEDdata[31:0];

   //Sst up dut
   onePoint_spi dut(clk, sck, MOSI, cs, readData, MISO, index, loadData);

   // Load fake "fft'ed" data into bank 0 (this is the data that will be read)
   // "Fake" fft'ed data simply is the index address.
   initial begin
           for (int i = 0; i < 32; i++) begin
                   memBank0[i][31:16] = i[15:0];
                   memBank0[i][15:0] = i[15:0];
```

```
            end

        // Reset spi module to known state.
        sck = 1'b0;          // Set sck = 0
        cs = 1'b1; #5;
        cs = 1'b0; #5;       // Drive cs low for a bit to reset spi module
        cs = 1'b1;
    end

// Initialize clk
initial
    forever begin
        clk = 1'b0; #5;
        clk = 1'b1; #5;
    end

// Set up assign statements for index and readData:
always @ (posedge clk) begin
        readData = memBank0[index];
        memBank1[index] = loadData;
    end

// Assign MISO and MOSI
assign MOSI = master_buffer[39];

initial begin
    // Perform SPI transmission for each index
    for (int j = 0; j < 32; j++) begin
            master_buffer = {j[4:0], 3'b0, j[15:0], j[15:0]};

            // Perform shifts
            for (int s = 0; s < 40; s++) begin
                    sck = 1'b1;
                    // On posedge sck: strobe MISO
                    MISOcapture = MISO;
                    #20;

                    sck = 1'b0;
                    master_buffer = {master_buffer[38:0], MISOcapture};
                    #20;
                end
            // After shifts, capture data out to "misoEDdata"
```

```
            misoEDdata[j[4:0]] = master_buffer[31:0];
        end
      // After transmission is done, set cs low
      cs = 1'b0;
      #1000; // Wait a bunch of clock cycles to see if things get messed up

      // Stop
      // At this point, misoED should have same contents as dataBank0
      //  and dataBank1 should have same contents as dataBank0
      $stop();
   end

endmodule

////////////////////////////////////////////////////////////////////////
// onePoint_spi reciever module that handles one data point transmissions
// Written by Weston Miller (wmiller@hmc.edu)
// CLK polarity = 0; (clock is idle low)
// CLK phase = 0; (strobes on rising edge, shifts on falling edge)
//
// Total Transmission length: 5 bytes.
// Bytes in:
//      First Byte: [5-bit index, 3-bit X]
//      Second Byte: MSB real data
//      Third Byte: LSB real data
//      Fourth Byte: MSB imag data (probably zeros)
//      Fifth Byte: LSB imag data (probably zeros)
//Bytes out:
//      First Byte: [8-bit X]
//      Second Byte: MSB real data
//      Third Byte: LSB real data
//      Fourth Byte: MSB imag data
//      Fifth Byte: LSB imag data
////////////////////////////////////////////////////////////////////////
module onePoint_spi (input  logic        clk, sck, sdi, cs,
                    input  logic [31:0] readData,
                    output logic        sdo,
                    output logic [4:0]  index,
                    output logic [31:0] loadData);
   logic sdi_strobed, notCaptured;
   logic [31:0] shift_register;
   logic [5:0] sCount;
```

```
// Input handling
// SDI strobe register
always_ff @(posedge sck)
    sdi_strobed <= sdi;

// sdi_shift register
always_ff @(negedge sck) begin
    if (cs) begin
        if(sCount == 7) shift_register <= readData;
        else shift_register <= {shift_register[30:0], sdi_strobed};
        end
    end

// Output handling
// Index output register/synchronization
logic [4:0] asyncIndex;

// Index output handling:
always_ff @(posedge sck)
    if(~cs) asyncIndex <= 5'b0;
    else if(sCount == 5) asyncIndex <= shift_register[4:0];

always_ff @(posedge clk)
    if(~cs) index <= 5'b0;
    else if(sCountSync > 5) index <= asyncIndex;
    // asyncIndex only changes when sCount == 5
    // If sCountSync >5, then asyncIndex will not be changing

// SDO output handling
assign sdo = shift_register[31];

/// LOAD DATA output handling
// f_sck can't be faster than 1/2-1/4 of f_clk
// sCountSync == 40 must happen with at least 1 posedge clk before
always_ff @(posedge clk)
    if (sCountSync == 40 && notCaptured) begin
            loadData <= shift_register;
            notCaptured <= 1'b0;            // This makes sure that loadData is
only sampled once
        end
```

```
       else if (sCountSync == 2) notCaptured <= 1'b1; // Once sCount passes 2,
notCaptured resets high
       else if (~cs) loadData <= 32'h0;

   // sck falling edge counter
   always_ff @(negedge sck or negedge cs) begin
           if (!cs)
               sCount <= 6'b0;
           else if (sCount == 40) begin // this allows for consecutive read/write
transmissions without driving cs low.
               sCount <= 6'b000001;
               end
           else
               sCount <= sCount + 6'b1;
       end

