

Dominic Myerchin

EE 271 Lab 8:

Market and Usability Analysis:

Ergonomics: The design is incredibly easy to use and understand. However, some of the display and audio elements could be extended. For instance, right now there are only 2 audio sources, one plays for each ear (Left ear gets bass drum, Right ear gets cymbal), also would be nice to display which beats have notes that play on them.

Suitability: The design is fun and simplistic, but very barebones. Unlike other drum sequencers there are no volume, tempo, or sound controls. However for the purpose of grooving out to a basic 8 bar rhythm, it works great.

Cost: The design is much more costly than our previous labs (The size calculated using the Resource Utilization page was ~522 ALUTs and ~382 Logic registers) However, more than %50 of the logic & ALUT size is compromised of the audio driver. The size of my code was 396 which is very low for an audio processing device. (Though it could definitely be lowered)

Analysis & Synthesis Resource Utilization by Entity								
<<Filter>>								
	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins	Full Hierarchy Name
1	▼ DE1_SoC	528 (1)	385 (0)	6144	0	75	0	DE1_SoC
1	> audio_drivers:setup	293 (3)	215 (3)	6144	0	0	0	DE1_SoC audio_drivers:setup
2	light_controller:comb_5	14 (14)	0 (0)	0	0	0	0	DE1_SoC light_controller:comb_5
3	note_player:play1	74 (74)	65 (65)	0	0	0	0	DE1_SoC note_player:play1
4	note_player:play2	40 (40)	1 (1)	0	0	0	0	DE1_SoC note_player:play2
5	note_register:render1	1 (1)	8 (8)	0	0	0	0	DE1_SoC note_register:render1
6	note_register:render2	1 (1)	8 (8)	0	0	0	0	DE1_SoC note_register:render2
7	saw_wav:sound2	63 (63)	55 (55)	0	0	0	0	DE1_SoC saw_wav:sound2
8	sqr_wav:sound1	41 (41)	33 (33)	0	0	0	0	DE1_SoC sqr_wav:sound1

Other considerations:

- Health & safety: The output volume levels have not been measured; prolonged use of the drum sequencer may result in hearing damage.
- Social: Headphone use recommended as to not drive the people around you insane

User Manual:

Capabilities: This 120-bpm drum sequencer has 2 programmable sounds that can be played at position in an 8-bar loop, as well as reset and mute functions. The drum sequencer will loop these 8 bars infinitely, displaying the current bar using LEDs 7-0 until the machine is shut off.

Program Notes: To program notes onto the sequencer, the user flips up the switches that correspond to each bar (For instance SW_0 = Bar_0, ..., SW_7 = Bar_7). Then to play the notes, the user can press KEY0 and it will play low bass notes on the switch position bars, or they can press KEY1 to play high notes on the switch positions. Once the button is pressed the SW positions are saved internally and will repeat forever until the sequencer is either reset, muted, mapped to new notes, or shut off.

Mute: To mute the sequencer the user can flip SW_8 up. Flip down to unmute.

