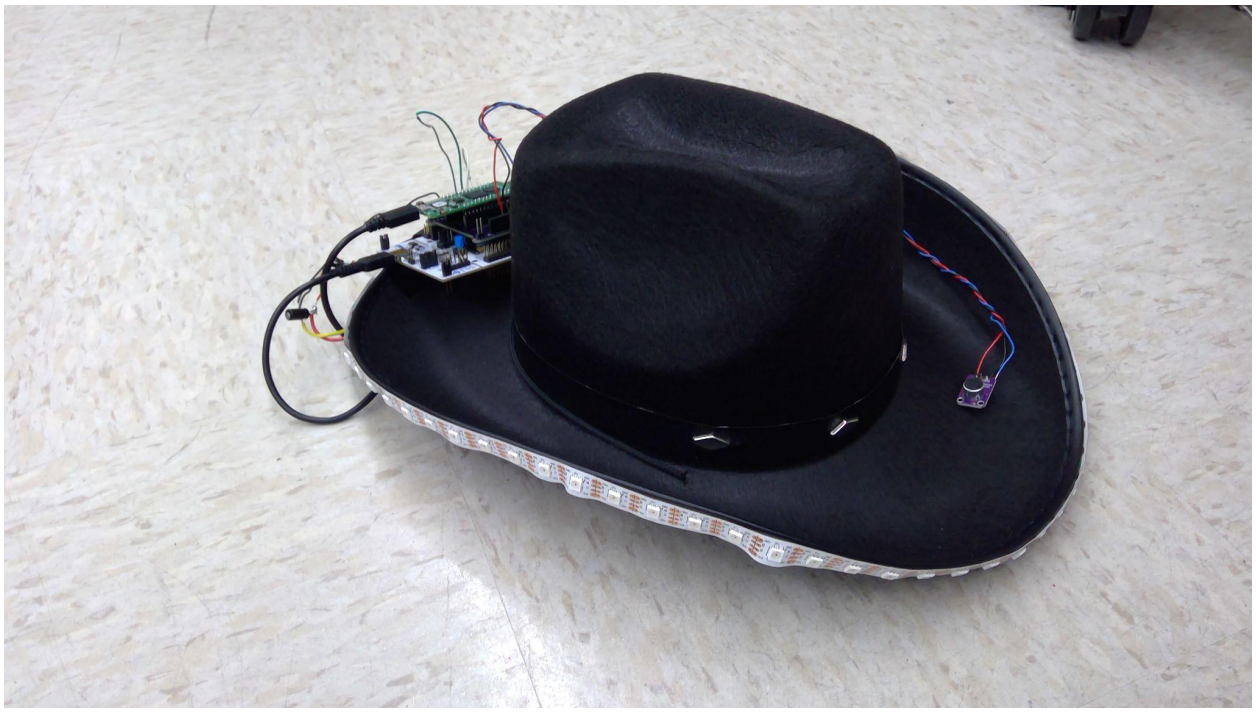
Musical Hat

Final Project Report
December 13, 2021
E155

Mason Wray and Robert Gallardo



Abstract

Our team attempted to create a “Musical Hat”. This hat had a strip of 60 addressable LEDs fixed to the top. A microphone on the hat would take in audio data from the environment and the LED’s would flash according to the audio’s spectrum data as determined by a 32 point fast fourier transform (FFT). While we were unable to represent the spectrum data on the LED’s correctly, we were successfully able to: convert analog audio data to digital data using an analog to digital converter (ADC), establish simultaneous serial peripheral interface (SPI) communication between the LED strip and FPGA, and develop a functioning 32 point FFT implementation using an FPGA.

1 Introduction

The end goal of this project was to be able to represent in real time the audio spectrum data present in the environment on a strip of 60 RGB LEDs. This would be done through sending sampled audio data to the

MAX1000 FPGA that contained an implementation of the 32 point FFT where the spectrum data would be calculated. Once the calculations were completed the FPGA would send the spectrum data back to the MCU. After receiving new spectrum data from the FPGA, the MCU would flash the LEDs on the hat depending on the frequencies present in the spectrum data. The dataflow and specifications for the project are detailed below:

1. Analog audio data is captured using the MAX4466 Electret Microphone Amplifier
2. 32 samples of audio data from the environment is sampled at a rate of 8000Hz using the Nucleo F401RE's onboard ADC. Signal duration of .125 ms.
3. Sampled audio data is sent to the MAX1000 FPGA using SPI communication
4. MAX1000 FPGA calculates the 32 point FFT on the sampled audio data
5. MAX1000 FPGA sends calculated spectrum data back to the MCU over SPI communication
6. Nucleo F401RE analyzes spectrum data and determines how the brightness of each LED on the hat needs to change.
7. Nucleo F401RE sends lighting data to the Addressable APA102 LED Strip over SPI communication.
8. Addressable APA102 LED Strip changes brightness of each LED according to lighting data.

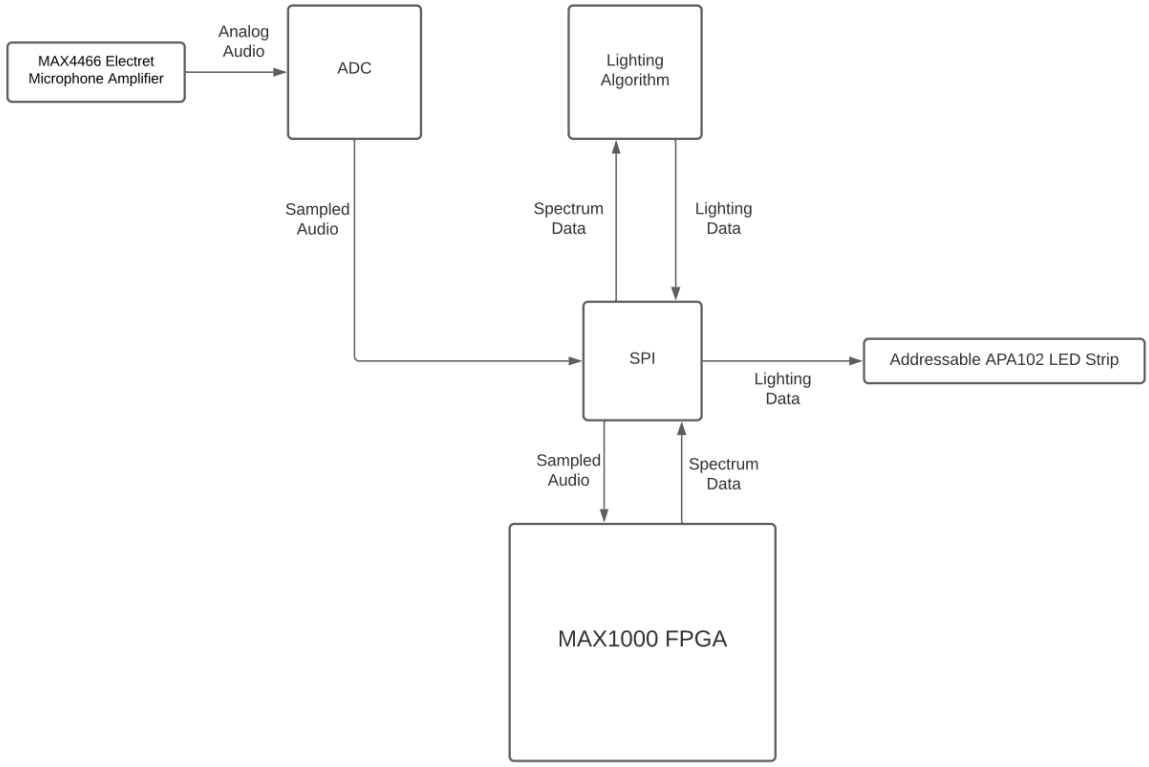


Fig 1: Project Dataflow Block Diagram

Ultimately we were unable to flash the lights on the LED strip according to the spectrum data calculated. This was due to a communication failure when sending spectrum data back to the Nucleo F401RE (5). The following is a detailed report of the results we were able to achieve as well as a deeper discussion of the system failure that we encountered.

2 New Hardware

To make the musical hat, we used two new pieces of hardware: a microphone to receive analog audio data, and an individually addressable strip of 60 LEDs.

2.1 Electret Microphone Amplifier MAX4466

Fig 2: Image of microphone amplifier board

The microphone we used is the Electret Microphone Amplifier MAX4466. It contains a 20-20KHz electret microphone soldered to a board containing a Maxim MAX4466 op-amp with adjustable gain. The mic has three ports: VCC, GND, and OUT. We powered the mic by connecting VCC and GND to the 3.3V supply and GND on the Nucleo-64, respectively.

To read data from the mic, we connected the OUT port to pin PA1 on the MCU, which read in the analog data to the board's ADC. The MAX4466 continuously outputs audio waveform data in the form of voltage corresponding to air pressure. The output voltage values generally range from 1 to 3V depending on noise level. With typical background noise in the lab, the output ranges from 2000 to 2050 mV.

2.2 Addressable APA102 LED Strip

We used a one meter strip of 60 APA102 LEDs to flash with music. The strip has four input pins: Vcc, GND, CLK, and DATA. Vcc requires 5V to power the LED strip. We supplied power to the strip with a USB to pinout converter connected to a 5V portable charger. We connected the 5V pin to Vcc and the GND pin to GND.

The APA102 LEDs can be individually controlled with SPI communication using the CLK and DATA pins. We set up the SPI protocol on the MCU, connecting SCK (pin PB10) to CLK and

MOSI (pin PB15) to DATA. The communication protocol is outlined in Fig 3. As an example, to set all LEDs to red at full brightness, we first send a start frame of 32 zeros to signal to the strip , followed by 60 LED frames each with three 1s, then ‘Global’ set to five 1s, ‘Blue’ set to eight 0s, ‘Green’ set to eight 0s, and ‘Red’ set to eight 1s. Finally, we send an end frame of 32 1s to signify the transmission is complete.

