Lab 6 Report: Live-Time Audio Visualizer

Hunter North and Peter Zhong

EE 371

Jun 2, 2021

Procedure

Lab 6 asked us to implement a project of our own with at least one input device and at least one output device. After some discussion between the group members, we decided to implement a live-time audio visualiser on FPGA. This visualizer takes an mp3 audio file as input, and outputs the average amplitude of the audio as colored bars with different heights moving across the VGA display.

Here are some module-by-module specifications of our live-time audio visualizer project:

- The width of the bars are fixed to 29 pixels with 2-pixel gaps (see “DE1_SoC.sv”).

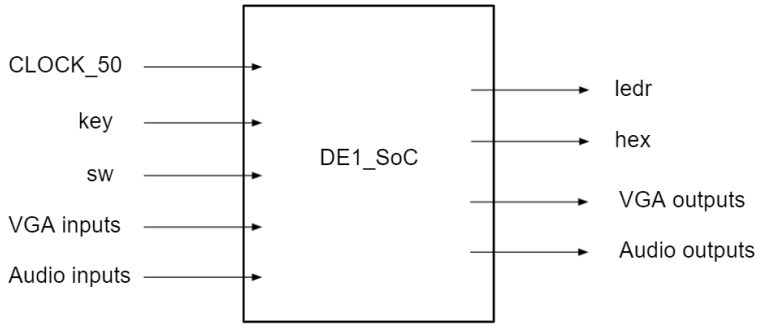

Figure 1. DE1_SoC Block Diagram. Image showing the correct input and output signals required to drive the top-level module.

- The heights of the bars are the average amplitude of the audio scaled down linearly to fit the 480 vertical pixels on the VGA display (see “height_picker.sv”).

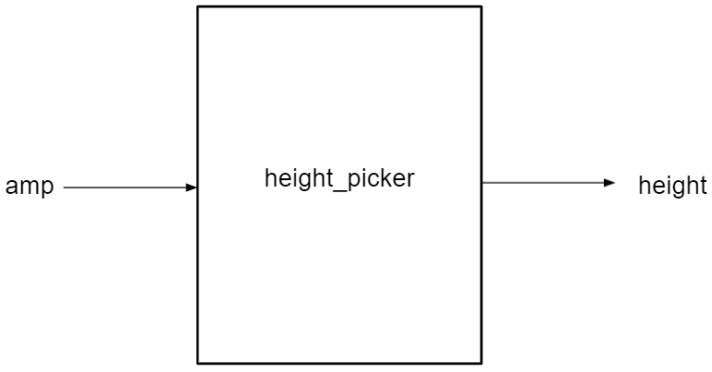

Figure 2. height_picker Block Diagram. Image showing the correct input and output signals required to drive the height_picker module.

- The RGB values of the colors of the bars are respectively determined by the first 8 bits, second 8 bits, and third 8 bits of the average amplitude of the audio (see “color_picker.sv”).

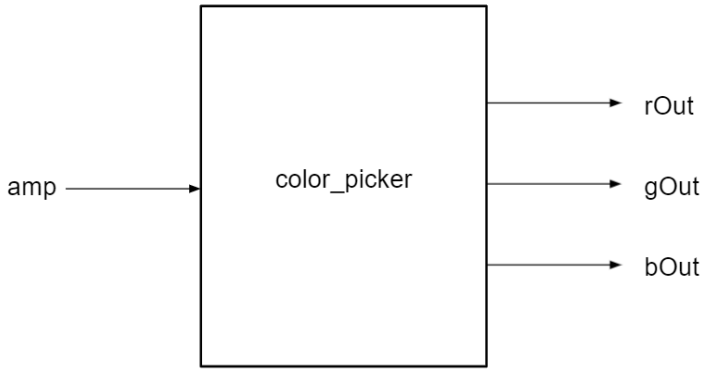

Figure 3. color_picker Block Diagram. Image showing the correct input and output signals required to drive the color_picker module.

- The RGB values and the heights of the bars are synchronously updated to the top-level module with a bar drawer module (see “bar_drawer.sv”).

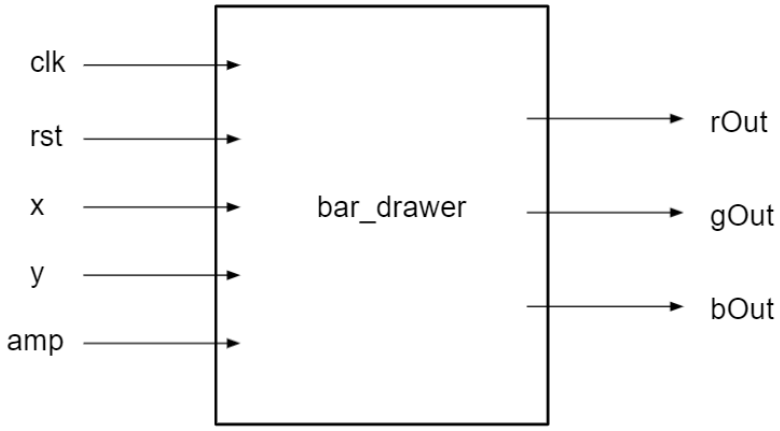

Figure 4 bar_drawer Block Diagram. Image showing the correct input and output signals required to drive the bar_drawer module.

