ATLAS: Audio Tracking Laser-Assisted Light Show

Arya Goutam and Alec Vercruysse

CONTENTS

***Abstract*—We propose a design of a system that projects a laser light show that synchronizes to to music. The system samples audio input with a PCM1808 stereo ADC, and a MAX10 FPGA analyzes the frequency content of the song to find percussive elements in order to determine beat onsets. An STM32 microcontroller uses the beat onset locations to control a servo that rotates a multi-pattern diffraction grating and two laser diodes which shine through this grating. While in practice the design requires the song to have percussive elements and the algorithm is sensitive to input audio volume, the system can accurately identify and rapidly responds to beats and produces a compelling visual effect.**

I. INTRODUCTION

A. Motivation

Light shows are often used to enhance music but coordinating lighting with music is challenging to do on the fly, especially with a low-cost setup. Our goal for this project was to design a system that automatically creates interesting lighting effects which are synchronized to a live audio input.

A. Goutam and A. Vercruysse are Engineering Majors in the class of 2023 at Harvey Mudd College, CA.

B. Overview

The system shown in Figure 1 above is described here:

1) Stereo audio is passed from a computer to the PCM.
2) The PCM passes digital audio output to the FPGA.
3) The FPGA reads the digital audio, computes beat onsets and sends pulses to the MCU when it detects a beat onset.
4) The MCU uses the beat onset pulses to compute the song's tempo, choose a light pattern according to the tempo, and synchronize the pattern with the beats of the song.
5) The synchronized light pattern is displayed via control signals to the lasers and the servo.

II. DESIGN

The system consists of a straightforward signal path that starts as the PCM1808 ADC samples the stereo audio input, and is then processed by the MAX10 FPGA. The FPGA then outputs beat onset information to the STM32 MCU, which drives a hobby servo motor with diffraction grating attached.

A. FPGA Design

The 10M08SAU169 FPGA is responsible for interfacing with the PCM1808 stereo audio ADC as well as performing the digital signal processing required to determine beat onsets from the audio signal. Beat detection is performed based on the threshold detection of a spectral-based novelty function.

TABLE I
10M08SAU169 FPGA RESOURCE USAGE

Total Logic Elements	6,060 / 8,064	75%
Total Pins	26 / 130	20%
Total Memory Bits	1,024 / 387,072	$< 1\%$
Total 9-bit multipliers	12 / 48	25%

This FPGA had just enough logic elements to implement all modules, including I2S, FFT, and beat detection. Table I demonstrates that 75% of logic elements are being used by the current design. However, since memory usage is extremely small (corresponding to a single 32 bit x 32 sample RAM), Quartus is preferring to infer register logic rather than memory for some memory modules including twiddle bit ROM and FFT RAM.

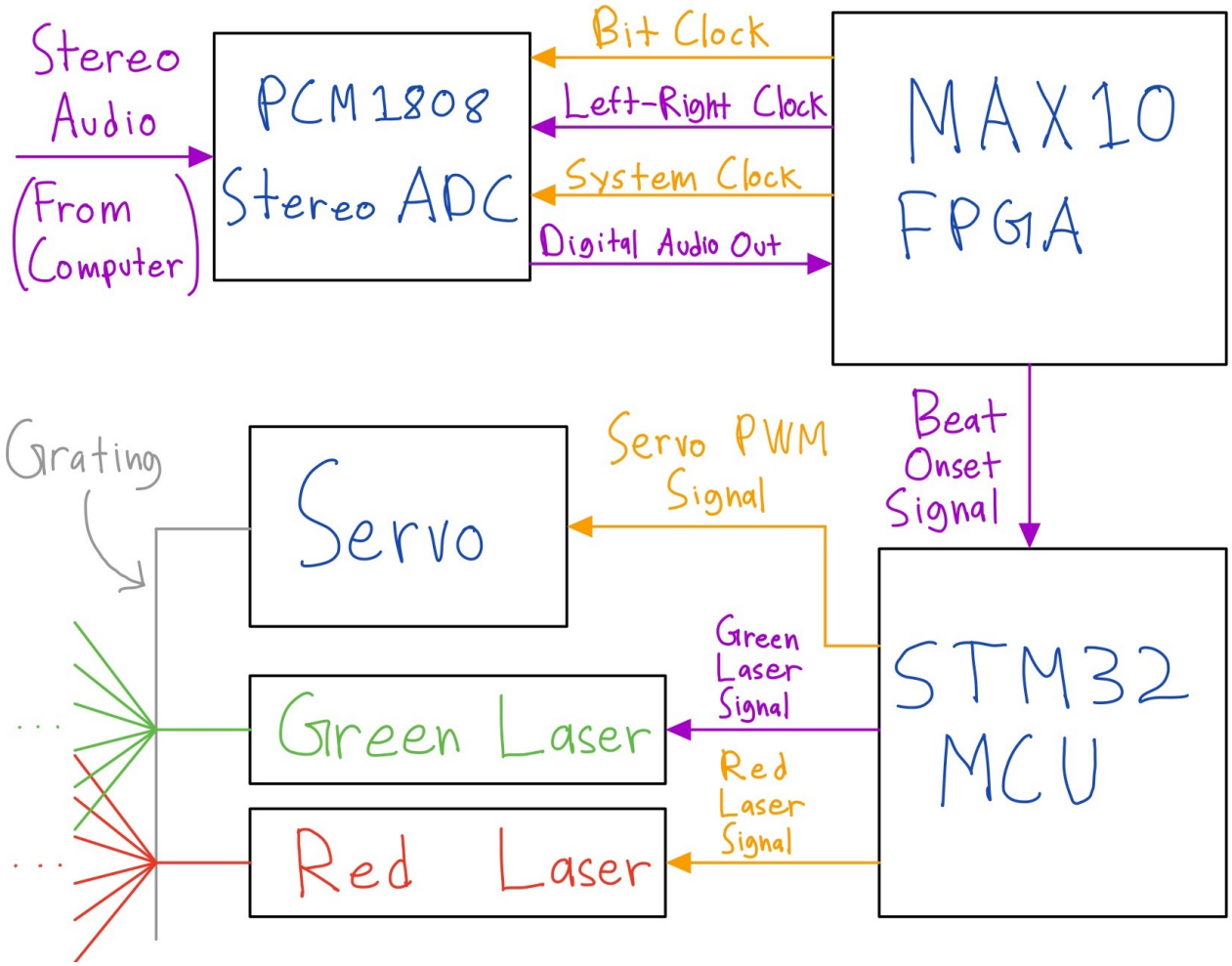

Fig. 1. System Block diagram displaying all components and signal paths within system.

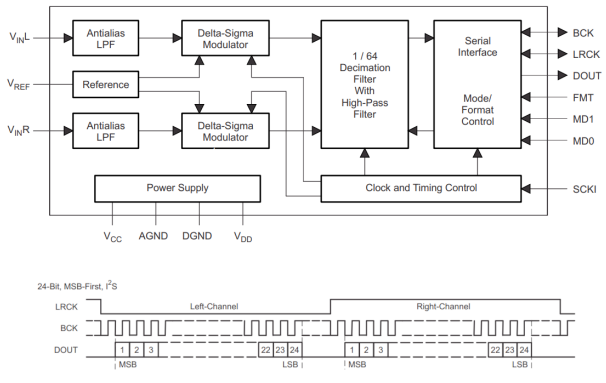

Fig. 2. Block diagram and relevant I2S communication protocol of the of the PCM1808 ADC used for audio input sampling. `MD1=0`, `MD0=0`, and `FMT=0` to set the operating mode to slave-mode with I2S communication. `AGND` and `DGND` were tied together, with `VCC` and `VDD` tied to the 5V and 3.3V outputs of the Nucleo-64 development board, respectively.

1) ADC Interface: Although the FPGA includes an on-board ADC, an external one was used to sample the audio. This was done in order to reduce the complexity of external hardware as well as provide for maximum possible flexibility and minimum possible noise.

Since conventional line-level audio signals are stereo, two ADC channels are required in order to sample the song. Furthermore, line-level audio is zero centered, while the FPGA can only work with positive voltage, so therefore the signal would require biasing to sample with an FPGA. Lastly, the PCM1808 is much higher-performance than the FPGA – not only does it have 24 bit resolution (versus 12), it also performs all required anti-aliasing filtering before sampling. For a nominal 44.875 kHz sample rate, the PCM1808 samples at 3 MSPS and performs decimation and high pass to ensure that no aliasing occurs and no DC component exists in the signal. In addition, the PCM1808 exposes an extremely simple I2S slave-mode interface that allows for a fully synchronous design with the FPGA driving all sampling clocks, to allow for simple integration with the rest of the system. Lastly, the device requires only 3.3V and 5V power supplies, both of which already exist on the MAX10/Nucleo-64 development platform that is used for all work in E155. The block diagram and I2S communication example waveform can be seen in Fig.

2.

Since the purpose of the E155 final project focused on digital logic design and no points were given for analog front end (AFE) design, the flexibility of the simple all-in-one solution of the PCM1808 was preferred to a more custom AFE that leveraged the existing ADC on-board the FPGA. The low component count required of this design also allowed for the extremely quick single-iteration development of a PCB for the PCM1808 and all associated audio jacks required to interface the system with input audio.

A custom PCB was developed that included input (and output) 3.5mm and RCA audio jacks that allow the system to sit as a "man-in-the-middle" between an audio source and amplifier or listening device. The schematic of the PCB can be seen in Fig. 3. For maximum flexibility, all control signals were routed to the FPGA. Ultimately, the slave-mode I2S interface was selected, sampling at 46.875 kHz. This sampling frequency allows for a Nyquist frequency above 20 kHz, which is considered to be the upper frequency limit of human hearing. Therefore this sampling frequency should accurately capture all audible information in the analog audio input. Since the FPGA runs at 12MHz, the PCM1808 system clock (`scki`), which runs at $256 * 46.875\text{KHz} = 12\text{MHz}$ was derived directly from the FPGA. Furthermore, a power of 2-based prescaler was used to derive the associated left/right clock (`lrck`) and bit clock (`bck`) signals required for I2S communication.