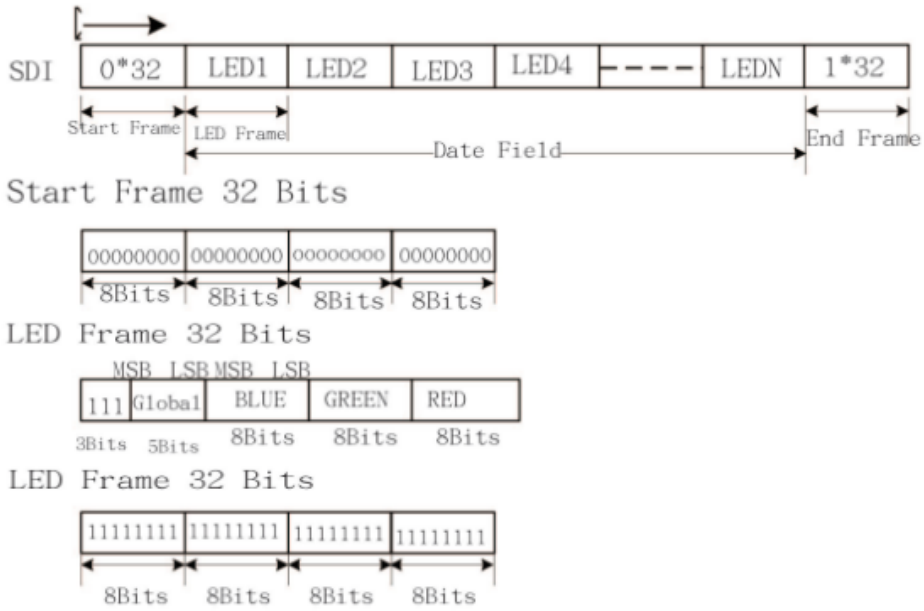


Fig 3: SPI protocol for APA102 LED strip

3 Schematics

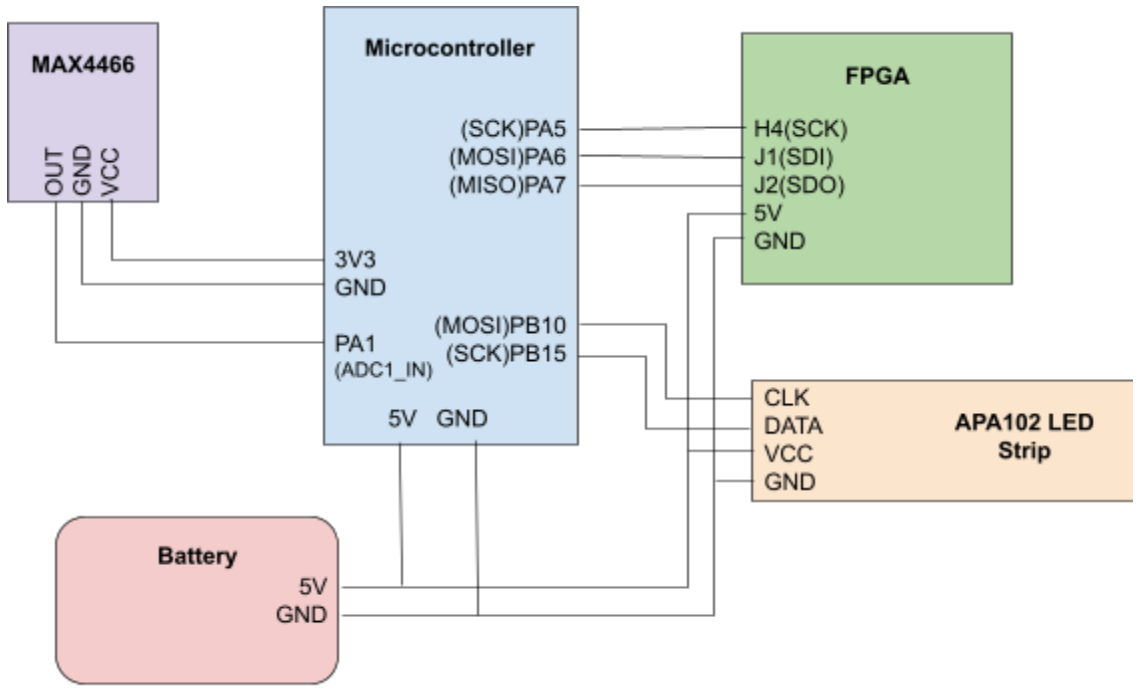


Fig 4: Breadboard Schematic

4 Microcontroller Design

The microcontroller orchestrates the overall data flow by reading from the mic using ADC, sending and receiving data from the FPGA over SPI, and sending control signals to the LED strip over SPI. This three step process is repeated continuously, with each cycle corresponding to a single 32 sample frequency and one update of the LEDs.

4.1 ADC

The first step the microcontroller completes is sampling 32 data points from the mic and converting the analog voltage to a digital 16 bit value. To do so, we initialized the ADC in ‘STM32F401RE_ADC.c’ with the configureADC() function. This function initializes the ADC to continuously make conversions on a single channel. Then, to sample at the desired frequency of 8kHz, we read the data stored in the ADC data register (ADC->DR.DATA) every 125 ms. We achieved this delay by initializing the MCU’s built-in timer 2 (TIM2) and calling

delay_micros(TIM2, 125). After sampling 32 times and storing each 16 bit value in an array, we then move on to sending the data to the FPGA for the FFT.

4.2 FPGA SPI

To send the audio waveform data to the FPGA, we used the MCU's SPI1. The FFT requires 32 points of data, each consisting of a 16 bit real part and a 16 bit imaginary part. Since the audio waveform signal consists only of real values, we simply sent 16 zeros for each imaginary part. For the real part, the ADC outputs 16 bit real values, however, these are unsigned integers and the ~~fft~~ requires 5 bits of overflow, meaning we can only send it values with 11 significant bits. To account for this, we divided each digital ADC output by 10 since the original values were all under 10,000 and any value under 1,000 has at most 10 significant bits. Finally, using the 16 bit SPI send/receive function 'spiSendReceive16', we sent the 32 real and imaginary data points to the FPGA to perform the FFT.

After sending the data, the MCU waits for the FPGA to assert the DONE pin, letting the MCU know the FFT has been completed and the FPGA is ready to send the data. Again using the 16 bit SPI send/receive function, the MCU reads in the 16 bit real part followed by the 16 bit imaginary part of the FFT output 32 times, corresponding to 32 frequency buckets.

4.3 APA102 LED Lighting

The final part of the MCUs control cycle is to update the LED strip based on the data received from the FPGA. We also designed a way to drive the LEDs using just the ADC output data in the case that the FPGA FFT data was not sufficient.

Using the FFT data, the MCU lights up the first 30 LEDs based on the amplitude of the first 30 frequencies of the discrete fourier transform (DFT) spectrum data. The updateBrightness function, in STM32F401RE_APA102.c, sets 30 brightness levels ranging from 0 to 31 based on the amplitude of the 30 frequencies. This is calculated for the DFT real imaginary pair by taking the sum of the real part squared and the imaginary part squared and then normalizing so the max value is 31 and the min is 0.

Without using the FFT data, the MCU uses the direct ADC output data by assigning each of the first 30 LEDs and the second 30 LEDs to the first 30 samples. Similar to above, we normalize

the sample data so all the values fall between 0 and 31. If the value for sample *i* is greater than 15, we set LED *i* and LED 30+*i* to full brightness. If the value is less than 15, we turn that LED off. This makes the LEDs show the approximate sound wave by lighting up the upper have of the wave and turning off the lower half.

At startup, before the above three step process runs continuously, we initialize the colors of the LEDs which form a rainbow. We then run a startup sequence where the LEDs light up sequentially several times. The colors of the LEDs remain constant for the duration of the program and only their brightnesses change.

5 FPGA Design

When designing the FPGA logic, there were 2 main functions that the FPGA needed to perform: the 32 point FFT, and the SPI communication link with the MCU. For reference, all FPGA logic was created using the SystemVerilog code located in Appendix B.

5.1 32 Point FFT Design

The 32 point FFT was designed according to an implementation specified in George Slade's "The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation" [1]. This document details the implementation of a pipelined FFT using hardware. While we did not implement the pipelined version of the FFT, many of the modules remain the same or only have slight modifications that will be detailed below.

5.1.1 Butterfly Unit Module (BFU)

The Butterfly Unit Module is the most basic element of the FFT. It is simply a 2-point FFT on 2 complex numbers. A graphical representation of the calculation is shown below in figure 5.

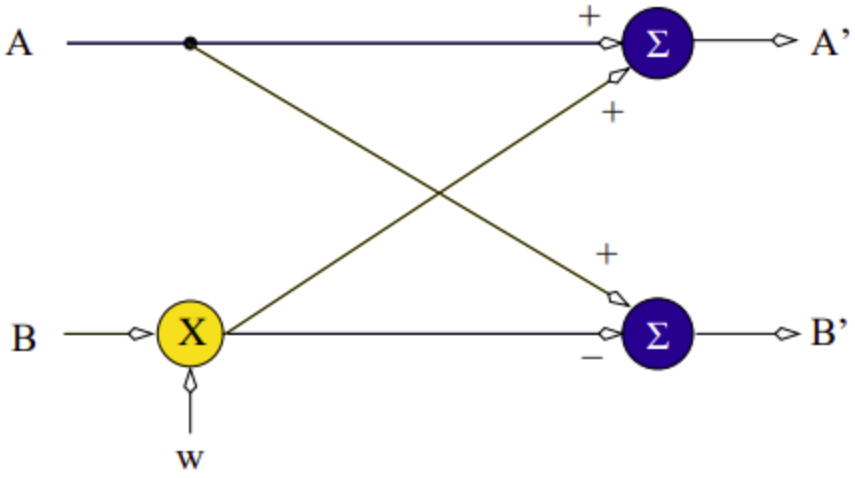


Fig 5: Graphical Representation of Butterfly Unit

Here A and B are the complex inputs, w is the twiddle factor described in section 5.1.2, and A' and b' are the outputs that will be stored in memory for future calculations. In essence, for a 32 point FFT, the BFU is used over and over with different A's and B's that are specified by the AGU.

5.1.2 Twiddle Factor ROMs

The twiddle factors are constants that are multiplied to the B input of the Butterfly Unit Module. These twiddle factors are constant regardless of the FFT input data and depend only on the current step of the FFT being calculated. For a 32 point FFT, there are 16 twiddle factors for the real component and 16 twiddle factors for the imaginary component. The twiddle factor used is determined by the AGU. The value of the twiddle factors are determined using the following equation:

$$w^n = e^{-\frac{j2\pi n}{N}}$$

Where N is the number of points of the FFT and n is defined as N/2. The twiddle factors for a 32 point FFT were calculated in the Slade paper [1], and are shown in figure 6:

Address k	$\cos(2\pi k/32)$ float	$\cos(2\pi k/32)$ 16-bit integer	$\sin(2\pi k/32)$ float	$\sin(2\pi k/32)$ 16-bit integer
0	1.0000e+00	0x7fff	0	0
1	9.8079e-01	0x7d89	1.9509e-01	0x1859
2	9.2388e-01	0x7641	3.8268e-01	0x30fb
3	8.3147e-01	0x6a6d	5.5557e-01	0x471c
4	7.0711e-01	0x5a82	7.0711e-01	0x5a82
5	5.5557e-01	0x471c	8.3147e-01	0x6a6d
6	3.8268e-01	0x30fb	9.2388e-01	0x7641
7	1.9509e-01	0x18f9	9.8079e-01	0x7d89
8	0	0x0	1.0e+00	0x7fff
9	-1.9509e-01	0xe707	9.8079e-01	0x7d89
10	-3.8268e-01	0xcf05	9.2388e-01	0x7641
11	-5.5557e-01	0xb8e4	8.3147e-01	0x6a6d
12	-7.0711e-01	0xa57e	7.0711e-01	0x5a82
13	-8.3147e-01	0x9593	5.5557e-01	0x471c
14	-9.2388e-01	0x89bf	3.8268e-01	0x30fb
15	-9.8079e-01	0x8277	1.9509e-01	0x1859

Fig 6: Table of Real & Imaginary Twiddle Factors

Because the twiddle factors are constant regardless of the input FFT data, the values for the real and imaginary twiddle factors were hardcoded into respective real and imaginary twiddle factor ROMs.