   // scount synchronizer for use with registers clocked with clk.
   logic [5:0] sCountHalfSync, sCountSync;
   always_ff @(posedge clk) begin
           sCountHalfSync <= sCount;
           sCountSync <= sCountHalfSync;
       end
endmodule
```

Top Level Testbench:

```
/////////////////////////////////////////////
// spectrum_analyzerTestBench.sv
// HMC E155
/////////////////////////////////////////////
`timescale 1ns/1ns

/////////////////////////////////////////////
// spectrum_analyzer testbench
/////////////////////////////////////////////
module spectrum_analyzerTestBench();
   logic clk, sck, sdi, sdo, load, FFTdone, MISOcapture;
   logic delay;

   spectrum_analyzer dut( clk, sck, sdi, sdo, load, FFTdone);
```

```
  // Reset everything by driving load low and then high for two clk cycles

  // Fixed arrays to store load and read data
  logic [15:0] loadData_r [31:0], loadData_i[31:0];
  logic [15:0] readData_r [31:0], readData_i[31:0];
  logic [39:0] sdiBuffer;

  logic [4:0] index;

  assign sdi = sdiBuffer[39];

  // Initialize load and read data
  // The read data will just be the storage index
  initial
      begin
          for (logic [5:0] j = 0; j < 32; j++) begin
                  if (j == 3) begin
                          loadData_r[j] <= 16'h00ff;
                          loadData_i[j] <= 16'h0000;
                      end
                  else begin
                          loadData_r[j] <= 16'h0000; // Should be hfa00 for square
wave
                          loadData_i[j] <= 16'h0000;
                      end
              end

          load = 1'b0;#2;
          load = 1'b1;
      end

  // Initialize FPGA clock
  initial
      forever begin
              clk = 1'b0; #5;
              clk = 1'b1; #5;
          end

  initial begin
          // Perform SPI transmission for each index
          #4;
          for (int j = 0; j < 32; j++) begin
```

```
        sdiBuffer = {j[4:0], 3'b0, loadData_r[j], loadData_i[j]};

        // Perform shifts
        for (int s = 0; s < 40; s++) begin
                sck = 1'b1;
                // On posedge sck: strobe MISO
                MISOcapture = sdo;
                #42;
                // On negedge sck: shift
                sck = 1'b0;
                sdiBuffer = {sdiBuffer[38:0], MISOcapture};
                #42;
            end

        // After shifts, capture data out to "misoEDdata"
        readData_r[j[4:0]] = sdiBuffer[31:16];
        readData_i[j[4:0]] = sdiBuffer[15:0];
    end

    // After transmission is done, set cs low
    load = 1'b0;
    # 70000
    load = 1'b1;

    // Perform another load in order to get out FFTed data
    // Perform SPI transmission for each index
    for (int j = 0; j < 32; j++) begin
            sdiBuffer = {j[4:0], 3'b0, loadData_r[j], loadData_i[j]};

            // Perform shifts
            for( int s = 0; s < 40; s++) begin
                sck = 1'b1;
                // On posedge sck: strobe MISO
                MISOcapture = sdo;
                #42;
                // On negedge sck: shift
                sck = 1'b0;
                sdiBuffer = {sdiBuffer[38:0], MISOcapture};
                #42;
            end
        // After shifts, capture data out to "readData"
        readData_r[j[4:0]] = sdiBuffer[31:16];
```

```
            readData_i[j[4:0]] = sdiBuffer[15:0];
        end

        // Stop.
        // At this point, misoED should have same contents as dataBank0
        //   and dataBank1 should have same contents as dataBank0

        $stop();
    end

  always @ (posedge clk) begin
        if (FFTdone) load = 1'b1;
    end
endmodule
```

twiddle_rom.txt

```
7fff0000
7d891859
764130fb
6a6d471c
5a825a82
471c6a6d
30fb7641
18f97d89
00007fff
e7077d89
cf057641
b8e46a6d
a57e5a82
9593471c
89bf30fb
82771859
```

Appendix B: Microcontroller Code

```
/**
   Main:
   @file spectrum_analyzer.c
   @author Ingrid Tsang (itsang@hmc.edu) and Weston Miller (wmiller@hmc.edu)
   @version 12/9/2021
*/
#include <stdio.h>
#include <math.h>
#include "STM32F401RE.h"

