Dominic Myerchin

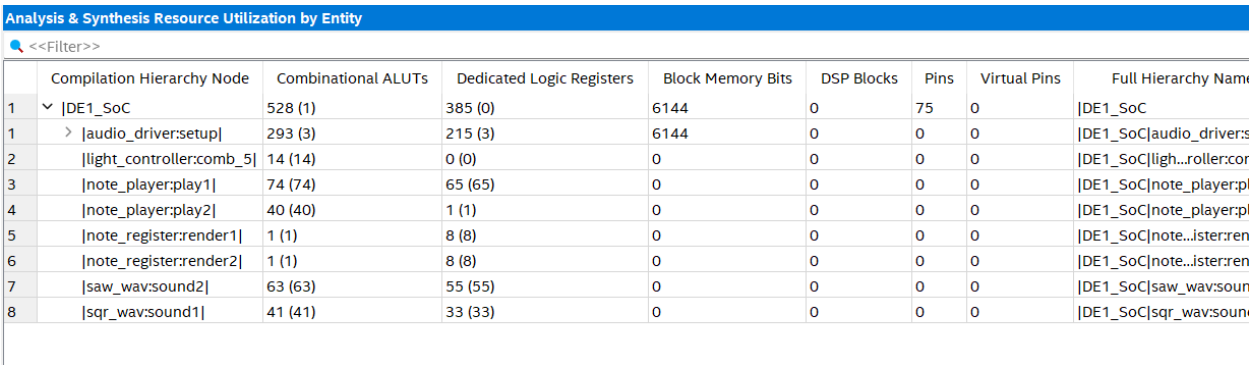
EE 271 Lab 8:

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 Nam
1	|DE1_SoC	528 (1)	385 (0)	6144	0	75	0	|DE1_SoC
1	> |audio_driver:setup|	293 (3)	215 (3)	6144	0	0	0	|DE1_SoC|audio_driver:s
2	|light_controller:comb_5|	14 (14)	0 (0)	0	0	0	0	|DE1_SoC|ligh...roller:cor
3	|note_player:play1|	74 (74)	65 (65)	0	0	0	0	|DE1_SoC|note_player:pl
4	|note_player:play2|	40 (40)	1 (1)	0	0	0	0	|DE1_SoC|note_player:pl
5	|note_register:render1|	1 (1)	8 (8)	0	0	0	0	|DE1_SoC|note...ister:ren
6	|note_register:render2|	1 (1)	8 (8)	0	0	0	0	|DE1_SoC|note...ister:ren
7	|saw_wav:sound2|	63 (63)	55 (55)	0	0	0	0	|DE1_SoC|saw_wav:soun
8	|sqr_wav:sound1|	41 (41)	33 (33)	0	0	0	0	|DE1_SoC|sqr_wav:soun

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

```
// Top-level module that defines the I/Os for the DE-1 SoC board

module DE1_SoC (CLOCK_50, FPGA_I2C_SCLK, FPGA_I2C_SDAT, AUD_XCK, AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK, AUD_ADCDAT, AUD_DACDAT, HEX0, HEX1, HEX2, HEX3, H
   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   output logic [9:0] LEDR;
   input CLOCK_50;
   // 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;
   // Audio data
   logic [23:0] dac_left, dac_right, sqr;
   logic [23:0] adc_left, adc_right;
   logic advance;
   logic [7:0] notes1, notes2;
   logic [23:0] wav1, wav2;
   integer beat;

   input logic [3:0] KEY;
   input logic [9:0] SW;

   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

   sqr_wav sound1(.CLOCK_50, .reset(SW[9]), .out(wav1));
   saw_wav sound2(.CLOCK_50, .reset(SW[9]), .out(wav2));

   note_register render1(.CLOCK_50, .reset(SW[9]), .load(~KEY[0]), .SW(SW[7:0]), .notes(notes1));
   note_register render2(.CLOCK_50, .reset(SW[9]), .load(~KEY[1]), .SW(SW[7:0]), .notes(notes2));

   note_player play1(.CLOCK_50, .reset(SW[9]), .SW(notes1), .mute(SW[8]), .aud_in(wav1), .aud_out(dac_left), .beat);
   note_player play2(.CLOCK_50, .reset(SW[9]), .SW(notes2), .mute(SW[8]), .aud_in(wav2), .aud_out(dac_right));

   light_controller(.beat, .LEDR);

   //notePlayer test(.CLOCK_50, .reset(SW[9]), .SW[7,0], .out(dac_left))

   //saw_wav sound2(.CLOCK_50, .reset(SW[9]), .out(dac_left));

   assign HEX0 = '1;
   assign HEX1 = '1;
   assign HEX2 = '1;
   assign HEX3 = '1;
   assign HEX4 = '1;
   assign HEX5 = '1;
endmodule
```

```
module wolfson_testbench ();
   reg clock;
   logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   logic [9:0] LEDR;
   // I2C Audio/Video config interface
   wire FPGA_I2C_SCLK;
   wire FPGA_I2C_SDAT;
   // Audio data
   logic [23:0] dac_left, dac_right, sqr;
   logic [23:0] adc_left, adc_right;
   logic advance;
   logic [3:0] KEY;
   logic [9:0] SW;
   wire AUD_XCK, AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK, AUD_ADCDAT, AUD_DACDAT;
   // Set up the clock.
