Audio Equalizer LED Display

Final Project Report
December 13th, 2019
E155

Kira Favakeh and Jonah Cartwright

Abstract:

Audio equalizers are frequently used in the music industry. In an audio equalizer, audio is broken down into a number of frequency components and visually displayed. In this project, audio from a microphone is broken up into 8 distinct frequency bins, and displayed on an 8x8 LED matrix. The audio input received by a microphone is amplified by a built in amplifier and converted via the ADC peripheral on the ATSAM processor into a digital signal. That signal is then sent via SPI from the ATSAM to the FPGA which then runs a 32-point digital FFT on the incoming signals. The FPGA then drives the LED display to show the relative amplitudes of the various frequency bins sorted by the FFT.

Introduction

This project was inspired by interesting and visually captivating equalizer displays often seen with music. This motivated the project to consists of fully assembling an LED display of an audio equalizer including the portions for audio input, signal processing, and driving the LED display. A block diagram of the project can be seen in Figure 1 below.

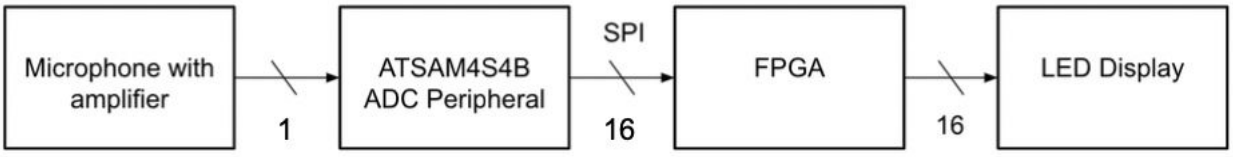

Figure 1. Whole system block diagram.

The system starts with a board housing a microphone and amplification circuit to pick up audio. A raw voltage value from the microphone is sent to the ATSAM4S4B ADC peripheral. With the ADC peripheral the microphone voltage in converted to a 16-bit digital value. From there, the ATSAM4S4B sends the audio signals to the FPGA via SPI. The FPGA collects and stores in RAM 32 16-bit signals from the microcontroller. Once the FPGA has collected those 32 signals, The FPGA is responsible for signal processing which involves running the data through a 32-point digital Fast Fourier Transform (FFT). The FPGA then drives the LEDs on an external 8x8 RGB LED display which will show the relative magnitudes of 8 frequency bins.

New Hardware

This project contains two new pieces of hardware not used before in E155. For this project, the Adafruit Electret Microphone Amplifier was used to capture audio input as seen in Figure 2 below on the left. This came as a small board that housed both a microphone and amplifier with adjustable gain from a small manual trimmer pot. There are three pinouts on the board including V_{CC}, ground, and output. The voltage supply range for the board is 2.4 - 5.5V so the team supplied 5 V to V_{CC} [1]. The output is connected to the peripheral PA17 on the µMudd Mark V.1 utility board corresponding to ADC channel 0 on the ATSAM4S4B [2].

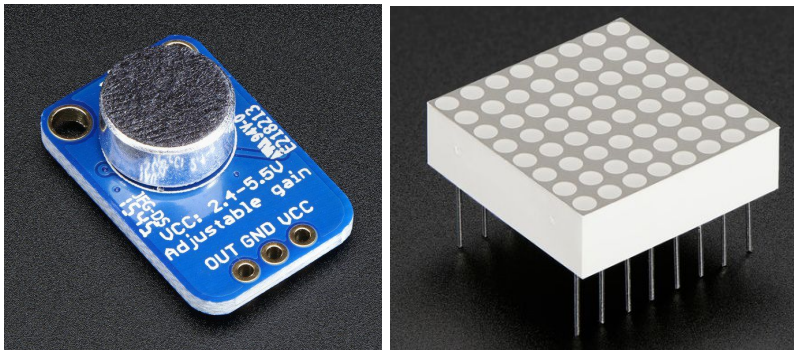

Figure 2. New hardware. (Left) Electret Microphone Amplifier. (Right) LED display/dot matrix.

The Luckylight LED display is a red dot matrix as seen above on the right. This part was a readily available stock part that the team had not used in E155. This 8x8 matrix was used for the LED display and was wired as can be seen in the schematic in the next section. This display has eight pins for each column and eight for each row that were simply driven high or low by the FPGA [3].

The team originally intended to use the Geeetech LED Matrix 8x8 is a triple color RGB common anode display. This display had PWM color intensity control and eight row enables and 24 column enables for red, green, and blue with one for each color in each column [4]. However, the team was not able to implement the final design with this board to due to time constraints that did not allow us to explore the color control on the board as the datasheet did not provide enough relevant information for straightforward implementation.