////////////////////////////////////////////////
// Constants
////////////////////////////////////////////////
#define LOAD_PIN           4      // PB
#define DONE_PIN           3      // PA
#define SCE_LCD_PIN        6      // PB
#define ADC_PIN            0      // PA
#define LCD_RESET_PIN      1      // PA
#define LCD_DC_PIN         9      // PA
#define LCD_BL_PIN         8      // PA (LCD LED backlight)
#define LCD_COLS_PER_FREQ  4      // number of LCD columns per frequency
#define LCD_NUM_COLS       84     // number of columns in LCD
#define LCD_NUM_ROWS       48     // number of rows in LCD
#define DISPLAY_RAM_SIZE   4032   // LCD has 48 rows * 84 columns
#define LCD_CHAR_COLS      17     // number of columns each character on LCD spans
#define LCD_CHAR_ROWS      6      // number of rows each character on LCD spans
#define LCD_CHAR_SIZE      102    // each character will take 6 rows * 17 columns
#define AUDIO_DATA_SIZE    32     // size of input array to FFT
#define FFT_DATA_SIZE      32     // size of output array from FFT
#define SAMPLING_FREQ      3200   // 3200 Hz (detect at most 1600 Hz in audio signal)
#define FUNDAMENTAL_FREQ   100    // sampling frequency/32 samples

char charZ[LCD_CHAR_SIZE] = {0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0,
```

```
                            0, 0, 0, 0, 0, 0,
                            1, 1, 1, 1, 1, 1,
                            0, 1, 0, 0, 0, 0,
                            0, 0, 1, 0, 0, 0,
                            0, 0, 0, 1, 0, 0,
                            0, 0, 0, 0, 1, 0,
                            1, 1, 1, 1, 1, 1,
                            0, 0, 0, 0, 0, 0};

char charH[LCD_CHAR_SIZE] = {0, 0, 0, 0, 0, 0,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 1, 1, 1, 1, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            0, 0, 0, 0, 0, 0};

char char0[LCD_CHAR_SIZE] = {0, 0, 0, 0, 0, 0,
                            1, 1, 1, 1, 1, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
```

```
                            1, 1, 1, 1, 1, 1,
                            0, 0, 0, 0, 0, 0};

char char1[LCD_CHAR_SIZE] = {0, 0, 0, 0, 0, 0,
                            0, 0, 1, 0, 0, 0,
                            0, 0, 1, 1, 0, 0,
                            0, 0, 1, 0, 1, 0,
                            0, 0, 1, 0, 0, 1,
                            0, 0, 1, 0, 0, 0,
                            0, 0, 1, 0, 0, 0,
                            0, 0, 1, 0, 0, 0,
                            0, 0, 1, 0, 0, 0,
                            0, 0, 1, 0, 0, 0,
                            0, 0, 1, 0, 0, 0,
                            0, 0, 1, 0, 0, 0,
                            0, 0, 1, 0, 0, 0,
                            0, 0, 1, 0, 0, 0,
                            0, 0, 1, 0, 0, 0,
                            1, 1, 1, 1, 1, 1,
                            0, 0, 0, 0, 0, 0};

char char2[LCD_CHAR_SIZE] = {0, 0, 0, 0, 0, 0,
                            1, 1, 1, 1, 1, 1,
                            1, 0, 0, 0, 0, 0,
                            1, 0, 0, 0, 0, 0,
                            1, 0, 0, 0, 0, 0,
                            1, 0, 0, 0, 0, 0,
                            1, 0, 0, 0, 0, 0,
                            1, 0, 0, 0, 0, 0,
                            1, 1, 1, 1, 1, 1,
                            0, 0, 0, 0, 0, 1,
                            0, 0, 0, 0, 0, 1,
                            0, 0, 0, 0, 0, 1,
                            0, 0, 0, 0, 0, 1,
                            0, 0, 0, 0, 0, 1,
                            0, 0, 0, 0, 0, 1,
                            1, 1, 1, 1, 1, 1,
                            0, 0, 0, 0, 0, 0};

char char3[LCD_CHAR_SIZE] = {0, 0, 0, 0, 0, 0,
                            1, 1, 1, 1, 1, 1,
                            1, 0, 0, 0, 0, 0,
```

```
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                1, 1, 1, 1, 1, 1,
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                1, 1, 1, 1, 1, 1,
                                0, 0, 0, 0, 0, 0};

char char4[LCD_CHAR_SIZE] = {0, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 1,
                                1, 0, 0, 0, 0, 1,
                                1, 0, 0, 0, 0, 1,
                                1, 0, 0, 0, 0, 1,
                                1, 0, 0, 0, 0, 1,
                                1, 0, 0, 0, 0, 1,
                                1, 0, 0, 0, 0, 1,
                                1, 1, 1, 1, 1, 1,
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                1, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0};

char char5[LCD_CHAR_SIZE] = {0, 0, 0, 0, 0, 0,
                                1, 1, 1, 1, 1, 1,
                                0, 0, 0, 0, 0, 1,
                                0, 0, 0, 0, 0, 1,
                                0, 0, 0, 0, 0, 1,
                                0, 0, 0, 0, 0, 1,
                                0, 0, 0, 0, 0, 1,
                                0, 0, 0, 0, 0, 1,
                                1, 1, 1, 1, 1, 1,
```