Since all I2S clock signals were derived from the same prescale counter, the value of the counter was also used to determine state for the I2S master input module on the FPGA. For example, the bit index was determined by sampling the 5 bits (6:2) above the bit index sampled to derive the bit clock (1).

The bit index and left-right select clock were used to determine when to shift input data into the left and right channel shift registers. At the end of each frame consisting of a left and right channel sample, stable left/right sample registers were written to and an output "sample ready" (`newsample_valid`) signal was generated.

The operation of the I2S input module was verified via a unit-testbench in ModelSim.

2) Fast Fourier Transform Computation: The 32-point Fast Fourier Transform (FFT) is computed using a multi-cycle module based on the non-pipelined reference implementation by Slade in [1]. The design consists of a single butterfly unit which reads from and writes to two dual-port RAM blocks "ping-pong" configuration. That is, for a certain level of FFT computation, one RAM is reading pairs of a and b inputs to the butterfly while the other is writing the resultant values. On the next level, the reading RAM becomes the writing one, and the writing RAM becomes the reading one. An address generation unit (AGU) controls the logic to handle both the ping-pong operation of the RAM as well as determine the pairs of values that need to serve as inputs to the butterfly operation at each clock cycle. A block diagram of this implementation is shown in Figure 4.

Additional control logic was implemented in order to load the RAM with samples prior to computation, read the output spectrum values after computation, and reset the module after computation.

The state of the FPGA was determined by an external control Finite State Machine composed of the states `FFT_LOADING`, `FFT_STARTING`, `FFT_WAITING`, `FFT_WRITING` and `FFT_RESETTING`. The state machine waits in the `FFT_LOADING` state until 32 samples are loaded into the FFT, at which point it enters the `FFT_STARTING` state to drive the start logic. The state machine then enters the `FFT_WAITING` logic until it receives the done signal from the AGU. At this point the state machine enters the `FFT_WRITING` state to allow the output spectrum values to be read into another module. This control module also generates the addresses required to properly load and write out the data into and out of the FFT module. Once all values are written, the state machine enters `FFT_RESETTING` to reset the AGU before the next computation. After this, the state goes back to `FFT_LOADING` and the cycle is repeated.

The load logic was implemented by inferring multiplexers between the AGU and RAM0 that allowed the write-enable, write-data, and address lines to be controlled by a load module that takes the input samples and writes them in the correct order to the FFT RAM [1].

The write logic was implemented in a similar manner, by inferring multiplexers that allowed the external FFT control module to access the address port of the RAM containing the output values.

The FFT computation is computed in less time than it takes to receive a single sample from the ADC, so all samples read in by the ADC are ultimately fed into the FFT module. This essentially results in an STFT module with a hop size equal to the window size of 32.

The FFT module was verified with a unit-testbench in ModelSim, specifically by comparing output spectra to ones computed on identical input in Python with NumPy. Note that the output values reported in Table V of Slade do not match the values reported in Figure 3 of Slade, and are incorrect. Furthermore, our implementation uses roots-of-unity (twiddle) values $w_m = e^{-j2\pi m/N}$, while Slade uses a less standard convention, $w_m = e^{j2\pi m/N}$, which means that the sign of the imaginary parts of the output values produced by the two different implementations of the FFT are flipped. A crucial resource used when testing the FFT module was referencing a working C implementation (adapted from the one in Slade), which allowed ground-truth intermediate values to be examined. This was crucial to verifying the operation of the butterfly unit and address generation unit in particular. The reference implementation is included in Appendix B-G.

3) Beat Onset Detection: The spectrum of a song can be considered to contain both harmonic and percussive information. Harmonic instruments and voices typically produce fundamental tones below 1 kHz, with harmonics of lower intensity appearing in frequency bins that are multiples of the fundamental. Percussive elements, however, such as drums, can be considered to generate impulses in the time domain. Impulses in the time domain correspond to spectral content at

[1] with bit reversed addresses, see [1]

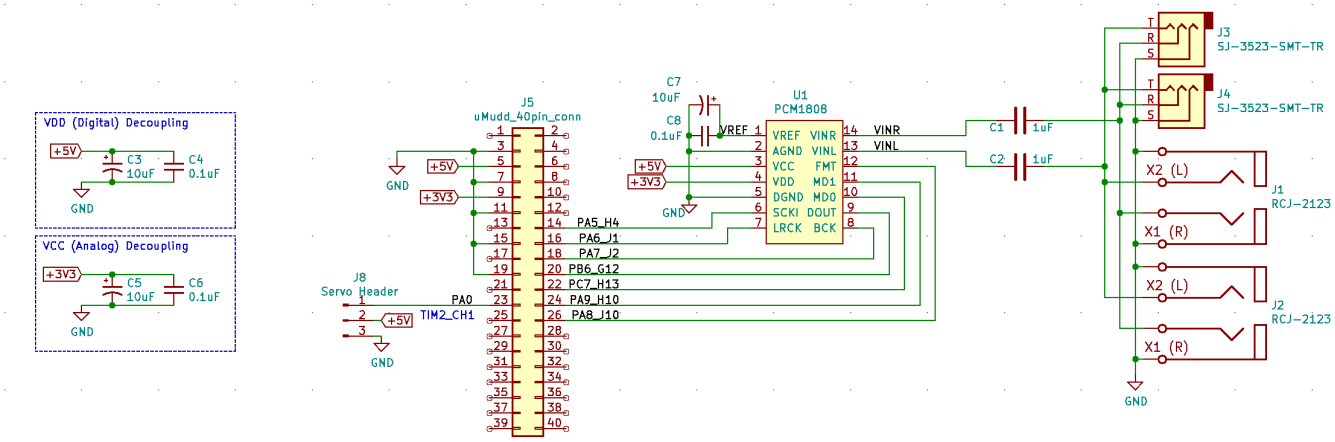

Fig. 3. Schematic of the ADC interface PCB. Note that the multiple audio jacks that allow it to be placed in the middle of an existing audio system.

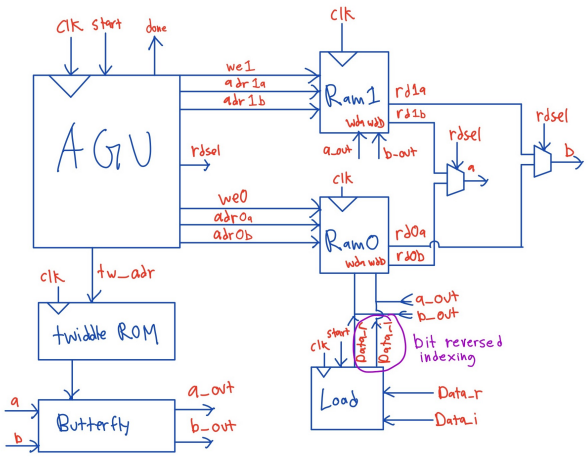

Fig. 4. Diagram of our Cooley-Tukey FFT System Verilog implementation based on Slade's paper and Prof. Brake's slides [1].

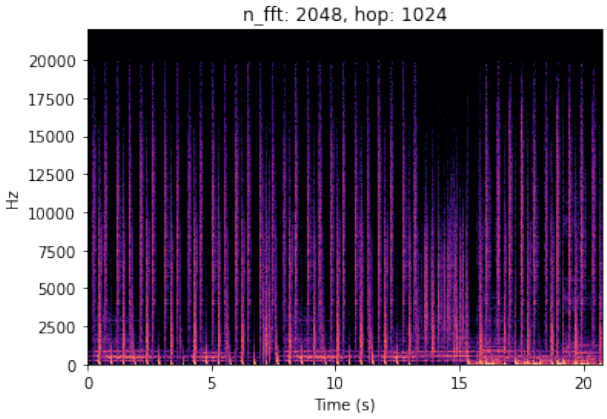

Fig. 5. Example log-magnitude Spectrum of a snippet of the EDM song "ACRAZE - Do it to It". Note that the upper frequency bins that correspond to the hi-hats provide steady tempo information, while the melodic and vocal parts of the song rest primarily in the lower frequency bins.

every frequency, so often drums and percussive elements will appear in high-frequency bins, depending on the mastering of the song and the specific instruments involved. Figure 5 displays an example spectrum of an electronic dance music (EDM) song that demonstrates this.

To take advantage of the high-frequency and broadband nature of percussive components, beat onsets were calculated by performing debounced thresholding of a novelty function computed by summing the magnitudes of the upper frequency bins.

As the output values of the FFT are being read sequentially, a complex multiplier computes the magnitude of each frequency bin by multiplying it with its complex conjugate. If the bin is between bins 10 and 15 inclusive, an accumulator accumulates this magnitude. The value of the accumulator is compared to a threshold to determine whether or not the sample could correspond to a beat. Hysteresis is implemented to ensure that a single beat in the song that spans multiple samples does not trigger the beat onset detector multiple times. Fig. 6 demonstrates this hysteresis for an example beat in a song.

B. Microcontroller Design

The STM32F401RE microcontroller (MCU) reads a pulsed beat onset signal from the FPGA and creates light patterns by controlling the servo and the two lasers.

The servo position is controlled by changing the duty cycle of a square wave signal. 770 μs of high time out of 20 ms corresponds to a position of zero degrees while 2270 μs corresponds to a position of 180 degrees. Positions are referred to as x μs. Our MCU simply writes out two binary signals for laser control.