Schematics

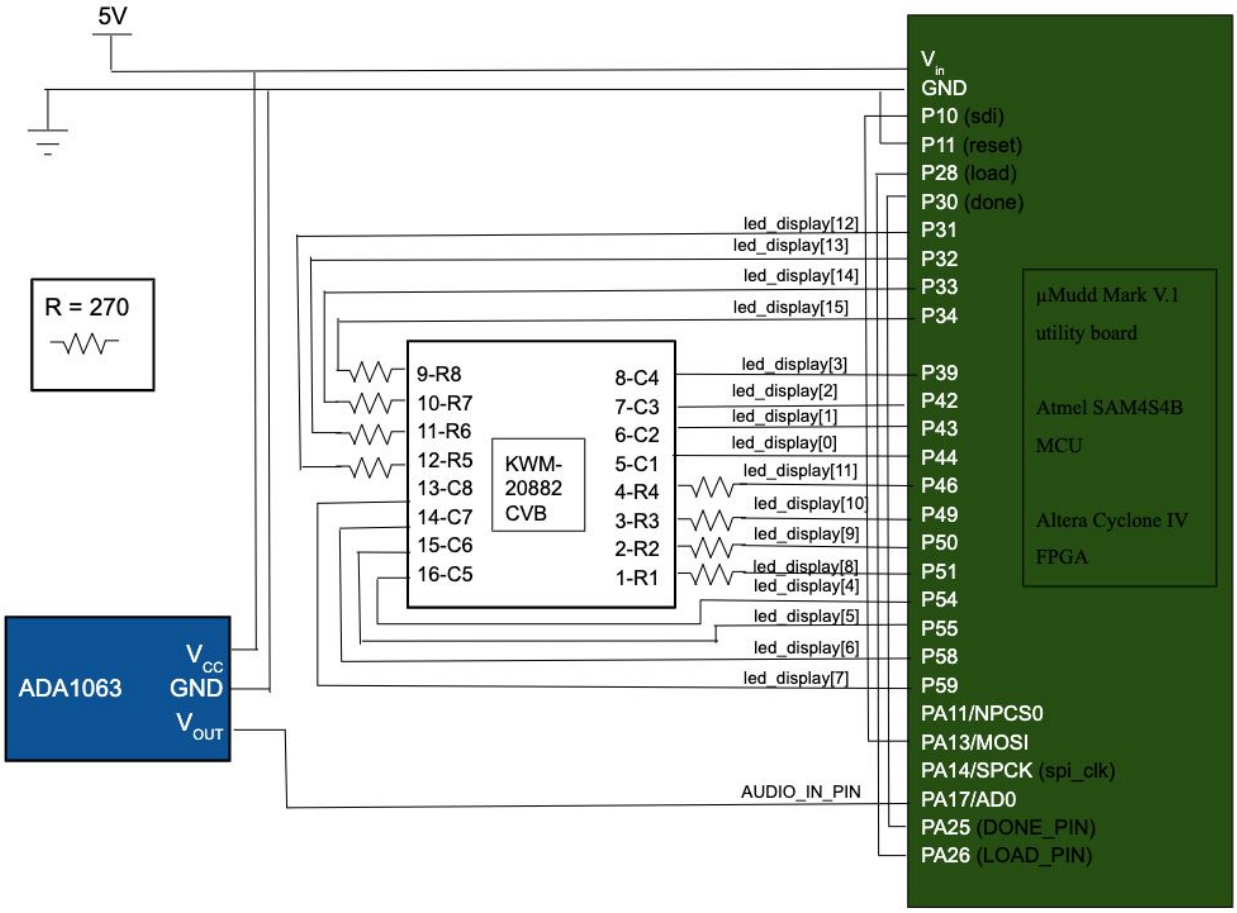

Figure 3. Breadboard schematic.

Above is a schematic of the final breadboard circuitry used for this project. The two new pieces of hardware are shown with their integration. This includes the ADA1063 microphone and amplifier and the KWM-20882CVB LED display. The ADA board is connected to power and ground and the output signal is sent to ADC channel 0. Additionally, the LED display has the rows and columns connected to I/O pins assigned to led_display that the FPGA controls to drive the display. The rows are connected to the peripherals in series with 270 Ω resistors. This was because the datasheet for the display says that the max forward voltage was 2.0 V for 20 mA forward current per dot. With this information, the team chose to use 270 Ω resistors to account for the remainder of the 5 V driven to the display. Lastly, the FPGA pins and ATSAM peripherals were connected such that they could communicate. This includes the signals for done, load, and MOSI.

Microcontroller Design

The microcontroller design is simple as its functionality is to receive audio input via ADC and export the signals to the FPGA via SPI. The microcontroller initialized the ATSAM as well as PIO, SPI, and ADC header files. Peripheral PA17 is used as the audio input, AUDIO_IN_PIN_PA17, to channel 0 of the ADC, ADC_CH0. After initializations, LOAD_PIN is immediately set high to allow the first iteration of data to be read by the FPGA. Then, a loop runs while DONE_PIN is asserted by the FPGA which is telling that it has finished the FFT processes and is ready for SPI ADC input. Within this loop, the controller reads from ADC_CH0 and sends the output as a short via spiSendReceive16 to the FPGA, and then de-asserts LOAD_PIN. The LOAD_PIN is then immediately reasserted again in order to avoid a lag from the controller to the FPGA that prevents the FPGA from staying in the spi state long enough to collect data.

FPGA Design

The FPGA does the bulk of the work as it implements the digital FFT for signal processing and is used to drive the LED display based on the outputted frequency bins from the FFT. The FPGA takes in amplified data from the microphone over SPI to run its signal processing on.

The logic on the FPGA is controlled by a state machine as in Figure 4 . The state machine starts at PRE_START where the variable newdata reset the addresses data will be loaded into. The state machine then loops through a set of tasks until 32 16-bit pieces of data are loaded in over SPI from the microcontroller and stored in memory. Then the FPGA calculates the FFT, simplifies the results into 8 bins of relative magnitude, and then drives the display. The FPGA then gets back to its PRE_START state and loads new data while holding the prior display.

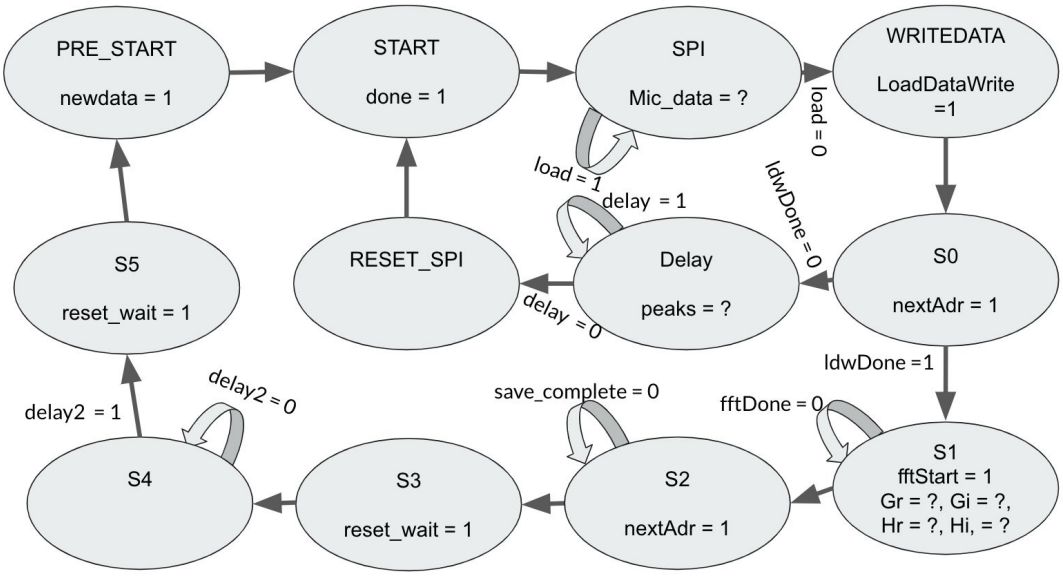

Figure 4 . FPGA State Machine

FFT:

The FFT hardware implementation is based on the paper "The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation" by G. William Slade [5]. The FFT has a sampling rate of about 2441 Hz and uses 32 samples. Therefore the FFT produces 16 bins with a width of about 76 Hz. The magnitude of each bin is then calculated. Because the LED matrix is only 8 columns, the magnitude of adjacent bins are combined (i.e bin 1 + bin 2, bin 3+ bin 4, etc) to get 8 magnitude outputs that will be used to control the display.

The System Verilog FFT module was written to implement the 32-point Cooley-Tukey Radix-2 FFT algorithm as described in the reference paper [5] on the FPGA. This system consists of several modules to implement the FFT. These modules are an address generation unit used to generate a defined set of addresses to store data in memory, a twiddle factor ROM used to provide the necessary twiddle factors based on the index of the FFT, a butterfly unit that performs the complex multiply add needed, and a memory bank made up of 4 2-port RAM modules to hold the real and imaginary components of the FFT.

Sampling:

In order to sample at a proper speed, the FPGA had to use a delay to wait some amount of time before it asked for another 16-bit data input from the microcontroller. This was implemented with a 14-bit counter that would delay the FPGA state machine for 0.004096 seconds until it asked for new data again. Once 32 16-bit data signals were collected the FPGA would exit the collected data and delay loop and calculate the FFT.