```
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 1, 1, 1, 1, 1,
                             0, 0, 0, 0, 0, 0};

char char6[LCD_CHAR_SIZE] = {0, 0, 0, 0, 0, 0,
                             1, 1, 1, 1, 1, 1,
                             0, 0, 0, 0, 0, 1,
                             0, 0, 0, 0, 0, 1,
                             0, 0, 0, 0, 0, 1,
                             0, 0, 0, 0, 0, 1,
                             0, 0, 0, 0, 0, 1,
                             0, 0, 0, 0, 0, 1,
                             1, 1, 1, 1, 1, 1,
                             1, 0, 0, 0, 0, 1,
                             1, 0, 0, 0, 0, 1,
                             1, 0, 0, 0, 0, 1,
                             1, 0, 0, 0, 0, 1,
                             1, 0, 0, 0, 0, 1,
                             1, 0, 0, 0, 0, 1,
                             1, 1, 1, 1, 1, 1,
                             0, 0, 0, 0, 0, 0};

char char7[LCD_CHAR_SIZE] = {0, 0, 0, 0, 0, 0,
                             1, 1, 1, 1, 1, 1,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
                             1, 0, 0, 0, 0, 0,
```

```
                            1, 0, 0, 0, 0, 0,
                            0, 0, 0, 0, 0, 0};

char char8[LCD_CHAR_SIZE] = {0, 0, 0, 0, 0, 0,
                            1, 1, 1, 1, 1, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 1, 1, 1, 1, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 1, 1, 1, 1, 1,
                            0, 0, 0, 0, 0, 0};

char char9[LCD_CHAR_SIZE] = {0, 0, 0, 0, 0, 0,
                            1, 1, 1, 1, 1, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 0, 0, 0, 0, 1,
                            1, 1, 1, 1, 1, 1,
                            1, 0, 0, 0, 0, 0,
                            1, 0, 0, 0, 0, 0,
                            1, 0, 0, 0, 0, 0,
                            1, 0, 0, 0, 0, 0,
                            1, 0, 0, 0, 0, 0,
                            1, 0, 0, 0, 0, 0,
                            1, 1, 1, 1, 1, 1,
                            0, 0, 0, 0, 0, 0};

////////////////////////////////////////////////
// Function Prototypes
////////////////////////////////////////////////
```

```
void initLCD();
void getAudioData(uint16_t *audioData);
void performFFT(uint16_t *fftInput, uint16_t *fftOutputReal, uint16_t *fftOutputImag);
void displayFFT(double *fftOutput);
void clearDisplay();
int displayDominantFrequency(char *displayRAM, int index, int dominantFreq);
int appendCharColumn(char *displayRAM, int index, char *charArray, int
charStartIndex);
double q15toDouble(uint16_t q15val);
double getMagnitude(double real, double imag);

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

   // Enable clocks
   RCC->AHB1ENR.GPIOAEN = 1;
   RCC->AHB1ENR.GPIOBEN = 1;
   RCC->AHB1ENR.DMA2EN = 1;
   RCC->APB2ENR.ADC1EN = 1;
   RCC->APB2ENR.SPI1EN = 1;
   RCC->APB1ENR.TIM2EN = 1;
   RCC->APB1ENR.TIM5EN = 1;

   // Initialize pins
   pinMode(GPIOA, ADC_PIN, GPIO_ANALOG);
   pinMode(GPIOB, LOAD_PIN, GPIO_OUTPUT);
   pinMode(GPIOA, DONE_PIN, GPIO_INPUT);
   pinMode(GPIOB, SCE_LCD_PIN, GPIO_OUTPUT);

   // Configure ADC
   configureADC();
   initTIM2(SAMPLING_FREQ);

   // Configure DMA
   uint16_t audioData[AUDIO_DATA_SIZE];
   initDMA2((uint32_t) &(ADC1->DR), (uint32_t) &audioData, AUDIO_DATA_SIZE);   //
Peripheral: ADC1 data register
```

```
                                                                                           //
Memory: array buffer

   // Iniitalize SPI
   // "clock divide" = master clock frequency / desired baud rate
   // the phase for the SPI clock is 0 and the polarity is 0
   spiInit(0b101, 0, 0);
   initTIM5();

   // Set serial chip select pins
   digitalWrite(GPIOB, SCE_LCD_PIN, 1);     // active low
   digitalWrite(GPIOB, LOAD_PIN, 0);        // active high

   // Initialize LCD
   initLCD();
   clearDisplay();

   // Initialize data arrays
   uint16_t fftInput[AUDIO_DATA_SIZE];
   uint16_t fftOutputReal[FFT_DATA_SIZE];
   uint16_t fftOutputImag[FFT_DATA_SIZE];
   double fftOutputMagnitudes[FFT_DATA_SIZE/2];
   while (1) {
       // Sample audio data
       getAudioData(audioData);

       // Perform FFT using FPGA
       // Test audio data (square wave)
       // for (int i = 0; i < FFT_DATA_SIZE; ++i){
       //     if (i >= 24 || (i >= 8 & i < 17)){
       //         audioData[i] = 0x00ff;
       //     }
       //     else {
       //         audioData[i] = 0x0000;
       //     }
       // }
       delay_micros(1000);
       performFFT(audioData, fftOutputReal, fftOutputImag);