The lasers are controlled with a dual laser driver circuit. To create this circuit, we used a prebuilt dual laser driver

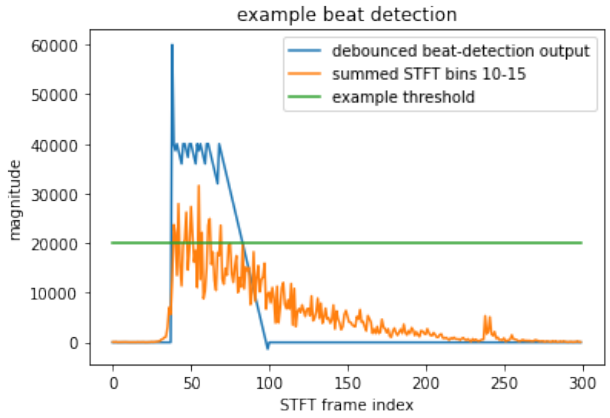

Fig. 6. Example beat-detection in the EDM song "ACRAZE - Do it to It". The orange trace corresponds to the sum of the upper frequency bins. The blue trace (beat detection out) goes high when the threshold is first reached. A "debounce" timer then counts down from there, resetting every time the threshold is reached, to only allow the beat to be detected again once the orange trace has been below the threshold for some minimum time. In practice, the blue signal only goes high when the beat is first detected, and the counter logic is stored and handled separately, but the logic displayed in this figure is helpful for visualization of the algorithm.

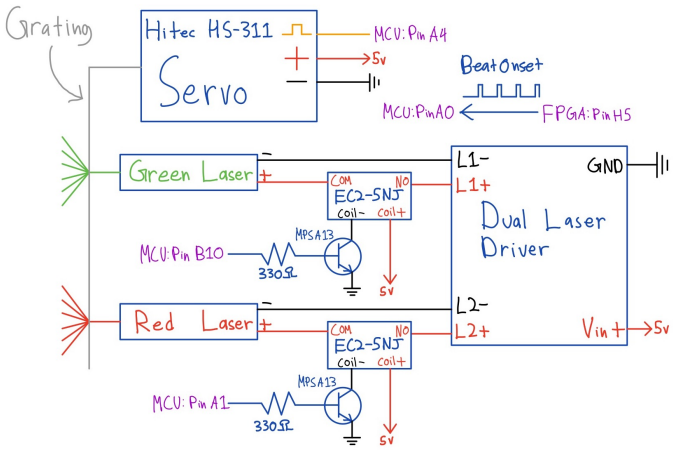

Fig. 7. Circuit diagram detailing physical connections between MCU, servo, and lasers.

board and inserted relays into the power wires going to the lasers. The MCU digital pins could not source enough current to activate the relays so we used NPN transistors to drive the relays and electrically control the flow of power to the lasers. We chose to use a relay instead of a transistor to avoid modifying the properties of the existing laser driver circuit. A circuit diagram of our MCU subsystem is presented in Figure 7.

A light pattern is encoded with four values:

1) Servo Starting Rotation (pulse width μs)
2) Servo Ending Rotation (pulse width μs)
3) Red Laser On (0 or 1)
4) Green Laser On (0 or 1)

Beats per minute (BPM) is calculated using the time difference between the last two beat onset pulses, and a pattern is assigned depending upon which bucket the BPM falls into from the following:

- $0 < \text{BPM} \leq 30$
- $30 < \text{BPM} \leq 60$
- $60 < \text{BPM} \leq 120$
- $120 < \text{BPM} \leq 240$

Once a pattern is selected, the servo oscillates between the servo start position and the servo end position. These transitions happen exactly at the beat onset times. The MCU also enables the lasers according to the selected pattern by writing the two binary "laser on" signals to our laser driver circuit.

III. Results

The efficacy of our system is highly dependent upon the song that is fed into it. Songs with a consistent percussive element, such as a clap or drum hit, produced the best results. In these cases, the beat onset detection algorithm captures nearly all of the beats and the lasers produce an aesthetically interesting light show that is closely synchronized to the input song. Additionally, the system functions best when the input song is sent at maximum volume.

There are a few ways we could have improved the system given more time:

1) Introduce an automatic gain control circuit. We believe this would allow for songs at varying volumes to perform well with ATLAS.
2) Compute a log-magnitude spectrogram. Humans perceive the loudness of sound on a log scale so operating on a log-magnitude spectrogram is likely to yield beat detections that better align with a human's perception of beats.
3) Reduce the hop size of the STFT to be half of the window size by double-buffering the input sample windows, which provides for a more conventional STFT computation.
4) Experiment with different FFT window sizes and bucket summation indices. Varying the window size and bucket summation indices could allow ATLAS to work on a greater variety of musical genres. For example, a higher FFT window size with beat detection based on a summation of the low frequency buckets could produce good beat tracking in rock songs with a prominent bassline. Higher frequency resolution that enables the spectrum to differentiate fundamental frequencies of different notes would also enable the system to operate successfully with more conventional beat-detection algorithms such as spectral-flux based novelty functions.

Acknowledgment

The authors would like to thank Prof. Joshua Brake for his invaluable guidance with our final project. We also appreciate the insightful feedback we received on our project by the class. Lastly, we thank Sam Abdelmuati in the stockroom for his help regarding part selection.

APPENDIX A
BILL OF MATERIALS

TABLE II
BILL OF MATERIALS

Ct.	Description	Part Number	Extended Price
1	stereo ADC	PCM1808PW	$2.23
2	cap cer 1uf 50v 0805	C2012JB1H105K085AB	$0.374
3	cap cer 0.1uf 50v 0805	CGA4J2X8R1H104K125AA	$0.396
3	cap cer 10uf 25v 0805	C2012JB1E106M085AC	$1.26
2	conn jack stereo 3.5mm SMD	SJ-3523-SMT-TR	$1.88
1	conn header vert 40pos 2.54mm	SBH11-PBPC-D20-ST-BK	$0.73
5	JLCPCB boards (+ shipping)	–	$8.98
1	laser + diffraction grating module	ZQ-B338	$20.99
1	Hitec hobby servo	HS-311	N/A [2]
2	relay	EC2-5NJ	N/A [3]
2	NPN darlington transistor	MPSA13	N/A [4]
2	330 ohm resistor	–	N/A [5]