Testing:

The FFT was tested using a square wave with a known output provided by the reference paper [5]. The 32 point input was tested in ModelSim and the FFT delivered the expected real and imaginary outputs as seen in Figure 5.

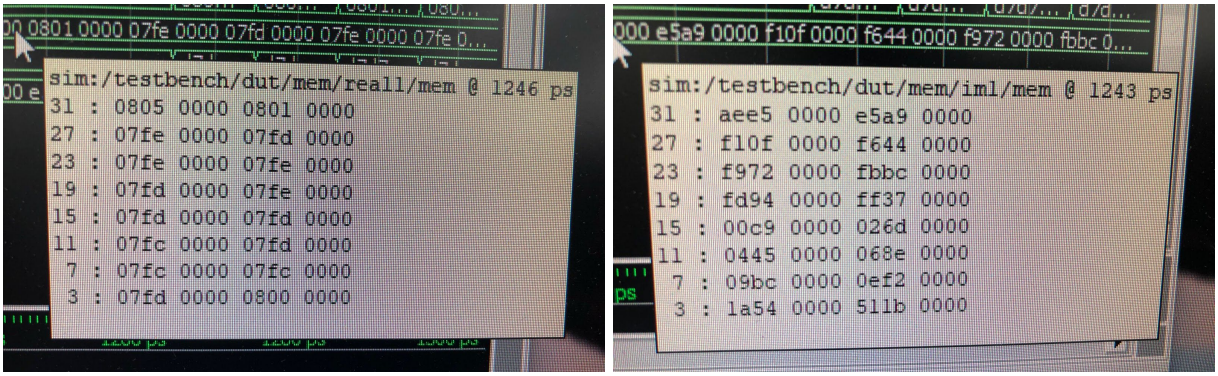

Figure 5. Real and Imaginary outputs of test of FFT

Driving the Display:

Once the FFT was finished the magnitudes were calculated for each bin. As previously mentioned these magnitudes were combined into 8 bins for the 8 columns of the LED Matrix. This bins represented the magnitude of the frequency bins 0-150 Hz, 150-300 Hz, …, 1050 -1200 Hz. The FPGA decodes the magnitude of each bin to drive the display. The higher the magnitude for a column the higher up the LEDs will light up in that column.

In order to properly visualize the 8x8 display, time multiplexing had to be utilized. This means that the FPGA had a counter that loops through the column values and actually only turns on one column at a time. However the columns turn on and off so fast that it is not noticeable to the human eye. Additionally, in order to make the FFT visualization line up more closely with the beat of most music the display can't just change every time an FFT is calculated. So we delay how often an FFT is calculated. Once a FFT is calculated and the display is driven a 22 bit delay counter pauses the state machine so the display can be seen before the FPGA asks for more data and recalculates the FFT. This delay is about 0.1 seconds.

Results

In the end, the project was successful while lacking some final polish and cleanliness. The most challenging part was implementing the FFT in System Verilog. The time that took limited the amount of time we had to polish up our design. The display flashed different columns on the matrix just not the way we expected. We saw from the logic analyzer on the oscilloscope, as in Figure 6, that the FPGA was sending 32 16-bit data signals at the proper sampling rate and that the FPGA only got new samples and reran the FFT every 0.1 seconds as desired.

Figure 6. Timing of sampling and timing of FFT calculation

Therefore, the display could be seen driving lighting up different columns to different heights along with the music as expected. However, the magnitudes in each bin were not as expected. When a square wave was delivered directly to the microphone, the display did not show the fundamental frequency and harmonics as expected. We think this issue is because of the small size of out FFT or because the magnitude in the bins weren't not sorted and calculated as expected. Overall, our display was able to visualize music in bar that somewhat, but not wholly represented the frequencies of the music being heard by the microphone as in Figure 7.

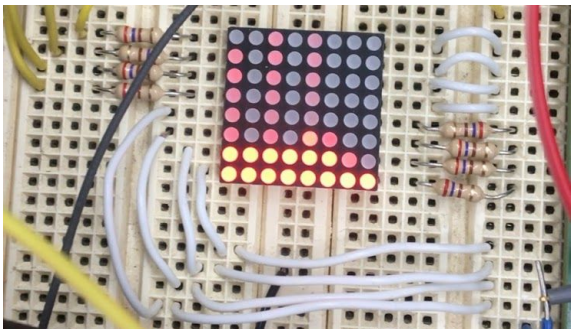

Figure 7. Example of lit up display

References

[1] Maxim, “Low-Cost, Micropower, SC70/SOT23-8, Microphone Preamplifiers with Complete Shutdown,” MAX4466 datasheet, 2001.
[2] Atmel, SAM4S Series, SMART ARM-based Flash MCU,” Atmel-11100K-ATARM-SAM4S-Datasheet, 09 Jun. 2015.
[3] Luckylight, “1.9mm (0.8”) 8x8 Hyper Red Dot Matrix LED Displays Technical Data Sheet,” W07088C/D Rev. V.2, 14 Sept. 2016.
[4] Genta, “深圳市共达光电器件有限公司,” 表单-工程-15-1.
[5] Slade, George. (2013). The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation.

Parts List

Part	**Source**	**Vendor #**	**Price**
The Adafruit Electret Microphone Amplifier - MAX4466 with Adjustable Gain	Adafruit	ADA1063	$6.95
LED Display/Dot Matrix	LuckyLight	KWM-20882CVB	N/A (Stock part)
Geeetech LED Matrix 8x8 - Triple Color RGB common Anode Display -5mm dia	Amazon	GET-0031	$8.95

