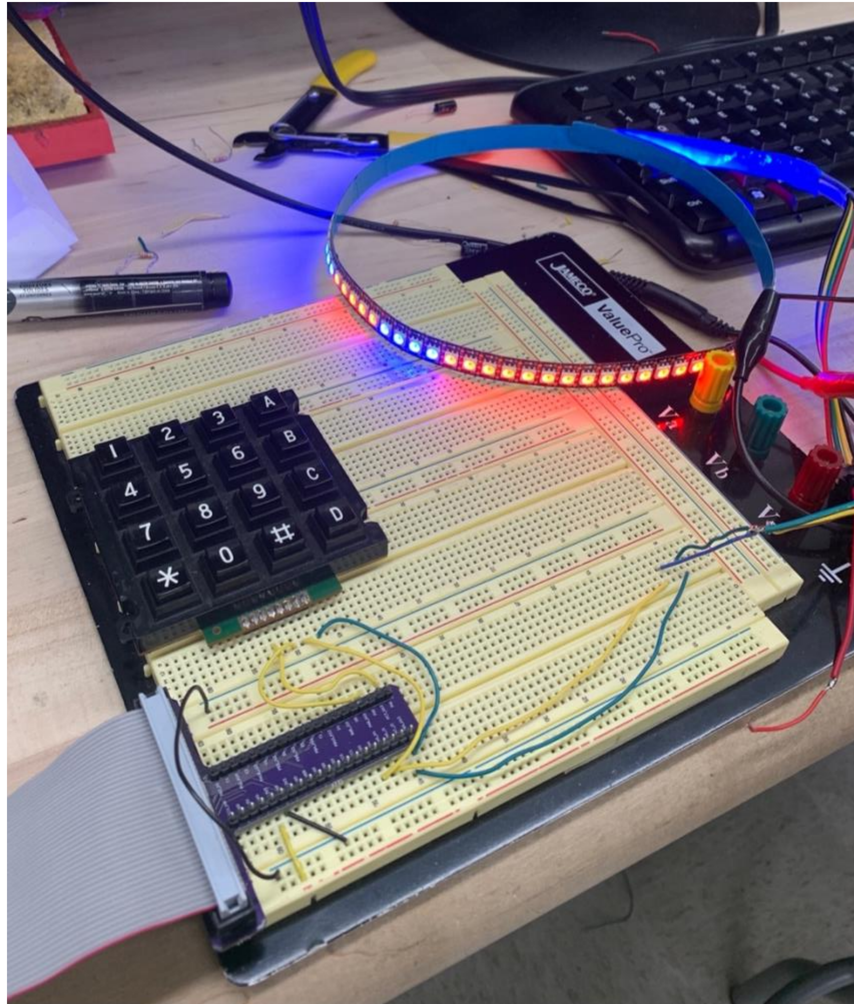
Programmable RGB LED Strip

Engineering 155 Final Project Report
November 4, 2021

Kevin Kong and John Hearn



Abstract:

The goal of this project was to program an LED strip to both display preset patterns and respond to music using a keypad, digital output microphone, FPGA, and microcontroller. Our goal was to take in sound data through the microphone, process the data using the fast Fourier Transform (FFT), and use I^{2}S to communicate between the FPGA and the microcontroller to determine the color and number of LEDs to light up. The keypad was meant to allow the user to switch between the preset patterns and the music mode.

Introduction:

LED strips have recently become increasingly common as TikTok trends sweep the nation. One major reason for their appeal is their ability to sync to music, creating a visually appealing light show for the user. Our final project was to implement this functionality, along with allowing the user to swap between preset patterns.

On the music side of things, we were to take in data using an input microphone and communicate the data to an FPGA, which was to process the input sound using the FFT. The output of the FFT was then to be communicated to the microcontroller using SPI, and based on the volume and frequency of the input sound, the microcontroller was to communicate to the LED strip through SPI to flash a certain number of LEDs with differing colors.

Similarly, the keypad was to output a signal to the MCU to allow the user to swap between the preset patterns we had coded in C. The keypad was to send a signal to the MCU, switching between the preset patterns as the user pleases.

A top-level block diagram of the system is shown below in Figure 1

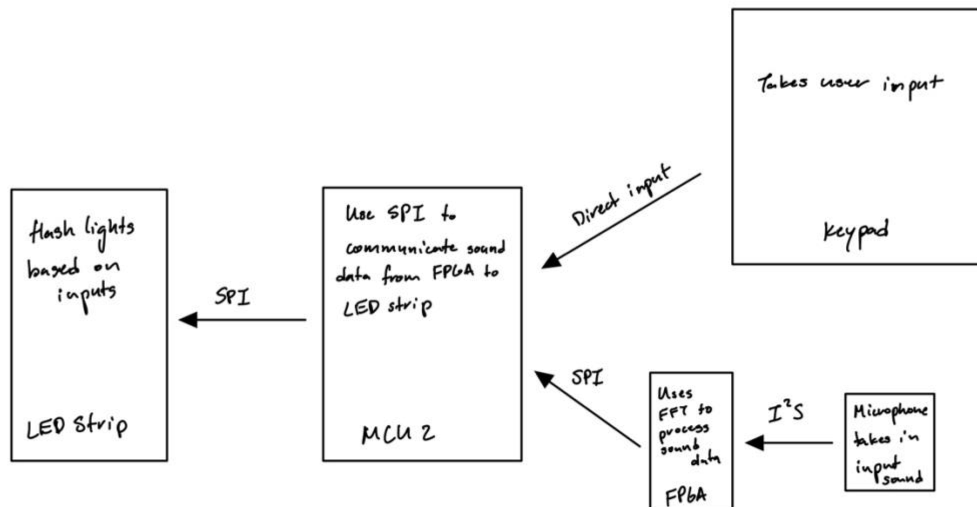


Fig. 1: Top-level block diagram of the system

New Hardware:

Our project used two pieces of new hardware: the SK9822 LED strip from BTF-Lighting and the I^2S MEMS microphone from Adafruit.