       // Convert FFT output to doubles and calculate magnitudes
       // Only half due to symmetry (audio signal is only real)
       for (int i = 0; i < FFT_DATA_SIZE/2; i++) {
```

```
            if (i == 0) {
                fftOutputMagnitudes[i] = 0;              // Set DC to zero
            } else {
                fftOutputMagnitudes[i] = getMagnitude(
                    q15toDouble(fftOutputReal[i]),
                    q15toDouble(fftOutputImag[i])
                );
            }
        }

        // Display amplitude vs frequency graph on LCD
        displayFFT(fftOutputMagnitudes);
    }
}

////////////////////////////////////////////////
// Functions
////////////////////////////////////////////////
// Initialize LCD to prepare for sending data from MCU
void initLCD() {
   // Set pins
   pinMode(GPIOA, LCD_RESET_PIN, GPIO_OUTPUT);
   pinMode(GPIOA, LCD_DC_PIN, GPIO_OUTPUT);
   pinMode(GPIOA, LCD_BL_PIN, GPIO_OUTPUT);

   // Turn on LED backlight (optional)
   digitalWrite(GPIOA, LCD_BL_PIN, 1);

   // Reset LCD
   digitalWrite(GPIOA, LCD_RESET_PIN, 0);
   digitalWrite(GPIOA, LCD_RESET_PIN, 1);

   // Set D/C pin for command
   digitalWrite(GPIOA, LCD_DC_PIN, 0);

   // Initialize settings for LCD
   digitalWrite(GPIOB, SCE_LCD_PIN, 0);
   spiSendReceive(0x22);     // Set PD = 0 (chip is active),
                             //     V = 1  (vertical addressing - p9),
                             //     H = 0  (use basic instruction set)
   digitalWrite(GPIOB, SCE_LCD_PIN, 1);
```

```
    digitalWrite(GPIOB, SCE_LCD_PIN, 0);
    spiSendReceive(0x0C);      // Set DE = 10 (normal mode)
    digitalWrite(GPIOB, SCE_LCD_PIN, 1);

    // Set D/C pin for data
    digitalWrite(GPIOA, LCD_DC_PIN, 1);
}

// Get audio data from microphone using ADC
void getAudioData(uint16_t *audioData) {
    // Sample microphone at SAMPLE_FREQUENCY (triggered by TIM2)
    for (int i = 0; i < AUDIO_DATA_SIZE; i++) {
        TIM2->SR &= ~(0x1);          // Clear UIF
        TIM2->CNT = 0;               // Reset count
        ADC1->CR2.SWSTART = 1;       // Start single ADC conversion
        while(!(TIM2->SR & 1));      // Wait for UIF to go high
    }
}

// Perform FFT using FPGA
void performFFT(uint16_t *fftInput, uint16_t *fftOutputReal, uint16_t *fftOutputImag)
{
    // Reset FFT
    digitalWrite(GPIOB, LOAD_PIN, 1);
    digitalWrite(GPIOB, LOAD_PIN, 0);

    // Load audio data onto FPGA
    // Transmission interface:
    //    send {5'b address, 000}
    //    send {MSB real_data}
    //    send {LSB real_data}
    //    send {MSB imag_data}
    //    send {LSB imag_data}
    digitalWrite(GPIOB, LOAD_PIN, 1);
    delay_micros(1000);
    for (int i = 0; i < AUDIO_DATA_SIZE; i++) {
        spiSendReceive(i << 3);                 // Send {5'b address, 000}
        while(SPI1->SR.BSY);
        spiSendReceive(fftInput[i] >> 8);       // Send {MSB real_data}
        while(SPI1->SR.BSY);
        spiSendReceive(fftInput[i]);            // Send {LSB real_data}
        while(SPI1->SR.BSY);
```

```
        spiSendReceive(0);                          // Send {MSB imag_data}
        while(SPI1->SR.BSY);
        spiSendReceive(0);                          // Send {LSB imag_data}
        while(SPI1->SR.BSY);
    }

    // Reset LOAD signal signifying that data loading is complete
    delay_micros(1000);
    digitalWrite(GPIOB, LOAD_PIN, 0);

    // Wait for DONE signal to be asserted by FPGA signifying that the data is ready to
be read out
    while(!digitalRead(GPIOB, DONE_PIN));