[2] already owned
[3] from stock room
[4] from stock room
[5] from stock room

APPENDIX B
MAX10 FPGA SYSTEMVERILOG CODE

A. i2s.sv

```
module final_fpga(input logic        clk,   //   12MHz MAX1000 clk, H6
                  input logic        nreset, //  global reset,      E6 (right btn)
                  input logic        din, //     I2S DOUT,          PB6_G12
                  input logic        uscki, //   SPI clk,           PB3_J12
                  input logic        umosi, //   SPI MOSI,          PB5_J13
                  input logic        uce, //     SPI CE,            PA10_L12
                  input logic [3:0]  sw1, //     threshold sel.     E1, C2, C1, D1
                   ↪  (DIP SW1)
                  output logic       bck, //     I2S bit clock,     PA7_J2
                  output logic       lrck, //    I2S l/r clk,       PA6_J1
                  output logic       scki, //    PCM1808 sys clk,   PA5_H4
                  output logic       fmt, //     PCM1808 FMT,       PA8_J10
                  output logic [1:0] md, //      PCM1808 MD1 & MD0, PC7_H13, PA9_H10
                  output logic       miso, //    SPI MISO,          PB4_K11
                  output logic [7:0] LEDs, //    debug LEDs         (see MAX1000 user
                   ↪  guide)
                  output logic       beat_out // beat (to MCU)      H5 (thru a jumper
                   ↪  to MCU)
                  );

    ////////////////////////////////// reset
    // active high reset, active low btns
    logic                           reset;
    assign reset = ~(nreset);
    //assign LEDs[0] = reset;
    ////////////////////////////////// end reset

    ////////////////////////////////// I2S/PCM1808
    assign md  = 2'b00; // see PCM1808 table 2 (slave mode)
    assign fmt = 0;     // see PCM1808 table 3 (i2s mode)

    logic                           newsample;
    logic [23:0]                    left, right;
    i2s pcm_in(clk, reset, din, bck, lrck, scki, left, right, newsample);
    ////////////////////////////////// end I2S/PCM1808

    ////////////////////////////////// FFT
    logic                           fft_start, fft_done, i2s_load, fft_load,
     ↪  fft_reset, fft_creset, fft_write;
    logic [31:0]                    fft_wd;
    logic signed [15:0]             left_msbs;
    logic signed [15:0]             fft_rd;
    logic [5:0]                     sample_ctr;

    assign fft_reset = reset | fft_creset;
    assign left_msbs = left[23:8];
    assign fft_rd    = left_msbs >>> 5; // arithmetic shift right
    fft fft(clk, fft_reset, fft_start, fft_load, fft_rd, fft_wd, fft_done);
    fft_control i2s_fft_glue(clk, reset, newsample, fft_done, fft_start, fft_load,
     ↪  fft_creset, fft_write, sample_ctr);
    ////////////////////////////////// end FFT

    ////////////////////////////////// store result
    logic [31:0]                    res_count; // result count
```

```
50    logic                                posedge_fft_done;
51    pos_edge pos_edge_fft_done(clk, fft_done, posedge_fft_done);
52    always_ff @(posedge clk)
53      begin
54         if       (reset)    res_count <= 0;
55         else if (posedge_fft_done) res_count <= res_count + 1'b1;
56      end
57
58    logic [31:0]                        spectrum_result [0:31];
59    always_ff @(posedge clk) begin
60       if (fft_write)
61         begin
62            spectrum_result[sample_ctr] <= fft_wd;
63         end
64    end
65    ////////////////////////////////// end store result
66
67    ////////////////////////////////// SPI
68    logic [31:0] spi_data;
69    logic [4:0]  spi_adr;
70    assign spi_data = spectrum_result[spi_adr];
71    spi_slave spi(clk, reset, uscki, umosi, uce, spi_data, spi_adr, miso);
72    ////////////////////////////////// end SPI
73
74    ////////////////////////////////// beat tracking
75    logic [7:0]  thresh, accum_stable;
76    logic        beat_ctr, posedge_beat_out;
77    assign thresh = {1'b0, sw1, 2'b0};
78    assign LEDs   = {accum_stable[7:1], beat_ctr};
79    beat_track beattrack(clk, reset, sample_ctr, fft_wd, fft_write, fft_done, thresh,
      ↪  beat_out, accum_stable);
80    pos_edge pos_edge_beat_out(clk, beat_out, posedge_beat_out);
81
82    always_ff @(posedge clk) begin
83       if (reset)
84         beat_ctr <= 0;
85       else if (posedge_beat_out)
86         beat_ctr <= beat_ctr + 1'b1;
87    end
88    ////////////////////////////////// end beat tracking
89
90  endmodule // final_fpga
91
92  typedef enum logic [3:0] {FFT_LOADING, FFT_STARTING, FFT_WAITING, FFT_WRITING,
      ↪  FFT_RESETTING, FFT_ERROR} statetype;
93  module fft_control(input logic        clk,
94                     input logic        reset,
95                     input logic        newsample,
96                     input logic        fft_done,
97                     output logic       fft_start,
98                     output logic       fft_load,
99                     output logic       fft_reset,
100                    output logic       fft_write,
101                    output logic [5:0] sample_ctr);
102
103   statetype                    state, nextstate;
104
105   always_ff @(posedge clk) begin
```

```
106       if      (reset || (state == FFT_RESETTING))
107         sample_ctr <= 0;
108       else if ((newsample && (state == FFT_LOADING)) || (state == FFT_WRITING ||
          ↪  nextstate == FFT_WRITING))
109         sample_ctr <= sample_ctr + 1'b1;
110       else if (state == FFT_WAITING)
111         sample_ctr <= 0; // this case needs to be checked after the previous
112       //                    in case state is WAITING but nextstate is WRITING.
113    end
114
115    always_ff @(posedge clk) begin
116       if (reset) state <= FFT_LOADING;
117       else state <= nextstate;
118    end
119
120    always_comb begin
121       case (state)
122         FFT_LOADING  :
123           if (sample_ctr == 32)   nextstate = FFT_STARTING;
124           else                    nextstate = FFT_LOADING;
125         FFT_STARTING :            nextstate = FFT_WAITING;
126         FFT_WAITING  :
127           if (fft_done)           nextstate = FFT_WRITING;
128           else                    nextstate = FFT_WAITING;
129         FFT_WRITING  :
130           if (sample_ctr == 32)   nextstate = FFT_RESETTING;
131           else                    nextstate = FFT_WRITING;
132         FFT_RESETTING:            nextstate = FFT_LOADING;
133         default      :            nextstate = FFT_ERROR; // debug
134       endcase // case (state)
135    end
136
137    assign fft_load  = (state == FFT_LOADING) & newsample;
138    assign fft_start = (state == FFT_STARTING);
139    assign fft_reset = (state == FFT_RESETTING);
140    assign fft_write = (state == FFT_WRITING || nextstate == FFT_WRITING);
141
142 endmodule // control
143
144 module i2s(input logic         clk,
145            input logic         reset,
146            input logic         din, //  PCM1808 DOUT,         PB6_G12
147            output logic        bck, //  bit clock,            PA7_J2
148            output logic        lrck, // left/right clk,       PA6_J1
149            output logic        scki, // PCM1808 system clock, PA5_H4
150            output logic [23:0] left,
151            output logic [23:0] right,
152            output logic        newsample_valid);
153
154    ////////////////// clock ////////////////////////////////
155
156    //   Fs = 46.875 KHz
157    logic [8:0]                prescaler; // 9-bit prescaler
158    assign scki = clk;         // 256 * Fs = 12 MHz
159    assign bck  = prescaler[1]; // 64  * Fs = 3 MHz = 12 MHz / 4
160    assign lrck = prescaler[7]; // 1   * Fs = 12 Mhz / 256
161
162    always_ff @(posedge clk)
```

```
163       begin
164          if (reset)
165            prescaler <= 0;
166          else
167            prescaler <= prescaler + 9'd1;
168       end
169     ////////////////// end clock ////////////////////////////////
170
171     // left and right shift registers
172     logic [23:0]                lsreg, rsreg;
173
174     // samples the prescaler to figure out what bit should currently be sampled.
175     // sampling occurs on bit 1 and bit 24, NOT bit 0!
176     logic [4:0]                 bit_state;
177     assign bit_state = prescaler[6:2];
178
179     // shift register enable logic
180     logic                       shift_en;
181     assign shift_en = ((bit_state >= 1) && (bit_state <= 24) && !reset);
182
183     // shift register operation. samples DOUT only when shift_en.
184     // this should be the only register that is not clocked directly from clk!!
185     always_ff @(posedge bck)
186       begin
187          if (!lrck && shift_en)     // left
188            begin
189               lsreg <= {lsreg[22:0], din};
190               rsreg <= rsreg;
191            end
192          else if (lrck && shift_en) // right
193            begin
194               rsreg <= {rsreg[22:0], din};
195               lsreg <= lsreg;
196            end
197       end // shift register operation
198
199     // load shift regs into output regs.
200     // update both regs at once, once every fs.
201     // this way, left and right will always contain a valid sample.
202     logic newsample;
203     assign newsample = (bit_state == 25 && lrck && prescaler[1:0] == 0); // once every
        ↪  cycle
204     assign newsample_valid = (bit_state == 26 && lrck && prescaler[1:0] == 0); // once
        ↪  we can sample it!
205     always_ff @(posedge clk)
206       begin
207          if (reset)
208            begin
209               left <= 0;
210               right <= 0;
211            end
212          else if (newsample)
213            begin
214               left <= lsreg;
215               right <= rsreg;
216            end
217          else
218            begin
```

```
219                 left <= left;
220                 right <= right;
221             end
222         end
223 
224 endmodule // i2s
```