Reset: To reset the machine the user must flip the switch up and then down

```

1 // Top-level module that defines the I/Os for the DE-1 SoC board
2
3 module DE1_SoC (CLOCK_50, FPGA_I2C_SCLK, FPGA_I2C_SDAT, AUD_XCK, AUD_DACLCK, AUD_ADCLCK, AUD_BCLK, AUD_ADCDAT, AUD_DACDAT, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5);
4     output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
5     output logic [9:0] LEDR;
6     input CLOCK_50;
7     // I2C Audio/Video config interface
8     output FPGA_I2C_SCLK;
9     inout FPGA_I2C_SDAT;
10    // Audio CODEC
11    output AUD_XCK;
12    input AUD_DACLCK, AUD_ADCLCK, AUD_BCLK;
13    input AUD_ADCDAT;
14    output AUD_DACDAT;
15    // Audio data
16    logic [23:0] dac_left, dac_right, sqr;
17    logic [23:0] adc_left, adc_right;
18    logic advance;
19    logic [7:0] notes1, notes2;
20    logic [23:0] wav1, wav2;
21    integer beat;
22
23    input logic [3:0] KEY;
24    input logic [9:0] SW;
25
26    audio_driver setup(CLOCK_50, .reset(SW[9]), .dac_left, .dac_right, .adc_left, .adc_right, .advance, .FPGA_I2C_SCLK, .FPGA_I2C_SDAT, .AUD_XCK, .AUD_DACLCK, .AUD_ADCLCK, .AUD_BCLK, .AUD_ADCDAT, .AUD_DACDAT);
27
28    sqr_wav sound1(CLOCK_50, .reset(SW[9]), .out(wav1));
29    saw_wav sound2(CLOCK_50, .reset(SW[9]), .out(wav2));
30
31    note_register render1(CLOCK_50, .reset(SW[9]), .load(~KEY[0]), .SW(SW[7:0]), .notes(notes1));
32    note_register render2(CLOCK_50, .reset(SW[9]), .load(~KEY[1]), .SW(SW[7:0]), .notes(notes2));
33
34    note_player play1(CLOCK_50, .reset(SW[9]), .SW(notes1), .mute(SW[8]), .aud_in(wav1), .aud_out(dac_left), .beat);
35    note_player play2(CLOCK_50, .reset(SW[9]), .SW(notes2), .mute(SW[8]), .aud_in(wav2), .aud_out(dac_right));
36
37    light_controller(.beat, .LEDR);
38
39    //notePlayer test(CLOCK_50, .reset(SW[9]), .SW[7,0], .out(dac_left))
40
41    //saw_wav sound2(CLOCK_50, .reset(SW[9]), .out(dac_left));
42
43    assign HEX0 = '1;
44    assign HEX1 = '1;
45    assign HEX2 = '1;
46    assign HEX3 = '1;
47    assign HEX4 = '1;
48    assign HEX5 = '1;
49 endmodule
50

```

```

42 module wolfson_testbench ();
43     reg clock;
44     logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
45     logic [9:0] LEDR;
46     // I2C Audio/Video config interface
47     wire FPGA_I2C_SCLK;
48     wire FPGA_I2C_SDAT;
49     // Audio data
50     logic [23:0] dac_left, dac_right, sqr;
51     logic [23:0] adc_left, adc_right;
52     logic advance;
53     logic [3:0] KEY;
54     logic [9:0] SW;
55     wire AUD_XCK, AUD_DACLCK, AUD_ADCLCK, AUD_BCLK, AUD_ADCDAT, AUD_DACDAT;
56     // Set up the clock.
57     parameter CLOCK_PERIOD=20;
58     initial clock<=1;
59     always begin
60         #(CLOCK_PERIOD/2);
61         clock <= ~clock;
62     end
63     wolfson chip (.AUD_XCK, .AUD_DACLCK, .AUD_ADCLCK, .AUD_BCLK, .AUD_ADCDAT, .AUD_DACDAT);
64     // leave I2C disconnected
65     DE1_SoC top (.CLOCK_50(clock), .FPGA_I2C_SCLK, .FPGA_I2C_SDAT, .AUD_XCK, .AUD_DACLCK, .AUD_ADCLCK, .AUD_BCLK, .AUD_ADCDAT, .AUD_DACDAT);
66     initial begin
67         // user test bench routine here
68         repeat(1) @(posedge clock);
69         KEY <= 4'b0000;
70         SW[8:0] <= 9'b000000000;
71         SW[9] <= 1; repeat(1) @(posedge clock); // Always reset FSMs at start
72         SW[9] <= 0; repeat(1) @(posedge clock);
73
74         KEY[0] <= 0; repeat(1) @(posedge clock); // Test case 1: Key 0 pressed
75         KEY[0] <= 1; repeat(1) @(posedge clock);
76
77         SW[1] <= 1; repeat(25000000) @(posedge clock); // Test case 2: SW 1 flipped
78
79         KEY[0] <= 0; repeat(1) @(posedge clock); // Test case 3: Key 0 pressed
80         KEY[0] <= 1; repeat(25000000) @(posedge clock);
81
82         KEY[1] <= 0; repeat(1) @(posedge clock); // Test case 4: Key 0 pressed
83         KEY[1] <= 1; repeat(25000000) @(posedge clock);
84
85         SW[2] <= 1; repeat(10) @(posedge clock); // Test case 2: SW 2 flipped
86         SW[4] <= 1; repeat(10) @(posedge clock); // Test case 2: SW 4 flipped
87         SW[6] <= 1; repeat(20) @(posedge clock); // Test case 2: SW 6 flipped
88         SW[8] <= 1; repeat(25000000) @(posedge clock); // Test case 2: SW 8 flipped
89
90         $stop; // End the simulation.
91     end
92 endmodule
93

