Vincent Kwok & Hunter Hatzenbeler
EE/CSE 371
April 24, 2023
Lab 3 Report

Design Procedure:

Lab 3 centers around using the audio CODEC from an FPGA to generate and filter noise from an externally downloaded file and an internal memory module. The tasks in this lab show how to take data and use it in a way that creates an output that is not just a light or LED. The tasks in this lab reference back to past labs or lecture concepts while adding the complexity of incorporating audio as a data input and output. Task 2 uses the concept of switching between modules that store memory and using the IP catalog to create memory, both were used in lab 2. Task 3 required the application of FIFO buffers, which were touched upon in past lectures.

Task 1:

The task involves playing a music file using the DE1-SoC board. The first step required modifying the provided starter kit circuit to allow data to pass from the audio input to the audio output of the device only when the CODEC was ready. Almost all of the required code was provided in the lab specification except for the assign statements shown in figure 1.

```
25  /***** Task 1 ********/
26
27      assign writedata_left = readdata_left & write_ready;
28      assign writedata_right = readdata_right & write_ready;
29
30      assign read = read_ready & write_ready;
31      assign write = read_ready & write_ready;
32
```

Figure 1: Assign Statements for task 1

Task 2:

In task 2, we generated a MIF file using a Python script given by the lab. The script generated a constant tone that could be changed by frequency (note), duration, and volume. The FPGA and ROMs that can be made through the IP catalog of Quartus can use a mif file to be read out and saved in memory. Using the noisypiano noise, we generated the equivalent mif file and used it is the initial storage of an initializable ROM created by the IP catalog. With this ROM module, we were able to instantiate it into the task 2 module. We made the task 2 module output its mif file data in order using a counter, looping it back to the start when all its data was used by the speaker. Finally, the Task 1 and 2 modules were combined together. The testbench was designed to alternate between a constant tone and the noisy piano audio.

```
module task2 (CLOCK_50, reset, write, dout);
   input logic CLOCK_50, reset, write;
   output logic [23:0] dout;

   logic [15:0] addr;

   // initializes ram that stores the note, data and wren are not needed since its read only
   ram ramInit (.address(addr), .clock(CLOCK_50), .data(1'd0), .wren(1'b0), .q(dout));

   // hits block every time write is given
   always @(posedge write) begin
      if (reset)
         addr = 1'd0;
      else if (addr == 16'b1011101101111111) // resets address when it hits the end
         addr = 1'd0;
      else
         addr = addr + 1'b1; // increments address every write
   end

endmodule
```

Additional code for task 2

Task 3:

In task3, we are tasked to create a finite impulse response averaging filter, though using the concept of a FIFO buffer as well as an accumulator, it is possible to create an averaging filter that takes samples equal to the length of the buffer. Using this buffer, we were able to reduce the noise in the noisy piano output by smoothing the waveform output by taking noisy_piano.mp3 as input.

```

module task3 #(parameter N = 3)(CLOCK_50, reset, read_ready, write_ready, din_left, din_right, dout_left, dout_right);
   input logic signed [23:0] din_left, din_right;
   input logic CLOCK_50, reset, read_ready, write_ready;
   output logic [23:0] dout_left, dout_right;

   // creates 24 bit signed variables to hold data from different stages
   logic signed [23:0] divided_left, divided_right, accumulated_left, accumulated_right, fifo_left, fifo_right;
   logic empty, full, rd, wr;

   // logic to account for first N-1 samples sent into filter
   logic [N:0] counter;
   logic first_N = 1'b1;

   // combination block to handle dividing the input, as well as read and write signals for the buffer
   always_comb begin
      divided_left = din_left >>> N;
      divided_right = din_right >>> N;

      if(first_N)
         rd = 1'b0; // doesnt read until buffer is full
      else
         rd = read_ready & write_ready; // reads from buffer when read and write are ready

      wr = read_ready & write_ready; // writes from buffer when read and write are ready
   end

   // creates left and right fifo buffers
   fifo #(.DATA_WIDTH(24), .ADDR_WIDTH(N)) leftBuffer (.clk(CLOCK_50), .reset(reset), .rd(rd),
                     .wr(wr), .empty(empty), .full(full), .w_data(divided_left), .r_data(fifo_left));
   fifo #(.DATA_WIDTH(24), .ADDR_WIDTH(N)) rightBuffer (.clk(CLOCK_50), .reset(reset), .rd(rd),
                     .wr(wr), .empty(empty), .full(full), .w_data(divided_right), .r_data(fifo_right));


```

```
46
47    always_ff @(posedge CLOCK_50) begin
48       if(reset) begin // on reset clears accumulator and first_n counter
49          first_N = 1'b1;
50          counter = 1'd0;
51
52          accumulated_left <= 1'd0;
53          accumulated_right <= 1'd0;
54
55          dout_left <= 1'd0;
56          dout_right <= 1'd0;
57
58       end else if (read_ready & write_ready) begin // only updates dout and accumulator when read and write occurs
59          if(~first_N) begin
60             counter = counter + 1'b1;
61             if(counter == N-1)
62                first_N = 1'b0;
63          end
64
65          dout_left <= accumulated_left + divided_left - fifo_left;
66          dout_right <= accumulated_right + divided_right - fifo_right;
67
68          accumulated_left <= accumulated_left + dout_left;
69          accumulated_right <= accumulated_right + dout_right;
70
71       end
72
73    end
74
75 endmodule
76
```

Figure 2: Implementation of the FIFO buffer

Overall System:

The block diagram given by the lab for task 1 is provided in Figure 3. A lot of control signals and status signals between the CODEC interface and the circuit interface are interconnected to each other. The read depends on the read_ready output while the write depends on the write_ready output. Readdata and writedata signals are only valid if read_ready is asserted and when write_ready is asserted respectively.