B. *fft.sv*

```
// the width is the bit width (e.g. if width=16, 16 real and 16 im bits).
// the input should be width-5 to account for bit growth.
module fft
  #(parameter width=16, N_2=5, hann=0) // N_2 is log base 2 of N (points)
   (input logic                 clk,
    input logic                 reset,
    input logic                 start,
    input logic                 load,
    input logic [width-1:0]     rd, // real    read data in
    output logic [2*width-1:0] wd, // complex write data out
    output logic                done);

   logic                        enable;  // for AGU operation
   logic                        rdsel;   // read from RAM0 or RAM1
   logic                        we0_agu, we0, we1; // RAMx write enable
   logic [N_2 - 1:0]            adr0a_agu, adr0b_agu, adr0a, adr0b, adr0a_load,
    ↪  adr0b_load, adr0a_load_agu, adr1a, adr1b, adr1a_agu;
   logic [N_2 - 2:0]            twiddleadr; // twiddle ROM adr
   logic [2*width-1:0]          twiddle, a, b, writea, writeb, aout, bout, rd0a, rd0b,
    ↪  rd1a, rd1b, val_in;

   // LOAD LOGIC
   fft_load #(width, N_2, hann) loader(clk, reset, load, rd, adr0a_load, adr0b_load,
    ↪  val_in);
   assign adr0a_load_agu = load ? adr0a_load : adr0a_agu;
   assign adr0b = load ? adr0b_load : adr0b_agu;
   assign writea = load ? val_in : aout;
   assign writeb = load ? val_in : bout;
   assign we0    = load ?   1'b1 : we0_agu;

   // AGU ENABLE LOGIC
   always_ff @(posedge clk)
     begin
        if      (start) enable <= 1;
        else if (done || reset)  enable <= 0;
     end

   // OUTPUT LOGIC
   logic [N_2-1:0] out_idx;
   assign wd    = N_2[0] ? rd1a : rd1b; // ram holding results depends on even-ness of
    ↪  log2(N-points)s?
   assign adr0a = done ? out_idx : adr0a_load_agu;
   assign adr1a = done ? out_idx : adr1a_agu;

   always_ff @(posedge clk)
     begin
        if      (reset) out_idx <= 0;
        else if (done)  out_idx <= out_idx + 1'b1;
     end

   fft_agu #(width, N_2) agu(clk, enable, reset, done, rdsel, we0_agu, adr0a_agu,
    ↪  adr0b_agu, we1, adr1a_agu, adr1b, twiddleadr);
   fft_twiddleROM #(width, N_2) twiddlerom(twiddleadr, twiddle);

   twoport_RAM #(width, N_2) ram0(clk, we0, adr0a, adr0b, writea, writeb, rd0a, rd0b);
   twoport_RAM #(width, N_2) ram1(clk, we1, adr1a, adr1b,   aout,   bout, rd1a, rd1b);
```

```
52    assign a = rdsel ? rd1a : rd0a;
53    assign b = rdsel ? rd1b : rd0b;
54
55    fft_butterfly #(width) bgu(twiddle, a, b, aout, bout);
56
57  endmodule // fft
58
59  module fft_load
60    #(parameter width=16, N_2=5, hann=0) // hann: bool, whether or not to window.
61     (input logic clk,
62      input logic                reset,
63      input logic                load,
64      input logic [width-1:0]    rd,
65      output logic [N_2-1:0]     adr0a_load,
66      output logic [N_2-1:0]     adr0b_load,
67      output logic [2*width-1:0] val_in);
68
69    logic [N_2-1:0]             idx;
70    logic [width-1:0]           val_in_re;
71
72    bit_reverse #(N_2) reverseaddr(idx, adr0a_load);
73    assign adr0b_load = adr0a_load;
74
75    always_ff @(posedge clk)
76      begin
77         if (reset) begin
78            idx <= 0;
79         end else if (load) begin
80            idx <= idx + 1'b1;
81         end
82      end
83
84    logic        [2*width-1:0] untruncated_mult;
85    logic signed [width-1:0]   hann_coeff;
86    hann_lut #(width, N_2) hann_rom(clk, idx, hann_coeff);
87    assign untruncated_mult = hann_coeff * rd;
88    assign val_in_re = hann ? untruncated_mult[2*width-2:width-1] : rd;
89    assign val_in    = {val_in_re, 16'b0}; // imaginary is all zeros!
90
91  endmodule // fft_load
92
93  module bit_reverse
94    #(parameter N_2=5)
95     (input logic [N_2-1:0] in,
96      output logic [N_2-1:0] out);
97
98    genvar                  i;
99    generate
100      for(i=0; i<N_2; i=i+1) begin : BIT_REVERSE
101         assign out[i] = in[N_2-i-1];
102      end
103   endgenerate
104
105 endmodule // bit_reverse
106
107 // 32-point FFT address generation unit
108 module fft_agu
109   #(parameter width=16, N_2=5)
```

```
    (input logic             clk,
     input logic             enable,
     input logic             reset,
     output logic            done,
     output logic            rdsel,
     output logic            we0,
     output logic [N_2-1:0]  adr0a,
     output logic [N_2-1:0]  adr0b,
     output logic            we1,
     output logic [N_2-1:0]  adr1a,
     output logic [N_2-1:0]  adr1b,
     output logic [N_2-2:0]  twiddleadr);

    logic [N_2-1:0]          fftLevel = 0;
    logic [N_2-1:0]          flyInd = 0;

    logic [N_2-1:0]          adrA;
    logic [N_2-1:0]          adrB;

    always_ff @(posedge clk) begin
       if (reset) begin
          fftLevel <= 0;
          flyInd <= 0;
       end
       // Increment fftLevel and flyInd
       else if(enable === 1 & ~done) begin
          if(flyInd < 2**(N_2 - 1) - 1) begin
             flyInd <= flyInd + 1'd1;
          end else begin
             flyInd <= 0;
             fftLevel <= fftLevel + 1'd1;
          end
       end
    end

    // sets done when we are finished with the fft
    assign done = (fftLevel == (N_2));
    calcAddr #(width, N_2) adrCalc(fftLevel, flyInd, adrA, adrB, twiddleadr);

    assign adr0a = adrA;
    assign adr1a = adrA;

    assign adr0b = adrB;
    assign adr1b = adrB;

    // flips every cycle
    assign we0 = fftLevel[0] & enable;
    assign we1 = ~fftLevel[0] & enable;

    // flips every cycle
    assign rdsel = fftLevel[0];

endmodule // fft_agu

// todo: parameterize for more than 32-point FFT.
module calcAddr
  #(parameter width=16, N_2=5)
    (input logic  [N_2-1:0]    fftLevel,
```

```
     input logic [N_2-1:0]  flyInd,
     output logic [N_2-1:0] adrA,
     output logic [N_2-1:0] adrB,
     output logic [N_2-2:0] twiddleadr);

    logic [N_2-1:0]         tempA;
    logic [N_2-1:0]         tempB;

    always_comb begin
       tempA = flyInd << 1'd1;
       tempB = tempA +  1'd1;
       adrA = ((tempA << fftLevel) | (tempA >> (N_2 - fftLevel))) & 5'h1f;

       adrB = ((tempB << fftLevel) | (tempB >> (N_2 - fftLevel))) & 5'h1f;
       twiddleadr = ((32'hffff_fff0 >> fftLevel) & 32'hf) & flyInd;
    end
endmodule // calcAddr

module fft_twiddleROM
  #(parameter width=16, N_2=5)
    (input logic  [N_2-2:0] twiddleadr, // 0 - 1023 = 10 bits
     output logic [2*width-1:0] twiddle);

    // twiddle table pseudocode: w[k] = w[k-1] * w,
    // where w[0] = 1 and w = exp(-j 2pi/N)
    // for k=0... N/2-1

    logic [2*width-1:0]         vectors [0:2**(N_2-1)-1];
    initial $readmemb("rom/twiddle.vectors", vectors);
    assign twiddle = vectors[twiddleadr];

endmodule // fft_twiddleROM


// make sure the script rom/hann.py has been run with
// the desired width! the `width` param should be equal to `q` in the script.
// todo: test hann windowing (does the read need to be clocked)?
module hann_lut
  #(parameter width=16, N_2=5)
    (input logic              clk,
     input logic [N_2-1:0]    idx,
     output logic [width-1:0] out);

    logic [width-1:0]         vectors[2**N_2-1:0];
    initial $readmemb("rom/hann.vectors", vectors);

    always @(posedge clk)
      out <= vectors[idx];

endmodule // hann_lut


// explicit so that it is inferred, and we control
// the truncation of the output.
module mult
  #(parameter width=16)
    (input logic signed [width-1:0]  a,
     input logic signed [width-1:0]  b,
```

```
226      output logic signed [width-1:0] out);
227
228    logic [2*width-1:0]              untruncated_out;
229
230    assign untruncated_out = a * b;
231    assign out = untruncated_out[30:15];
232    // see slade paper. this works as long as we're not
233    // multiplying two maximum mag. negative numbers.
234
235  endmodule // mult
236
237
238  module complex_mult
239    #(parameter width=16)
240     (input logic [2*width-1:0]  a,
241      input logic [2*width-1:0]  b,
242      output logic [2*width-1:0] out);
243
244    logic signed [width-1:0]    a_re, a_im, b_re, b_im, out_re, out_im;
245    assign a_re = a[31:16]; assign a_im = a[15:0];
246    assign b_re = b[31:16]; assign b_im = b[15:0];
247
248    logic signed [width-1:0]    a_re_be_re, a_im_b_im, a_re_b_im, a_im_b_re;
249    mult #(width) m1 (a_re, b_re, a_re_be_re);
250    mult #(width) m2 (a_im, b_im, a_im_b_im);
251    mult #(width) m3 (a_re, b_im, a_re_b_im);
252    mult #(width) m4 (a_im, b_re, a_im_b_re);
253
254    assign out_re = (a_re_be_re) - (a_im_b_im);
255    assign out_im = (a_re_b_im) + (a_im_b_re);
256    assign out = {out_re, out_im};
257  endmodule // complex_mult
258
259  module fft_butterfly
260    #(parameter width=16)
261     (input logic [2*width-1:0] twiddle,
262      input logic [2*width-1:0]  a,
263      input logic [2*width-1:0]  b,
264      output logic [2*width-1:0] aout,
265      output logic [2*width-1:0] bout);
266
267    logic signed [width-1:0]    a_re, a_im, aout_re, aout_im, bout_re, bout_im;
268    logic signed [width-1:0]    b_re_mult, b_im_mult;
269    logic [2*width-1:0]         b_mult;
270
271
272    // expand to re and im components
273    assign a_re = a[2*width-1:width];
274    assign a_im = a[width-1:0];
275
276    // perform computation
277    complex_mult #(width) twiddle_mult(b, twiddle, b_mult);
278    assign b_re_mult = b_mult[31:16];
279    assign b_im_mult = b_mult[15:0];
280
281    assign aout_re = a_re + b_re_mult;
282    assign aout_im = a_im + b_im_mult;
283
```

```
284     assign bout_re = a_re - b_re_mult;
285     assign bout_im = a_im - b_im_mult;
286 
287     // pack re and im outputs
288     assign aout = {aout_re, aout_im};
289     assign bout = {bout_re, bout_im};
290 
291 endmodule // fft_butterfly
292 
293 
294 // adapted from HDL example 5.7 in Harris TB
295 module twoport_RAM
296   #(parameter width=16, N_2=5)
297    (input logic                 clk,
298     input logic                 we,
299     input logic [N_2-1:0]       adra,
300     input logic [N_2-1:0]       adrb,
301     input logic [2*width-1:0]   wda,
302     input logic [2*width-1:0]   wdb,
303     output logic [2*width-1:0] rda,
304     output logic [2*width-1:0] rdb);
305 
306     reg [2*width-1:0]           mem [2**N_2-1:0];
307 
308     always @(posedge clk)
309       if (we)
310         begin
311            mem[adra] <= wda;
312            mem[adrb] <= wdb;
313         end
314 
315     assign rda = mem[adra];
316     assign rdb = mem[adrb];
317 
318 endmodule // twoport_RAM
319 
320 module complex_mag
321   #(parameter width=16)
322    (input logic [2*width-1:0]   a,
323     output logic [2*width-1:0] out);
324 
325     logic [2*width-1:0]         b;
326     logic signed [width-1:0]    b_re, b_im;
327 
328     assign b_re = a[31:16];
329     assign b_im = -a[15:0];
330     assign b = {b_re, b_im};
331 
332     complex_mult mag_mult(a, b, out);
333 
334 endmodule // magnitude
```