```

```

1 module light_controller (beat, LEDR);
2   output logic [9:0] LEDR;
3   input integer beat;
4
5   integer counter;
6   // State variables
7   enum { low, high } ps, ns;
8
9   // Next State logic
10  always_comb begin
11    case (beat)
12      0: LEDR = 10'b0000000001;
13      1: LEDR = 10'b0000000010;
14      2: LEDR = 10'b00000000100;
15      3: LEDR = 10'b00000001000;
16      4: LEDR = 10'b00000010000;
17      5: LEDR = 10'b00000100000;
18      6: LEDR = 10'b00001000000;
19      7: LEDR = 10'b00010000000;
20      default: LEDR = 10'b0000000000; // or some other default value
21    endcase
22  end
23 endmodule
24
25 module light_controller_testbench();
26   logic [9:0] LEDR;
27   integer beat;
28
29   light_controller dut (.beat, .LEDR);
30
31   // Test the design.
32   integer i;
33   initial begin
34     #10;
35     for (i=0; i<8; i++) begin
36       beat <= i; #10;
37     end
38
39     beat <= 0; #30;
40
41     $stop; // End the simulation.
42   end
43 endmodule
44
45

```

```

1 module note_register (CLOCK_50, reset, load, SW, notes);
2   output logic [7:0] notes;
3   input logic CLOCK_50, reset, load;
4   input logic [7:0] SW;
5
6   // DFFs
7   always_ff @(posedge CLOCK_50) begin
8     if (reset)
9       notes <= 8'b00000000;
10    else if (load)
11      notes <= SW;
12  end
13 endmodule
14
15 module note_register_testbench();
16   logic CLOCK_50, reset, load;
17   logic [7:0] SW, notes;
18
19   note_register dut (.CLOCK_50, .reset, .load, .SW, .notes);
20
21   // Set up a simulated clock.
22   parameter CLOCK_PERIOD=100;
23   initial begin
24     CLOCK_50 <= 0;
25     forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
26   end
27
28   // Test the design.
29   integer i;
30   initial begin
31     @(posedge CLOCK_50);
32     load <= 0;
33     SW <= 8'b00000000;
34     reset <= 1; repeat(1) @(posedge CLOCK_50); // Always reset FSMs at start
35     reset <= 0; repeat(2) @(posedge CLOCK_50);
36
37     SW <= 8'b10010110; repeat(3) @(posedge CLOCK_50);
38     load <= 1; repeat(1) @(posedge CLOCK_50);
39     load <= 0; repeat(2) @(posedge CLOCK_50);
40
41     reset <= 1; repeat(1) @(posedge CLOCK_50); // Always reset FSMs at start
42     reset <= 0; repeat(2) @(posedge CLOCK_50);
43     $stop; // End the simulation.
44   end
45 endmodule

```

```

1 module note_player (CLOCK_50, reset, SW, mute, aud_in, aud_out, beat);
2   output logic [23:0] aud_out;
3   output integer beat;
4   input logic [23:0] aud_in;
5   input logic [7:0] SW;
6   input logic CLOCK_50, reset, mute;
7
8   integer counter;
9   // State variables
10  enum { off, on } ps, ns;
11
12  // Next State logic
13  always_comb begin
14    case (ps)
15      off: if (counter < 100 & SW[beat])
16            ns = on;
17          else ns = off;
18      on: if (counter >= 6250000)
19            ns = off;
20          else ns = on;
21    endcase
22  end
23
24  always_comb begin
25    if (mute)
26      aud_out = 24'b000000000000000000000000;
27    else begin
28      case (ps)
29        off: aud_out = 24'b000000000000000000000000;
30        on: aud_out = aud_in;
31      endcase
32    end
33  end
34
35  // DFFs
36  always_ff @(posedge CLOCK_50) begin
37    if (reset) begin
38      ps <= off;
39      counter <= 0;
40      beat <= 0;
41    end
42    else begin
43      if (counter >= 12500000) begin
44        counter <= 0;
45        if (beat == 7)
46          beat <= 0;
47        else
48          beat++;
49      end
50      else
51        counter++;
52      ps <= ns;