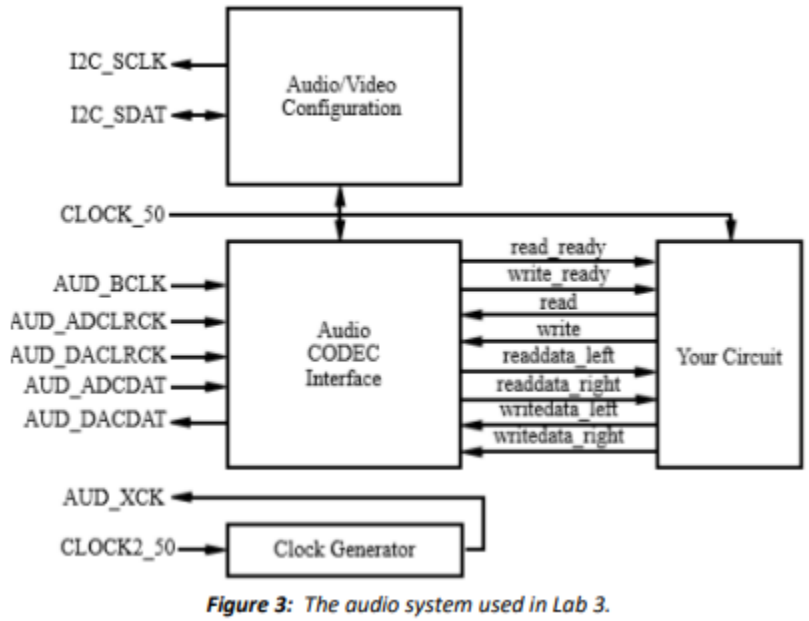


Figure 3: *The audio system used in Lab 3.*

Figure 3: Task 1 Audio system

Results

Task 1:

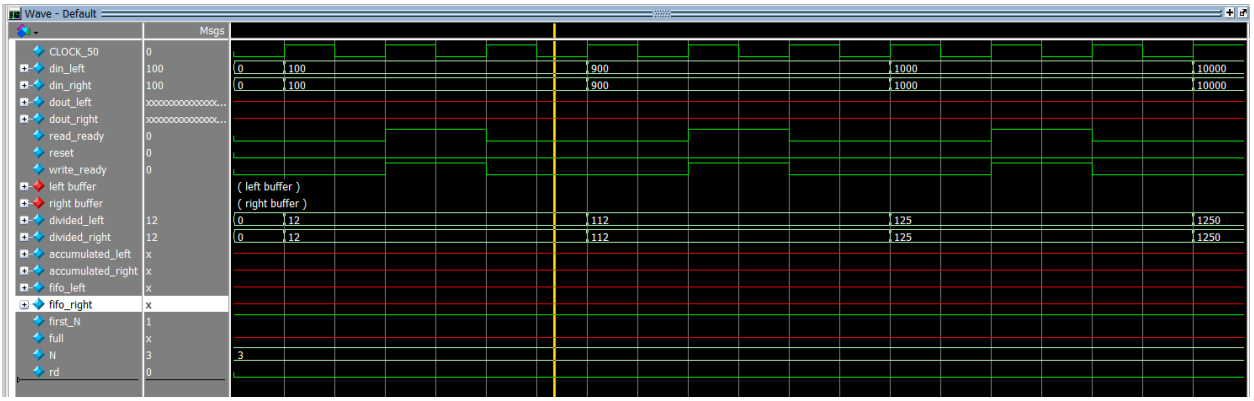
For task 1, the results were fairly straightforward. Since there were only 4 lines of code to update, it didn't take us that long to figure out what we had to do to get it working. Once we changed the values of the 4 lines, we added all of the given codec files, along with the file for part 1 to lablands. We figured it would be easier to test it out this way rather than simulate an entire mp3 file in modelsim. Once we synthesized and ran it in lablands, it all worked after a couple tries of fine tuning the exact values for the read and write signals. For this part, we also sent a screenshot of our work and got it checked off by the TA's to make sure we had it correct.