We used an SK9882 LED strip, which interfaces with SPI. The strip was 1 meter long and with 144 individually addressable LEDs.The LEDs are RGB and the configuration for each LED are the same. The first byte determines the brightness of the LED, the next byte controls the amount of blue, and the next 2 bytes controlling the amount of green and red respectively. There is a port on the LED strip for Clock In, Data In, 5 V, and ground. We used power supply in the lab to provide the 5 V since powering the LED strip from the microcontroller could potentially draw too much current.

The microphone required a 3.3 V input, along with bitclock (BCLK), left-right clock (LRCLK), and channel select (SEL) signals and output the sound data (DOUT) signal, bit by bit.

The LRCLK signal was to run at the desired sampling frequency and BCLK signal was to run at 64 times the frequency of LRCLK. We chose a combination of 46.9 kHz and 3MHz, respectively because 46.9kHz is considered by many to be close to the ideal sampling frequency for sound while also making it easy to achieve the BCLK frequency through division of the FPGA's clock, which runs at 12MHz.

The SEL signal was connected to ground to sample exclusively from the left channel. With two of these microphones, each would act as either the left or right channel and the SEL signal would be driven either high or low to designate which microphone to sample from. Because we were working with one microphone, we drove the signal low.

Schematics:

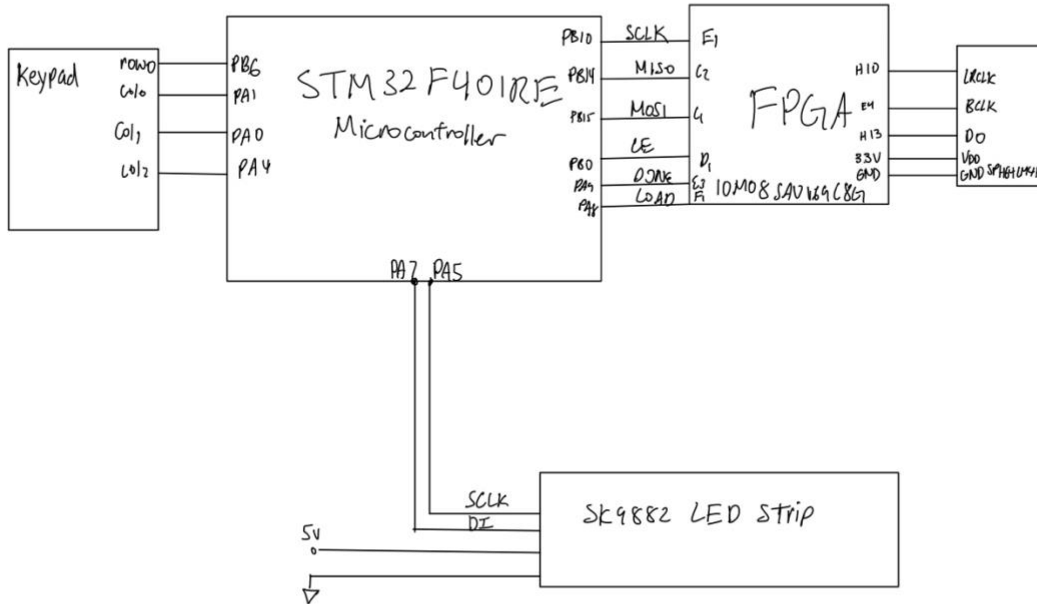


Fig 2: Overall system schematic

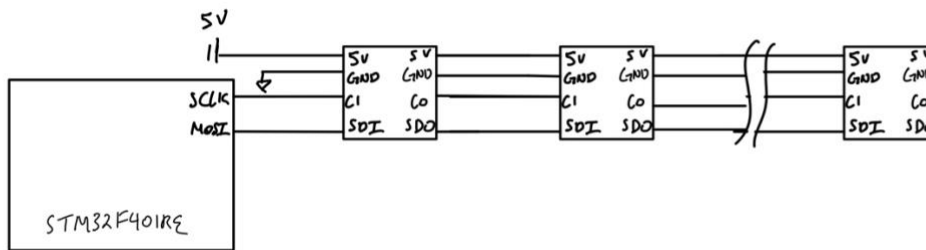


Fig. 3: Schematic of LED strip and addressable LEDs

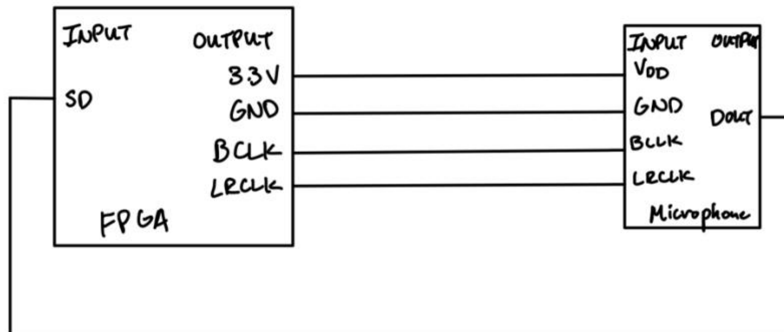


Fig. 4: Schematic of microphone interfacing with FPGA

FPGA Design:

The FPGA design for our involved setting up a system which was able to use I²S to take in sound input and process it using the FFT.