C. spi.sv

```
typedef enum logic [3:0] {SPI_IDLE, SPI_LOAD, SPI_WAIT, SPI_SHIFT, SPI_ERROR}
  ↪  spi_state;
module spi_slave(input logic        clk,
                 input logic        reset,
                 input logic        uscki, // non-synchronus miso
                 input logic        umosi, // non-synchronus mosi
                 input logic        uce,   // non-synchronus chip-enable
                 input logic [31:0] data,  // 32-bit. based on adr.
                 output logic [4:0] adr,   // to address data ram
                 output logic       miso);

   // cpol = 0 (idle low)
   // cpha = 1 (shift on leading edge (posedge), sample on lagging edge (negedge)
   logic                       scki, mosi, ce;
   sync scki_sync(clk, uscki, scki);
   sync mosi_sync(clk, umosi, mosi);
   sync   ce_sync(clk,   uce,   ce);

   logic                       posedge_scki;
   pos_edge pos_edge_scki(clk, scki, posedge_scki);

   logic [31:0]                data_out;
   logic [4:0]                 bit_idx;
   spi_state state, nextstate;

   assign miso = data_out[31]; // MSB. shifted out

   // SPI state register
   always_ff @(posedge clk) begin
      if (reset) state <= SPI_IDLE;
      else state <= nextstate;
   end

   // SPI state transition logic
   always_comb begin
      case (state)
        SPI_IDLE : if (ce) nextstate <= SPI_LOAD;
        else    nextstate <= SPI_IDLE;
        SPI_LOAD : nextstate <= SPI_WAIT;
        SPI_WAIT : if (posedge_scki) begin
           if (bit_idx != 31) nextstate <= SPI_SHIFT;
           else nextstate <= SPI_LOAD;
        end
        else if (~ce) nextstate <= SPI_IDLE;
        else   nextstate <= SPI_WAIT;
        SPI_SHIFT : nextstate <= SPI_WAIT;
        default : nextstate <= SPI_ERROR;
      endcase // case (state)
      end

   // adr logic
   always_ff @(posedge clk) begin
      if (state == SPI_IDLE)
        adr <= 0;
      else if (state == SPI_LOAD)
        adr <= adr + 1'b1;
```

```
56      end
57
58      // bit_idx logic
59      always_ff @(posedge clk) begin
60         if (state == SPI_IDLE || state == SPI_LOAD)
61           bit_idx <= 0;
62         else if (state == SPI_SHIFT)
63           bit_idx <= bit_idx + 1'b1;
64      end
65
66      // data_out logic
67      always_ff @(posedge clk) begin
68         if (state == SPI_LOAD)
69           data_out <= data;
70         else if (state == SPI_SHIFT)
71           data_out <= {data_out[30:0], 1'b0};
72      end
73
74  endmodule // spi_slave
75
76  // positive edge detection in synchronus logic
77  module pos_edge(input logic clk,
78                  input logic  in,
79                  output logic out);
80
81      logic                       last;
82      always_ff @(posedge clk) begin
83         last <= in;
84      end
85      assign out = (last == 0 && in == 1);
86
87  endmodule // pos_edge
88
89  // negative edge detection in synchronus logic
90  // (unused)
91  module neg_edge(input logic clk,
92                  input logic  in,
93                  output logic out);
94
95      logic                       last;
96      always_ff @(posedge clk) begin
97         last <= in;
98      end
99      assign out = (last == 1 && in == 0);
100
101 endmodule // pos_edge
102
103 // synchronizer chain
104 module sync(input logic clk,
105             input logic  in,
106             output logic out);
107
108     logic                   m1;
109     always_ff @(posedge clk) begin
110        m1 <= in;
111        out <= m1;
112     end
113
```

```
114 endmodule // sync
```

D. beat_track.sv

```
typedef enum logic [1:0] {BEAT_LOAD, BEAT_EVAL, BEAT_RESET, BEAT_ERROR}
 ↪  beat_load_state;
module beat_track
  #(parameter wait_samples=30)
   (input logic        clk,
    input logic        reset,
    input logic [5:0]  sample_ctr,
    input logic [31:0] data,
    input logic        fft_write,
    input logic        fft_done,
    input logic [7:0]  thresh,
    output logic       beat_out,
    output logic [7:0] debug);

   ////////////////////////////////// load control
   beat_load_state             state, nextstate;
   always_ff @(posedge clk) begin
      if (reset) state <= BEAT_RESET;
      else       state <= nextstate;
   end

   always_comb
     case (state)
       BEAT_LOAD  :
         if (~fft_write)  nextstate = BEAT_EVAL;
         else             nextstate = BEAT_LOAD;
       BEAT_EVAL  :       nextstate = BEAT_RESET;
       BEAT_RESET :
         if (fft_write)   nextstate = BEAT_LOAD;
         else             nextstate = BEAT_RESET;
       BEAT_ERROR :       nextstate = BEAT_ERROR;
       default    :       nextstate = BEAT_ERROR;
     endcase // case (state)
   ////////////////////////////////// end load control

   ////////////////////////////////// accumulator
   logic [31:0]                data_mag;
   logic signed [31:0]         accum, accum_stable;
   logic signed [15:0]         mag_real;

   assign debug = accum_stable[7:0];

   assign mag_real = data_mag[31:16];
   complex_mag magnitude(data, data_mag);

   // todo: potential error if we were to try to accumulate
   //       the first fft output sample (since we need
   //       the state to be out of BEAT_RESET to accum.)
   always_ff @(posedge clk) begin
      if (reset || (state == BEAT_RESET))
        begin
           accum <= 0;
           // accum_stable <= 0; // debug, easier not to reset
        end
      // choose bins 10..16 (see sim .ipynb)
      else if ((sample_ctr >= 10) && (sample_ctr <= 16) && fft_write)
```

```
56          begin
57             accum <= accum + mag_real;
58          end
59       if (state == BEAT_EVAL)
60          accum_stable <= accum;
61    end
62    ////////////////////////////////// end accumulator
63
64    ////////////////////////////////// beat filter
65    logic [31:0]                wait_ctr;
66    logic                       over;
67
68    // debouncing & thresholding logic
69    always_ff @(posedge clk) begin
70       if (reset) begin
71          wait_ctr <= wait_samples + 1'b1;
72          over <= 0;
73       end
74       else if (state == BEAT_EVAL) begin
75          if (accum > thresh) begin // threshold met
76             wait_ctr <= 0;
77             over <= 1;
78          end else begin            // threshold not met
79             if (wait_ctr > wait_samples) begin
80                over <= 0;
81                wait_ctr <= wait_ctr;
82             end else begin
83                wait_ctr <= wait_ctr + 1'b1;
84                over <= over;
85             end
86          end
87       end // if (state == BEAT_EVAL)
88    end // always_ff @ (posedge clk)
89
90    always_ff @(posedge clk) begin
91       if (state == BEAT_EVAL) begin
92          if ((accum > thresh) && ~over && (wait_ctr > wait_samples))
93            beat_out <= 1;
94          else
95            beat_out <= 0;
96       end
97    end
98    ////////////////////////////////// end beat filter
99
100
101 endmodule // beat_track
```