   parameter CLOCK_PERIOD=20;
   initial clock<=1;
   always begin
      #(CLOCK_PERIOD/2);
      clock <= ~clock;
   end
   wolfson chip (.AUD_XCK, .AUD_DACLRCK, .AUD_ADCLRCK, .AUD_BCLK, .AUD_ADCDAT, .AUD_DACDAT);
   // leave I2C disconnected
   DE1_SoC top (.CLOCK_50(clock), .FPGA_I2C_SCLK, .FPGA_I2C_SDAT, .AUD_XCK, .AUD_DACLRCK, .AUD_ADCLRCK, .AUD_BCLK, .AUD_ADCDAT, .
   initial begin
      // user test bench routine here
                  repeat(1) @(posedge clock);
      KEY <= 4'b0000;
      SW[8:0] <= 9'b000000000;
      SW[9] <= 1; repeat(1) @(posedge clock); // Always reset FSMs at start
      SW[9] <= 0; repeat(1) @(posedge clock);

      KEY[0] <= 0;repeat(1) @(posedge clock); // Test case 1: Key 0 pressed
      KEY[0] <= 1;repeat(1) @(posedge clock);

      SW[1] <= 1; repeat(25000000) @(posedge clock); // Test case 2: SW 1 flipped

      KEY[0] <= 0;repeat(1) @(posedge clock); // Test case 3: Key 0 pressed
      KEY[0] <= 1;repeat(25000000) @(posedge clock);

      KEY[1] <= 0;repeat(1) @(posedge clock); // Test case 4: Key 0 pressed
      KEY[1] <= 1;repeat(25000000) @(posedge clock);

      SW[2] <= 1; repeat(10) @(posedge clock); // Test case 2: SW 2 flipped
      SW[4] <= 1; repeat(10) @(posedge clock); // Test case 2: SW 4 flipped
      SW[6] <= 1; repeat(20) @(posedge clock); // Test case 2: SW 6 flipped
      SW[8] <= 1; repeat(25000000) @(posedge clock); // Test case 2: SW 8 flipped

      $stop; // End the simulation.
   end
endmodule
```

```
module light_controller (beat, LEDR);
   output logic [9:0] LEDR;
   input integer beat;

   integer counter;
   // State variables
   enum { low, high } ps, ns;

   // Next State logic
   always_comb begin
   case (beat)
      0: LEDR = 10'b0000000001;
      1: LEDR = 10'b0000000010;
      2: LEDR = 10'b0000000100;
      3: LEDR = 10'b0000001000;
      4: LEDR = 10'b0000010000;
      5: LEDR = 10'b0000100000;
      6: LEDR = 10'b0001000000;
      7: LEDR = 10'b0010000000;
      default: LEDR = 10'b0000000000; // or some other default value
   endcase
   end
endmodule


module light_controller_testbench();
   logic [9:0] LEDR;
   integer beat;

   light_controller dut (.beat, .LEDR);

   // Test the design.
   integer i;
   initial begin
      #10;
      for (i=0;i<8;i++) begin
         beat <= i; #10;
      end

      beat<= 0; #30;

      $stop; // End the simulation.
   end
endmodule
```

```
module note_register (CLOCK_50, reset, load, SW, notes);
   output logic [7:0] notes;
   input logic CLOCK_50, reset, load;
   input logic [7:0] SW;

   // DFFs
   always_ff @(posedge CLOCK_50) begin
      if (reset)
         notes <= 8'b00000000;
      else if (load)
         notes <= SW;
   end
endmodule

module note_register_testbench();
   logic CLOCK_50, reset, load;
   logic [7:0] SW, notes;

   note_register dut (.CLOCK_50, .reset, .load, .SW, .notes);

   // Set up a simulated clock.
   parameter CLOCK_PERIOD=100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   // Test the design.
   integer i;
   initial begin
      @(posedge CLOCK_50);
      load <= 0;
      SW <= 8'b00000000;
      reset <= 1; repeat(1) @(posedge CLOCK_50); // Always reset FSMs at start
      reset <= 0; repeat(2) @(posedge CLOCK_50);

      SW <= 8'b10010110;   repeat(3) @(posedge CLOCK_50);
      load <= 1;           repeat(1) @(posedge CLOCK_50);
      load <= 0;           repeat(2) @(posedge CLOCK_50);

      reset <= 1; repeat(1) @(posedge CLOCK_50); // Always reset FSMs at start
      reset <= 0; repeat(2) @(posedge CLOCK_50);
      $stop; // End the simulation.