- The updating frequency of the average amplitude of the bar and the number of samples to take the average amplitude of is determined by the parameter UPDATE_FREQ in the audio averaging module. The average values of the samples are shifted across the bars through a shift register implemented with an always_ff block (see “audio_averaging_shifting.sv”).

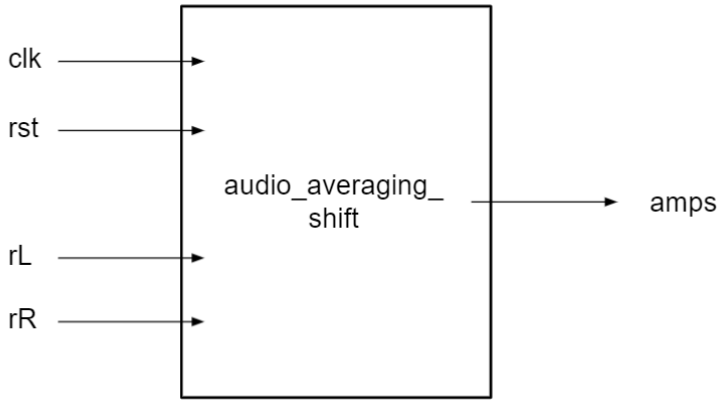

Figure 5. audio_averaging_shift Block Diagram. Image showing the correct input and output signals required to drive the audio_averaging_shift module.

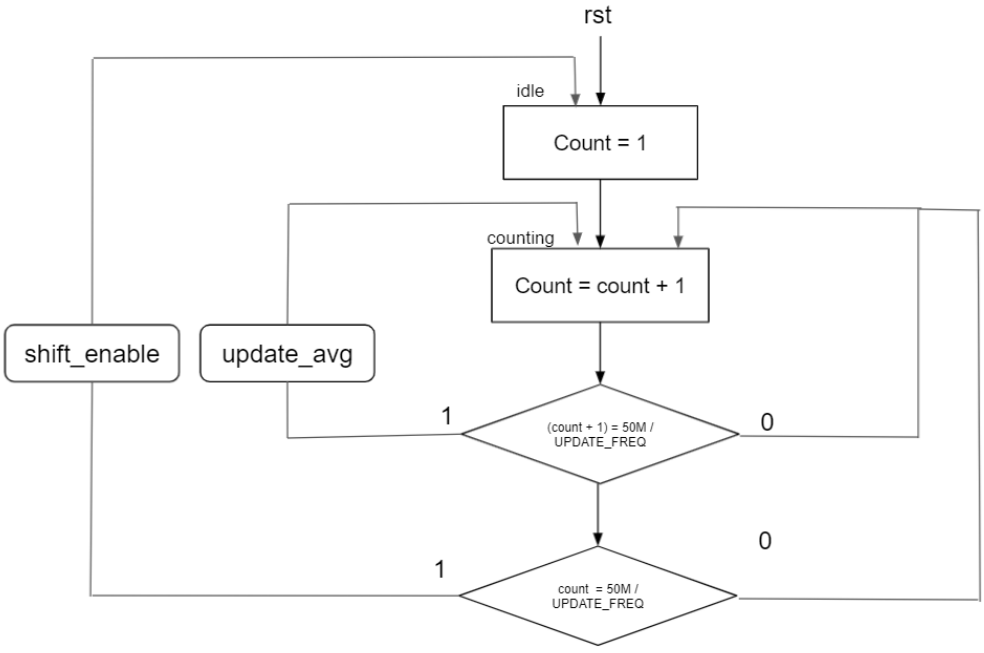

Figure 6. ASMD chart for the control signals in the audio_averaging_shift module

Lastly, we went about wiring up all of these modules together. We created the audio averaging shift module and wired each output to its own bar_drawer module. Each bar_drawer module holds a color and height picker which decide the characteristics of the audio bar. The desired information is utilized by the video driver to paint an image on the VGA monitor.