Task 2:

The second task didn't take us as long either. We already had python installed on the computer, and running the script didn't take too long to get the note. Additionally, since we already had made a ram before in the previous lab, it didn't take us very long to get the ram working. We initialized the ram in a separate task 2 module, and made a simple counter to run through each address. This didn't take us long, however it did take us a little bit to modify the 2:1 mux to work for 24 bit values. Once we did this all however, we got the code to run in both simulation in lablands. Using SW[9], we were able to toggle between the noisy piano and our generated note. Below is the simulation we ran, which runs through the different addresses of the created ram showing the different dout values.

```
35  module task2_testbench();
36     logic CLOCK_50, reset, write;
37     logic [23:0] dout;
38
39     task2 dut (.*);
40
41     // Set up a simulated clock.
42     parameter CLOCK_PERIOD=100;
43
44     initial begin
45        CLOCK_50 <= 0;
46        forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
47     end
48
49
50     initial begin
51
52     CLOCK_50 = 1'b0; reset = 1'b0; write = 1'b0; @(posedge CLOCK_50);
53     write = 1'b0; @(posedge CLOCK_50);
54     write = 1'b1; @(posedge CLOCK_50);
55     write = 1'b0; @(posedge CLOCK_50);
56     write = 1'b1; @(posedge CLOCK_50);
57     write = 1'b0; @(posedge CLOCK_50);
58     write = 1'b1; @(posedge CLOCK_50);
59     write = 1'b0; @(posedge CLOCK_50);
60     write = 1'b1; @(posedge CLOCK_50);
61     write = 1'b0; @(posedge CLOCK_50);
62     write = 1'b1; @(posedge CLOCK_50);
63     write = 1'b0; @(posedge CLOCK_50);
```

Task 3:
The third task is the one that gave us the most trouble. In theory, it was easy to piece together that we needed two fifos, for left and right, using the files given to us in HW3. To do the accumulator, it also didn't take us very long to figure out that we needed to create a single register that would increment on itself each time. However, what was hard for us was figuring out the different control signals needed to run the fifo, as well as what to do for the edge cases when the buffer wasn't filled yet. In the end, the filter was still bugged for us. We were able to get the unfiltered values through, but when we switched on the filter it was not working. Below is a screenshot of the simulation. As you can see, the control signals for the buffer are working, the input data is being properly divided, but for some reason the fifo is not being filled.



Flow Summary:

Flow Status: Successful - Fri Apr 28 23:28:42 2023 Quartus Prime Version: 17.0.0 Build 595 04/25/2017 SJ Lite Edition Revision Name: part1 Top-level Entity Name: part1 Family: Cyclone V Device: 5CSEMA5F31C6 Timing Models: Final Logic utilization (in ALMs): N/A Total registers: 270 Total pins: 21 Total virtual pins: 0 Total block memory bits: 6,400 Total DSP Blocks: 0 Total HSSI RX PCSs: 0 Total HSSI PMA RX Deserializers: 0 Total HSSI TX PCSs: 0 Total HSSI PMA TX Serializers: 0 Total PLLs: 1 Total DLLs: 0	Flow Status: Successful - Fri Apr 28 23:27:03 2023 Quartus Prime Version: 17.0.0 Build 595 04/25/2017 SJ Lite Edition Revision Name: part1 Top-level Entity Name: part1 Family: Cyclone V Device: 5CSEMA5F31C6 Timing Models: Final Logic utilization (in ALMs): N/A Total registers: 315 Total pins: 21 Total virtual pins: 0 Total block memory bits: 1,164,288 Total DSP Blocks: 0 Total HSSI RX PCSs: 0 Total HSSI PMA RX Deserializers: 0 Total HSSI TX PCSs: 0 Total HSSI PMA TX Serializers: 0 Total PLLs: 1 Total DLLs: 0	Flow Status: Successful - Fri Apr 28 23:25:58 2023 Quartus Prime Version: 17.0.0 Build 595 04/25/2017 SJ Lite Edition Revision Name: part1 Top-level Entity Name: part1 Family: Cyclone V Device: 5CSEMA5F31C6 Timing Models: Final Logic utilization (in ALMs): Total registers: 315 Total pins: 21 Total virtual pins: 0 Total block memory bits: 1,164,288 Total DSP Blocks: 0 Total HSSI RX PCSs: 0 Total HSSI PMA RX Deserializers: 0 Total HSSI TX PCSs: 0 Total HSSI PMA TX Serializers: 0 Total PLLs: 1 Total DLLs: 0
The ModelSim Flow Summary of the task 2 module compilation	The ModelSim Flow Summary of the task 2 module compilation	The ModelSim Flow Summary of the task 2 module compilation

Experience Report:

We found this lab to be quite confusing at times due to our lack of coverage and examples in lectures of audio related System Verilog programming. In task 1, we found trouble correctly defining the statements to ensure that the system would function in all circumstances due to us not really understanding why our assign statements may have failed to function as expected in some cases when we were starting off the lab. We also found the instruction in task2 a bit confusing as there were no directions given on where to continue programming the task, causing us to write a sequential memory block in a Verilog file which failed to compile. This problem took a while for us to realize. We believe that it was not only our group that found the lab not being very thorough in instruction and more challenging compared to the last. Still it was an interesting lab experience.

This lab took approximately 8 hours, broken down as follows:

- Reading - 30 minutes
- Planning - 40 minutes
- Design- 80 minutes
- Coding- 150 minutes
- Testing- 60 minutes
- Debugging- 120 minutes