Appendix A: MCU Code, FPGA Verilog

```
//////////////////////////////////////////////////
// E155 Final Project
//
// File: controller.c
// Date: 12/12/19
// Authors: Jonah Cartwright and Kira Favakeh
// Contact: jcartwright@hmc.edu, kfavakeh@hmc.edu
//
// Convert analog audio signal to digital using ADC and send to FPGA over SPI
//////////////////////////////////////////////////

////////////////////////////////////////////////
// #includes
////////////////////////////////////////////////

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "SAM4S4B_lab7\SAM4S4B_lab7\SAM4S4B.h"

#define AUDIO_IN_PIN PIO_PA17
#define LOAD_PIN    26
#define DONE_PIN    25

////////////////////////////////////////////////
// Main
////////////////////////////////////////////////

int main(void) {

  samInit();
  pioInit();
  spiInit(MCK_FREQ/244000, 0, 1);
  // "clock divide" = master clock frequency / desired baud rate
  // the phase for the SPI clock is 1 and the polarity is 0

  pioPinMode(AUDIO_IN_PIN, PIO_INPUT);
```

```
  pioPinMode(LOAD_PIN, PIO_OUTPUT);
  pioPinMode(DONE_PIN, PIO_INPUT);

adcChannelInit(ADC_CH0, ADC_CGR_GAIN_X2, ADC_COR_OFFSET_ON);
//Enables an ADC channel 0 and initialize gain and offset

adcInit(ADC_MR_LOWRES_BITS_12);
//Enable the ADC peripheral and initialize the resolution

pioDigitalWrite(LOAD_PIN, 1);                   //write load high for initial adc/spi sequence

while(1){
        while(pioDigitalRead(DONE_PIN))         //read and send data when FPGA is ready
        {
                float audioIn = adcRead(ADC_CH0); //Read analog voltage at ADC channel 0
                float audioInx = audioIN*100;   //multiply signal such that accuracy is not
lost in conversion to short
                short audioOut;
                audioOut = (short)audioInx;
                spiSendReceive16(audioOut);     //send audio data to FPGA
                pioDigitalWrite(LOAD_PIN, 0);   //write load low such that the FPGA knows
to end SPI
                        pioDigitalWrite(LOAD_PIN, 1);   //write load high immediately such
that there's no lag when in SPI state
        }
}

}
```

```
1
2  ////////////////////////////////////////////////
3  ////// final_jc_kf4.sv
4  ////// HMC E155 12 December 2019
5  ////// Kira Favakeh and Jonah cartwright
6  ////// kfavakeh@hmc.edu, jcartwright@hmc.edu
7  ////////////////////////////////////////////////
8
9  ////////////////////////////////////////////
10 // testbench
11 //   Tests final
12 ////////////////////////////////////////////
13
14 module testbench();
15   logic clk, reset, sdi, load;
16   logic [15:0] led_display;
17   logic spi_clk, done;
18
19   final_jc_kf4 dut(clk, spi_clk, reset, sdi, load, led_display, done);
20
21   initial begin
22     reset = 1'b0; #5;
23     reset = 1'b1; #5;
24     reset = 1'b0; #5;
25   end
26
27   initial
28     forever begin
29       clk = 1'b0; #12500;
30       clk = 1'b1; #12500;
31     end
32
33   initial
34     forever begin
35       spi_clk = 1'b0; #125000;
36       spi_clk = 1'b1; #125000;
37     end
38
39   initial
40     forever begin
41       sdi = 1'b0; #100000000;
42       sdi = 1'b0; #100000000;
43       sdi = 1'b0; #100000000;
44       sdi = 1'b0; #100000000;
45       sdi = 1'b0; #100000000;
46       sdi = 1'b1; #100000000;
47       sdi = 1'b1; #100000000;
48       sdi = 1'b1; #100000000;
49       sdi = 1'b1; #100000000;
50       sdi = 1'b1; #100000000;
51     end
52
53     assign load = 1'b0;
54
55
56
57 endmodule
58
59
60 module final_jc_kf4(input logic clk, spi_clk, reset, sdi, load,
```

```
61              output logic [15:0] led_display,
62          output logic done);
63
64
65    // Internal Logic
66
67    // Real and Imaginary twiddle factors for butterfly unit
68    logic [15:0] twiddleFactor1, twiddleFactor2;
69    logic [3:0] twiddleAdr;
70
71    // Write Addresses for FFT RAM
72    logic [4:0] writeAddrA, writeAddrB;
73    // Read Addresses for FFT RAM
74    logic [4:0] adrA, adrB;
75
76    // Write to RAM for FFT, Write to RAM for data loading
77    logic memWrite, LoadDataWrite;
78
79    // Choose which RAM to read from
80    logic BankReadSelect;
81
82    // Reset the delay before we display again
83    logic reset_wait;
84
85    // Choose which bank to write to
86    logic Bank0WriteEn, Bank1WriteEn;
87
88    // Addresses to load Data, we use bit reversed addresses
89    logic [4:0] LoadDataAddr, LoadDataAddrRev;
90
91    // Real and Imaginary compentents of FFT
92    logic [15:0] Gr, Gi, Hr, Hi, Xr, Yr, Xi, Yi;
93
94    // Load data is done, Start FFT, done saving FFT data, Display result
95    logic ldwDone, fftStart, save_complete, show;
96
97    // FFT is done, get new data
98    logic fftDone, newdata;
99
100   // Data in from microphone
101   logic [15:0] Data_imag_in, Data_real_in;
102
103   // delays for sampling and display
104   logic delay, delay2, delay3;
105
106   // Iterate load data address
107   logic nextAdr;
108
109   // Magnitude from a bin
110   logic [19:0] final_mag;
111
112   // State Machine to control FPGA
113   typedef enum logic [3:0] {START, SPI, DELAY, RESET_SPI, WRITEDATA,
    S0,S1,S2,S3, S4, S5, PRE_START} statetype;
114   statetype state, nextState;
115
116
117   always_ff @(posedge clk, posedge reset)
118   begin
119     if(reset) state <= START;
```

```
120     else state <= nextState;
121   end
122
123   always_comb
124     case(state)
125       // Declare that we want new data
126       PRE_START: nextState <= START;
127       // Ask for data from Microcontroller
128       START: nextState <= SPI;
129       // Load in first 16 bits of data
130       SPI: if (~load) nextState <= WRITEDATA;
131           else nextState <= SPI;
132       // Write Data to memory
133       WRITEDATA: nextState <= S0;
134       // Delay unless memory is full
135       S0: if (ldwDone) nextState <= S1;
136         else nextState <= DELAY;
137       // Delay before we take new sample (sampling rate ~2441 Hz)
138       DELAY: if (delay) nextState <= RESET_SPI;
139             else nextState <= DELAY;
140       // RESET_SPI
141       RESET_SPI: nextState <= START;
142       // Perform FFT
143       S1: if (fftDone) nextState = S2;
144         else nextState = S1;
145       // Save FFT Magnitude Data
146       S2: if (save_complete) nextState = S3;
147         else nextState = S2;
148       // Reset Delay
149       S3: nextState = S4;
150       // Delay ~ 0.1 seconds until we perform get data and perform fft again
151       S4: if(delay2) nextState <= S5;
152           else nextState <= S4;
153       // Run agin
154       S5: nextState = PRE_START;
155     endcase
156
157
158   // Sampling Delay (40000000 / 2^14 = ~2441 Hz)
159   logic [13:0] delay_counter;
160   always_ff @(posedge clk, posedge reset, posedge LoadDataWrite)
161     if (reset) delay_counter <= 14'b0;
162     else if (LoadDataWrite) delay_counter <= 14'b0;