5.1.3 RAM Module

Due to the fact that the BFU could only compute the FFT for 2 points at a time, in order to determine a 32 point FFT we needed to create modules that continually stored the values output by the BFU so that they could be used by the BFU again later. To implement this we determined it was easiest to implement 4 separate basic RAM modules. Each module would contain either the real or imaginary components of either the A or B inputs to the BFU. Each RAM module would be controlled by the AGU individually.

5.1.4 Address Generation Unit (AGU)

The Address Generation Unit is the control module responsible for determining which step of the 32 point FFT calculation is to be computed, selecting the correct values from the RAMs to be calculated, and selecting the correct twiddle factors that are to be used. This module is also responsible for generating the write signals to each of the RAMs that will allow us to store the most recently computed BFU outputs. The way in which the RAM addresses and twiddle factors are generated are stated concisely in the Slade paper and is cited in figure 7:

```
/* Generate addresses for data and twiddles. */
        ja = j << 1;  // Multiply by 2 using left shift.
        jb = ja + 1;
        ja = ((ja << i) | (ja >> (5 - i))) & 0x1f; // Address A; 5 bit circular left shift
        jb = ((jb << i) | (jb >> (5 - i))) & 0x1f ; // Address B; implemented using C statements

        TwAddr = ((0xfffffff0 >> i) & 0xf) & j; // Twiddle addresses
```

Fig 7: Generation of RAM Addresses and Twiddle Factors

To ensure proper data flow of the FFT calculation a FSM was created. This FSM determined when write enabled for the RAMs should be asserted, began FFT calculations when data was correctly loaded in and informed the top level module when the calculation was completed.The transitions are shown in figure 8 and the next state logic is provided in appendix B.

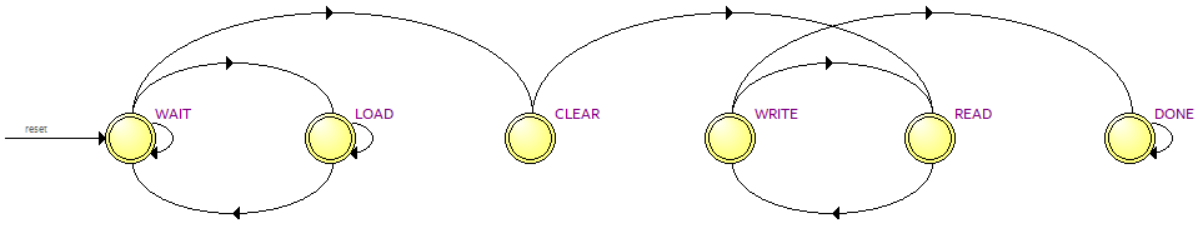


Fig 8: Address Generation Unit FSM.

5.2 FPGA SPI Module

This module was responsible for collecting data to be transformed from the MCU and sending the transformed data back to the MCU. Because we were computing a 32 point FFT on 32 bit complex data the SPI module sent and received 32 x 32 = 1024 bits for each FFT calculation performed.

5.3 FPGA Top Level Control

This is the module responsible for the control of SPI communication on the FPGA, and the initiation of the FFT calculation. Additionally, this is the part of the project where the failure described in the abstract and introduction occurred. The SPI communication was designed to send 1024 bits at a time back to the MCU and the FFT outputs were designed such that only 32 bits, 1 frequency output, were output from the FFT module at each clock cycle. As a result we created an additional FSM that was responsible for counting clock cycles allowing us to have enough time to load data into the FFT, wait for the FFT calculation to begin producing outputs, and capture the correct outputs of the FFT. The next state diagram for this FSM is shown in figure 9 and the next state logic is located Appendix B:

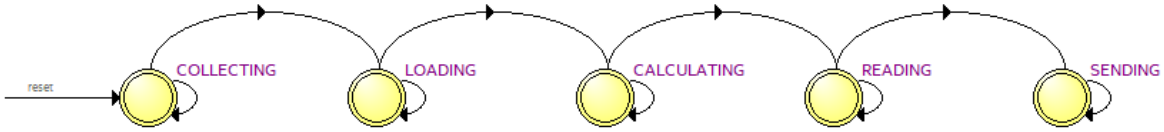


Fig 9: Top-Level Module FSM

The problem that we encountered was during the READING state the FFT top level module was not reading the correct data from the FFT output. Instead of reading the expected FFT data, only 0s for both real and imaginary magnitudes were collected and sent back to the SPI. We believe that this was due to us not creating an elegant solution for our FSM. As mentioned above, this FSM switches states according to a number of clock cycles. We determined that it would take a variable number of clock cycles to collect data, 35 clock cycles to load data, 131 clock cycles to calculate data, and 32 clock cycles to read data. Each state of the FSM included a counter that was intended to count to each of these values in each state although there must have been an error in our determined number of clock cycles or next state logic as the values that were read from the FFT RAM's during the READING state were incorrect.

6 Results

6.1 FFT Testbench Outputs & Stored Output Values

The waveforms and corresponding output values were calculated using the square wave described in the Slade paper [1].

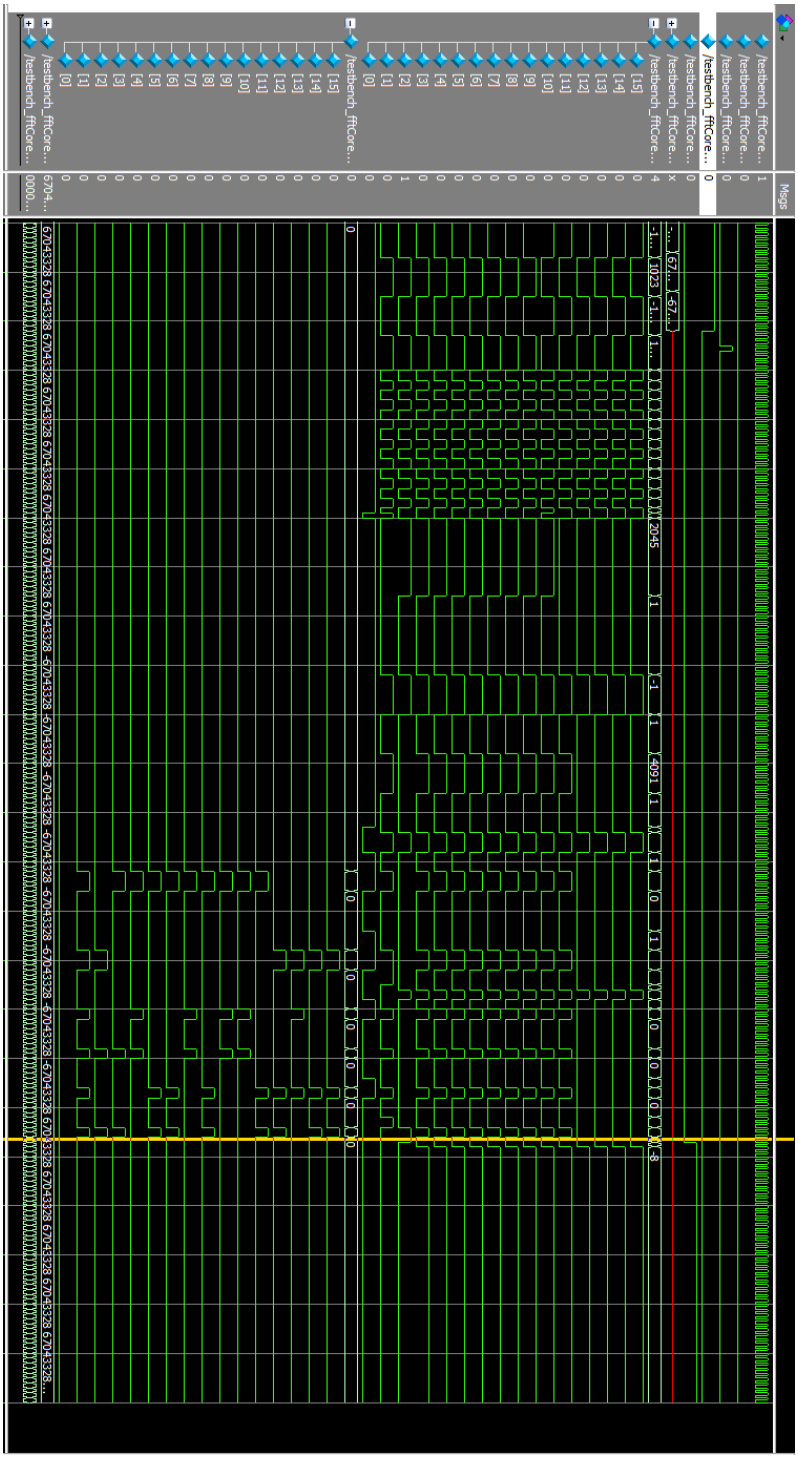


Fig 10: Testbench waveforms

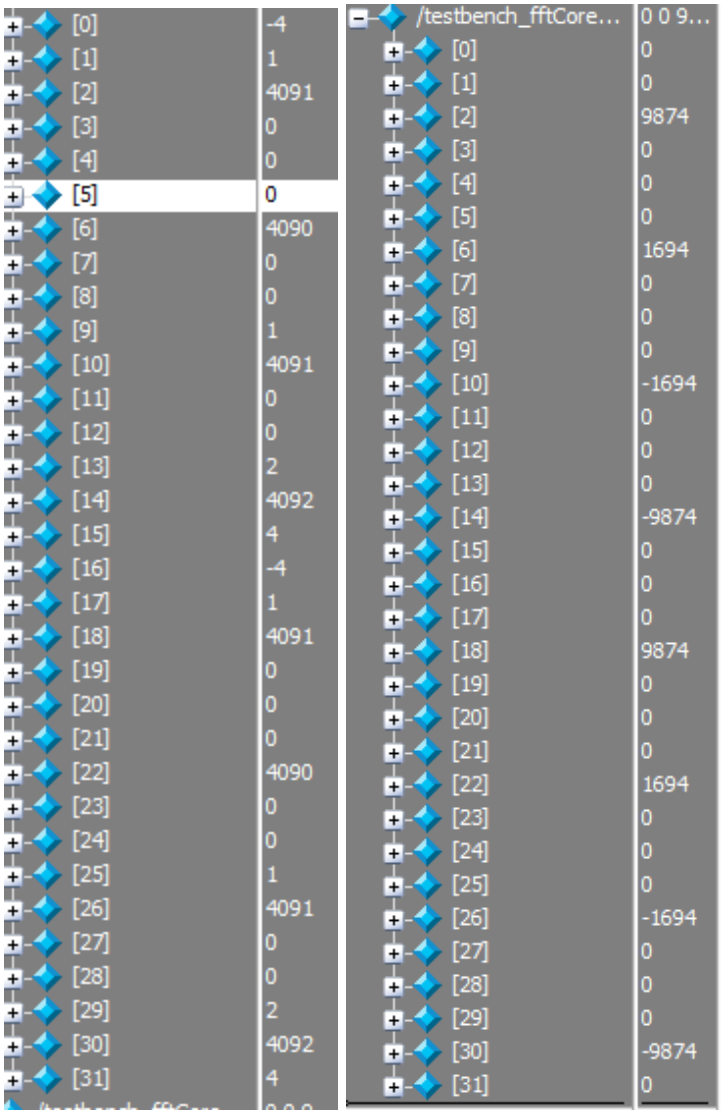


Fig 11: stored real & imaginary outputs

Because these values are what correspond to the FFT values of the square wave used in the slade paper, we are confident that the hardware we developed is a valid implementation of a 32 point FFT