E. testbenches.sv

```
module i2s_testbench();
   logic clk, reset, din, bck, lrck, scki;
   logic [24:0] left, right; // 23 + 1 msb pad
   logic [8:0]  i;

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

   initial begin
      reset = 1'b1;
      i = 0;
   end

   i2s dut(clk, reset, din, bck, lrck, scki, left, right);

   always @(posedge clk) begin
      if (i == 10)
        reset = 0;
      i = i + 1;
   end
endmodule // i2s_testbench


// requires manually checking! compare to values in file.
// need to copy the rom/ dir into the simulation/modelsim/ dir
module hannrom_testbench #(parameter width=16, N_2=5)();
   logic clk;
   logic [N_2-1:0] idx;
   logic [width-1:0] out;

   // clk. if ns scale, then we're running @ 11.9 MHz
   initial
     forever begin
        clk = 1'b0; #5;
        clk = 1'b1; #5;
     end

   initial begin
      idx = 0;
   end

   always @(posedge clk) begin
      idx =idx+1'b1;
   end

   hann_lut dut(clk, idx, out);

endmodule // twiddlerom_testbench

// tested
module bgu_testbench #(parameter width=16)();
   logic clk;
   logic [2*width-1:0] twiddle;
```

```
57      logic [2*width-1:0] a;
58      logic [2*width-1:0] b;
59      logic [2*width-1:0] aout;
60      logic [2*width-1:0] bout;
61
62      initial
63        forever begin
64           clk = 1'b0; #5;
65           clk = 1'b1; #5;
66        end
67
68      initial begin
69         twiddle = 0;
70         a = 0;
71         b = 0;
72         aout = 0;
73         bout = 0; #10;
74         assert (aout===0 && bout===0) else $error("case 1 failed.");
75
76         twiddle = {16'h7FFF, 16'b0}; #10; assert(aout===0 && bout===0) else $error("case
           ↪  2 failed.");
77         b = {16'h7FFF, 16'b0}; #10; assert(aout==={16'h7FFE,16'b0} && bout==={16'h8002,
           ↪  16'b0}) else $error("case 3 failed.");
78
79         // real test case:
80         // real b*w out: 0xEE59 (-4520). im b*w out: 0x58C2 (22722).
81         // aout: 141 + j27382. bout:  9179 - j18062.
82         twiddle = {16'h471C, 16'h6A6C}; a={16'h1234, 16'h1234}; b={16'h3FFF, 16'h3FFF};
           ↪  #10;
83         assert(aout==={16'h008C, 16'h6AF6} && bout==={16'h23DC, 16'hB972}) else
           ↪  $error("case 4 failed!");
84
85         $display("BGU tests complete.");
86      end
87
88      fft_butterfly dut(twiddle, a, b, aout, bout);
89
90  endmodule // twiddlerom_testbench
91
92  // verify that RAM works, and is fully two-port.
93  // tested.
94  module ram_testbench #(parameter width=16, N_2=5)();
95      logic clk;
96      initial
97        forever begin
98           clk = 1'b0; #5;
99           clk = 1'b1; #5;
100       end
101
102     logic                 we;
103     logic [N_2-1:0]       adra;
104     logic [N_2-1:0]       adrb;
105     logic [2*width-1:0] wda;
106     logic [2*width-1:0] wdb;
107     logic [2*width-1:0] rda;
108     logic [2*width-1:0] rdb;
109
110     twoport_RAM #(width, N_2) dut(clk, we, adra, adrb, wda, wdb, rda, rdb);
```

```
111
112     initial begin
113         we = 0; adra = 0; adrb = 0; wda = 0; wdb = 0; rda = 0; rdb = 0; #10
114           adra = 10; adrb = 12; wda = 10; wdb = 12; we = 1; #10;
115         we = 0; #20; adra = 12; adrb = 10; #10; assert(rda === 12 && rdb === 10) else
            ↪  $error("ram test failed.");
116     end
117
118 endmodule // ram_testbench
119
120
121 // outdated (reset signal). previously tested working.
122 module agu_testbench #(parameter width=16, N_2=5)();
123
124     // inputs
125     logic clk;
126     initial
127       forever begin
128         clk = 1'b0; #5;
129         clk = 1'b1; #5;
130       end
131     logic  start;
132
133     // outputs
134     logic  done;
135     logic  rdsel;
136     logic  we0;
137     logic [N_2-1:0] adr0a;
138     logic [N_2-1:0] adr0b;
139     logic           we1;
140     logic [N_2-1:0] adr1a;
141     logic [N_2-1:0] adr1b;
142     logic [N_2-2:0] twiddleadr;
143
144     fft_agu #(width, N_2) agu(clk, start, done, rdsel, we0, adr0a, adr0b, we1, adr1a,
        ↪  adr1b, twiddleadr);
145
146     initial begin
147         // init inputs and outputs to zero/default
148         start = 0; done = 0; rdsel = 0; we0 = 0; adr0a = 0; adr0b = 0; we1 = 0; adr1a =
            ↪  0; adr1b = 0; twiddleadr = 0; #10;
149
150         // init inputs to starting values.
151         start = 1; #10;
152     end
153
154 endmodule // agu_testbench
155
156 // fft module test.
157 // loads input from          rom/slade_test_in.memh,
158 // compares output to        rom/slade_test_out.memh,
159 // writes computed output to rom/fft_test_out.memh.
160 // (see "/sim/verify sim fft.ipynb" to process output)
161 //
162 // Note that the slade output values are incorrect, so we should
163 // verify in the jupyter notebook, not with slade_test_out.memh
164 module slade_fft_testbench();
165     logic clk;
```

```
166     logic start, load, done, reset;
167     logic signed [15:0] rd, expected_re, expected_im, wd_re, wd_im;
168     logic [31:0]        wd;
169     logic [31:0]        idx, out_idx, expected;
170
171     logic [15:0]        input_data [0:31];
172     logic [31:0]        expected_out [0:31];
173
174     //
        ↪  https://stackoverflow.com/questions/25607124/test-bench-for-writing-verilog-output-to-a
175     integer             f; // file pointer?
176
177     fft #(16, 5, 0) dut(clk, reset, start, load, rd, wd, done); // no hann
178
179     // clk
180     always
181       begin
182          clk = 1; #5; clk=0; #5;
183       end
184
185     // start of test
186     initial
187       begin
188          $readmemh("rom/slade_test_in.memh", input_data);
189          $readmemh("rom/slade_test_out.memh", expected_out);
190          f = $fopen("rom/fft_test_out.memh", "w"); // write computed vals
191          idx=0; reset=1; #40; reset=0;
192       end
193
194     always @(posedge clk)
195       if (~reset) idx <= idx + 1;
196       else idx <= idx;
197
198     always @(posedge clk)
199       if (load) out_idx <= 0;
200       else if (done) out_idx <= out_idx + 1;
201
202     // load/start logic
203     assign load =  idx < 32;
204     assign start = idx === 32;
205     assign rd = load ? input_data[idx[4:0]] : 0;
206     assign expected = expected_out[out_idx[4:0]];
207     assign expected_re = expected[31:16];
208     assign expected_im = expected[15:0];
209     assign wd_re = wd[31:16];
210     assign wd_im = wd[15:0];
211
212     always @(posedge clk)
213       if (done) begin
214          if (out_idx <= 31) begin
215             $fwrite(f, "%h\n", wd);
216             if (wd !== expected) begin
217                $display("Error @ out_idx %d: expected %b (got %b)    expected: %d+j%d,
                   ↪  got %d+j%d", out_idx, expected, wd, expected_re, expected_im, wd_re,
                   ↪  wd_im);
218             end
219          end else begin
220             $display("Slade FFT test complete.");
```

```
            $fclose(f);
        end
    end
endmodule // fft_testbench

module toplevel_testbench();
   logic clk, nreset, din, uscki, umosi, miso, bck, lrck, scki, fmt, uce, beat_out;
   logic [7:0] LEDs;
   logic [1:0] md;

   logic [63:0] idx;
   logic [31:0] sample_idx, bck_idx;
   logic [24:0] input_sample;
   logic [23:0] input_data [0:9720234];

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

   always_ff @(posedge clk)
     if (~nreset) idx <= idx + 1;
     else idx <= idx;

   initial begin
      $readmemh("rom/toplevel_test_in.memh", input_data);
      sample_idx = 0;
      nreset = 0; #100; nreset = 1;
   end

   // feed in samples!
   always @(negedge lrck) begin
      input_sample <= {1'b0, input_data[sample_idx]};
      sample_idx <= sample_idx + 1;
   end
   always @(negedge bck) begin
      input_sample <= {input_sample[23:0], 1'b0}; // 25 bits wide to account for first
       ↪  non-sampling bck.
   end
   assign din = lrck ? 0 : input_sample[24];

   final_fpga dut(clk, nreset, din, uscki, umosi, uce, bck, lrck, scki, fmt, md, miso,
    ↪  LEDs, beat_out);

   // spi stuff
   always begin
      uscki = 1; #30; uscki = 0; #30;
   end

   initial begin
      umosi = 0;
      uce=0;
      #200000;
      uce=1;
   end


```

```
277 endmodule // toplevel_testbench
278
279 module spi_testbench();
280    logic clk, reset;
281    logic uscki, umosi, miso, uce;
282    logic [31:0] data;
283    logic [4:0]  adr;
284    logic [31:0] sreg_in;
285
286    assign data = {27'b0, adr}; // simple
287
288    // clk
289    always
290      begin
291         clk = 1; #5; clk=0; #5;
292      end
293
294    always begin
295       uscki = 1; #30; uscki = 0; sreg_in = {sreg_in[30:0], miso}; #30;
296    end
297
298    initial begin
299       umosi = 0; uce = 0; reset=1; #100; reset=0; #100; uce=1;
300    end
301
302    spi_slave dut(clk, reset, uscki, umosi, uce, data, adr, miso);
303
304 endmodule // spi_testbench
```

F. *twiddle.py*

```python
# Alec Vercruysse
# 2021-11-16
# generate N-point q-bit two's complement integer twiddle bits

import numpy as np


# required since we need signed represntation
# https://stackoverflow.com/questions/699866/python-int-to-binary-string

def int2bin(integer, digits):
    if integer >= 0:
        return bin(integer)[2:].zfill(digits)
    else:
        return bin(2**digits + integer)[2:]


i2b = np.vectorize(int2bin)

N = 32
q = 16

n = np.arange(N/2)
#wi = np.exp(1j * np.pi*2/N)
#w = np.power(wi, n)

#w_re = np.real(w)
w_re = np.cos(2*np.pi*n/N)
w_re = (w_re * (2**(q-1) - 1)).astype('int')
w_re = i2b(w_re, q)

# w_im = np.imag(w)
w_im = -np.sin(2*np.pi*n/N)
w_im = (w_im * (2**(q-1) - 1)).astype('int')
w_im = i2b(w_im, q)

with open("twiddle.vectors", 'w') as f:
    for i in range(len(w_re)):
        f.write(w_re[i] + "" + w_im[i] + "\n")

with open("../simulation/modelsim/rom/twiddle.vectors", 'w') as f:
    for i in range(len(w_re)):
        f.write(w_re[i] + "" + w_im[i] + "\n")
```

G. *fft.cpp*

Used to verify the FPGA FFT operation.

```
//32 point FFT implementation in C++ based on Slade paper

#include <stdio.h>

int main(){

    typedef int16_t newType;

    // Initial real data
     newType Data_r[32] = {
        0x3FF,
        0x3FF,
        0xfc01,
        0xfc01,

        0x3FF,
        0x3FF,
        0xfc01,
        0xfc01,

        0x3FF,
        0x3FF,
        0xfc01,
        0xfc01,

        0x3FF,
        0x3FF,
        0xfc01,
        0xfc01,

        0x3FF,
        0x3FF,
        0xfc01,
        0xfc01,

        0x3FF,
        0x3FF,
        0xfc01,
        0xfc01,

        0x3FF,
        0x3FF,
        0xfc01,
        0xfc01,

        0x3FF,
        0x3FF,
        0xfc01,
        0xfc01
    };

    // Initial imaginary data
     newType Data_i[32] = {
        0x0,
        0x0,
```

```
56          0x0,
57          0x0,
58          0x0,
59          0x0,
60          0x0,
61          0x0,
62          0x0,
63          0x0,
64          0x0,
65          0x0,
66          0x0,
67          0x0,
68          0x0,
69          0x0,
70          0x0,
71          0x0,
72          0x0,
73          0x0,
74          0x0,
75          0x0,
76          0x0,
77          0x0,
78          0x0,
79          0x0,
80          0x0,
81          0x0,
82          0x0,
83          0x0,
84          0x0,
85          0x0
86      };
87
88      // twiddle factors real
89      newType Tw_r[16] = {
90          0x7fff,
91          0x7d89,
92          0x7641,
93          0x6a6d,
94          0x5a82,
95          0x471c,
96          0x30fb,
97          0x18f9,
98          0x0,
99          0xe707,
100         0xcf05,
101         0xb8e4,
102         0xa57e,
103         0x9593,
104         0x89bf,
105         0x8277
106     };
107
108     // twiddle factors imaginary
109     newType Tw_i[16] = {
110         0,
111         0x1859,
112         0x30fb,
113         0x471c,
```

```
114             0x5a82,
115             0x6a6d,
116             0x7641,
117             0x7d89,
118             0x7fff,
119             0x7d89,
120             0x7641,
121             0x6a6d,
122             0x5a82,
123             0x471c,
124             0x30fb,
125             0x1859
126         };
127 
128         for(newType i = 0; i < 5; i++){ // 5 levels for 2^5 = 32 point FFT
129             for(newType j = 0; j < 16; j++){ // operating on 16 bit ints
130                 newType ja=j<<1;
131                 newType jb=ja+1;
132                 ja = ((ja << i) | (ja >> (5-i))) & 0x1f; // Address A; 5 bit circular left
                    ↪  shift
133                 jb = ((jb << i) | (jb >> (5-i))) & 0x1f ; // Address B; implemented using
                    ↪  C statements
134                 newType TwAddr = ((0xfffffff0 >> i) & 0xf) & j; // Twiddle addresses
135 
136                 newType temp_r = ((Data_r[jb] * Tw_r[TwAddr]) / 32768) - ((Data_i[jb] *
                    ↪  Tw_i[TwAddr]) /32768);
137                 newType temp_i = ((Data_r[jb] * Tw_i[TwAddr]) / 32768) + ((Data_i[jb] *
                    ↪  Tw_r[TwAddr]) / 32768);
138                 if(ja == 0 || jb == 0){
139                     printf("2nd val, ja: %d, jb: %d, realA: %x, realB %x, tempR: %x,
                        ↪  tempI: %x, TwR: %x, TwI: %x, TwAddr: %d, \n",ja, jb, Data_r[ja],
                        ↪  Data_r[jb],temp_r, temp_i, Tw_r[TwAddr], Tw_i[TwAddr], TwAddr);
140                 }
141 
142                 //Sets data for level
143                 Data_r[jb] = Data_r[ja] - temp_r;
144                 Data_i[jb] = Data_i[ja] - temp_i;
145                 Data_r[ja] += temp_r;
146                 Data_i[ja] += temp_i;
147             }
148             for(newType k = 0; k < 32; k++){
149                 printf("%d %x %x \n", k, Data_r[k], Data_i[k]);
150             }
151 
152         }
153     }
```