The module for I²S was implemented by reading through the datasheet for the microphone, specifically page 7 of the referenced document. The block diagram for the module is shown below in figure 5. The inputs to the module for the I²S module were clk and sd (sound data), while the outputs were bclk, sel (lrclk), and dataOut[23:0]. The clk signal was driven by the on board 12MHz clock, while the sd signal was taken from the DOUT pin on on the microphone. Using a clock divider on the clk signal, we ran bclk at 3MHz and sel at 46.9kHz to sample the sound data. As suggested by the data sheet, after the lrclk signal changes, the dataOut[23:0] signal takes in the first 18 bits of data from sd as bits [23:6] and loads in six 0s as bits [5:0]. dataOut[23:0] is unchanged until the lrclk signal inverts, signifying that it's time to sample again. Our SystemVerilog implementation of the I²S module is included in the Appendix.

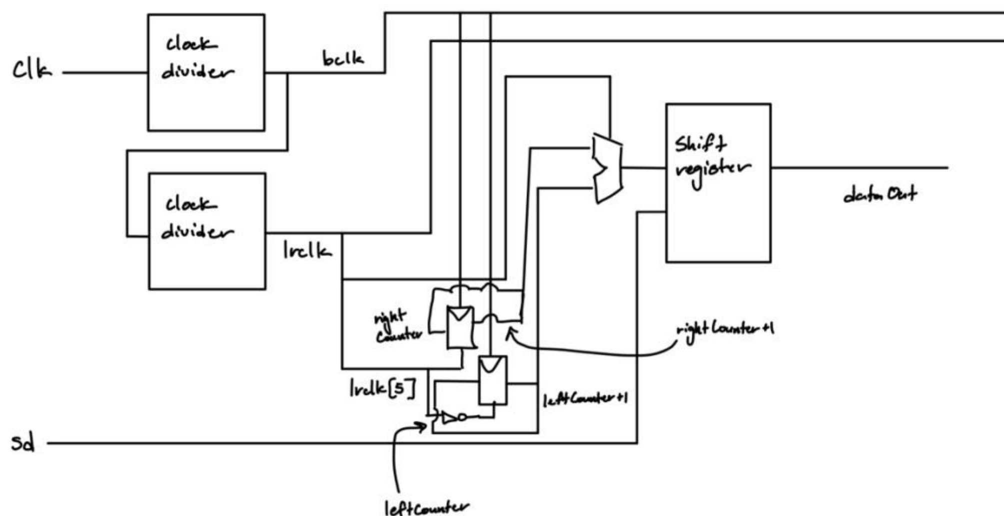


Fig. 5: Block diagram of I²S module

The FFT was implemented through careful inspection of George Slade's paper from 2013. The overall block diagram is shown below in figure 7. It was necessary to implement each of the lower-level modules shown in the diagram, and a short description of each is given below:

Address Generating Unit:

The AGU is responsible for outputting addresses to each of our RAMs, the addresses of our desired twiddle factors, and write signals for our RAMS. We should note that our addresses were to be outputted in bit reversed order.

We followed the pseudo code in Table III of Slade's paper closely for our SystemVerilog implementation. We used a state machine to model the pseudo code. We have 6 states: WAIT, CLEAR, LOAD, READ, WRITE, and DONE. We use the LOAD state to indicate when we want to load data into our RAM. We cycle between the READ and WRITE states, and using counters we were able to cycle the proper number of times before entering the DONE state. Table IV of Slade's paper was used to check our addresses to confirm functionality.

Butterfly Unit:

The butterfly unit is essential to the FFT. It performs the recursive two point transforms that build up the whole 32 point FFT. The butterfly unit performs the complex addition and multiplication between the data samples and the twiddle factors. Although we are using 11 bits for the data, the input is a total of 16 bits to accommodate for bit growth. Slade stressed that 5 bits for growth was essential for precision, and to also note that multiplication between two 16 bit numbers results in a 32 bit number. We index from [30:15], the result of the multiplication for future operations.

Twiddle ROM:

The Twiddle ROM is essentially a look up table for the twiddle factors based on the twiddle address. We use a .txt file with the twiddle factors, and read those in. We then output the desired twiddle factor based on the twiddle address input from the AGU. The twiddle factors were taken from Table II of the Slade article.

RAM:

We use two dual port RAMs. In the FFT, we read from one RAM and write to the other simultaneously. The RAMs take the address from the AGU and reads them in bit reversed order. The RAMs also take in the write enable signals from the AGU. It was necessary to use a "ping-pong scheme" in this implementation.

Our SystemVerilog implementation of the FFT is included in the Appendix.