Lighting Configuration

Because the MCU was unable to receive the correct FFT output data, we used the ADC data directly as described in section 4.3 above. The hat was able to display different wavelengths corresponding to different frequencies.

7 References

[1] https://web.mit.edu/6.111/www/f2017/handouts/FFTtutorial121102.pdf
[2] http://cdn.sparkfun.com/datasheets/Components/LED/APA102C.pdf

8 Bill of Materials

- Microphone
    - HiLetgo 2pcs Electret Microphone Amplifier MAX4466 Module Adjustable Gain Blue Breakout Board for Arduino
        - Amazon : Amazon.com: HiLetgo 2pcs Electret Microphone Amplifier MAX4466 Module Adjustable Gain Blue Breakout Board for Arduino : Electronics
- $8.39 for 2
- 

- LED Strip
    - Sparkfun: https://www.sparkfun.com/products/14015
    - $15.95
- 

- Cowboy Hat
    - Amazon: https://www.amazon.com/Fun-Central-Studded-Cowboy-Western
    - $10.99

-

Total Cost: $35.33

9 Appendices

Appendix A: Microcontroller Code

Main.c

```
/**
    Main: Contains main function for FFT SPI communication with FPGA.
    Receives mic analog data and converts to digital with onboard ADC.
    Sends the converted data over SPI to the FPGA and then receives back
    the fourier transformed data. Updates LEDs based on fft data

    Note: Currently does not use fft output because spi received all 0s
          from fpga due to ongoing errors in the system verilog FFT
          implementation. Instead uses data directly from the ADC to
create
          patterns on LEDs

    @file main.c
    @author Robert Gallardo, Mason Wray
*/
#include <stdio.h>
// #include <stdlib.h> // only need for abs() function used in FFT code
#include "STM32F401RE.h"

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

#define SUCCESS_LED 1 //PC
#define FAIL_LED    0 //PC
#define LOAD_PIN    5 //PB
#define DONE_PIN    3 //PB

uint16_t packets[32] = {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
};
```

```
// test fft input
uint16_t testPackets[32] = {
  0x03ff, 0x03ff, 0x03ff, 0x03ff, 0x03ff, 0x03ff, 0x03ff, 0x03ff,
  0xfc01, 0xfc01, 0xfc01, 0xfc01, 0xfc01, 0xfc01, 0xfc01, 0xfc01,
  0x03ff, 0x03ff, 0x03ff, 0x03ff, 0x03ff, 0x03ff, 0x03ff, 0x03ff,
  0xfc01, 0xfc01, 0xfc01, 0xfc01, 0xfc01, 0xfc01, 0xfc01, 0xfc01,
};

// example expected fft output
int16_t testDfft[32][2] = {
  {(int16_t)0xfffc,(int16_t)0x0000},
  {(int16_t)0x0001,(int16_t)0x0000},
  {(int16_t)0x0003,(int16_t)0x2692},
  {(int16_t)0x0000,(int16_t)0x0000},
  {(int16_t)0x0000,(int16_t)0x0000},
  {(int16_t)0x0000,(int16_t)0x0000},
  {(int16_t)0x0ffa,(int16_t)0x069e},
  {(int16_t)0x0000,(int16_t)0x0000},
  {(int16_t)0x0000,(int16_t)0x0000},
  {(int16_t)0x0001,(int16_t)0x0000},
  {(int16_t)0x0ffb,(int16_t)0xf962},
  {(int16_t)0x0000,(int16_t)0x0000},
  {(int16_t)0x0000,(int16_t)0x0000},
  {(int16_t)0x0002,(int16_t)0x0000},
  {(int16_t)0x0ffc,(int16_t)0xd96e},
  {(int16_t)0x0004,(int16_t)0x0000},
  {(int16_t)0xfffc,(int16_t)0x0000},
  {(int16_t)0x0001,(int16_t)0x0000},
  {(int16_t)0x0ffb,(int16_t)0x2692},
  {(int16_t)0x0000,(int16_t)0x0000},
  {(int16_t)0x0000,(int16_t)0x0000},
  {(int16_t)0x0000,(int16_t)0x0000},
  {(int16_t)0x0ffa,(int16_t)0x069e},
  {(int16_t)0x0000,(int16_t)0x0000},
  {(int16_t)0x0000,(int16_t)0x0000},
  {(int16_t)0x0001,(int16_t)0x0000},
  {(int16_t)0x0ffb,(int16_t)0xf962},
  {(int16_t)0x0000,(int16_t)0x0000},
  {(int16_t)0x0000,(int16_t)0x0000},
  {(int16_t)0x0002,(int16_t)0x0000},
```

```
  {(int16_t)0x0ffc,(int16_t)0xd96e},
  {(int16_t)0x0004,(int16_t)0x0000},
};

////////////////////////////////////////////////
// Main
////////////////////////////////////////////////
int main(void) {
  // Configure flash latency and set clock to run at 84 MHz
  configureFlash();
  configureClock();
  configureADC();
  // Enable GPIOA clock
  RCC->AHB1ENR.GPIOAEN = 1;
  RCC->AHB1ENR.GPIOCEN = 1;
  // debugging LEDs
  pinMode(GPIOC, SUCCESS_LED, GPIO_OUTPUT);
  pinMode(GPIOC, FAIL_LED, GPIO_OUTPUT);
  digitalWrite(GPIOC, SUCCESS_LED, 0);
  digitalWrite(GPIOC, FAIL_LED, 0);
  // Timer init
  RCC->APB1ENR |= (1 << 0); // TIM2EN
  initTIM(TIM2);

  // "clock divide" = master clock frequency / desired baud rate
  // the phase for the SPI clock is 1 and the polarity is 0
  initSPI1(1, 0, 0); // FPGA SPI
  initSPI2(1, 0, 0); // LED SPI
  // Load and done pins
  pinMode(GPIOB, LOAD_PIN, GPIO_OUTPUT);
  pinMode(GPIOB, DONE_PIN, GPIO_INPUT);
  // pinMode(GPIOB, CS_LED, GPIO_OUTPUT);
  // digitalWrite(GPIOB, CS_LED, 0);

  // Initialize colors
  setColorsRainbow();
  updateLEDs();
  startSeq(4);
  // digitalWrite(GPIOB, DO)
  while(1){
```

```
    //////////////////////
    // sample ADC
    //////////////////////
    for(int i = 0; i < 32; i++){
      uint16_t micOut = ADC->DR.DATA;
      // TEST FFT
      // packets[i] = testPackets[i]; //
      // Actual value
      packets[i] = (uint16_t) ((float)micOut/10); // round to nearest 10
mV
      // delay 125 microseconds for sampling freq of 8kHz (2x20kHz)
      delay_micros(TIM2, 125);
    }
    //////////////////////
    // send/receive to/from FPGA
    //////////////////////
    digitalWrite(GPIOB, LOAD_PIN, 1);
    for(int i = 0; i < 32; i++) {
      spiSendReceive16(SPI1, packets[i]); // TODO: check loop order and
16bit order correct
      spiSendReceive16(SPI1, 0x0000); // imaginary part
    }
    while(SPI1->SR.BSY); // Confirm all SPI transactions are complete
    digitalWrite(GPIOB, LOAD_PIN, 0); // signal to FPGA to begin FFT
    // Wait for DONE signal to be asserted by FPGA signifying that the
data is ready to be read out.
    while(!digitalRead(GPIOB, DONE_PIN));
    dfftMaxVal = 0;
    for(int i = 0; i < 32; i++) { // read in SPI data from FPGA
      // FFT CODE (using expected spi data):
      // dfft[i][0] = testDfft[i][0];
      // dfft[i][1] = testDfft[i][1];
      // FFT CODE:
      // dfft[i][0] = (int16_t)spiSendReceive16(SPI1, 0); // read real
part of ith bucket
      // dfft[i][1] = (int16_t)spiSendReceive16(SPI1, 0); // read
imaginary part of ith bucket
      // if(abs(dfft[i][0]) > dfftMaxVal || abs(dfft[i][1]) > dfftMaxVal){
```

```
      //   dfftMaxVal = abs(dfft[i][0]) > abs(dfft[i][1]) ?
abs(dfft[i][0]) : abs(dfft[i][1]);
      // }

      // DIRECT ADC CODE:
      dfft[i][0] = packets[i];
      dfft[i][1] = 0;
    }

    ////////////////////
    // update LEDs w/ SPI
    ////////////////////
    updateBrightness(); // update brightness based on dfft
    updateLEDs();
    delay_millis(TIM2, 200);
  } // END WHILE(1)

  return 0;

} // END MAIN(VOID)
```