163     else delay_counter <= delay_counter + 1'b1;
164   assign delay = delay_counter[13];
165
166   // Display Delay (~0.1 seconds)
167   logic [22:0] delay2_counter;
168   always_ff @(posedge clk, posedge reset, posedge reset_wait)
169     if (reset) delay2_counter <= 23'b0;
170     else if (reset_wait) delay2_counter <= 23'b0;
171     else delay2_counter <= delay2_counter + 1'b1;
172   assign delay2 = delay2_counter[22];
173
174
175   // Write Data to memory for data loading
176   assign LoadDataWrite = (state == WRITEDATA);
177   // Iterate Load Address
178   assign nextAdr = (state == S0);
179   // Start FFT
```

```
180   assign fftStart = (state == S1);
181
182   // Reset Load Data Addresses
183   assign newdata = (state == PRE_START);
184   // Reset display delay counter
185   assign reset_wait = (state == S3);
186   // Tell microcontroller that we want data
187   assign done = (state == START | state == SPI);
188
189
190   // Iterate Load Data address and tell us when it is full
191   always_ff @(posedge nextAdr, posedge newdata, posedge reset)
192     begin
193       if (reset) LoadDataAddr <= 0;
194       else if (newdata)
195           begin
196             LoadDataAddr <= 0;
197             ldwDone <= 0;
198           end
199       else
200         if (nextAdr)
201           {ldwDone, LoadDataAddr} <= LoadDataAddr + 1'b1;
202   end
203
204   // Get bit reversed address we are actually writing to
205   assign LoadDataAddrRev = {LoadDataAddr[0], LoadDataAddr[1],
LoadDataAddr[2],
206               LoadDataAddr[3], LoadDataAddr[4]};
207
208   // Generate Addresses
209    AGU #(5) agu(clk, fftStart, fftDone, adrA, adrB, memWrite, BankReadSelect,
twiddleAdr);
210   // get twiddle factors
211   twiddleFactors twiddle(twiddleAdr, twiddleFactor1, twiddleFactor2);
212   // read and write from memory
213   memory mem(clk, LoadDataWrite, Bank0WriteEn, Bank1WriteEn, Data_real_in,
Data_imag_in, BankReadSelect, LoadDataAddrRev, adrA, writeAddrA, adrB,
writeAddrB, Xr, Xi, Yr, Yi, Gr, Gi, Hr, Hi, final_mag);
214   // Perform butterfly
215   BFU butterfly(twiddleFactor1, twiddleFactor2, Gr, Gi, Hr, Hi, Xr, Xi, Yr,
Yi);
216
217   // FFT Magnitude Data
218   logic [159:0] fft_data;
219
220   // Counter to save data
221   logic [9:0] k = 9'd0;
222
223   // enable save new fft data
224   logic save_enable;
225   assign save_enable = (fftDone & (state == S2));
226
227   // Output FFT magnitude
228   FFTout savedata(clk,reset, save_enable, final_mag, k, fft_data,
save_complete);
229
230   // Change write address to next be read addresses
231   always_ff @(posedge clk)
232       begin
233             writeAddrA <= adrA;
```

```
234             writeAddrB <= adrB;
235       end
236
237     assign Bank0WriteEn = BankReadSelect & memWrite; //Read from 1, write to
    0
238     assign Bank1WriteEn = ~BankReadSelect & memWrite; //Read from 0, write to
    1
239
240
241   // Iterate save data addresses
242   always_ff @(posedge clk)
243     if (reset)
244       k <= 9'b0;
245     else if (LoadDataWrite)
246       k <= 9'b0;
247     else if (save_enable)
248       k <= k + 9'd20;
249
250
251    // Time Multiplex columns
252   logic [2:0] column;
253
254   always_ff @(posedge clk, posedge reset)
255     if (reset) column <= 3'b0;
256     else column <= column + 1'b1;
257
258   // drive display
259   drive_display drive_led(clk, reset, show, column, fft_data, led_display);
260
261   // Get data over spi
262   spi spi_master(clk, spi_clk, sdi, load, Data_real_in, Data_imag_in);
263
264 endmodule
265
266
267 module spi (input logic clk, spi_clk,
268         input logic sdi, load,
269         output logic [15:0] Data_real_in,
270         output logic [15:0] Data_imag_in);
271
272
273    // Apply 16 spi_clk to shift in microphone data
274     always_ff @(posedge spi_clk)
275         if (~load)  {Data_real_in} <= {Data_real_in};
276         else
277           begin
278             {Data_real_in} <= {Data_real_in[14:0], sdi};
279           end
280
281   // Imaginary Data is always zero
282   assign Data_imag_in = Data_real_in & 16'b0;
283
284
285 endmodule
286
287
288
289
290
291
```

```
292 //lookup• •table• •for• •twiddle• •addresses
293 module twiddleFactors(input logic [3:0] twiddleAdr,
294                 output logic [15:0] twiddleFactor1, // Real Twiddle Factor
295                 output logic [15:0] twiddleFactor2); // Imaginary Twiddle
    Factor
296
297     always_comb
298     begin
299         case(twiddleAdr)
300             4'b0000:
301                 begin
302                     twiddleFactor1 = 16'h7fff;
303                     twiddleFactor2 = 16'h0000;
304                 end
305             4'b0001:
306                 begin
307                     twiddleFactor1 = 16'h7d89;
308                     twiddleFactor2 = 16'h18f9;
309                 end
310             4'b0010:
311                 begin
312                     twiddleFactor1 = 16'h7641;
313                     twiddleFactor2 = 16'h30fb;
314                 end
315             4'b0011:
316                 begin
317                     twiddleFactor1 = 16'h6a6d;
318                     twiddleFactor2 = 16'h471c;
319                 end
320         4'b0100:
321           begin
322             twiddleFactor1 = 16'h5a82;
323             twiddleFactor2 = 16'h5a82;
324           end
325         4'b0101:
326           begin
327             twiddleFactor1 = 16'h471c;
328             twiddleFactor2 = 16'h6a6d;
329           end
330         4'b0110:
331           begin
332             twiddleFactor1 = 16'h30fb;
333             twiddleFactor2 = 16'h7641;
334           end
335         4'b0111:
336           begin
337             twiddleFactor1 = 16'h18f9;
338             twiddleFactor2 = 16'h7d89;
339           end
340
341         4'b1000:
342           begin
343             twiddleFactor1 = 16'h0000;
344             twiddleFactor2 = 16'h7fff;
345           end
346
347         4'b1001:
348           begin
349             twiddleFactor1 = 16'he707;
350             twiddleFactor2 = 16'h7d89;
```

```
351           end
352         4'b1010:
353           begin
354             twiddleFactor1 = 16'hcf05;
355             twiddleFactor2 = 16'h7641;
356           end
357         4'b1011:
358           begin
359             twiddleFactor1 = 16'hb8e4;
360             twiddleFactor2 = 16'h6a6d;
361           end
362         4'b1100:
363           begin
364             twiddleFactor1 = 16'ha57e;
365             twiddleFactor2 = 16'h5a82;
366           end
367         4'b1101:
368           begin
369             twiddleFactor1 = 16'h9593;
370             twiddleFactor2 = 16'h471c;
371           end
372 
373         4'b1110:
374           begin
375             twiddleFactor1 = 16'h89bf;
376             twiddleFactor2 = 16'h30fb;
377           end
378 
379         4'b1111:
380           begin