    // Transmission interface:
    //    send {5'b index, 000} rec: {8'bX}
    //    send {8'b0}           rec: {MSB_r}
    //    send {8'b0}           rec: {LSB_r}
    //    send {8'b0}           rec: {MSB_i}
    //    send {8'b0}           rec: {LSB_i}
    delay_micros(1000);
    digitalWrite(GPIOB, LOAD_PIN, 1);
    for (int i = 0; i < FFT_DATA_SIZE; i++) {
        spiSendReceive(i << 3);                     // Send {5'b index, 000}
        while(SPI1->SR.BSY);
        fftOutputReal[i] = spiSendReceive(i);     // Receive {MSB_r}
        fftOutputReal[i] = fftOutputReal[i] << 8;
        while(SPI1->SR.BSY);
        fftOutputReal[i] |= spiSendReceive(i);    // Receive {LSB_r}
        while(SPI1->SR.BSY);
        fftOutputImag[i] = spiSendReceive(i);     // Receive {MSB_i}
        fftOutputImag[i] = fftOutputImag[i] << 8;
        while(SPI1->SR.BSY);
        fftOutputImag[i] |= spiSendReceive(i);    // Receive {LSB_i}
        while(SPI1->SR.BSY);
    }
    digitalWrite(GPIOB, LOAD_PIN, 0);
}

// Display FFT output as frequency vs amplitude graph on LCD
void displayFFT(double *fftOutputMagnitudes) {
    // Test fftOutput
```

```
// double testFftOutput[32] =
//     {0x04, 0x10, 0x18, 0x03, 0x10, 0x11, 0x04, 0x0F,
//      0x01, 0x04, 0x03, 0x11, 0x12, 0x01, 0x05, 0x13,
//      0x00, 0x18, 0x11, 0x08, 0x02, 0x03, 0x05, 0x11,
//      0x10, 0x12, 0x14, 0x16, 0x18, 0x1A, 0x1C, 0x1E};
// fftOutputMagnitudes = testFftOutput;

// Get max amplitude of FFT output to normalize
double maxAmplitude = 0;
int dominantFreq = 0;
for (int i = 0; i < FFT_DATA_SIZE/2; i++) {
    if (fftOutputMagnitudes[i] > maxAmplitude) {
        maxAmplitude = fftOutputMagnitudes[i];
        dominantFreq = i * FUNDAMENTAL_FREQ;
    }
}
if (!maxAmplitude) {
    maxAmplitude = 1;
}

// Convert FFT output to display pixels (vertical addressing)
char displayRAM[DISPLAY_RAM_SIZE];
volatile int index = 0;
for (int i = 0; i < FFT_DATA_SIZE/2; i++) {
    for (int col = 0; col < LCD_COLS_PER_FREQ; col++) {
        // Calculate height of column by normalizing by fixed amplitude (0.2)
        double height = fftOutputMagnitudes[i] / .2 * (double) LCD_NUM_ROWS;
        for (int row = 0; row < LCD_NUM_ROWS; row++) {
            displayRAM[index] = row > (LCD_NUM_ROWS - height);
            index += 1;
        }
    }
}

// Clear three columns of display in between graph and text
for (int i = 0; i < LCD_NUM_ROWS * 3; i++) {
    displayRAM[index] = 0;
    index += 1;
}

// Display dominant frequency on the LCD
index = displayDominantFrequency(displayRAM, index, dominantFreq);
```

```
    // Clear rest of display
    while (index < DISPLAY_RAM_SIZE) {
        displayRAM[index] = 0;
        index += 1;
    }

    // Send pixel array to LCD through SPI
    for (int i = 0; i < DISPLAY_RAM_SIZE; i+=8) {
        uint8_t pixels = displayRAM[i] << 0   | displayRAM[i+1] << 1 |
                         displayRAM[i+2] << 2 | displayRAM[i+3] << 3 |
                         displayRAM[i+4] << 4 | displayRAM[i+5] << 5 |
                         displayRAM[i+6] << 6 | displayRAM[i+7] << 7;
        digitalWrite(GPIOB, SCE_LCD_PIN, 0);
        spiSendReceive(pixels);
        digitalWrite(GPIOB, SCE_LCD_PIN, 1);
    }
}

// Clear LCD display
void clearDisplay() {
    for (int i = 0; i < DISPLAY_RAM_SIZE; i+=8) {
        digitalWrite(GPIOB, SCE_LCD_PIN, 0);
        spiSendReceive(0b0);
        digitalWrite(GPIOB, SCE_LCD_PIN, 1);
    }
}

// Display dominant frequency on the LCD
// Assumptions:
//   - 0 <= dominantFreq <= 9999
//   - displayRAM has >= LCD_CHAR_COLS columns remaining (i.e. index <= LCD_NUM_ROWS *
(DISPLAY_RAM_SIZE - LCD_CHAR_COLS))
int displayDominantFrequency(char *displayRAM, int index, int dominantFreq) {
    // Get digits of dominant frequency
    char digitsReversed[4];
    volatile int i = 0;
    if (dominantFreq == 0) {
        digitsReversed[0] = 0;
        i += 1;
    }
    while (dominantFreq > 0) {
```

```
        digitsReversed[i] = dominantFreq % 10;
        dominantFreq /= 10;
        i += 1;
    }
    // Append pixels to pixel array
    for (int col = 0; col < LCD_CHAR_COLS; col++) {
        index = appendCharColumn(displayRAM, index, charZ, col * LCD_CHAR_ROWS);     //
Add 'z'
        index = appendCharColumn(displayRAM, index, charH, col * LCD_CHAR_ROWS);     //
Add 'H'