ADC.h

```
// STM32F401RE_ADC.h
// Header for ADC functions

#ifndef STM32F4_ADC_H
#define STM32F4_ADC_H

#include <stdint.h>

////////////////////////////////////////////////////////////////////////////////
/////
// Definitions
////////////////////////////////////////////////////////////////////////////////
/////
```

```
#define __IO volatile

// Base addresses for GPIO ports
#define ADC_BASE (0x40012000UL) // base address of ADC

////////////////////////////////////////////////////////////////////////////////
/////
// Bitfield structs
////////////////////////////////////////////////////////////////////////////////
/////

typedef struct {
  __IO uint32_t AWD     :1;
  __IO uint32_t EOC     :1;
  __IO uint32_t JEOC    :1;
  __IO uint32_t JSTRT   :1;
  __IO uint32_t STRT    :1;
  __IO uint32_t OVR     :1;
  __IO uint32_t         :26;
} ADC_SR_bits;

typedef struct {
  __IO uint32_t AWDCH : 5;
  __IO uint32_t EOCIE : 1;
  __IO uint32_t AWDIE : 1;
  __IO uint32_t JEOCIE : 1;
  __IO uint32_t SCAN   : 1;
  __IO uint32_t       : 15;
  __IO uint32_t RES   : 2;
  __IO uint32_t OVRIE : 1;
  __IO uint32_t       : 5;
} ADC_CR1_bits;

typedef struct {
  __IO uint32_t ADON    : 1;
  __IO uint32_t CONT    : 1; // 0 for single conversion 1 for continuous
  __IO uint32_t         : 6;
  __IO uint32_t DMA     : 1; // 0 for DMA disabled ...
  __IO uint32_t DDS     : 1;
  __IO uint32_t EOCS    : 1;
```

```
  __IO uint32_t ALIGN   : 1; // 0 right alignment
  __IO uint32_t         : 4;
  __IO uint32_t         : 7;
  __IO uint32_t         : 1; // reserved bit 23
  __IO uint32_t EXTSEL  : 4;
  __IO uint32_t EXTEN   : 2; // Ext trigger enable for reg channels
  __IO uint32_t SWSTART : 1;
  __IO uint32_t         : 1;
} ADC_CR2_bits;

typedef struct {
  __IO uint32_t SMP0 : 3;
  __IO uint32_t      : 29;
} ADC_SMPR2_bits;

typedef struct {
  __IO uint32_t HT : 12;
  __IO uint32_t    : 20;
} ADC_HTR_bits;

typedef struct {
  __IO uint32_t LT : 12;
  __IO uint32_t    : 20;
} ADC_LTR_bits;

typedef struct {
  __IO uint32_t   : 20;
  __IO uint32_t L : 4;
  __IO uint32_t   : 8;
} ADC_SQR1_bits;

typedef struct {
  __IO uint32_t SQ1 : 5;
  __IO uint32_t L : 4;
  __IO uint32_t   : 8;
} ADC_SQR3_bits;

typedef struct {
  __IO uint32_t DATA : 16;
  __IO uint32_t      : 16;
```

```
} ADC_DR_bits;

typedef struct {
  __IO uint32_t : 16; // reserved and kept at reset value
  __IO uint32_t ADCPRE : 2; // ADC prescaler to determine clk freq
  __IO uint32_t        : 14;
} ADC_CCR_bits;

typedef struct {
  __IO ADC_SR_bits SR;       /*!< ADC status register,   Address offset:
0x00 */
  __IO ADC_CR1_bits CR1;    /*!< ADC control register 1, Address offset:
0x04 */
  __IO ADC_CR2_bits CR2;    /*!< ADC control register 2, Address offset:
0x08 */
  __IO uint32_t SMPR1;     /*!< ADC sample time reg 1, Address offset: 0x0C
*/
  __IO ADC_SMPR2_bits SMPR2;     /*!< ADC sample time reg 1, Address
offset: 0x10 */
  __IO uint32_t JOFR1;     /* Address offset: 0x14 */
  __IO uint32_t JOFR2;     /* Address offset: 0x18 */
  __IO uint32_t JOFR3;     /* Address offset: 0x1C */
  __IO uint32_t JOFR4;     /* Address offset: 0x20 */
  __IO ADC_HTR_bits HTR;       /*!< ADC watchdog higher threshold register,
Address offset: 0x24 */
  __IO ADC_LTR_bits LTR;       /*!< ADC watchdog higher threshold register,
Address offset: 0x28 */
  __IO ADC_SQR1_bits SQR1;       /*!< ADC reg seq register 1,   Address
offset: 0x2C */
  __IO uint32_t SQR2;      /* Address offset: 0x30 */
  __IO ADC_SQR3_bits SQR3;       /*!< ADC reg seq register 1,   Address
offset: 0x34 */
  __IO uint32_t JSQR;      /* Address offset: 0x38 */
  __IO uint32_t JDR1;      /* Address offset: 0x3C */
  __IO uint32_t JDR2;      /* Address offset: 0x40 */
  __IO uint32_t JDR3;      /* Address offset: 0x44 */
  __IO uint32_t JDR4;      /* Address offset: 0x48 */
  __IO ADC_DR_bits DR;     /*!< ADC regular data register, Address offset:
0x4C */
```

```
  __IO ADC_CCR_bits CCR;    /*!< ADC common control register, Address
offset: 0x04 relative to ADC1 base addr + 0x300 */
} ADC_TypeDef;

#define ADC ((ADC_TypeDef *) ADC_BASE)

////////////////////////////////////////////////////////////////////////////
/////
// Function prototypes
////////////////////////////////////////////////////////////////////////////
/////

void configureADC();

#endif
```

ADC.c

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

#include "STM32F401RE_ADC.h"
#include "STM32F401RE_RCC.h"
#include "STM32F401RE_GPIO.h"

void configureADC() {
    RCC->AHB1ENR.GPIOAEN = 1; // Enable GPIOA clock domain
    pinMode(GPIOA, 1, GPIO_ANALOG); // PA1, ADC1_IN1
    RCC->APB2ENR |= (1 << 8); // Turn on ADC1 clock domain (ADC1EN bit in
APB2ENR)
    ADC->SQR1.L = 0b0000; // 1 conversion for regular channel sequence
length
    ADC->SQR3.SQ1 = 0b00001; // Select channel 1 for 1st register
    ADC->CR1.SCAN = 0; // Disable scan mode
    ADC->CR1.RES = 0b00; // 12-bit resolution, 12+3=15 ADCCLK cycles
    ADC->CR2.ALIGN = 0; // 1:left/0:RIGHT alignment
    ADC->CR2.CONT = 1; // Continuous mode enabled
    ADC->CR2.EXTEN = 0b00; // trigger detection disabled (Set to ... to
enable)
```

```
    // 11.3.1 ADC on-off control
    ADC->CR2.ADON = 1; // wake up ADC from power-down mode
    ADC->CR2.SWSTART = 1; // begin regular conversions
}
```

APA102.h

```
#ifndef STM32F4_APA102_H
#define STM32F4_APA102_H

#include <stdint.h> // Include stdint header

////////////////////////////////////////////////////////////////////////////
/////
// Definitions
////////////////////////////////////////////////////////////////////////////
/////
#define NUM_LEDS 60
#define NUM_REPEATS 1 // repeat pattern twice

uint8_t colors[NUM_LEDS][3]; // array of LED colors (r,g,b)
uint8_t brightness[NUM_LEDS]; // array of LED brightnesses
int16_t dfft[32][2]; // output from FFT 32x(signed 16 bit real, signed 16
bit imaginary)
int16_t dfftMaxVal; // max dfft value per sample batch to normalize values

/* Sends the specified color and brightness to an LED over SPI2
 *   -- global: 5 bit brightness value (leftmost 3 bits become 1s and have
no effect)
 *   -- r,g,b: 8 bit red, green, blue values respectively */
void ledFrame(uint8_t global, uint8_t r, uint8_t g, uint8_t b);

/* Sends the start frame of 32 0s to tell the LED strip to begin reading
data */
void startFrame(void);

/* Sends the end frame of 32 1s to tell the LED strip to stop reading data
*/
```

```
void endFrame(void);

/* Set global 'colors' array to make LEDS rainbow */
void setColorsRainbow();

/* Update LEDs based on the brightness and colors arrays*/
void updateLEDs();

/* update 'brightness' array based on either fft output or
 * direct ADC data */
void updateBrightness();

/* Sequentially brightens LEDs from first to last
 * -- num: number of times to repeat sequence
 * Notes: LED color is determined by values in global 'colors' array */
void startSeq(int num);

#endif
```

APA102.c

```
// STM32F401RE_APA102.c
// Source code for APA102 LED strip functions

#include "STM32F401RE_RCC.h"
#include "STM32F401RE_GPIO.h"
#include "STM32F401RE_SPI.h"
#include "STM32F401RE_TIM.h"
#include "STM32F401RE_APA102.h"

void ledFrame(uint8_t global, uint8_t r, uint8_t g, uint8_t b) {
  // combine start sequence 111 with 5 bit global (brightness)
  uint8_t firstByte = (0b111 << 5) | global;
  spiSendReceive(SPI2, firstByte);
  spiSendReceive(SPI2, b);
  spiSendReceive(SPI2, g);
  spiSendReceive(SPI2, r);
}

void startFrame() {
```

```
    for(int i = 0; i < 4; i++) {
      spiSendReceive(SPI2, 0x00);
    }
}

void endFrame() {
    for(int i = 0; i < 4; i++) {
      spiSendReceive(SPI2, 0xFF);
    }
}

void setColorsRainbow() {
    int lenSection = NUM_LEDS / NUM_REPEATS;
    float lenChunk = lenSection / 6; // length of each rainbow color chunk
    uint8_t r = 0xFF;
    uint8_t g = 0xFF;
    uint8_t b = 0xFF;
    for(int i = 0; i < NUM_REPEATS; i++) {
        for(int j = 0; j < lenSection; j++) {
            float k = j % lenSection; // index within repeated sections
            // Set color
            if (k < lenChunk) {
                // red --> red green
                r = 0xFF;
                g = (uint8_t)((k / lenChunk) * 0xFF);
                b = 0x00;
            } else if (k >= lenChunk && k < 2 * lenChunk) {
                // red green --> green
                r = (uint8_t)(((2 * lenChunk - k) / lenChunk) * 0xFF);
                g = 0xFF;
                b = 0x00;
            } else if (k >= 2 * lenChunk && k < 3 * lenChunk) {
                // green --> green blue
                r = 0x00;
                g = 0xFF;
                b = (uint8_t)(((k - 2 * lenChunk) / lenChunk) * 0xFF);
            } else if (k >= 3 * lenChunk && k < 4 * lenChunk) {
                // green blue --> blue
                r = 0x00;
                g = (uint8_t)(((4 * lenChunk - k) / lenChunk) * 0xFF);
```