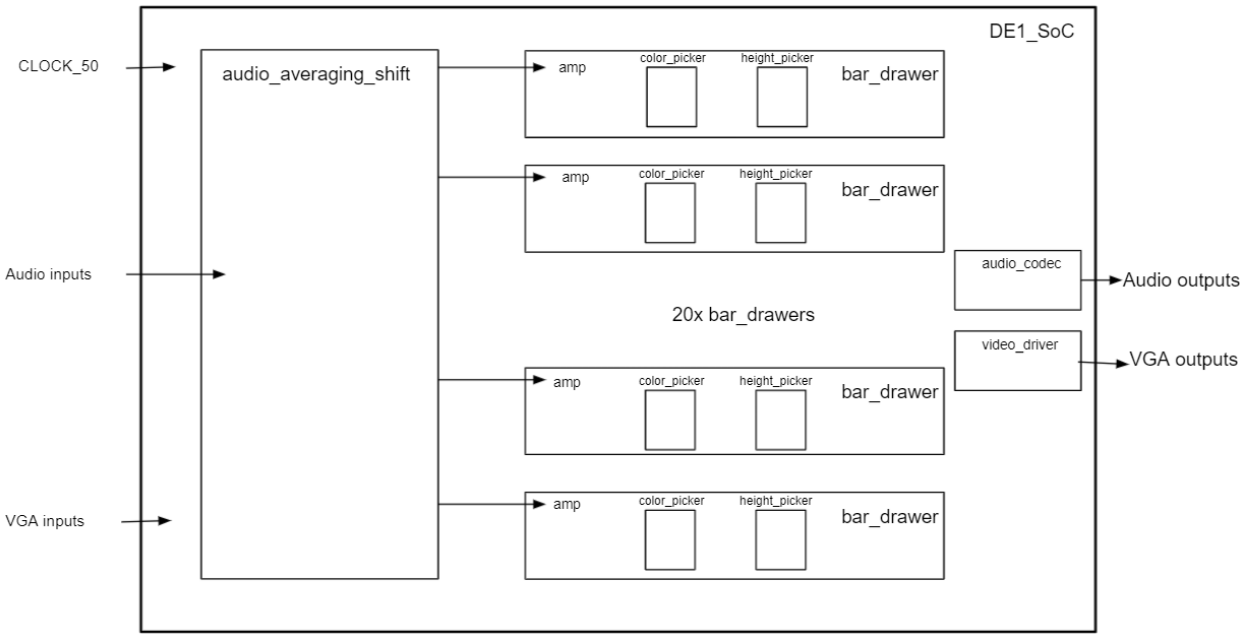

Figure 7. System Structure. Diagram that shows how all the individual modules are wired together in order to build the Live Time Audio Visualizer.

Results

height_picker.sv

This waveform below (Figure 8) shows the successful testing of the height_picker module. In order to test the functionality of the height_picker module, we picked three random 24-bit numbers and passed them into the module. As can be seen from the waveform below, the height_picker module successfully scaled the 24-bit numbers down to a number between 0 and 480.

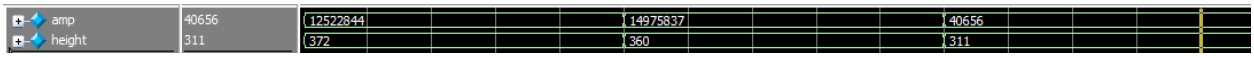

Figure 8. height_picker testbench. Testbench that is run to confirm that the height_picker module is functioning correctly.

color_picker.sv

This waveform below (Figure 9) shows the successful testing of the color_picker module. In order to test the functionality of the color_picker module, we picked three random 24-bit numbers to be the average amplitude of audio, and passed them into the module. As can be seen from the waveform below, the color_picker module successfully determined the RGB values of the colors of the bars through the first, second and third 8 bits of the 24-bit numbers.

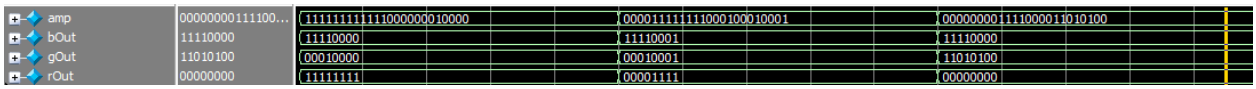

Figure 9. color_picker testbench. Testbench that is run to confirm that the color_picker module is functioning correctly.

bar_drawer.sv

This waveform below (Figure 10) shows the successful testing of the bar_drawer module. In order to test the functionality of the bar_drawer module, we tested a medium sized amplitude of 8 million. We began by resetting the module, and passed the amplitude value into the module. We also tested the case when the VGA driver is drawing pixels that are outside of the height of the current bar. As can be seen from the waveform below, the bar_drawer module returns the correct color when the currently drawn pixel is within the height of the bar, and returns all zeros when the pixel is out of range to draw black.