381             twiddleFactor1 = 16'h8277;
382             twiddleFactor2 = 16'h18f9;
383           end
384       endcase
385     end
386 
387 
388 endmodule
389 
390 // Perform Butterfly Operation
391 module butterflyUnit(input logic [15:0] twiddleFactor1,
392                          input logic [15:0] twiddleFactor2,
393                          input logic [15:0] BFUdataA_real,
394                          input logic [15:0] BFUdataB_real,
395                          input logic [15:0] BFUdataA_imag,
396                          input logic [15:0] BFUdataB_imag,
397                          output logic [15:0] memDataA_real,
398                          output logic [15:0] memDataB_real,
399                          output logic [15:0] memDataA_imag,
400                          output logic [15:0] memDataB_imag);
401 
402     logic signed [31:0] BMult_real, BMult_imag;
403 
404     // Make signed values to do signed multiplaction
405     logic signed [15:0] s_twiddleFactor1, s_twiddleFactor2;
406     logic signed [15:0] s_BFUdataB_real, s_BFUdataB_imag;
407     logic signed [31:0] s_BMult_real, s_BMult_imag;
408 
409     assign s_twiddleFactor1 = twiddleFactor1;
410     assign s_twiddleFactor2 = twiddleFactor2;
```

```
411     assign s_BFUdataB_real = BFUdataB_real;
412     assign s_BFUdataB_imag = BFUdataB_imag;
413     assign s_BMult_real = (s_BFUdataB_real*s_twiddleFactor1) -
    (s_BFUdataB_imag*s_twiddleFactor2);
414     assign s_BMult_imag = (s_BFUdataB_real*s_twiddleFactor2) +
    (s_BFUdataB_imag*s_twiddleFactor1);
415 
416     assign BMult_real = s_BMult_real;
417     assign BMult_imag = s_BMult_imag;
418 
419     assign memDataA_real = (BFUdataA_real+BMult_real[30:15]);
420     assign memDataA_imag = (BFUdataA_imag+BMult_imag[30:15]);
421     assign memDataB_real = (BFUdataA_real-BMult_real[30:15]);
422     assign memDataB_imag = (BFUdataA_imag-BMult_imag[30:15]);
423 
424 endmodule
425 
426 
427 
428 module memory(input logic clk,
429             input logic LoadDataWrite,
430             input logic Bank0WriteEN, Bank1WriteEN,
431             input logic [15:0] Data_real_in,
432             input logic [15:0] Data_imag_in,
433             input logic BankReadSelect,
434             input logic [4:0] LoadDataAddr,
435             input logic [4:0] ReadAddrG,
436             input logic [4:0] WriteAddrG,
437             input logic [4:0] ReadAddrH,
438             input logic [4:0] WriteAddrH,
439             input logic [15:0] Xr,
440             input logic [15:0] Xi,
441             input logic [15:0] Yr,
442             input logic [15:0] Yi,
443             output logic [15:0] Gr, Gi, Hr, Hi,
444         output logic [19:0] final_mag);
445 
446     logic [4:0] addrA0, addrA1, addrB0, addrB1;
447     logic [15:0] DataA0_r, DataA0_i;
448     logic [15:0] preG0r, preG0i, preH0r, preH0i, preG1r, preG1i, preH1r,
    preH1i;
449     logic Bank0_A_WR, Bank0_B_WR;
450     logic [15:0] DataB_r, DataB_i, DataA1_r, DataA1_i;
451     logic Bank1WrEnDelay;
452 
453     assign Bank0_A_WR = Bank0WriteEN | LoadDataWrite;
454 
455    assign DataA0_r = LoadDataWrite ? Data_real_in : Xr;
456    assign DataA0_i = LoadDataWrite ? Data_imag_in : Xi;
457 
458 
459     assign addrA0 = LoadDataWrite ? LoadDataAddr : (Bank0WriteEN ? WriteAddrG
    : ReadAddrG);
460     assign addrB0 = Bank0WriteEN ? WriteAddrH : ReadAddrH;
461     assign addrA1 = LoadDataWrite ? LoadDataAddr : (Bank1WriteEN ? WriteAddrG
    : ReadAddrG);
462     assign addrB1 = Bank1WriteEN ? WriteAddrH : ReadAddrH;
463 
464     // Instantiate RAM
```

```
465     twoportram
real0(clk,DataA0_r,Yr,addrA0,addrB0,Bank0_A_WR,Bank0WriteEN,preG0r,preH0r);
466     twoportram
im0(clk,DataA0_i,Yi,addrA0,addrB0,Bank0_A_WR,Bank0WriteEN,preG0i,preH0i);
467
468     twoportram
real1(clk,Xr,Yr,addrA1,addrB1,Bank1WriteEN,Bank1WriteEN,preG1r,preH1r);
469     twoportram
im1(clk,Xi,Yi,addrA1,addrB1,Bank1WriteEN,Bank1WriteEN,preG1i,preH1i);
470
471
472     assign Gr = BankReadSelect ? preG1r : preG0r;
473     assign Gi = BankReadSelect ? preG1i : preG0i;
474     assign Hr = BankReadSelect ? preH1r : preH0r;
475     assign Hi = BankReadSelect ? preH1i : preH0i;
476
477    // Calculate Magnitude of FFT bins
478    // Technically realtive magnitude becuase we don't square the results
479    logic signed [15:0] signed_real_even;
480    logic signed [15:0] signed_imag_even;
481    logic signed [31:0] signed_mag_even;
482    logic [15:0] even_mag;
483
484    assign signed_real_even = preH1r;
485    assign signed_imag_even = preH1i;
486
487    assign signed_mag_even = (signed_real_even * signed_real_even) +
(signed_imag_even * signed_imag_even);
488
489    assign even_mag = signed_mag_even[30:15];
490
491    logic signed [15:0] signed_real_odd;
492    logic signed [15:0] signed_imag_odd;
493    logic signed [31:0] signed_mag_odd;
494    logic [15:0] odd_mag;
495
496    assign signed_real_odd = preG1r;
497    assign signed_imag_odd = preG1i;
498
499    assign signed_mag_odd = (signed_real_odd * signed_real_odd) +
(signed_imag_odd * signed_imag_odd);
500
501    assign odd_mag = signed_mag_odd[30:15] ;
502
503
504    // We only have 8 columns but 16 bins below one half of the sampling
frequency
505    // so we combine adjacent bins (i.e. 0 and 1, 2 and 3, etc.)
506    assign final_mag =  even_mag + odd_mag;
507
508 endmodule
509
510
511 // Two port RAM module
512 module twoportram
513                 (input logic clk,
514                     input logic [15:0] dataA, dataB,
515                     input logic [4:0] addrA, addrB,
516                     input logic writeEnA, writeEnB,
517                     output logic [15:0]qA, qB);
```

```
518
519     logic [15:0] mem[31:0];
520
521     always_ff @(posedge clk)
522         begin
523             if (writeEnA)
524                 begin
525                     mem[addrA] <= dataA;
526                     qA <= dataA;
527                 end
528             else
529                 qA <= mem[addrA];
530         end
531     always_ff @(posedge clk)
532         begin
533             if (writeEnB)
534                 begin
535                     mem[addrB] <= dataB;
536                     qB <= dataB;
537                 end
538             else
539                 qB <= mem[addrB];
540             end
541
542
543 endmodule
544
545 // Address Generation Unit
546 module AGU #(parameter logN = 5)
547             (input logic clk,
548             input logic fftStart,
549             output logic fftDone,
550             output logic [logN-1:0] memA_addr,
551             output logic [logN-1:0] memB_addr,
552             output logic memWrite, bankReadSelect,
553             output logic [logN-2:0] twiddleAdr
554             );
555
556     logic prevfftStart = 1'b0;
557     logic clear;
558
559     logic [1:0] delayCounter = 2'b00;
560     logic [3:0] i = 4'b000;       // level counter