```
                b = 0xFF;
            } else if (k >= 4 * lenChunk && k < 5 * lenChunk) {
                // blue --> blue red
                r = (uint8_t)(((k - 5 * lenChunk) / lenChunk) * 0xFF);
                g = 0x00;
                b = 0xFF;
            } else if (k >= 5 * lenChunk && k <= 6 * lenChunk) {
                // blue red --> red
                r = 0xFF;
                g = 0x00;
                b = (uint8_t)(((6 * lenChunk - k) / lenChunk) * 0xFF);
            } else {
                r = 0xFF;
                g = 0xFF;
                b = 0xFF;
            }
            colors[i*lenSection + j][0] = r;
            colors[i*lenSection + j][1] = g;
            colors[i*lenSection + j][2] = b;
        }
    }
}

void updateLEDs() {
    startFrame();
    for(int i = 0; i < NUM_LEDS; i++) {
        ledFrame(brightness[i], colors[i][0], colors[i][1], colors[i][2]);
    }
    endFrame();
}

void updateBrightness() {
    for(int i = 0; i < 2; i++) {
        for(int j = 0; j < 30; j++) {
            // FFT CODE:
            // float realNormed = ((float)(dfft[j][0]))/dfftMaxVal;
            // float imagNormed = ((float)(dfft[j][1]))/dfftMaxVal;
            // set brightness to be proportionally related to frequency
amplitude in jth fft bucket
```

```
            // brightness[30*i+j] =
(uint8_t)(((float)(realNormed*realNormed + imagNormed*imagNormed))/2 *
31.0);

            // DIRECT ADC CODE:
            // turn on LED if its corresponding normalized sample is
higher than 50% brightness (15)
            brightness[30*i+j] = (uint8_t)(((float)(dfft[j][0]-150)) *
(31.0/100.0)) > 15 ? 0x00 : 0x1F;
        }
    }
}

/* Sequentially brightens LEDs from first to last 'num' times
 * LED color is determined by global 'colors' array */
void startSeq(int num) {
    for(int x = 0; x < num; x++) { // repeat seq num times
        for(int k = 0; k < NUM_LEDS/2; k++) {
            // set brightness for all LEDs on round k
            for(int i = 0; i < 2; i++) {
                for(int j = 0; j < 30; j++) {
                    // fully light all LEDs before the kth LED
                    brightness[30*i+j] = (j < k) ? 0x05 : 0x01;
                }
            }
            updateLEDs();
            delay_millis(TIM2, 20);
        }
        delay_millis(TIM2, 100);
    }
}
```

SPI.h

```
// STM32F401RE_SPI.h
// Header for SPI functions

#ifndef STM32F4_SPI_H
#define STM32F4_SPI_H
```

```c
#include <stdint.h> // Include stdint header

//////////////////////////////////////////////////////////////////////////////
//////
// Definitions
//////////////////////////////////////////////////////////////////////////////
//////

#define SPI1_BASE (0x40013000UL)
#define SPI2_BASE (0x40003800UL)
#define __IO volatile

//////////////////////////////////////////////////////////////////////////////
//////
// Bitfield structs
//////////////////////////////////////////////////////////////////////////////
//////

typedef struct {
  __IO uint32_t CPHA        : 1;
  __IO uint32_t CPOL        : 1;
  __IO uint32_t MSTR        : 1;
  __IO uint32_t BR          : 3;
  __IO uint32_t SPE         : 1;
  __IO uint32_t LSBFIRST    : 1;
  __IO uint32_t SSI         : 1;
  __IO uint32_t SSM         : 1;
  __IO uint32_t RXONLY      : 1;
  __IO uint32_t DFF         : 1;
  __IO uint32_t CRCNEXT     : 1;
  __IO uint32_t CRCEN       : 1;
  __IO uint32_t BIDIOE      : 1;
  __IO uint32_t BIDIMODE    : 1;
  __IO uint32_t             : 16;
} SPI_CR1_bits;

typedef struct {
  __IO uint32_t RXDMAEN     : 1;
  __IO uint32_t TXDMAEN     : 1;
  __IO uint32_t SSOE        : 1;
```

```
  __IO uint32_t             : 1;
  __IO uint32_t FRF         : 1;
  __IO uint32_t ERRIE       : 1;
  __IO uint32_t RXNEIE      : 1;
  __IO uint32_t TXEIE       : 1;
  __IO uint32_t             : 24;
} SPI_CR2_bits;

typedef struct {
  __IO uint32_t RXNE        : 1;
  __IO uint32_t TXE         : 1;
  __IO uint32_t CHSIDE      : 1;
  __IO uint32_t UDR         : 1;
  __IO uint32_t CRCERR      : 1;
  __IO uint32_t MODF        : 1;
  __IO uint32_t OVR         : 1;
  __IO uint32_t BSY         : 1;
  __IO uint32_t FRE         : 1;
  __IO uint32_t DFF         : 1;
  __IO uint32_t CRCNEXT     : 1;
  __IO uint32_t CRCEN       : 1;
  __IO uint32_t BIDIOE      : 1;
  __IO uint32_t BIDIMODE    : 1;
  __IO uint32_t             : 16;
} SPI_SR_bits;

typedef struct {
  __IO uint32_t DR  : 16;
  __IO uint32_t     : 16;
} SPI_DR_bits;


typedef struct {
  __IO SPI_CR1_bits CR1;        /*!< SPI control register 1 (not used in
I2S mode),      Address offset: 0x00 */
  __IO SPI_CR2_bits CR2;        /*!< SPI control register 2,
Address offset: 0x04 */
  __IO SPI_SR_bits SR;         /*!< SPI status register,
Address offset: 0x08 */
```

```
  __IO SPI_DR_bits DR;          /*!< SPI data register,
Address offset: 0x0C */
  __IO uint32_t CRCPR;       /*!< SPI CRC polynomial register (not used in
I2S mode), Address offset: 0x10 */
  __IO uint32_t RXCRCR;      /*!< SPI RX CRC register (not used in I2S
mode),          Address offset: 0x14 */
  __IO uint32_t TXCRCR;      /*!< SPI TX CRC register (not used in I2S
mode),          Address offset: 0x18 */
  __IO uint32_t I2SCFGR;     /*!< SPI_I2S configuration register,
Address offset: 0x1C */
  __IO uint32_t I2SPR;       /*!< SPI_I2S prescaler register,
Address offset: 0x20 */
} SPI_TypeDef;

// Pointers to GPIO-sized chunks of memory for each peripheral
#define SPI1 ((SPI_TypeDef *) SPI1_BASE)
#define SPI2 ((SPI_TypeDef *) SPI2_BASE)

////////////////////////////////////////////////////////////////////////////
/////
// Function prototypes
////////////////////////////////////////////////////////////////////////////
/////

/* Enables the SPI peripheral and intializes its clock speed (baud rate),
polarity, and phase.
 *    -- clkdivide: (0x01 to 0xFF). The SPI clk will be the master clock /
clkdivide.
 *    -- cpol: clock polarity (0: inactive state is logical 0, 1: inactive
state is logical 1).
 *    -- cpha: clock phase (1: data changed on leading edge of clk and
captured on next edge,
 *          0: data captured on leading edge of clk and changed on next
edge)
 * Note: the SPI mode register is set with the following unadjustable
settings:
 *    -- Master mode
 *    -- Fixed peripheral select
 *    -- Chip select lines directly connected to peripheral device
 *    -- Mode fault detection enabled
```

```
 *     -- WDRBT disabled
 *     -- LLB disabled
 *     -- PCS = 0000 (Peripheral 0 selected), means NPCS[3:0] = 1110
 * Refer to the datasheet for more low-level details. */
void initSPI1(uint32_t clkdivide, uint32_t cpol, uint32_t ncpha);
void initSPI2(uint32_t clkdivide, uint32_t cpol, uint32_t ncpha);

/* 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(SPI_TypeDef * SPIx, uint8_t send);

/* 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(SPI_TypeDef * SPIx, uint16_t send);

#endif
```

SPI.c

```
// 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(SPI_TypeDef * SPIx, uint32_t br, uint32_t cpol, uint32_t
cpha) {
    SPIx->CR1.BR = br;       // Set the clock divisor
    SPIx->CR1.CPOL = cpol;   // Set the polarity
    SPIx->CR1.CPHA = cpha;   // Set the phase
    SPIx->CR1.LSBFIRST = 0; // Set least significant bit first
    // SPIx->CR1.DFF = 0;       // Set data format to 8 bits
    SPIx->CR1.SSM = 0;       // Turn off software slave management
    SPIx->CR2.SSOE = 1;      // Set the NSS pin to output mode
    SPIx->CR1.MSTR = 1;      // Put SPI in master mode
    SPIx->CR1.SPE = 1;       // Enable SPI
}

void initSPI1(uint32_t br, uint32_t cpol, uint32_t cpha) {
    // Turn on GPIOA and GPIOB clock domains (GPIOAEN and GPIOBEN bits in
AHB1ENR)
    RCC->AHB1ENR.GPIOAEN = 1;
    RCC->AHB1ENR.GPIOBEN = 1;

    RCC->APB2ENR |= (1 << 12); // Turn on SPI1 clock domain (SPI1EN bit in
APB2ENR)
    // Initially assigning SPI pins
    pinMode(GPIOA, 5, GPIO_ALT); // PA5, Arduino D13, SPI1_SCK
    pinMode(GPIOA, 6, GPIO_ALT); // PA6, Arduino D12, SPI1_MISO
    pinMode(GPIOA, 7, GPIO_ALT); // PA7, Arduino D11, SPI1_MOSI
    pinMode(GPIOB, 6, GPIO_OUTPUT); // PB6, Arduino D10, Manual CS
    // Set output speed type to high for SCK
    GPIOA->OSPEEDR |= (0b11 << 2*5);
    // Set to AF05 for SPI alternate functions
    GPIOA->AFRL |= (1 << 18) | (1 << 16);
    GPIOA->AFRL |= (1 << 22) | (1 << 20);
    GPIOA->AFRL |= (1 << 26) | (1 << 24);
    GPIOA->AFRL |= (1 << 30) | (1 << 28);

    SPI1->CR1.DFF = 1; // Set data format to 16 bits
    spiInit(SPI1, br, cpol, cpha);
}

void initSPI2(uint32_t br, uint32_t cpol, uint32_t cpha) {
```

```
    // Turn on GPIOA and GPIOB clock domains (GPIOAEN and GPIOBEN bits in
AHB1ENR)
    RCC->AHB1ENR.GPIOAEN = 1;
    RCC->AHB1ENR.GPIOBEN = 1;

    RCC->APB1ENR |= (1 << 14); // Turn on SPI1 clock domain (SPI2EN bit in
APB1ENR)
    // Initially assigning SPI pins
    pinMode(GPIOB, 10, GPIO_ALT); // PB10, Arduino D13, SPI1_SCK
    pinMode(GPIOB, 15, GPIO_ALT); // PA7, Arduino D11, SPI1_MOSI
    // Set output speed type to high for SCK
    GPIOB->OSPEEDR |= (0b11 << 2*10);
    // Set to AF05 for SPI alternate functions
    GPIOB->AFRH |= (1 << 10) | (1 << 8);  // PB10 SCK
    GPIOB->AFRH |= (1 << 30) | (1 << 28); // PB15 MOSI