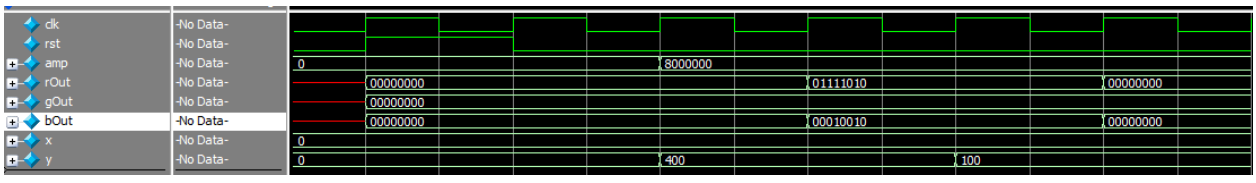

Figure 10. bar_drawer testbench. Testbench that is run to confirm that the color_picker module is functioning correctly.

audio_averaging_shift.sv

This waveform below (Figure 11) shows the successful testing of the audio_averaging_shift module. In order to test the functionality of the audio_averaging_shift module, we set the dut's updating frequency to once every 5 clock cycles, and passed a sequence of random input amplitude values into the module. As can be seen from the waveform below, the module successfully computes the average value of the previous amplitudes and updates the most recent average value before shifting it into the shift registers used to support the bar_drawers modules

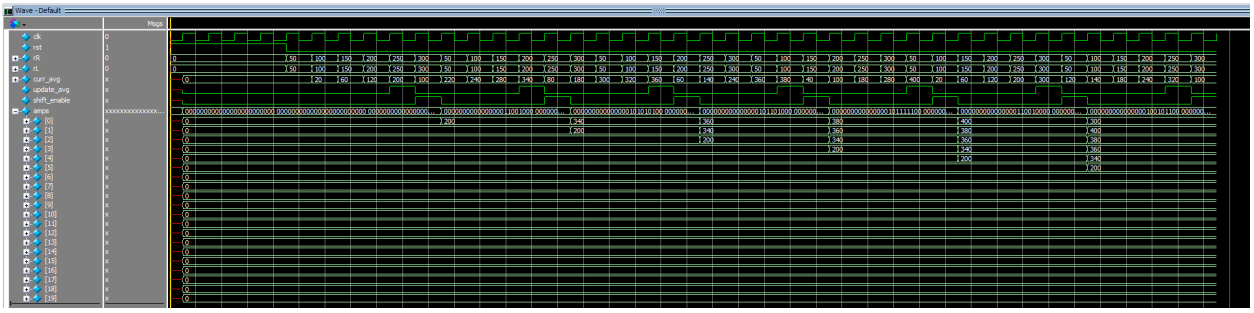

Figure 11. Waveform generated by Appendix 1.E that demonstrates a successful test of the audio_averaging_shift module.