        // Add digits
        for (int j = 0; j < i; j++) {
            switch (digitsReversed[j]) {
                case 0:
                    index = appendCharColumn(displayRAM, index, char0, col *
LCD_CHAR_ROWS);
                    break;
                case 1:
                    index = appendCharColumn(displayRAM, index, char1, col *
LCD_CHAR_ROWS);
                    break;
                case 2:
                    index = appendCharColumn(displayRAM, index, char2, col *
LCD_CHAR_ROWS);
                    break;
                case 3:
                    index = appendCharColumn(displayRAM, index, char3, col *
LCD_CHAR_ROWS);
                    break;
                case 4:
                    index = appendCharColumn(displayRAM, index, char4, col *
LCD_CHAR_ROWS);
                    break;
                case 5:
                    index = appendCharColumn(displayRAM, index, char5, col *
LCD_CHAR_ROWS);
                    break;
                case 6:
                    index = appendCharColumn(displayRAM, index, char6, col *
LCD_CHAR_ROWS);
                    break;
```

```
            case 7:
                index = appendCharColumn(displayRAM, index, char7, col *
LCD_CHAR_ROWS);
                break;
            case 8:
                index = appendCharColumn(displayRAM, index, char8, col *
LCD_CHAR_ROWS);
                break;
            case 9:
                index = appendCharColumn(displayRAM, index, char9, col *
LCD_CHAR_ROWS);
                break;
            }
        }

        volatile int row = (1 + LCD_CHAR_ROWS) * (2 + i);
        while (row < LCD_NUM_ROWS) {                               // Clear rest of
column
            displayRAM[index] = 0;
            index += 1;
            row += 1;
        }
    }
    return index;
}

// Helper function for displayDominantFrequency to append one column of a char to the
pixel array
int appendCharColumn(char *displayRAM, int index, char *charArray, int charStartIndex)
{
    // Add empty row
    displayRAM[index] = 0;
    index += 1;

    // Add character
    for (int j = 0; j < LCD_CHAR_ROWS; j++) {
        displayRAM[index] = charArray[charStartIndex + j];
        index += 1;
    }
    return index;
}
```

```c
// Convert Q15 to double
double q15toDouble (uint16_t q15val) {
   uint16_t mask = 1 << 15;
   double doubleValue = 0;
   double difference;

   if (mask == (mask & q15val)) {
       doubleValue = -1;
   }

   for(int i = 0; i < 15; ++i) {
       mask = 1 << i;
       if (mask == (mask & q15val)) {
           difference = 15-i;
           doubleValue += pow(0.5, difference);
       }
   }
   return doubleValue;
}

// Calculate magnitude of vector from real and imaginary components
double getMagnitude(double real, double imag) {
   return sqrt(pow(real, 2) + pow(imag, 2));
}
```

```c
// STM32F401RE_ADC.c
// Source code for ADC functions

#include "STM32F401RE_ADC.h"

void configureADC() {
   // Use regular channel 0
   ADC1->SQR3.SQ1 = 0;     // 1st conversion: channel 0

   // Turn on ADC
   ADC1->CR2.ADON = 1;

   // Enable DMA mode
   ADC1->CR2.DMA = 1;
   ADC1->CR2.DDS = 1;
}
```

```c
// STM32F401RE_DMA.c
// Source code for DMA functions

#include "STM32F401RE_DMA.h"

void initDMA2(uint32_t peripheralAddress, uint32_t memoryAddress, uint16_t numData) {
   // DMA2 stream 0 (channel 0) for ADC1
   // Select channel 0
   DMA2->S0CR.CHSEL = 0b000;

   // Set priority level to high
   DMA2->S0CR.PL = 0b10;

   // Set memory data size to half-word (16-bit)
   DMA2->S0CR.MSIZE = 0b01;

   // Set peripheral data size to be half-word (16-bit)
   DMA2->S0CR.PSIZE = 0b01;

   // Set memory address pointer to be incremented after each data transfer
   DMA2->S0CR.MINC = 1;

   // Set data transfer to be peripheral-to-memory
   DMA2->S0CR.DIR = 0b00;

   // Enable circular mode
   DMA2->S0CR.CIRC = 1;

   // Set DMA source and destination addresses
   DMA2->S0PAR = peripheralAddress;         // Source: peripheral
   DMA2->S0M0AR = memoryAddress;            // Destination: memory

   // Set DMA data transfer length (# of samples)
   DMA2->S0NDTR = numData;

   // Enable DMA
   DMA2->S0CR.EN = 1;
}
```

```c
// STM32F401RE_FLASH.c
// Source code for FLASH functions
```

```
#include "STM32F401RE_FLASH.h"

void configureFlash() {
   FLASH->ACR.LATENCY = 2; // Set to 0 waitstates
   FLASH->ACR.PRFTEN = 1; // Turn on the ART
}
```