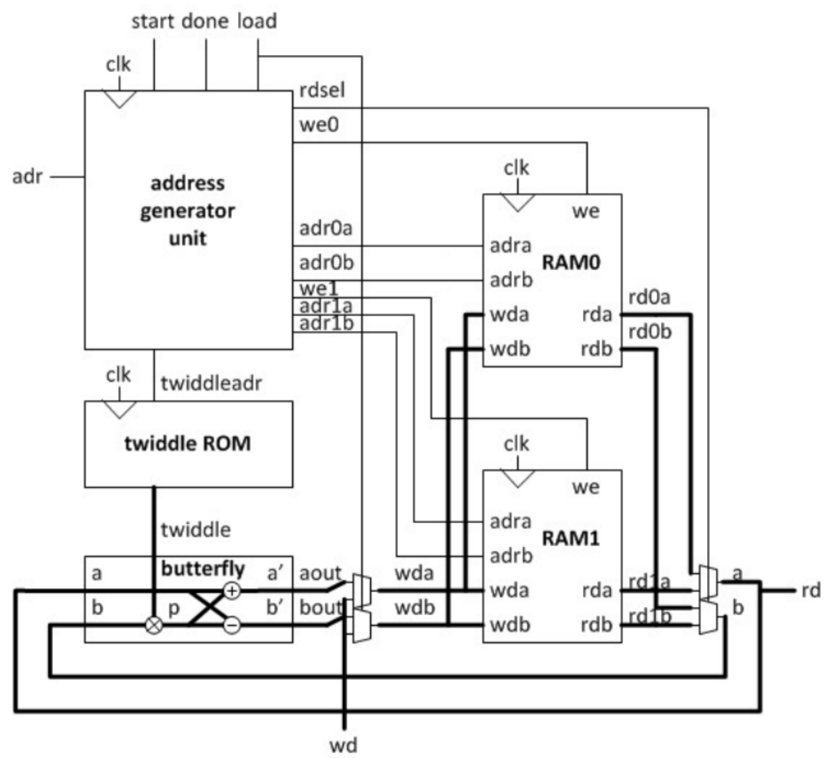


Fig. 8: FFT block diagram

Microcontroller Design:

Utilizing the code from Lab 6, we set up SPI interfacing between our STM32F401RE microcontroller and our SK9882 RGB LED strip. In the SPI protocol, our microcontroller acted as the master and the LED strip was the slave. Pins PA5 and PA7 were used for SCLK and DI respectively, while pins for Chip Enable nor "Master in Slave Out" were ignored as the LED strip did not have those ports. To display our desired pattern on the LED strip, we needed to send 2-bytes of 0's four times to act as the start frame, and 2-bytes of 1'S four times to act as the end frame. In between our two frames, we sent our data to configure the LED strip's pattern.

We used a global variable called LEDARR that puts our LEDS into a two-dimensional array of uint8_t's. The LED array is 94x4, signifying 94 LEDS used as wells as an uint8_t value for global, red, green, and blue. To address each LED, we would use their index on the array, and set the other values accordingly. The global value always leads with three 1's. This structure means that whatever value we input for global, we should OR it with 0xE0. The red, blue, and green values have 8 bits of resolution, meaning their values can be set from 0 to 255. To use this LEDARR properly, we have a function to clear the array, and another to set the array to desired values before any SPI transmission. We can then send these values over after using a start frame and end it with an end frame.

We also used our microcontroller to read inputs from the digital keypad to switch between three patterns on the LED strip. We set ports A0, A1, and A4 as input pins to read in which button in the first row of the keypad was pressed. We then stay in the mode of the most recently pressed key, only changing until another valid button is pressed.

Results:

We were successful in programming interfacing the SPI between the MCU and LED strip to display preset patterns based on a user's keypad input. The preset patterns are visually appealing and fun to watch.

However, we did not meet all the proposed specifications. We experienced difficulty in extracting the sound data from the microphone which inhibited us from being able to implement the music mode of the LED strip. This was very disappointing to us as it was the main component of our project and had gotten the software components working in simulation to facilitate the functionality. However, we acknowledge that attempting this project was no easy task. Implementing the FFT, I^2S, and using SPI to communicate between not only the FPGA and MCU but also between the MCU and LED strip is not trivial, and we may have bitten off more than we could chew in the time allotted for the project.

In pursuing projects of the sort in the future, we agree that it is important to understand and consider the difficulty of implementing each component in the time given. With more time and room for error, we could have either resolved the issue with the current microphone or ordered a new one that was easier to interface with.

Overall, we still feel as though this project was an incredible learning experience and gave us a better understanding of how to choose hardware applicable to a certain problem and interface the new hardware with the other components.