Appendix

1.A - DE1_SoC.sv

```
// Hunter North and Peter Zhong
// 05/26/2021
// EE 371
// Lab 6
//
// Top level module which will create the live time audio visualizer
// Parameters required:
//    HEX             - 7 bit structures representing hex displays on the DE1_SoC
//    KEY             - 4 different signals represnting the press on a key button on a DE1_SoC
//    LEDR            - 10 different signals representing the LEDs on DE1_SoC
//    SW              - 10 different signals represetning the switches on the DE1_SoC
//    CLOCK_50        - 50 Mhz clock signal
//    VGA drivers     - A set of drivers used to make the VGA function
//    Audio drivers   - A set of drivers used to make the audio function
//
module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW,
                CLOCK_50, CLOCK2_50, VGA_R, VGA_G, VGA_B, VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, VGA_VS,
                FPGA_I2C_SCLK, FPGA_I2C_SDAT, AUD_XCK, AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK,
                AUD_ADCDAT, AUD_DACDAT);

    // DE1_SoC hardware
    output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    output logic [9:0] LEDR;
    input logic [3:0] KEY;
    input logic [9:0] SW;

    // DE1_SoC drivers for VGA
    input CLOCK_50, CLOCK2_50;
    output [7:0] VGA_R;
    output [7:0] VGA_G;
    output [7:0] VGA_B;
    output VGA_BLANK_N;
    output VGA_CLK;
    output VGA_HS;
    output VGA_SYNC_N;
    output VGA_VS;

    // I2C Audio/Video config interface
    output FPGA_I2C_SCLK;
    inout FPGA_I2C_SDAT;
    // Audio CODEC
    output AUD_XCK;
    input AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK;
    input AUD_ADCDAT;
    output AUD_DACDAT;
```

```
    // Make system resetable with key0
    logic reset;
    assign reset = ~KEY[0];

    // Instantiate vga driver module
    logic [9:0] x;
    logic [8:0] y;
    logic [7:0] r, g, b;
    video_driver #(.WIDTH(640), .HEIGHT(480))
        v1 (.CLOCK_50, .reset, .x, .y, .r, .g, .b,
            .VGA_R, .VGA_G, .VGA_B, .VGA_BLANK_N,
            .VGA_CLK, .VGA_HS, .VGA_SYNC_N, .VGA_VS);

    // Logic units to drive audio  modules
    logic read_ready, write_ready, read, write;
    logic signed [23:0] readdata_left, readdata_right;
    logic signed [23:0] writedata_left, writedata_right;
    logic signed [23:0] noisy_left, noisy_right;

    // Outputs the input audio
    assign writedata_left = readdata_left;
    assign writedata_right = readdata_right;
    // Only read or write when both are possible
    assign read = read_ready & write_ready;
    assign write = read_ready & write_ready;

    // Instantiate audio modules
    clock_generator my_clock_gen(CLOCK2_50, 1'b0, AUD_XCK);
    audio_and_video_config cfg(CLOCK_50, 1'b0, FPGA_I2C_SDAT, FPGA_I2C_SCLK);
    audio_codec codec(CLOCK_50, 1'b0, read, write, writedata_left, writedata_right,
                      AUD_ADCDAT, AUD_BCLK, AUD_ADCLRCK, AUD_DACLRCK, read_ready, write_ready,
                      readdata_left, readdata_right, AUD_DACDAT);

    // Instantiate audio segement averaging to be shifted into bar modules
    logic [23:0] amps [0:19];
    audio_averaging_shift #(3) aas (.clk(CLOCK_50), .rst(reset), .amps, .rL(readdata_left), .rR(readdata_right));

    // Create 20 different bars to visually represent audio
    logic [7:0] r1, g1, b1;
    bar_drawer bar_1 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[0]), .rOut(r1), .gOut(g1), .bOut(b1));

    logic [7:0] r2, g2, b2;
    bar_drawer bar_2 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[1]), .rOut(r2), .gOut(g2), .bOut(b2));
```

```
90
91      logic [7:0] r3, g3, b3;
92      bar_drawer bar_3 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[2]), .rOut(r3), .gOut(g3), .bOut(b3));
93
94      logic [7:0] r4, g4, b4;
95      bar_drawer bar_4 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[3]), .rOut(r4), .gOut(g4), .bOut(b4));
96
97      logic [7:0] r5, g5, b5;
98      bar_drawer bar_5 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[4]), .rOut(r5), .gOut(g5), .bOut(b5));
99
100     logic [7:0] r6, g6, b6;
101     bar_drawer bar_6 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[5]), .rOut(r6), .gOut(g6), .bOut(b6));
102
103     logic [7:0] r7, g7, b7;
104     bar_drawer bar_7 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[6]), .rOut(r7), .gOut(g7), .bOut(b7));
105
106     logic [7:0] r8, g8, b8;
107     bar_drawer bar_8 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[7]), .rOut(r8), .gOut(g8), .bOut(b8));
108
109     logic [7:0] r9, g9, b9;
110     bar_drawer bar_9 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[8]), .rOut(r9), .gOut(g9), .bOut(b9));
111
112     logic [7:0] r10, g10, b10;
113     bar_drawer bar_10 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[9]), .rOut(r10), .gOut(g10), .bOut(b10));
114
115     logic [7:0] r11, g11, b11;
116     bar_drawer bar_11 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[10]), .rOut(r11), .gOut(g11), .bOut(b11));
117
118     logic [7:0] r12, g12, b12;
119     bar_drawer bar_12 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[11]), .rOut(r12), .gOut(g12), .bOut(b12));
120
121     logic [7:0] r13, g13, b13;
122     bar_drawer bar_13 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[12]), .rOut(r13), .gOut(g13), .bOut(b13));
123
124     logic [7:0] r14, g14, b14;
125     bar_drawer bar_14 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[13]), .rOut(r14), .gOut(g14), .bOut(b14));
126
127     logic [7:0] r15, g15, b15;
128     bar_drawer bar_15 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[14]), .rOut(r15), .gOut(g15), .bOut(b15));
129
130     logic [7:0] r16, g16, b16;
131     bar_drawer bar_16 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[15]), .rOut(r16), .gOut(g16), .bOut(b16));
132
133     logic [7:0] r17, g17, b17;
134     bar_drawer bar_17 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[16]), .rOut(r17), .gOut(g17), .bOut(b17));
```

```
136     logic [7:0] r18, g18, b18;
137     bar_drawer bar_18 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[17]), .rOut(r18), .gOut(g18), .bOut(b18));
138
139     logic [7:0] r19, g19, b19;
140     bar_drawer bar_19 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[18]), .rOut(r19), .gOut(g19), .bOut(b19));
141
142     logic [7:0] r20, g20, b20;
143     bar_drawer bar_20 (.clk(CLOCK_50), .rst(reset), .x, .y, .amp(amps[19]), .rOut(r20), .gOut(g20), .bOut(b20));
144
145     // Draw to the VGA
146     always_ff @(posedge CLOCK_50) begin
147         if (x > 0 && x < 31) begin
148             r <= r1;
149             g <= g1;
150             b <= b1;
151         end else if (x > 32 && x < 63) begin
152             r <= r2;
153             g <= g2;
154             b <= b2;
155         end else if (x > 64 && x < 95) begin
156             r <= r3;
157             g <= g3;
158             b <= b3;
159         end else if (x > 96 && x < 127) begin
160             r <= r4;
161             g <= g4;
162             b <= b4;
163         end else if (x > 128 && x < 159) begin
164             r <= r5;
165             g <= g5;
166             b <= b5;
167         end else if (x > 160 && x < 191) begin
168             r <= r6;
169             g <= g6;
170             b <= b6;
171         end else if (x > 192 && x < 223) begin
172             r <= r7;
173             g <= g7;
174             b <= b7;
175         end else if (x > 224 && x < 255) begin
176             r <= r8;
177             g <= g8;
178             b <= b8;
179         end else if (x > 256 && x < 287) begin
```

```
            r <= r9;
            g <= g9;
            b <= b9;
        end else if (x > 288 && x < 319) begin
            r <= r10;
            g <= g10;
            b <= b10;
        end else if (x > 320 && x < 351) begin
            r <= r11;
            g <= g11;
            b <= b11;
        end else if (x > 352 && x < 383) begin
            r <= r12;
            g <= g12;
            b <= b12;
        end else if (x > 384 && x < 415) begin
            r <= r13;
            g <= g13;
            b <= b13;
        end else if (x > 416 && x < 447) begin
            r <= r14;
            g <= g14;
            b <= b14;
        end else if (x > 448 && x < 479) begin
            r <= r15;
            g <= g15;
            b <= b15;
        end else if (x > 480 && x < 511) begin
            r <= r16;
            g <= g16;
            b <= b16;
        end else if (x > 512 && x < 543) begin
            r <= r17;
            g <= g17;
            b <= b17;
        end else if (x > 544 && x < 575) begin
            r <= r18;
            g <= g18;
            b <= b18;
        end else if (x > 576 && x < 607) begin
            r <= r19;
            g <= g19;
            b <= b19;
        end else if (x > 608 && x < 639) begin
            r <= r20;
```

```
            g <= g20;
            b <= b20;
        end else begin
            r <= SW[7:0];
            g <= '0;
            b <= '0;
        end
    end

    // Turn off hex displays
    assign HEX0 = '1;
    assign HEX1 = '1;
    assign HEX2 = '1;
    assign HEX3 = '1;
    assign HEX4 = '1;
    assign HEX5 = '1;

endmodule
```