```

```

53   end
54 end
55 endmodule
56
57 module note_player_testbench();
58   logic [23:0] aud_out;
59   logic [23:0] aud_in;
60   logic [7:0] SW;
61   logic CLOCK_50, reset;
62
63   sqr_wav in (.CLOCK_50, .reset, .out(aud_in));
64
65   note_player dut (.CLOCK_50, .reset, .SW, .aud_in, .aud_out);
66
67   // Set up a simulated clock.
68   parameter CLOCK_PERIOD=100;
69   initial begin
70     CLOCK_50 <= 0;
71     forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
72   end
73
74   // Test the design.
75   integer i;
76   initial begin
77     @(posedge CLOCK_50);
78     SW <= 8'b00000000;
79     SW[0] <= 1;
80     reset <= 1; repeat(1) @(posedge CLOCK_50); // Always reset FSMs at start
81     reset <= 0; repeat(50000000) @(posedge CLOCK_50);
82
83     $stop; // End the simulation.
84   end
85 endmodule
86

```

```

1  module sqr_wav (CLOCK_50, reset, out);
2  output logic [23:0] out;
3  input logic CLOCK_50, reset;
4
5  integer counter;
6  // State variables
7  enum { low, high } ps, ns;
8
9  // Next State logic
10 always_comb begin
11     case (ps)
12         low: if (counter == 50000)
13             ns = high;
14             else ns = low;
15         high: if (counter == 50000)
16             ns = low;
17             else ns = high;
18     endcase
19 end
20
21 always_comb begin
22     case (ps)
23         low: out = 24'b000000000000000000000000;
24         high: out = 24'b0000000011110000000000000000;
25     endcase
26 end
27
28 // DFFs
29 always_ff @(posedge CLOCK_50) begin
30     if (reset) begin
31         ps <= low;
32         counter <= 0;
33     end
34     else begin
35         if (ns != ps) begin
36             counter <= 0;
37             ps <= ns;
38         end
39         else
40             counter++;
41     end
42 end
43 endmodule

```

```

45 module sqr_wav_testbench();
46     logic CLOCK_50, reset;
47     logic [23:0] out;
48
49     sqr_wav dut (.CLOCK_50, .reset, .out);
50
51     // Set up a simulated clock.
52     parameter CLOCK_PERIOD=100;
53     initial begin
54         CLOCK_50 <= 0;
55         forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
56     end
57
58     // Test the design.
59     integer i;
60     initial begin
61         @(posedge CLOCK_50);
62         reset <= 1; repeat(1) @(posedge CLOCK_50); // Always reset FSMs at start
63         reset <= 0; repeat(100000) @(posedge CLOCK_50);
64
65         $stop; // End the simulation.
66     end
67 endmodule
68
69

```

```

1  module saw_wav (CLOCK_50, reset, out);
2      output logic [23:0] out;
3      input logic CLOCK_50, reset;
4
5      integer counter;
6
7      // DFFs
8      always_ff @(posedge CLOCK_50) begin
9          if (reset) begin
10             counter <= 0;
11             out <= 24'b000000000000000000000000;
12         end
13         else begin
14             counter++;
15             if (counter == 5000) begin
16                 counter <= 0;
17                 out <= 24'b000000000000000000000000;
18             end
19             else
20                 out <= out + 70;
21         end
22     end
23 endmodule
24
25 module saw_wav_testbench();
26     logic CLOCK_50, reset;
27     logic [23:0] out;
28
29     saw_wav dut (.CLOCK_50, .reset, .out);
30
31     // Set up a simulated clock.
32     parameter CLOCK_PERIOD=100;
33     initial begin
34         CLOCK_50 <= 0;
35         forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
36     end
37
38     // Test the design.
39     integer i;
40     initial begin
41         @(posedge CLOCK_50);
42         reset <= 1; repeat(1) @(posedge CLOCK_50); // Always reset FSMs at start
43         reset <= 0; repeat(10000) @(posedge CLOCK_50);
44
45         $stop; // End the simulation.
46     end
47 endmodule
48

```