References:

[1] Slade, G. The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation. **2013**.
https://web.mit.edu/6.111/www/f2017/handouts/FFTtutorial121102.pdf

[2] https://www.adafruit.com/product/3421?gclid=CjwKCAiA78aNBhAlEiwA7B76p9Z_wAs43FQ Za5DdLtXsj1JTxZONY0kYzY5jyMh1MzIzvI8w1dXm6hoCK8oQAvD_BwE

Bill of Materials:

Part	Part Number	Vendor	Quantity	Price/Unit	Total Price
1m LED Strip with 144 LEDs	SK9882	Amazon	1	$31.99	31.99
I^2S Microphone	SPH0645LM4H	Adafruit	1	$6.95	$6.95

Appendix A – SystemVerilog Code:

```
// Top level module connecting the i2s and FFTcore modules
// state machine allows for correct timing of feeding output of i2s to input
of fft
module finalProject(input logic clk, reset, sd,
                                      output logic [31:0] freqR, freqI);

      logic sel, bclk, load;

      typedef enum logic [2:0] {s0, s1, s2, s3} statetype;
      statetype state, nextState;

      always_ff @(posedge bclk, posedge reset) begin
            if (reset) state <= s0;
            else state <= nextState;
      end


      always_comb
            case(state)
                  s0: if (~load) nextState = s1;  // want to stay in s0 until
load isn't asserted
                         else nextState = s0;

                  s1: if (load) nextState = s1;  // stay in load until FFT is
done loading
                         else nextState = s2;

                  s2: nextState = s3; // intermediary state to assert start

                  s3: nextState = s3; // stay in s3 for the rest of the
process, done will be asserted and stop process

                  default: nextState = s0;
      endcase


      assign start = (state == s2);  // let FFT know when to start loading
the data


      logic [23:0] dataOut;
      i2s micIn(clk, reset, sd, sel, bclk, load, dataOut);
      fftCore fft(bclk, reset, start, load, {dataOut[23:0], 8'b00000000},
freqR, freqI, done);

endmodule


// I2S MODULE
// module to drive signals for the i2s microphone
module i2s(input logic clk, reset, sd,
                    output logic sel, bclk, load,
                    output logic [23:0] dataOut);
```

```
	logic [19:0] prescaler;  // SERIAL CLOCK -- should run at 2MHz
	logic [5:0] lrclk;  // LEFT-RIGHT CLOCK -- TRANSMIT LEFT CHANNEL WHEN LOW, RIGHT WHEN HIGH
	logic [4:0] leftCounter, rightCounter;  // Might have to switch the bitwidth to avoid overflow
	logic [23:0] dataReg;

	// update the value of prescaler, runs at 3MHz
	always_ff @(posedge clk) begin
		if (reset) prescaler <= 0;
		else prescaler <= prescaler + 246625;
	end

	assign bclk = prescaler[19];


	// sel should run at bclk / 64 = 46.9kHz
	always_ff @(posedge bclk, posedge reset) begin
		if (reset) lrclk <= 0;
		else lrclk <= lrclk + 1;
	end

	assign sel = lrclk[5];

	assign load = ((sel && rightCounter >= 25 && rightCounter <= 31) || (~sel && leftCounter >= 25 && leftCounter <= 31));

	// updating rightCounter
	always_ff @(posedge bclk, posedge reset) begin
		if (reset) rightCounter <= 0;
		else if (sel) rightCounter <= rightCounter + 1;
		else rightCounter <= 0;
	end


	// updating leftCounter
	always_ff @(posedge bclk, posedge reset) begin
		if (reset) leftCounter <= 0;
		else if (~sel) leftCounter <= leftCounter + 1;
		else leftCounter <= 0;
	end

	// shift register to update dataReg
	always_ff @(posedge bclk) begin
		if (sel) begin
			if (rightCounter >= 1 && rightCounter <= 18)
				dataReg <= {dataReg[22:0], sd};
		end else begin
			if (leftCounter >= 1 && leftCounter <= 18)
				dataReg <= {dataReg[22:0], sd};
		end if ((rightCounter >= 19 && rightCounter <= 24) || (leftCounter >= 19 && leftCounter <= 24))
			dataReg <= {dataReg[22:0], 1'b0};
	end

	assign dataOut = ((sel && rightCounter >= 25 && rightCounter <= 31) || (~sel && leftCounter >= 25 && leftCounter <= 31)) ? dataReg : dataOut;
```

```
endmodule

module testbench_fftCore();
	logic clk, reset, start, load, done;
	logic signed [31:0] wd;
	logic signed [15:0] outputReal, outputImag;
	logic signed [31:0] testvectors[0:31];


	fftCore dut(clk, reset, start, load, wd, outputReal, outputImag, done);

	always
		begin
			clk = 1; #5; clk = 0; #5;
		end

	logic [31:0] counter;
	always_ff @(posedge clk)
		begin
			if (reset) counter <= 0;
			else counter <= counter + 1;
		end

	assign wd = testvectors[counter];

	always_comb
		begin
			load  = (counter < 32);
			start = (counter == 35);
		end

	initial
		begin
			$readmemh("inputs.txt", testvectors);
			reset = 1; #10; reset = 0;
		end

endmodule


// FFT CORE
// Utilizes the lower level modules to perform the FFT
module fftCore(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] twiddleadr;
	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
      AGU agu(clk, reset, start, load, done, wen0, wen1, clear, rdsel,
twiddleadr, adr0a, adr0b);

      assign adr1a = adr0a;

   assign adr1b = adr0b;

      twiddleRom tw(clk, twiddleadr, twiddleReal, 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;

      butterfly 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 = ain[15:0];
endmodule


// BUTTERFLY UNIT
// module to perform the recursive two-point transforms that build up the 32
point FFT
// given two 32-bit inputs along with a twiddle factor, outputs two 32-bit
complex numbers as output
module butterfly(input logic signed [15:0] ainr, aini, binr, bini,
                                    input logic signed [15:0] twiddleReal,
twiddleImag,
                                    output logic signed [15:0] aOutReal,
bOutReal, aOutImag, bOutImag);

            logic signed [31:0] bwkReal, bwkImag;

            assign bwkImag = binr * twiddleImag + bini * twiddleReal;
            assign bwkReal = binr * twiddleReal - bini * twiddleImag;

            // Outputs
            assign aOutReal = ainr + bwkReal[30:15];
            assign aOutImag = aini  + bwkImag[30:15];

            assign bOutReal = ainr - bwkReal[30:15];
            assign bOutImag = aini  - bwkImag[30:15];
```