1.B - height_picker.sv

```
1  // Hunter North and Peter Zhong
2  // 05/26/2021
3  // EE 371
4  // Lab 6
5  //
6  // Module that will output the correct height based off an audio segment
7  // Parameters Used:
8  //    amp      - Amplitude of sound to be visually represented by bar
9  //    height   - Decided height of bar
10 module height_picker(amp, height);
11
12    input logic [23:0] amp;       // Dimension might need to be adjusted
13
14    output logic [8:0] height;
15
16    // Scale amp by 480
17    integer scaled_amp;
18    assign scaled_amp = amp * 23'h0111E0;
19
20    // Divide amp down by max of 24 bit number
21    logic [23:0] shrunk_amp;
22    assign shrunk_amp = scaled_amp / 24'hffffff;
23
24    // Assign final height
25    assign height = 480 - shrunk_amp;
26
27 endmodule
28
29 // Module to test the height_picker module
30 module height_picker_testbench();
31
32    logic [23:0] amp;       // Dimension might need to be adjusted
33    logic [8:0] height;
34
35    height_picker dut (.*);
36
37    initial begin
38       amp <= 24'hbf155c; #10;
39       amp <= 24'he4835d; #10;
40       amp <= 24'h009ed0; #10;
41    end
42 endmodule
```

1.C - color_picker.sv

```
1  // Hunter North and Peter Zhong
2  // 05/26/2021
3  // EE 371
4  // Lab 6
5  //
6  // Module that will output the correct color based off of audio segment
7  // Parameters Used:
8  //    amp      - Amplitude of sound to be visually represented by bar
9  //    rOut     - Amount of red coloring in output color
10 //    gOut     - Amount of green coloring in output color
11 //    bOut     - Amount of blue coloring in output color
12
13 module color_picker(amp, rOut, gOut, bOut);
14
15    input logic [23:0] amp;       // Dimension might need to be adjusted
16
17    output logic [7:0] rOut, gOut, bOut;
18
19    assign rOut = amp[23:16];
20    assign bOut = amp[15:8];
21    assign gOut = amp[7:0];
22
23 endmodule
24
25 // Module to test the color_picker module
26 module color_picker_testbench();
27
28    logic [23:0] amp;       // Dimension might need to be adjusted
29    logic [7:0] rOut, gOut, bOut;
30
31    color_picker dut (.*);
32
33    initial begin
34       amp <= 24'hfff010; #10;
35       amp <= 24'h0ff111; #10;
36       amp <= 24'h00f0d4; #10;
37    end
38 endmodule
```