561     logic [logN-2:0] j = 4'b0000;   // index counter
562     logic [logN-2:0] tw_addr;
563     logic [logN-1:0] j_shift;
564
565     logic [logN-2:0] ones = ~4'b0;
566     logic [logN-2:0] zeros = 4'b0;
567
568     always_ff @(posedge clk)
569         begin
570             prevfftStart <= fftStart;
571         end
572
573     assign clear = ~prevfftStart & fftStart;
574
575    always_ff @(posedge clk, posedge clear)
576         begin
577             if (clear)
```

```
578                 begin
579                     j <= 0;
580                     delayCounter <= 0;
581                 end
582             else if (j != (1 << (logN-1)) - 1) // j < 16
583           j <= j + 1'b1;
584             else
585                 begin
586                     delayCounter <= delayCounter + 1'b1;
587                     if (delayCounter == 2)
588                         begin
589                             j <= 0;
590                             delayCounter <= 0;
591                         end
592                 end
593         end
594 
595     always_ff @(posedge clk, posedge clear)
596         begin
597             if(clear)
598                 begin
599                     i <= 0;
600                     fftDone <= 0;
601                 end
602             else if (delayCounter == 2)
603                 if (i < logN) // i < 5
604                     i <= i + 1'b1;
605                 else
606                     begin
607                         i <= 0;
608                         fftDone <= 1;
609                     end
610         end
611 
612     assign j_shift = j << 1;
613     assign memA_addr = (j_shift << i) | (j_shift >> (logN - i));
614     assign memB_addr = ((j_shift + 1'b1) << i) | ((j_shift + 1'b1) >> (logN -
    i));
615 
616     assign tw_addr = ({ones, zeros} >> i) & j;
617 
618     always_ff @(posedge clk)
619         twiddleAdr <= tw_addr;
620 
621     assign bankReadSelect = i[0];
622 
623     assign memWrite = (~fftDone & ~((delayCounter == 2) | (j == 0)));
    //memWrite low when j = 0 and the clock cycle before that
624 
625 endmodule
626 
627 // Save FFT Magbitude to give to display
628 module FFTout(input logic clk, reset,
629           input logic enable,
630           input logic [19:0] final_mag,
631           input logic [9:0] SaveDataAdr,
632           output logic [159:0] fftOut,
633           output logic save_complete);
634 
635     assign save_complete = (SaveDataAdr == 9'd160);
```

```
636
637     always_ff @(posedge clk, posedge reset)
638         begin
639         if (reset)
640             begin
641               fftOut <= 0;
642             end
643         else if (enable & ~save_complete)
644           fftOut[SaveDataAdr +: 20] <= final_mag;
645         end
646
647
648 endmodule
649
650
651
652 // Drive LED display
653 module drive_display (input logic clk, reset,
654                 input logic [2:0] column,
655                         input logic [159:0] fft_data,
656                         output logic [15:0] display);
657
658   // For Each column turn on that column and display the relative magnitude
659     always_ff @(posedge clk, posedge reset)
660         if (reset) display <= 16'b1111111100000000;
661         else
662         begin
663           if (column == 3'b000)
664             begin
665               if  (fft_data[159:140] < 16'h0002)
666                   display <= 16'b1111111000000001;
667               else if  (16'h0002 < fft_data[159:140] & fft_data[159:140] <
    16'h0004)
668                   display <= 16'b1111111000000001;
669               else if ( 16'h0004 < fft_data[159:140] & fft_data[159:140] <
    16'h0021)
670                 display <= 16'b1111110000000001;
671               else if  (16'h0021 < fft_data[159:140] & fft_data[159:140] <
    16'h0064)
672                 display <= 16'b1111100000000001;
673               else if  (16'h0064 < fft_data[159:140] & fft_data[159:140] <
    16'h0256)
674                 display <= 16'b1111000000000001;
675               else if  (16'h0256 < fft_data[159:140] & fft_data[159:140] <
    16'h0512)
676                 display <= 16'b1110000000000001;
677               else if  (16'h0512 < fft_data[159:140] & fft_data[159:140] <
    16'h0768)
678                 display <= 16'b1100000000000001;
679               else if  (16'h0768 < fft_data[159:140] & fft_data[159:140] <
    16'h1280)
680                 display <= 16'b1000000000000001;
681               else
682                 display <= 16'b0000000000000001;
683             end
684
685
686           else if (column == 3'b001)
687             begin
688               if  (fft_data[139:120] < 16'h0002)
```

```
689                 display <= 16'b1111110000000010;
690               else if  (16'h0002 < fft_data[139:120] & fft_data[139:120] <
    16'h0004)
691                 begin
692                   flag <= 1'b1;
693                   display <= 16'b1111111000000010;
694                 end
695               else if  (16'h0004 < fft_data[139:120] & fft_data[139:120] <
    16'h0021)
696                 display <= 16'b0111110000000010;
697               else if  (16'h0021 < fft_data[139:120] & fft_data[139:120] <
    16'h0064)
698                 display <= 16'b1111100000000010;
699               else if  (16'h0064 < fft_data[139:120] & fft_data[139:120] <
    16'h0256)
700                 display <= 16'b1111000000000010;
701               else if  (16'h0256 < fft_data[139:120] & fft_data[139:120] <
    16'h0512)
702                 display <= 16'b1110000000000010;
703               else if  (16'h0512 < fft_data[139:120] & fft_data[139:120] <
    16'h0768)
704                 display <= 16'b1100000000000010;
705               else if  (16'h0768 < fft_data[139:120] & fft_data[139:120] <
    16'h1280)
706                 display <= 16'b1000000000000010;
707               else
708                 display <= 16'b0000000000000010;
709             end
710
711
712           else if (column == 3'b010)
713             begin
714               if  (fft_data[119:100] < 16'h0002)
715                 display <= 16'b1111100000000100;
716               else if  (16'h0002 < fft_data[119:100] & fft_data[119:100] <
    16'h0004)
717                 display <= 16'b1111111000000100;
718               else if ( 16'h0004 < fft_data[119:100] & fft_data[119:100] <
    16'h0021)
719                 display <= 16'b1111110000000100;
720               else if  (16'h0021 < fft_data[119:100] & fft_data[119:100] <
    16'h0064)
721                 display <= 16'b1111100000000100;
722               else if  (16'h0064 < fft_data[119:100] & fft_data[119:100] <
    16'h0256)
723                 display <= 16'b1111000000000100;
724               else if  (16'h0267 < fft_data[119:100] & fft_data[119:100] <
    16'h0512)
725                 display <= 16'b1110000000000100;