APPENDIX C
STM32F401RE C CODE

Requires standard E155 libraries provided by Prof. Brake.

A. *Servo.c*

```
#include <stdio.h>
#include "STM32F401RE.h"

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

#define SERVO_PIN    4 // GPIOA
#define BEAT_PIN     0 // GPIOA
#define GREEN_PIN    10 // GPIOB
#define RED_PIN      1 // GPIOA

#define USART_ID USART2_ID

#define MIN_MICROS 770
#define MAX_MICROS 2270
#define CYCLE_MICROS 20000

// Used for testing
const float usPerDeg = (MAX_MICROS - MIN_MICROS)/180;
// takes in an angle from 0-180, spits out microseconds to write to servo
long angleToMicros(float angle){
  return (long)(angle * usPerDeg) + MIN_MICROS;
}

// Servo Pattern points

// Red laser pattern locations
#define RPP0 770
#define RPP1 917
#define RPP2 1022
#define RPP3 1136
#define RPP4 1242
#define RPP5 1370
#define RPP6 1465
#define RPP7 1599
#define RPP8 1724
#define RPP9 1852
#define RPPA 1960
#define RPPB 2084
#define RPPC 2204
#define RPPD 2269

// Green laser pattern locations
#define GPP0 770
#define GPP1 870
#define GPP2 966
#define GPP3 1091
#define GPP4 1208
#define GPP5 1328
#define GPP6 1442
#define GPP7 1532
```

```
#define GPP8 1664
#define GPP9 1792
#define GPPA 1862
#define GPPB 1978
#define GPPC 2134
#define GPPD 2269

//Pattern set 1

// start, end, red, green
//  int pats[6][4] = {
//                          {GPP0,  GPP1, 1, 1},
//                          {RPP0,  RPP1, 0, 1},
//                          {GPP2, GPP3,  1, 1},
//                          {RPP3, RPP4,  1, 0},
//                          {RPP5, RPP6,  1, 0},
//                          {RPP7, RPP8,  1, 0} //1328, 1442
//                          };

// Pattern set 2

int pats[6][4] = {
                   {GPPC,  GPPD, 1, 1},
                   {GPPA,  GPPB, 1, 1},
                   {GPP8, GPP9,  0, 1},
                   {RPP6, RPP7,  1, 0},
                   {RPP4, RPP5,  1, 0},
                   {RPP2, RPP3,  1, 0}
                   };

// used for printing
char text[50];

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

  /* Configure APB prescalers
      1. Set APB2 (high-speed bus) prescaler to no division
      2. Set APB1 (low-speed bus) to divide by 2.
  */
  RCC->CFGR.PPRE2 = 0b000;
  RCC->CFGR.PPRE1 = 0b100;

  configureClock();

  // Enable GPIOA clock
  RCC->APB2ENR |= (1 << 16); //Enable TIM10
  RCC->APB2ENR |= (1 << 17); //Enable TIM10
  RCC->APB2ENR |= (1 << 18); //Enable TIM11
  RCC->AHB1ENR.GPIOAEN = 1; // Enable GPIOA Clock
  RCC->AHB1ENR.GPIOBEN = 1; // Enable GPIOA Clock

  // Servo Pin
  pinMode(GPIOA, SERVO_PIN, GPIO_OUTPUT);
  // Beat Pin from FPGA
  pinMode(GPIOA, BEAT_PIN, GPIO_INPUT);
  // Red Laser Transistor -> Relay
```

```
111   pinMode(GPIOA, RED_PIN, GPIO_OUTPUT);
112   // Green Laser Transistor -> Relay
113   pinMode(GPIOB, GREEN_PIN, GPIO_OUTPUT);
114
115
116   USART_TypeDef* USART = initUSART(USART_ID);
117
118   // Print millis timer
119   // PSC + 1 = 42000
120   // 84MHz with a 42000 prescaler gives 0.5 ms resolution
121   // Can count t0 32 seconds maximum
122   TIM9->PSC = 41999;
123   TIM9->CR1 |= (1 << 0); // Enable the counter
124   TIM9->EGR |= (1 << 0);  // Generate an update event
125
126   // Servo Micros Timer:
127   // Goal, Count to 20,000, microsecond resolution
128   // PSC + 1 = 84
129   // 84MHz with a 84 prescaler gives microsecond resolution
130   // Can count to 2^16 > 20,000
131   TIM10->PSC = 83;
132   TIM10->CR1 |= (1 << 0); // Enable the counter
133   TIM10->EGR |= (1 << 0);  // Generate an update event
134
135   // Beat Pin Millis Length Timer:
136   // PSC + 1 = 42000
137   // 84MHz with a 42000 prescaler gives 0.5 ms resolution
138   // Can count t0 32 seconds maximum
139   TIM11->PSC = 41999;
140   TIM11->CR1 |= (1 << 0); // Enable the counter
141   TIM11->EGR |= (1 << 0);  // Generate an update event
142
143
144   TIM9->CNT = 0;  // Print
145   TIM10->CNT = 0; // Servo Micros
146   TIM11->CNT = 0; // BPM
147
148   uint8_t prevBeatRead = 0; // boolean variable that tracks previous read value
149
150   int beatIntervalMs = 10000;
151
152   int patInd = 0;
153
154   uint8_t patState = 0;
155
156   digitalWrite(GPIOB, GREEN_PIN, 1);
157   digitalWrite(GPIOA, RED_PIN, 1);
158
159   while(1){
160
161     // Pattern selection
162
163     if(beatIntervalMs < 250){ //< 0.25sec, bpm > 240
164       patInd = 0;
165     } else if(beatIntervalMs < 500){ //< 0.5sec, bpm > 120
166       patInd = 2;
167     } else if(beatIntervalMs < 1000){ //< 1sec, bpm > 60
168       patInd = 3;
```

```
169       } else if (beatIntervalMs < 2000) { // < 2sec, 60 > bpm > 30
170         patInd = 4;
171       } else if (beatIntervalMs >= 2000){ // > 2sec, 30 > bpm
172         patInd = 5;
173       }
174
175       // Writes Servo PWM signal out
176       long curMicros = TIM10->CNT;
177       long microsDes = pats[patInd][patState];
178       if(curMicros < microsDes){
179         digitalWrite(GPIOA, SERVO_PIN, 1);
180       } else if(curMicros < CYCLE_MICROS){
181         digitalWrite(GPIOA, SERVO_PIN, 0);
182       } else {
183         TIM10->CNT = 0;
184       }
185
186       uint8_t curBeatRead = digitalRead(GPIOA, BEAT_PIN);
187
188       if (curBeatRead == 1 && prevBeatRead == 0){ // only on the rising edge
189         beatIntervalMs = (TIM11->CNT)/2;
190
191         digitalWrite(GPIOA, RED_PIN, pats[patInd][2]);
192         digitalWrite(GPIOB, GREEN_PIN, pats[patInd][3]);
193
194         // sprintf(text, "\n interval: %d | ind: %d | red: %d | green: %d\n",
          ↪  beatIntervalMs, patInd, pats[patInd][2], pats[patInd][3]);
195         // for (size_t j = 0; text[j]; j++) {
196         //     sendChar(USART, text[j]);
197         // }
198         TIM11->CNT = 0;
199       }
200       prevBeatRead = curBeatRead;
201
202       if(TIM9->CNT > 200){
203           // sprintf(text, "\n%d  %d  %d\n", beatIntervalMs, patInd, patState);
204           // for (size_t j = 0; text[j]; j++) {
205           //     sendChar(USART, text[j]);
206           // }
207           TIM9->CNT = 0;
208       }
209     }
210   }
```

REFERENCES

[1] G. Slade, “The fast fourier transform in hardware: A tutorial based on an FPGA implementation,” 03 2013.