```
// STM32F401RE_SPI.c
// SPI function declarations

#include "STM32F401RE_SPI.h"
#include "STM32F401RE_RCC.h"
#include "STM32F401RE_GPIO.h"

/* Enables the SPI peripheral and intializes its clock speed (baud rate), polarity,
and phase.
*    -- br: (0b000 - 0b111). The SPI clk will be the master clock / 2^(BR+1).
*    -- cpol: clock polarity (0: inactive state is logical 0, 1: inactive state is
logical 1).
*    -- cpha: clock phase (0: data captured on leading edge of clk and changed on next
edge,
*          1: data changed on leading edge of clk and captured on next edge)
* Refer to the datasheet for more low-level details. */
void spiInit(uint32_t br, uint32_t cpol, uint32_t cpha) {
   // Initially assigning SPI pins
   pinMode(GPIOA, 5, GPIO_ALT);        // SPI1_SCK
   pinMode(GPIOA, 6, GPIO_ALT);        // SPI_MISO (SDI)
   pinMode(GPIOA, 7, GPIO_ALT);        // SPI_MOSI (SDO)

   // Set output speed type to high for SCK
   GPIOA->OSPEEDR |= (0b11 << 2*5);

   // Set to AF05 for SPI alternate functions
   GPIOA->AFRL |= (0b101 << 4*5) | (0b101 << 4*6) | (0b101 << 4*7);

   SPI1->CR1.BR = br;       // Set the clock divisor
   SPI1->CR1.CPOL = cpol;   // Set the polarity
   SPI1->CR1.CPHA = cpha;   // Set the phase
   SPI1->CR1.LSBFIRST = 0; // Set most significant bit first
   SPI1->CR1.DFF = 0;       // Set data format to 8 bits
   SPI1->CR1.SSM = 0;       // Turn off software slave management
```

```
    SPI1->CR2.SSOE = 1;      // Set the NSS pin to output mode
    SPI1->CR1.MSTR = 1;      // Put SPI in master mode
    SPI1->CR1.SPE = 1;       // Enable SPI
}

/* Transmits a character (1 byte) over SPI and returns the received character.
*     -- send: the character to send over SPI
*     -- return: the character received over SPI */
uint8_t spiSendReceive(uint8_t send) {
    while(!(SPI1->SR.TXE));          // Wait until the transmit buffer is empty
    SPI1->DR.DR = send;              // Transmit the character over SPI
    while(!(SPI1->SR.RXNE));         // Wait until data has been received
    uint8_t rec = SPI1->DR.DR;
    return rec;                      // Return received character
}

/* Transmits a short (2 bytes) over SPI and returns the received short.
*     -- send: the short to send over SPI
*     -- return: the short received over SPI */
uint16_t spiSendReceive16(uint16_t send) {
    SPI1->CR1.SPE = 1;
    SPI1->DR.DR = send;
    while(!(SPI1->SR.RXNE));
    uint16_t rec = SPI1->DR.DR;
    SPI1->CR1.SPE = 0;
    return rec;
}
```

```
// STM32F401RE_TIM.c
// TIM functions

#include "STM32F401RE_TIM.h"
#include "STM32F401RE_RCC.h"

void initTIM2(uint32_t samplingFreq) {
    // Set prescaler and autoreload to issue DMA request at samplingFreq
    TIM2->PSC = 0x0000;
    TIM2->ARR = SystemCoreClock/samplingFreq;

    // Enable trigger output on timer update events
    TIM2->CR2.MMS = 0b010;       // Set master mode to update
    TIM2->DIER.UDE = 1;
```

```
   // Generate an update event to force update
   TIM2->EGR |= 1;

   // Enable counter
   TIM2->CR1 |= 1;              // Set CEN = 1
}

void initTIM5(){
 // Set prescaler to give 1 µs time base
 uint32_t psc_div = (uint32_t) ((SystemCoreClock/1e6)-1);
  // Set prescaler division factor
 TIM5->PSC = (psc_div - 1);
  // Enable update interrupt
 TIM5->DIER.UIE = 1;

 // Generate an update event to update prescaler value
 TIM5->EGR |= 1;

 // Enable counter
 TIM5->CR1 |= 1; // Set CEN = 1
}

void delay_millis(uint32_t ms) {
 TIM5->ARR = ms*1000;// Set timer max count
 TIM5->EGR |= 1;      // Force update
 TIM5->SR &= ~(0x1); // Clear UIF
 TIM5->CNT = 0;      // Reset count

 while(!(TIM5->SR & 1)); // Wait for UIF to go high
}

void delay_micros(uint32_t us) {
 TIM5->ARR = us;      // Set timer max count
 TIM5->EGR |= 1;      // Force update
 TIM5->SR &= ~(0x1); // Clear UIF
 TIM5->CNT = 0;      // Reset count

 while(!(TIM5->SR & 1)); // Wait for UIF to go high
}
```