726               else if  (16'h0512 < fft_data[119:100] & fft_data[119:100] <
    16'h0768)
727                 display <= 16'b1100000000000100;
728               else if  (16'h0768 < fft_data[119:100] & fft_data[119:100] <
    16'h1280)
729                 display <= 16'b1000000000000100;
730               else
731                 display <= 16'b0000000000000100;
732             end
733
734
```

```
735           else if (column == 3'b011)
736             begin
737               if  (fft_data[99:80] < 16'h0002)
738                 display <= 16'b1111111000001000;
739               else if  (16'h0002 < fft_data[99:80] & fft_data[99:80] <
    16'h0004)
740                 display <= 16'b1111111000001000;
741               else if ( 16'h0004 < fft_data[99:80] & fft_data[99:80] <
    16'h0021)
742                 display <= 16'b1111110000001000;
743               else if  (16'h0021 < fft_data[99:80] & fft_data[99:80] <
    16'h0064)
744                 display <= 16'b1111100000001000;
745               else if  (16'h0064 < fft_data[99:80] & fft_data[99:80] <
    16'h0256)
746                 display <= 16'b1111000000001000;
747               else if  (16'h0256 < fft_data[99:80] & fft_data[99:80] <
    16'h0512)
748                 display <= 16'b1110000000001000;
749               else if  (16'h0512 < fft_data[99:80] & fft_data[99:80] <
    16'h0768)
750                 display <= 16'b1100000000001000;
751               else if  (16'h0768 < fft_data[99:80] & fft_data[99:80] <
    16'h1280)
752                 display <= 16'b1000000000001000;
753               else
754                 display <= 16'b0000000000001000;
755             end
756
757
758           else if (column == 3'b100)
759             begin
760               if  (fft_data[79:60] < 16'h0002)
761                 display <= 16'b1111110000010000;
762               else if  (16'h0002 < fft_data[79:60] & fft_data[79:60] <
    16'h0004)
763                 display <= 16'b1111111000010000;
764               else if ( 16'h0004 < fft_data[79:60] & fft_data[79:60] <
    16'h0021)
765                 display <= 16'b1111110000010000;
766               else if  (16'h0021 < fft_data[79:60] & fft_data[79:60] <
    16'h0064)
767                 display <= 16'b1111100000010000;
768               else if  (16'h0064 < fft_data[79:60] & fft_data[79:60] <
    16'h0256)
769                 display <= 16'b1111000000010000;
770               else if  (16'h0256 < fft_data[79:60] & fft_data[79:60] <
    16'h0512)
771                 display <= 16'b1110000000010000;
772               else if  (16'h0512 < fft_data[79:60] & fft_data[79:60] <
    16'h0768)
773                 display <= 16'b1100000000010000;
774               else if  (16'h0768 < fft_data[79:60] & fft_data[79:60] <
    16'h1280)
775                 display <= 16'b1000000000010000;
776               else
777                 display <= 16'b0000000000010000;
778             end
779
780
```

```
781           else if (column == 3'b101)
782             begin
783               if  (fft_data[59:40] < 16'h0002)
784                 display <= 16'b1111100000100000;
785               else if  (16'h0002 < fft_data[59:40] & fft_data[59:40] <
    16'h0004)
786                 display <= 16'b1111111000100000;
787               else if ( 16'h0004 < fft_data[59:40] & fft_data[59:40] <
    16'h0021)
788                 display <= 16'b1111110000100000;
789               else if  (16'h0021 < fft_data[59:40] & fft_data[59:40] <
    16'h0064)
790                 display <= 16'b1111100000100000;
791               else if  (16'h0064 < fft_data[59:40] & fft_data[59:40] <
    16'h0256)
792                 display <= 16'b1111000000100000;
793               else if  (16'h0256 < fft_data[59:40] & fft_data[59:40] <
    16'h0512)
794                 display <= 16'b1110000000100000;
795               else if  (16'h0512 < fft_data[59:40] & fft_data[59:40] <
    16'h0768)
796                 display <= 16'b1100000000100000;
797               else if  (16'h0768 < fft_data[59:40] & fft_data[59:40] <
    16'h1280)
798                 display <= 16'b1000000000100000;
799               else
800                 display <= 16'b0000000000100000;
801             end
802
803
804           else if (column == 3'b110)
805             begin
806               if  (fft_data[39:20] < 16'h0002)
807                 display <= 16'b1111111001000000;
808               else if  (16'h0002 < fft_data[39:20] & fft_data[39:20] <
    16'h0004)
809                 display <= 16'b1111111001000000;
810               else if ( 16'h0004 < fft_data[39:20] & fft_data[39:20] <
    16'h0021)
811                 display <= 16'b1111110001000000;
812               else if  (16'h0021 < fft_data[39:20] & fft_data[39:20] <
    16'h0064)
813                 display <= 16'b1111100001000000;
814               else if  (16'h0064 < fft_data[39:20] & fft_data[39:20] <
    16'h0256)
815                 display <= 16'b1111000001000000;
816               else if  (16'h0256 < fft_data[39:20] & fft_data[39:20] <
    16'h0512)
817                 display <= 16'b1110000001000000;
818               else if  (16'h0512 < fft_data[39:20] & fft_data[39:20] <
    16'h0768)
819                 display <= 16'b1100000001000000;
820               else if  (16'h0768 < fft_data[39:20] & fft_data[39:20] <
    16'h1280)
821                 display <= 16'b1000000001000000;
822               else
823                 display <= 16'b0000000001000000;
824             end
825
826
```

```
827           else if (column == 3'b111)
828             begin
829               if  (fft_data[19:0] < 16'h0002)
830                 display <= 16'b1111111110000000;
831               else if  (16'h0002 < fft_data[19:0] & fft_data[19:0] <
    16'h0004)
832                 display <= 16'b1111111010000000;
833               else if ( 16'h0004 < fft_data[19:0] & fft_data[19:0] <
    16'h0021)
834                 display <= 16'b1111110010000000;
835               else if  (16'h0021 < fft_data[19:0] & fft_data[19:0] <
    16'h0064)
836                 display <= 16'b1111100010000000;
837               else if  (16'h0064 < fft_data[19:0] & fft_data[19:0] <
    16'h0256)
838                 display <= 16'b1111000010000000;
839               else if  (16'h0256 < fft_data[19:0] & fft_data[19:0] <
    16'h0512)
840                 display <= 16'b1110000010000000;
841               else if  (16'h0512 < fft_data[19:0] & fft_data[19:0] <
    16'h0768)
842                 display <= 16'b1100000010000000;
843               else if  (16'h0768 < fft_data[19:0] & fft_data[19:0] <
    16'h1280)
844                 display <= 16'b1000000010000000;
845               else
846                 display <= 16'b0000000010000000;
847             end
848 
849           else
850             display <= 16'b1111111100000000;
851         end
852 
853 endmodule
854 
855 
856 
857 
858 
859 
860 
861 
862 
863 
864 
```