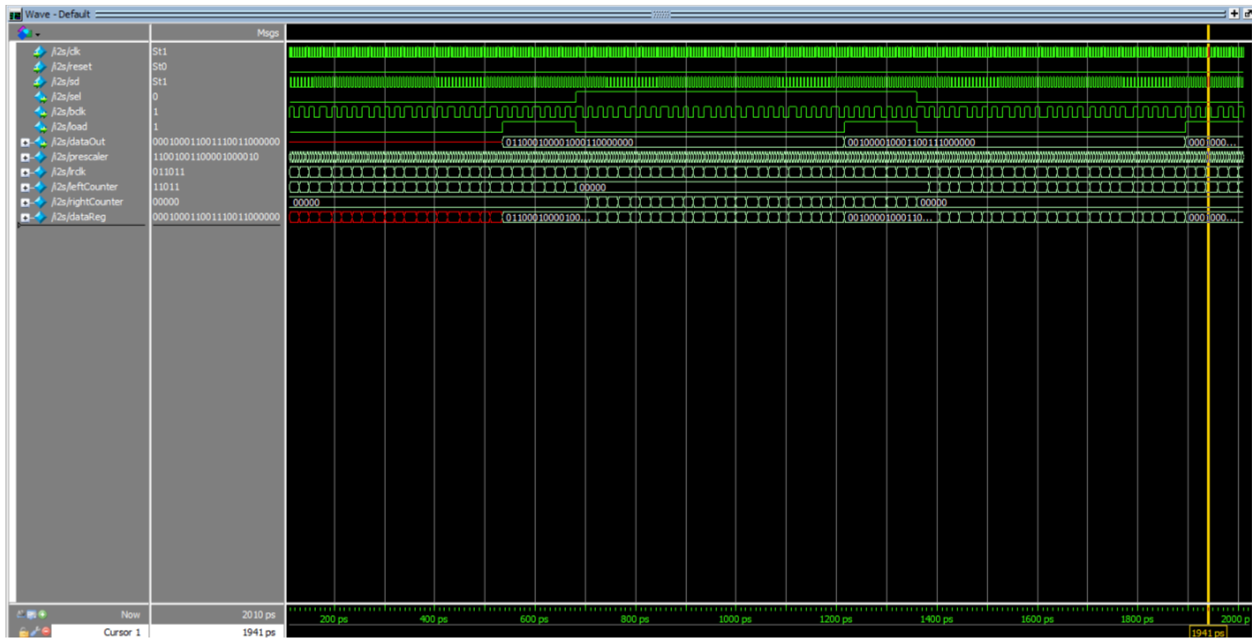
```
endmodule


// TWIDDLE ROM MODULE
// given a twiddle address, outputs the real and imaginary twiddle factors as
specified in Slade's paper
module twiddleRom(input logic clk,
                                    input logic [3:0] twiddleAdr,
                                    output logic signed [15:0] twiddleReal,
twiddleImag);

            logic [31:0] twiddleROM[0:15];
            logic [31:0] twiddle;

            initial $readmemh("twiddleROM.txt", twiddleROM);


            assign twiddle = twiddleROM[twiddleAdr];

            assign twiddleReal = twiddle >> 16;
            assign twiddleImag = twiddle;
endmodule


// ADDRESS GENERATING UNIT
// generates the addresses to determine where in RAM to store the processed
data
module AGU(input logic clk, reset, start, load,
                    output logic done, wen0, wen1, clear, rdSel,
                    output logic [3:0] twiddleadr,
                    output logic [4:0] adra, adrb);
      typedef enum logic [2:0] {WAIT, LOAD, READ, WRITE, CLEAR, DONE}
statetype;  // Might need a clear state in between read and write
      statetype state, nextState;

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

      logic [2:0] i, inext;
      logic [4:0] j, jnext, jshift;

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

```
                state <= nextState;
                i <= inext;
                j <= jnext;
            end

    // Combination logic for nextState
    always_comb
        case(state)
            WAIT: if (start) nextState = CLEAR;
                    else if (load) nextState = LOAD;
                    else nextState = WAIT;

            LOAD: if (counter < 32) nextState = LOAD;
                    else nextState = WAIT;

            READ: nextState = WRITE;

            WRITE: if (i == 4 && j == 15) nextState = DONE;
                     else nextState = READ;

            CLEAR: nextState = READ;