1.D - bar_drawer.sv

```
// Hunter North and Peter Zhong
// 05/26/2021
// EE 371
// Lab 6
//
// Module that will draw a bar on the screen based on an audio segment
// Parameters Used:
//    clk      - Clock to coordinate timing in module
//    rst      - Signal to reset bar to all black
//    x        - X component of location that is being drawn to
//    y        - Y component of location that is being drawn to
//    amp      - Amplitude of sound to be visually represented by bar
//    rOut     - Red coloring in bar
//    gOut     - Green coloring in bar
//    bOut     - Blue coloring in bar
module bar_drawer (clk, rst, x, y, amp, rOut, gOut, bOut);

   input logic          clk;
   input logic          rst;
   input logic [9:0]    x;
   input logic [8:0]    y;
   input logic [23:0]   amp;        // Dimension might need to be adjusted

   output logic [7:0]   rOut, gOut, bOut;

   /****Determine Height of Bar*******/
   logic [8:0] height;
   height_picker hp (.*);


   /****Determine Color of Bar********/
   logic [7:0] rBar, gBar, bBar;
   color_picker cp (.amp, .rOut(rBar), .gOut(gBar), .bOut(bBar));

   /****Draw to Board ***************/
   always_ff @(posedge clk) begin
      // Black when resetting
      if (rst) begin
         rOut <= '0;
         gOut <= '0;
         bOut <= '0;
      // Black if outisde the determined bar height
      end else if (y < height) begin
         rOut <= '0;
         gOut <= '0;
         bOut <= '0;
      // Draw bar calculated colors if inside bar height
      end else if (y >= height) begin
         rOut <= rBar;
         gOut <= gBar;
         bOut <= bBar;
      end
   end
endmodule
```

```
55
56  // Module to test the bar_drawer module
57  module bar_drawer_testbench ();
58
59      // Inputs to drive dut
60      logic       clk;
61      logic       rst;
62      logic [9:0] x;
63      logic [8:0] y;
64      logic [23:0] amp;       // Dimension might need to be adjusted
65
66      // Outputs for dut
67      logic [7:0] rOut, gOut, bOut;
68
69      // Simulated clock for testing
70      parameter clock_period = 100;
71      initial begin
72          clk <= 0;
73          forever #(clock_period / 2) clk <= ~clk;
74      end
75
76      // Instantiate module for testing
77      bar_drawer dut (.*);
78
79      // Test dut
80      initial begin
81          // Set defaults
82          rst <= 0;   x <= 0;  y <= 0;  amp <= 0;      @(posedge clk);
83
84          // Reset system
85          rst <= 1;                           repeat(1)@(posedge clk);
86
87          // Undo reset
88          rst <= 0;                           repeat(1)@(posedge clk);
89
90          // Test large (250) amp
91          // inside bar...
92          y <= 400;   amp <= 8000000;         repeat(2)@(posedge clk);
93          // Outside bar...
94          y <= 100;                           repeat(2)@(posedge clk);
95
96          $stop;
97      end
98  endmodule
```

1.E - audio_averaging_shift.sv

```
// Hunter North and Peter Zhong
// 05/26/2021
// EE 371
// Lab 6
//
// Module that will calculate the average value over a set of audio samples. This module
// will store 20 difference averages in a shift register that will continually shift right
// in order to make room for the newest average.
// Parameters required:
//    UPDATE_FREQ    - The rate at which we are shifting in the newest average value
//    clk            - Signal to coordinate timing in module
//    rst            - Signal to reset the module
//    rL             - The right amplitude from audio file
//    rR             - The left amplitude from audio file
//    amps           - A 20 unit structure which stores the previous 20 amplitudes
//
module audio_averaging_shift #(parameter UPDATE_FREQ = 5) (clk, rst, amps, rL, rR);

   input logic clk, rst;
   input logic [23:0] rL, rR;
   output logic [23:0] amps [0:19];

   // control logic for updating average value and shifting registers
   integer count;
   logic update_avg, shift_enable;

   always_ff @(posedge clk) begin
      if (rst) begin
         count <= 1;
         update_avg <= 0;
         shift_enable <= 0;
      end
      // update avg one cycle before shifting registers
      else if ( (count + 1) == 50000000 / UPDATE_FREQ) begin
         count <= count + 1;
         update_avg <= 1;
         shift_enable <= 0;
      end
      else if (count == 50000000 / UPDATE_FREQ) begin
         count <= 1;
         update_avg <= 0;
         shift_enable <= 1;
      end
      else begin
         count <= count + 1;
         shift_enable <= 0;
         update_avg <= 0;
      end
   end