    SPI2->CR1.DFF = 0; // Set data format to 8 bits
    spiInit(SPI2, br, cpol, cpha);
}

/* 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(SPI_TypeDef * SPIx, uint8_t send) {
    while(!(SPIx->SR.TXE)); // Wait until the transmit buffer is empty
    SPIx->DR.DR = send; // Transmit the character over SPI
    while(!(SPIx->SR.RXNE)); // Wait until data has been received
    uint8_t rec = SPIx->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(SPI_TypeDef * SPIx, uint16_t send) {
    digitalWrite(GPIOB, 6, 0);
    SPIx->CR1.SPE = 1;
    SPIx->DR.DR = send;
```

```
    while(!(SPIx->SR.RXNE));
    uint16_t rec = SPIx->DR.DR;

    SPIx->CR1.SPE = 0;
    digitalWrite(GPIOB, 6, 1);

    return rec;
}
```

Appendix B: System Verilog

FFT Top-Level Module

```
module FinalMWRG(input logic clk, reset, sck, sdi, spiload,
                         output logic done, sdo);

    //define logic variables for the input and output data
    logic [1023:0] fftinput, fftoutput;

    //placeholder values
    logic [1023:0] fftDataIn, fftDataOut;

    //create iterator logic for each state to count number of clock cycles
    logic [5:0] ittLoad = 0; //LOAD State
    logic [7:0] ittCalc = 0; //CALCULATION State
    logic [5:0] ittRead = 0; //READ State

    //logic for the data words from input
    logic [31:0] wd, tempwd;
    //logic for the ourputs from fft
    logic [15:0] outputReal, outputImag;

    logic load, start;

    //instantiate sub-modules
    fft_spi spi(sck, sdi, sdo, done, fftinput, fftoutput);
    fft_core core(clk, reset, start, load, wd, outputReal, outputImag,
done);
```

```
    //define states for the FSM
    typedef enum logic [2:0] {COLLECTING, LOADING, CALCULATING, READING,
SENDING, NULL} statetype;
    statetype currentState, nextState;

    always_ff @(posedge clk) begin

        //always switch to next state
        currentState = nextState;

        //when spiload is set high begin
        if(spiload) begin
            nextState = COLLECTING;
            fftDataIn = fftinput;

            end


        case(currentState)
            //state for collecting input data over SPI
            COLLECTING:
                begin
                load = 0;
                start = 0;
                if(spiload)begin
                    nextState = COLLECTING;
                    end
                else
                    begin
                    nextState = LOADING;
                    end
                end

            //state for loading input data into the RAM
            LOADING:
            begin
                load = 1;
                start = 0;
                if(ittLoad < 32) begin
```

```
                nextState = LOADING;
                ittLoad = ittLoad + 1;
            end
            else begin
                ittLoad = 0;
                nextState = CALCULATING;
            end
        end

        //state for calculating the FFT
        CALCULATING:
        begin
            load = 0;
            start = 1;
            if(ittCalc < 131)begin
                nextState =  CALCULATING;
                ittCalc = ittCalc + 1;
            end
            else begin
                ittCalc = 0;
                nextState = READING;

            end
        end

        //state for reading the fft ourputs into the SPI output
register
        READING:
        begin
            load = 0;
            start = 0;
            if(ittRead < 32)begin
            nextState = READING;
            ittRead = ittRead + 1;
            end
            else begin
                ittRead = 0;
                nextState = SENDING;

            end
```

```
        end

        //state for sending output data over SPI
        SENDING:
        begin
            load = 0;
            start = 0;
            nextState = SENDING;
        end

        //state where nothing is happening
        NULL:
        begin
            if(spiload) nextState = COLLECTING;
            else begin

                load = 0;
                start = 0;

            end
            //nextState <= NULL;
        end

        default:
        begin
            load = 0;
            start = 0;
            //nextState <= NULL;
        end
    endcase

    case(ittRead)
        0: fftDataOut[31:0] = {outputReal, outputImag};
        1: fftDataOut[63:32] = {outputReal, outputImag};
        2: fftDataOut[95:64] = {outputReal, outputImag};
        3: fftDataOut[127:96] = {outputReal, outputImag};
        4: fftDataOut[159:128] = {outputReal, outputImag};
        5: fftDataOut[191:160] = {outputReal, outputImag};
        6: fftDataOut[223:192] = {outputReal, outputImag};
        7: fftDataOut[255:224] = {outputReal, outputImag};
```

```
            8: fftDataOut[287:256] = {outputReal, outputImag};
            9: fftDataOut[319:288] = {outputReal, outputImag};
            10: fftDataOut[351:320] = {outputReal, outputImag};
            11: fftDataOut[383:352] = {outputReal, outputImag};
            12: fftDataOut[415:384] = {outputReal, outputImag};
            13: fftDataOut[447:416] = {outputReal, outputImag};
            14: fftDataOut[479:448] = {outputReal, outputImag};
            15: fftDataOut[511:480] = {outputReal, outputImag};
            16: fftDataOut[543:512] = {outputReal, outputImag};
            17: fftDataOut[575:544] = {outputReal, outputImag};
            18: fftDataOut[607:576] = {outputReal, outputImag};
            19: fftDataOut[639:608] = {outputReal, outputImag};
            20: fftDataOut[671:640] = {outputReal, outputImag};
            21: fftDataOut[703:672] = {outputReal, outputImag};
            22: fftDataOut[735:704] = {outputReal, outputImag};
            23: fftDataOut[767:736] = {outputReal, outputImag};
            24: fftDataOut[799:768] = {outputReal, outputImag};
            25: fftDataOut[831:800] = {outputReal, outputImag};
            26: fftDataOut[863:832] = {outputReal, outputImag};
            27: fftDataOut[895:864] = {outputReal, outputImag};
            28: fftDataOut[927:896] = {outputReal, outputImag};
            29: fftDataOut[959:928] = {outputReal, outputImag};
            30: fftDataOut[991:960] = {outputReal, outputImag};
            31: fftDataOut[1023:992] = {outputReal, outputImag};
            default: ;
        endcase

        case(ittLoad)
            0: tempwd = fftinput[31:0];
            1: tempwd = fftinput[63:32];
            2: tempwd = fftinput[95:64];
            3: tempwd = fftinput[127:96];
            4: tempwd = fftinput[159:128];
            5: tempwd = fftinput[191:160];
            6: tempwd = fftinput[223:192];
            7: tempwd = fftinput[255:224];
            8: tempwd = fftinput[287:256];
            9: tempwd = fftinput[319:288];
            10: tempwd = fftinput[351:320];
            11: tempwd = fftinput[383:352];
```

```
            12: tempwd = fftinput[415:384];
            13: tempwd = fftinput[447:416];
            14: tempwd = fftinput[479:448];
            15: tempwd = fftinput[511:480];
            16: tempwd = fftinput[543:512];
            17: tempwd = fftinput[575:544];
            18: tempwd = fftinput[607:576];
            19: tempwd = fftinput[639:608];
            20: tempwd = fftinput[671:640];
            21: tempwd = fftinput[703:672];
            22: tempwd = fftinput[735:704];
            23: tempwd = fftinput[767:736];
            24: tempwd = fftinput[799:768];
            25: tempwd = fftinput[831:800];
            26: tempwd = fftinput[863:832];
            27: tempwd = fftinput[895:864];
            28: tempwd = fftinput[927:896];
            29: tempwd = fftinput[959:928];
            30: tempwd = fftinput[991:960];
            31: tempwd = fftinput[1023:992];
            default tempwd = 32'b0;
        endcase
    end

    assign wd = tempwd;
    assign fftoutput = fftDataOut;
endmodule
```

FFT SPI Module

```
module fft_spi(input  logic sck,
               input  logic sdi,
               output logic sdo,
               input  logic done,
               output logic [1023:0] fftinput,
               input  logic [1023:0] fftoutput);

    logic          sdodelayed, wasdone;
    logic [1023:0] fftoutputcaptured;
```

```
    // assert load
    // apply 1024 sclks to shift in key and fftinput, starting with
fftinput[0]
    // then deassert load, wait until done
    // then apply 1024 sclks to shift out fftoutput, starting with
fftoutput[0]
    always_ff @(posedge sck)
        if (!wasdone)  {fftoutputcaptured, fftinput} = {fftoutput,
fftinput[1022:0], sdi};
        else           {fftoutputcaptured, fftinput} =
{fftoutputcaptured[1022:0], fftinput, sdi};

    // sdo should change on the negative edge of sck
    always_ff @(negedge sck) begin
        wasdone = done;
        sdodelayed = fftoutputcaptured[1022];
    end

    // when done is first asserted, shift out msb before clock edge
    assign sdo = (done & !wasdone) ? fftoutput[1023] : sdodelayed;
endmodule
```

fft_core Module

```
module fft_core(input logic clk, reset, start, load,
                    input logic signed [31:0] wd,
                    output logic signed [15:0] outputReal, outputImag,
                    output logic done);

    logic wen0, wen1, clear, rdsel;
    logic [3:0] k;
    logic [4:0] adr0a, adr0b, adr1a, adr1b;
    logic signed [15:0] twiddleReal, twiddleImag;
    logic [31:0] ain, bin, aout, bout, rd0a, rd1a, rd0b, rd1b, wda, wdb;

    // To load the data in
    assign wda = load ? wd : aout;
    assign wdb = load ? wd : bout;
```

```
    // Substantiates the AGU
    AddressGenerationUnit agu(clk, reset, start, load, done, wen0, wen1,
clear, rdsel, k, adr0a, adr0b);

    assign adr1a = adr0a;

   assign adr1b = adr0b;

    RealTwiddles twReal(clk, k, twiddleReal);
    ImagTwiddles twImag(clk, k,  twiddleImag);


    RAM r0r(clk, wen0, adr0a, adr0b, wda[31:16], wdb[31:16], rd0a[31:16],
rd0b[31:16]);
    RAM r0i(clk, wen0, adr0a, adr0b, wda[15:0], wdb[15:0], rd0a[15:0],
rd0b[15:0]);
    RAM r1r(clk, wen1, adr1a, adr1b, wda[31:16], wdb[31:16], rd1a[31:16],
rd1b[31:16]);
    RAM r1i(clk, wen1, adr1a, adr1b, wda[15:0], wdb[15:0], rd1a[15:0],
rd1b[15:0]);

    // Muxes for which bank to read
    assign ain = rdsel ? rd1a : rd0a;
    assign bin = rdsel ? rd1b : rd0b;

    ButterflyUnit bf(ain[31:16], ain[15:0], bin[31:16], bin[15:0],
                 twiddleReal, twiddleImag,
                 aout[31:16], bout[31:16], aout[15:0], bout[15:0]);