            DONE: nextState = DONE;

            default: nextState = WAIT;
        endcase


    // Combinational logic for incrementation
    always_comb
        case(state)
            WRITE:
                begin
                    if(j == 15) begin
                        jnext = 0;
                        inext = i + 1;
                    end else begin
                        jnext = j + 1;
                        inext = i;
                    end
                end
            default:
                begin
                    inext = i;
                    jnext = j;
                end
        endcase

    logic [4:0] cntRev;
    bitreverse br1(counter[4:0], cntRev);
    // Assign values of addresses according to AGU pseudocode in Table 3 of
the Slade paper
    assign jshift = j << 1;
    assign adra = (state == LOAD) ? cntRev : ((jshift << i) | (jshift >> (5
- i))) & 8'h1f;
    assign adrb = (nextState == LOAD) ? cntRev : (((jshift + 1) << i) |
((jshift + 1) >> (5 - i))) & 8'h1f;
```

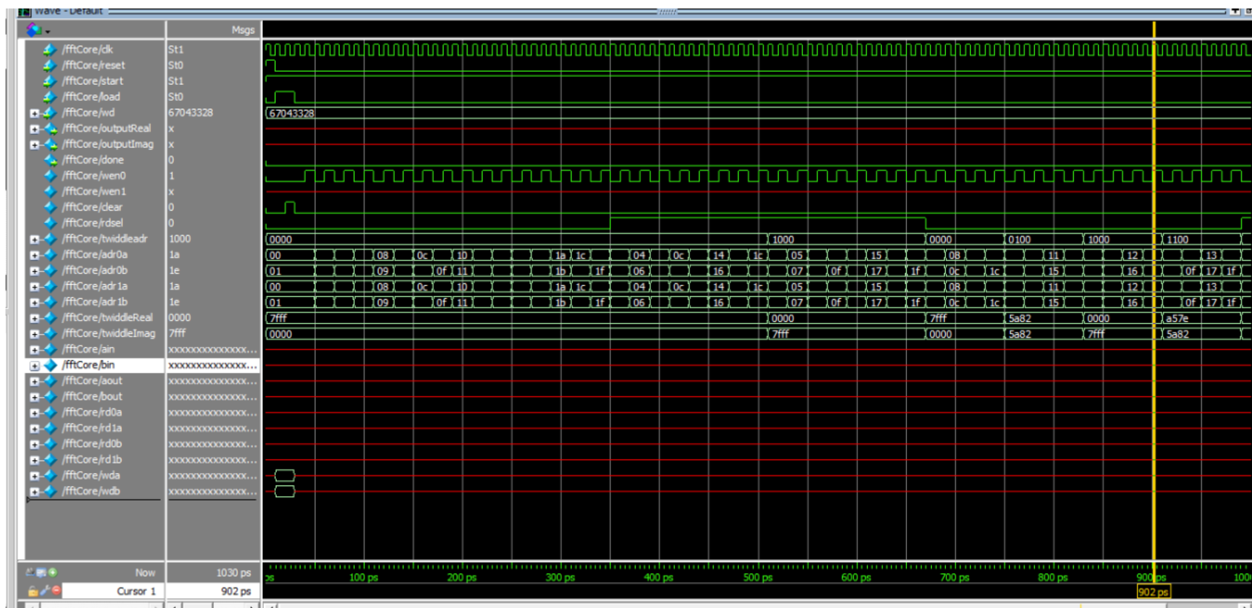
```
	assign twiddleadr = ((32'hfffffff0 >> i) & 4'hf) & j;

	// Assign output based on states
	assign wen0 = load | ((state == WRITE) && rdSel);
	assign wen1 = ~rdSel && (state == WRITE);
	assign done = (state == DONE);
	assign clear = (state == CLEAR);

	// Want to alternate between RAMs
	assign rdSel = i[0];


endmodule


// Dual ported RAM module
// given two addresses and two 16 bit inputs, stores them in memory
module RAM(input logic clk, wen,
			input logic [4:0] adrb, adra,
			input logic [15:0] wda, wdb,
			output logic [15:0] rda, rdb);

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

	always_ff@(posedge clk) begin
		if (wen) begin
			ram[adrb] <= wda;
			ram[adra] <= wdb;
			rda <= wda;
			rdb <= wdb;
		end
		else begin
			rda <= ram[adrb];
			rdb <= ram[adra];
		end
	end
endmodule

module bitreverse(input logic [4:0] adr,
				output logic [4:0] reversed);
	assign reversed = {adr[0], adr[1], adr[2], adr[3], adr[4]};
endmodule
```

Appendix B – FFT and I2S Simulation:

Appendix C – I2S Simulation:



Appendix D – FFT Simulation:



Appendix E - C Code:

```
// John Hearn & Kevin Kong
// Microcontroller code for SPI interfacing with LED and FPGA
//headers
#include "STM32F401RE.h"
#include <stdint.h> // for integer types (i.e., uint32_t)
//pins for GPIO outputs
#define MOSI 7
#define SCLK 5
#define DONE_PIN 8
#define LOAD_PIN 9
//define for SPI configurations
#define clkdivide 0b101
#define cpol 1
#define ncpha 1
// Size and indices of array LEDARR
#define NUMLED 94
#define RED 3
#define GREEN 2
#define BLUE 1
#define GLOBAL 0
#define mode0 1
#define mode1 0
#define mode3 4
// Frames our LED strip into an array based on 9. Feature Description of
datasheet
uint8_t LEDARR[NUMLED][4];
int16_t fft[32][2];
```

```
uint8_t COLOR[4];