```

```
50
51
52      // audio averaging, update average
53      logic [23:0] curr_avg, avg;
54      integer samples; // number of samples to average for every update operation
55      // assign samples = 48000 / UPDATE_FREQ; // 48k samples per second
56      // for testbench
57      assign samples = 50000000 / UPDATE_FREQ;
58
59      always_ff @(posedge clk) begin
60          if (rst) begin
61              curr_avg <= 0;
62              avg <= 0;
63          end
64          else if (update_avg) begin
65              avg <= curr_avg;
66              curr_avg <= (rL / samples)  + (rR / samples);
67          end
68          else
69              curr_avg <= curr_avg + (rL / samples)  + (rR / samples);
70      end
71
72
73
74      // shift registers w/ count, update frequency 0.2s
75      always_ff @(posedge clk) begin
76          if (rst) begin
77              amps <= '{default: '0};
78          end
79          else if (shift_enable) begin
80              amps[19] <= amps[18];
81              amps[18] <= amps[17];
82              amps[17] <= amps[16];
83              amps[16] <= amps[15];
84              amps[15] <= amps[14];
85              amps[14] <= amps[13];
86              amps[13] <= amps[12];
87              amps[12] <= amps[11];
88              amps[11] <= amps[10];
89              amps[10] <= amps[9];
90              amps[9] <= amps[8];
91              amps[8] <= amps[7];
92              amps[7] <= amps[6];
93              amps[6] <= amps[5];
94              amps[5] <= amps[4];
95              amps[4] <= amps[3];
96              amps[3] <= amps[2];
97              amps[2] <= amps[1];
98              amps[1] <= amps[0];
99              amps[0] <= avg;
100         end
101     end
102
103 endmodule
```

```
// Module to test the audio_averaging_shift module
module audio_averaging_shift_testbench();
   logic clk, rst;
   logic [23:0] rL, rR;
   logic [23:0] amps [0:19];

   // shifting once every 2 cycles
   audio_averaging_shift #(10000000) dut(.*);

   // Setting up a simulated clock.
   parameter CLOCK_PERIOD = 100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
   end

   initial begin
      rst <= 1; rL <= 0; rR <= 0;  repeat(5) @(posedge clk);
      rst <= 0; rL <= 50; rR <= 50;   @(posedge clk);
                rL <= 100; rR <= 100; @(posedge clk);
                rL <= 150; rR <= 150; @(posedge clk);
                rL <= 200; rR <= 200; @(posedge clk);
                rL <= 250; rR <= 250; @(posedge clk);
                rL <= 300; rR <= 300; @(posedge clk);

                rL <= 50; rR <= 50;   @(posedge clk);
                rL <= 100; rR <= 100; @(posedge clk);
                rL <= 150; rR <= 150; @(posedge clk);
                rL <= 200; rR <= 200; @(posedge clk);
                rL <= 250; rR <= 250; @(posedge clk);
                rL <= 300; rR <= 300; @(posedge clk);

                rL <= 50; rR <= 50;   @(posedge clk);
                rL <= 100; rR <= 100; @(posedge clk);
                rL <= 150; rR <= 150; @(posedge clk);
                rL <= 200; rR <= 200; @(posedge clk);
                rL <= 250; rR <= 250; @(posedge clk);
                rL <= 300; rR <= 300; @(posedge clk);

                rL <= 50; rR <= 50;   @(posedge clk);
                rL <= 100; rR <= 100; @(posedge clk);
                rL <= 150; rR <= 150; @(posedge clk);
                rL <= 200; rR <= 200; @(posedge clk);
                rL <= 250; rR <= 250; @(posedge clk);
                rL <= 300; rR <= 300; @(posedge clk);

                rL <= 50; rR <= 50;   @(posedge clk);
                rL <= 100; rR <= 100; @(posedge clk);
                rL <= 150; rR <= 150; @(posedge clk);
                rL <= 200; rR <= 200; @(posedge clk);
                rL <= 250; rR <= 250; @(posedge clk);
                rL <= 300; rR <= 300; @(posedge clk);

                rL <= 50; rR <= 50;   @(posedge clk);
                rL <= 100; rR <= 100; @(posedge clk);
                rL <= 150; rR <= 150; @(posedge clk);
                rL <= 200; rR <= 200; @(posedge clk);
                rL <= 250; rR <= 250; @(posedge clk);
                rL <= 300; rR <= 300; @(posedge clk);
      $stop;
   end
endmodule
```