   end
endmodule
```

```
module note_player (CLOCK_50, reset, SW, mute, aud_in, aud_out, beat);
   output logic [23:0] aud_out;
   output integer beat;
   input logic [23:0] aud_in;
   input logic [7:0] SW;
   input logic CLOCK_50, reset, mute;

   integer counter;
   // State variables
   enum { off, on } ps, ns;

   // Next State logic
   always_comb begin
   case (ps)
      off: if (counter < 100 & SW[beat])
              ns = on;
           else ns = off;
      on: if (counter >= 6250000)
             ns = off;
          else ns = on;
   endcase
   end

   always_comb begin
   if (mute)
      aud_out = 24'b000000000000000000000000;
   else begin
      case (ps)
         off:  aud_out = 24'b000000000000000000000000;
         on: aud_out = aud_in;
      endcase
   end
   end

   // DFFs
   always_ff @(posedge CLOCK_50) begin
      if (reset) begin
         ps <= off;
         counter <= 0;
         beat <= 0;
      end
      else begin
         if (counter >= 12500000) begin
            counter <= 0;
            if (beat == 7)
               beat <= 0;
            else
               beat++;
         end
         else
            counter++;
         ps <= ns;
```

```
         end
   end
endmodule

module note_player_testbench();
   logic [23:0] aud_out;
   logic [23:0] aud_in;
   logic [7:0] SW;
   logic CLOCK_50, reset;

   sqr_wav in (.CLOCK_50, .reset, .out(aud_in));

   note_player dut (.CLOCK_50, .reset, .SW, .aud_in, .aud_out);

   // Set up a simulated clock.
   parameter CLOCK_PERIOD=100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   // Test the design.
   integer i;
   initial begin
      @(posedge CLOCK_50);
      SW <= 8'b00000000;
      SW[0] <= 1;
      reset <= 1; repeat(1) @(posedge CLOCK_50); // Always reset FSMs at start
      reset <= 0; repeat(50000000) @(posedge CLOCK_50);

      $stop; // End the simulation.
   end
endmodule
```

```
module sqr_wav (CLOCK_50, reset, out);
   output logic [23:0] out;
   input logic CLOCK_50, reset;

   integer counter;
   // State variables
   enum { low, high } ps, ns;

   // Next State logic
   always_comb begin
   case (ps)
      low: if (counter == 50000)
            ns = high;
         else ns = low;
     high: if (counter == 50000)
            ns = low;
         else ns = high;
   endcase
   end

   always_comb begin
   case (ps)
      low:  out = 24'b000000000000000000000000;
      high: out = 24'b000000011100000000000000;
   endcase
   end

   // DFFs
   always_ff @(posedge CLOCK_50) begin
      if (reset) begin
         ps <= low;
         counter <= 0;
      end
      else begin
         if (ns != ps) begin
            counter <= 0;
            ps <= ns;
         end
         else
            counter++;
      end
   end
endmodule
```

```
module sqr_wav_testbench();
   logic CLOCK_50, reset;
   logic [23:0] out;

   sqr_wav dut (.CLOCK_50, .reset, .out);

   // Set up a simulated clock.
   parameter CLOCK_PERIOD=100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   // Test the design.
   integer i;
   initial begin
      @(posedge CLOCK_50);
      reset <= 1; repeat(1) @(posedge CLOCK_50); // Always reset FSMs at start
      reset <= 0; repeat(100000) @(posedge CLOCK_50);


      $stop; // End the simulation.
   end
endmodule
```

```
module saw_wav (CLOCK_50, reset, out);
   output logic [23:0] out;
   input logic CLOCK_50, reset;

   integer counter;

   // DFFs
   always_ff @(posedge CLOCK_50) begin
      if (reset) begin
         counter <= 0;
         out <= 24'b000000000000000000000000;
      end
      else begin
      counter++;
         if (counter == 5000) begin
            counter <= 0;
            out <= 24'b000000000000000000000000;
         end
         else
            out <= out + 70;
      end
   end
endmodule
```

```
module saw_wav_testbench();
   logic CLOCK_50, reset;
   logic [23:0] out;

   saw_wav dut (.CLOCK_50, .reset, .out);

   // Set up a simulated clock.
   parameter CLOCK_PERIOD=100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   // Test the design.
   integer i;
   initial begin
      @(posedge CLOCK_50);
      reset <= 1; repeat(1) @(posedge CLOCK_50); // Always reset FSMs at start
      reset <= 0; repeat(10000) @(posedge CLOCK_50);

      $stop; // End the simulation.
   end
endmodule
```