// Clears the LEDARR for future updates
void clearLEDARR(){
    volatile uint8_t i;
    volatile uint8_t j;
    for( i=0; i < NUMLED; i ++){
        LEDARR[i][GLOBAL] = 0xE0;
        for (j = 1; j < 4; j++){
            LEDARR[i][j] = 0x00;
        }
    }
}
// Structuring the LEDARR to what we want to send
void setLED(uint8_t ledNum, uint8_t global, uint8_t red, uint8_t green,
uint8_t blue){
    LEDARR[ledNum][GLOBAL] = (0xE0 | global);   // Global always starts
with 111
    LEDARR[ledNum][RED] = red;
    LEDARR[ledNum][GREEN] = green;
    LEDARR[ledNum][BLUE] = blue;
}

// Starts SPI transmission
void updateLEDSPI(){
    // Start Frame
   int i;
   for(i=0;i<4;i++){
        spiSendReceive(0x00);
   }
    // LED Frame
    for (i = 0; i <NUMLED; i ++){
            spiSendReceive(LEDARR[i][GLOBAL]);
            spiSendReceive(LEDARR[i][BLUE]);
            spiSendReceive(LEDARR[i][GREEN]);
            spiSendReceive(LEDARR[i][RED]);
        }
    // End Frame
    for(i=0;i<4;i++){
        spiSendReceive(0xFF);
   }
}

// Determines pattern from input
int getPattern(uint8_t blue, uint8_t red, uint8_t green){
    int i;
    int x= 0;
    int isMode0 = 0;
    int isMode1 = 0;
    int isMode3 = 0;
    int currPattern;

    while (1) {
```

```
        // wait for input
        while (!isMode0 && !isMode1 && !isMode3) {
            isMode0 = digitalRead(GPIOA, mode0);
            isMode1 = digitalRead(GPIOA, mode1);
            isMode3 = digitalRead(GPIOA, mode3);
        }

        if (isMode0) {
            currPattern = 0;
        } else if (isMode1) {
            currPattern = 1;
        } else if (isMode3) {
            currPattern = 3;
        }
        break;
    }
    // Pattern 0 is white LED
    if (currPattern == 0) {
      for (i = 0; i < NUMLED; i++) {
            if((x + i) % 2 == 0) {
                red = 200;
                green = 0;
            } else {
                red = 0;
                green = 200;
            }
            setLED(i,02,red,green, 0 );
            updateLEDSPI();
        }
        x++;
      // Pattern 1 to a moving color on other color
    } else if(currPattern == 1) {
        for (i = 0; i < NUMLED; i++) {
            setLED(i, 6 ,200, 0, 178);
            updateLEDSPI();
            delay_micros(TIM2, 800);
            setLED(i+4, 5 ,190, 0, 0);
            updateLEDSPI();
        }

      // Pattern 2 alternating colors
    } else if (currPattern == 3) {
        for (i=47;i<NUMLED; i++) {
            setLED(i, 4, 0, 0, 250);
            setLED(94 -i, 5, 124, 0, 0);
            updateLEDSPI();
            delay_millis(TIM2, 20);
            setLED(i-47, 4, 0,0, 250);
            setLED(NUMLED+ 47-i,4,200, 0, 0);
        }
    }
    return currPattern;
}
```

```
void playPattern(int pattern, uint8_t blue, uint8_t red, uint8_t green) {
    int i = 0;
    if (pattern == 0) {
        int x = 0;
        for (i = 0; i < NUMLED; i++) {
            if((x + i) % 2 == 0) {
                red = 200;
                green = 0;
            } else {
                red = 0;
                green = 200;
            }
            setLED(i,02,red,green, 0 );
            updateLEDSPI();
        }
        x++;
    } else if (pattern == 1) {
        for (i = 0; i < NUMLED; i++) {
            setLED(i, 6 ,200, 0, 178);
            updateLEDSPI();
            delay_micros(TIM2, 750);
            setLED(i+4, 5 ,190, 0, 0);
            updateLEDSPI();
        }
    } else if (pattern == 3) {
        for (i=47;i<NUMLED; i++) {
            setLED(i, 4, 0, 0, 250);
            setLED(94 -i, 5, 124, 0, 0);
            updateLEDSPI();
            delay_millis(TIM2, 20);
            setLED(i-47, 4, 0,0, 250);
            setLED(NUMLED+ 47-i,4,200, 0, 0);
        }
    }
}

int main(void){
    //SPI SET UP
    configureFlash();
    configureClock();
    RCC->AHB1ENR.GPIOAEN = 1;

    spiInit(clkdivide,cpol,ncpha);
    // TIMER SET UP
    RCC->APB1ENR |= (1 << 0);
    initTIM(TIM2);
    clearLEDARR();
    int x;
    uint8_t green;
    uint8_t red;

    pinMode(GPIOA, mode0, GPIO_INPUT);
    pinMode(GPIOA, mode1, GPIO_INPUT);
    pinMode(GPIOA, mode3, GPIO_INPUT);
```

```
    while(1) {
        int currentPattern = getPattern(0, 0, 0);
        while (!digitalRead(GPIOA, mode0) && !digitalRead(GPIOA, mode1)
&& !digitalRead(GPIOA, mode3)) {
            playPattern(currentPattern, 0, 0, 0);
        }
    }  //END WHILE
}  //END MAIN
```