    // Assigns output from ain
    assign outputReal = ain[31:16];
    assign outputImag = bin[15:0];

endmodule
```

Address Generation Unit Module

```
module AddressGenerationUnit(input logic clk, reset, start, load,
              output logic done, we1, we2, clear, memSelect,
              output logic [3:0] k,
```

```
            output logic [4:0] addressA, addressB);

    //define states for the FSM's
    typedef enum logic [2:0] {WAIT, LOAD, READ, WRITE, CLEAR, DONE}
statetype;  // Might need a clear state in between read and write
    statetype currentState, nextState;


    logic [5:0] counter;
    always_ff @(posedge clk)
        begin
            if (reset) counter <= 0;
            else counter <= counter + 1;
        end

    //define loop incrementors
    logic [2:0] i, inext;
    logic [4:0] j, jnext, jshift;

    // Next state fsm
    always_ff @(posedge clk, posedge reset)
        if (reset) begin
            currentState <= WAIT;
            i <= 0;
            j <= 0;
        end else if (clear) begin
            currentState <= nextState;
            i <= 0;
            j <= 0;
        end else begin
            currentState <= nextState;
            i <= inext;
            j <= jnext;
        end

    //next state logic
    always_comb
        case(currentState)

            //state FFT is in when waiting for start signal
```

```
        WAIT: if (start) nextState <= CLEAR;
                else if (load) nextState <= LOAD;
                else nextState <= WAIT;

        //state FFT is in when data is being loaded in
        LOAD: if (counter < 32) nextState <= LOAD;
                else begin
                    nextState <= WAIT;
                end

        //state when RAM data is being read
        READ: nextState <= WRITE;

        //state when RAM data is being written
        WRITE: if (i == 4 && j == 15) nextState <= DONE;
                 else nextState <= READ;

        //state when clear input is asserted
        CLEAR:
            nextState <= READ;

        //finished FFT calculation. Stay here.
        DONE: nextState <= DONE;

        default: nextState <= WAIT;
    endcase


//increment based on current state, i.e. calc and frame number
always_comb
    case(currentState)

        //ONLY INCREMEMNTS ON WRITE STATE
        WRITE:
            begin
                //if finished operations
                if(j == 15) begin
                    jnext = 0;
                    inext = i + 1;
```

```
                    //othersiwe continue
                    end else begin
                        jnext = j + 1;
                        inext = i;
                    end
                end
            default:
                begin
                    inext = i;
                    jnext = j;
                end
        endcase

    logic [4:0] counterRev;
    bitreverse br1(counter[4:0], counterRev);

    // Assign values of addresses according to AGU pseudocode in Table 3
of the Slade paper
    assign jshift = j << 1;

    //addressA take the value of bit reversed count when 1 and slade value
otherwisep.14
    assign addressA = (currentState == LOAD) ? counterRev : ((jshift << i)
| (jshift >> (5 - i))) & 8'h1f;

    //addressB does the same as A but clk cycle before
    assign addressB = (nextState == LOAD) ? counterRev : (((jshift + 1) <<
i) | ((jshift + 1) >> (5 - i))) & 8'h1f;

    //k values assigned according to slade paper p.13
    assign k = ((32'hfffffff0 >> i) & 4'hf) & j;

    // Assign output based on states

    //we1 remains on while loading in data or reqwriting after BFU
    assign we1 = load | ((currentState == WRITE) && memSelect);

    //we2 remains on in the write state when we1 is off (memSelect
controls)
    assign we2 = ~memSelect && (currentState == WRITE);
```

```
    assign done = (currentState == DONE);
    assign clear = (currentState == CLEAR);

    //memselect alternates in value
    assign memSelect = i[0];


endmodule
```

ButterflyUnit Module

```
module ButterflyUnit(input logic signed [15:0] aReal, aImag, bReal, bImag,
                     input logic signed [15:0] tReal, tImag,
                     output logic signed [15:0] xReal, yReal, xImag,
yImag);

        //nodal logic for multiplication output
        logic signed [31:0] tbReal, tbImag;

        //complete the complex multiplication
        assign tbImag = bReal * tImag + bImag * tReal;
        assign tbReal = bReal * tReal - bImag * tImag;

        //assign the outputs
        //weord 30:15 logic is from slade paper p.10
        assign xReal = aReal + tbReal[30:15];
        assign xImag = aImag  + tbImag[30:15];

        assign yReal = aReal - tbReal[30:15];
        assign yImag = aImag  - tbImag[30:15];

endmodule
```

Twiddle ROMs Modules

```
/*
Lookup table for the real values of the
```

```
complex twiddle factors.
Table was generated using the values from
Table II of the Slade Paper

inputs:
k = index of the twidle factors

outputs:
twiddle = twiddle factor, 16 bits
*/
module RealTwiddles (input logic clk,
                            input logic [3:0] k,
                            output logic [15:0] twiddle);
     always_comb begin
         case(k)
             4'b0000: twiddle = 16'h7fff;
             4'b0001: twiddle = 16'h7d89;
             4'b0010: twiddle = 16'h7641;
             4'b0011: twiddle = 16'h6a6d;
             4'b0100: twiddle = 16'h5a82;
             4'b0101: twiddle = 16'h471c;
             4'b0110: twiddle = 16'h30fb;
             4'b0111: twiddle = 16'h18f9;
             4'b1000: twiddle = 16'h0000;
             4'b1001: twiddle = 16'he707;
             4'b1010: twiddle = 16'hcf05;
             4'b1011: twiddle = 16'hb8e4;
             4'b1100: twiddle = 16'ha57e;
             4'b1101: twiddle = 16'h9593;
             4'b1110: twiddle = 16'h89bf;
             4'b1111: twiddle = 16'h8277;
             default: twiddle = 16'h1111;
         endcase
     end
endmodule: RealTwiddles


/*
Lookup table for the real values of the
complex twiddle factors.
```

```
Table was generated using the values from
Table II of the Slade Paper

inputs:
k = index of the twidle factors

outputs:
twiddle = twiddle factor, 16 bits
*/
module ImagTwiddles (input logic clk,
                            input logic [3:0] k,
                            output logic [15:0] twiddle);
    always_comb begin
        case(k)
            4'b0000: twiddle = 16'h0000;
            4'b0001: twiddle = 16'h1859;
            4'b0010: twiddle = 16'h30fb;
            4'b0011: twiddle = 16'h471c;
            4'b0100: twiddle = 16'h5a82;
            4'b0101: twiddle = 16'h6a6d;
            4'b0110: twiddle = 16'h7641;
            4'b0111: twiddle = 16'h7d89;
            4'b1000: twiddle = 16'h7fff;
            4'b1001: twiddle = 16'h7d89;
            4'b1010: twiddle = 16'h7641;
            4'b1011: twiddle = 16'h6a6d;
            4'b1100: twiddle = 16'h5a82;
            4'b1101: twiddle = 16'h471c;
            4'b1110: twiddle = 16'h30fb;
            4'b1111: twiddle = 16'h1859;
            default: twiddle = 16'h0000;
        endcase
    end
endmodule: ImagTwiddles
```

RAM Module

```
/*
Module for the RAM
inputs:
```

```
clk = system clock
we = write enable
adressA = address for the a data
addressB = address for the b data
aWriteData = a data to be written in
addressB = b data to be written in

output:
aReadData = a data that will be fed to BFU
bReadData = b data that will be fed to BFU
*/
module RAM(input logic clk, we,
                input logic [4:0] addressB, addressA,
                input logic [15:0] aWriteData, bWriteData,
                output logic [15:0] aReadData, bReadData);

    //create registers for the data
    logic [15:0] mem_array [0:31];

    always_ff@(posedge clk) begin

        //if write enable is set
        if (we) begin
            //write the a and b data to specified address in ram
            mem_array [addressB] <= aWriteData;
            mem_array [addressA] <= bWriteData;

            //set the output data to the data you want to read
            aReadData <= aWriteData;
            bReadData <= bWriteData;
        end

        else begin
            //set the output data to the current address
            aReadData <= mem_array[addressB];
            bReadData <= mem_array[addressA];
        end
    end
Endmodule
```

AGU FSM Next State Logic

	Source State	Destination State	
1	CLEAR	READ	
2	DONE	DONE	
3	LOAD	LOAD	(!counter[0]).(!counter[1]).(!counter[2]).(!counter[3]).(!counter[4]) + (!counter[0]).(!counter[1]).(!counter[2]).(!counter[3]).(co
4	LOAD	WAIT	(!counter[0]).(!counter[1]).(!counter[2]).(!counter[3]).(counter[4]).(counter[5]) + (!counter[0]).(!counter[1]).(!counter[2]).(cou
5	READ	WRITE	
6	WAIT	LOAD	(load).(!start)
7	WAIT	WAIT	(!load).(!start)
8	WAIT	CLEAR	(start)
9	WRITE	READ	(!always2)
10	WRITE	DONE	(always2)

Top-Level Module Next State Logic

	Source State	Destination State	
1	CALCULATING	READING	(!ittCalc[2]).(!ittCalc[3]).(!ittCalc[4]).(!ittCalc[5]).(ittCalc[6]).(ittCalc[7]) + (!ittCalc[2]).(!ittCalc[3]).(!ittCalc[4]).(ittCalc[5]).(ittC
2	CALCULATING	CALCULATING	(!ittCalc[2]).(!ittCalc[3]).(!ittCalc[4]).(!ittCalc[5]).(!ittCalc[6]) + (!ittCalc[2]).(!ittCalc[3]).(!ittCalc[4]).(!ittCalc[5]).(ittCalc[6]).(!itt
3	COLLECTING	COLLECTING	(spiload)
4	COLLECTING	LOADING	(!spiload)
5	LOADING	LOADING	(!ittLoad[0]).(!ittLoad[1]).(!ittLoad[2]).(!ittLoad[3]).(!ittLoad[4]) + (!ittLoad[0]).(!ittLoad[1]).(!ittLoad[2]).(!ittLoad[3]).(ittLoad[4
6	LOADING	CALCULATING	(!ittLoad[0]).(!ittLoad[1]).(!ittLoad[2]).(!ittLoad[3]).(ittLoad[4]).(ittLoad[5]) + (!ittLoad[0]).(!ittLoad[1]).(!ittLoad[2]).(ittLoad[3]
7	READING	READING	(!ittRead[0]).(!ittRead[1]).(!ittRead[2]).(!ittRead[3]).(!ittRead[4]) + (!ittRead[0]).(!ittRead[1]).(!ittRead[2]).(!ittRead[3]).(ittRead[4
8	READING	SENDING	(!ittRead[0]).(!ittRead[1]).(!ittRead[2]).(!ittRead[3]).(ittRead[4]).(ittRead[5]) + (!ittRead[0]).(!ittRead[1]).(!ittRead[2]).(ittRead[3]
9	SENDING	SENDING	