Jiovany Soliman, 1951338

Chloe’ Miller, 2031648

EE/CSE 371 Lab 3 Report: Digital Signal Processing

October 27, 2023

Design Procedure

The purpose of this lab was to interface with the DE1_SoC Audio CODEC to record and playback audio from an external source and an internal memory and apply an averaging FIR filter to minimize noise. To achieve this, we first modified the starter kit circuit to pass CODEC audio input to CODEC audio output only when the CODEC was ready. Then, we generated a static tone from memory by storing samples from a sound waveform into ROM that could be read back sequentially to re-produce the sound. Finally, we developed an averaging FIR filter using a FIFO buffer and accumulator that could be applied to either sound sources with user inputs from the onboard switches. Figure 1 shows the top-level block diagram of our system.

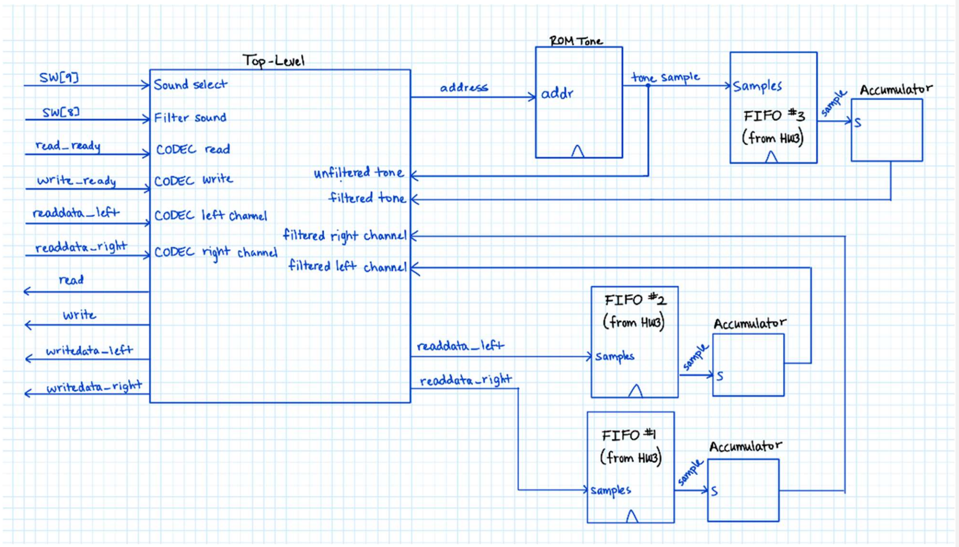
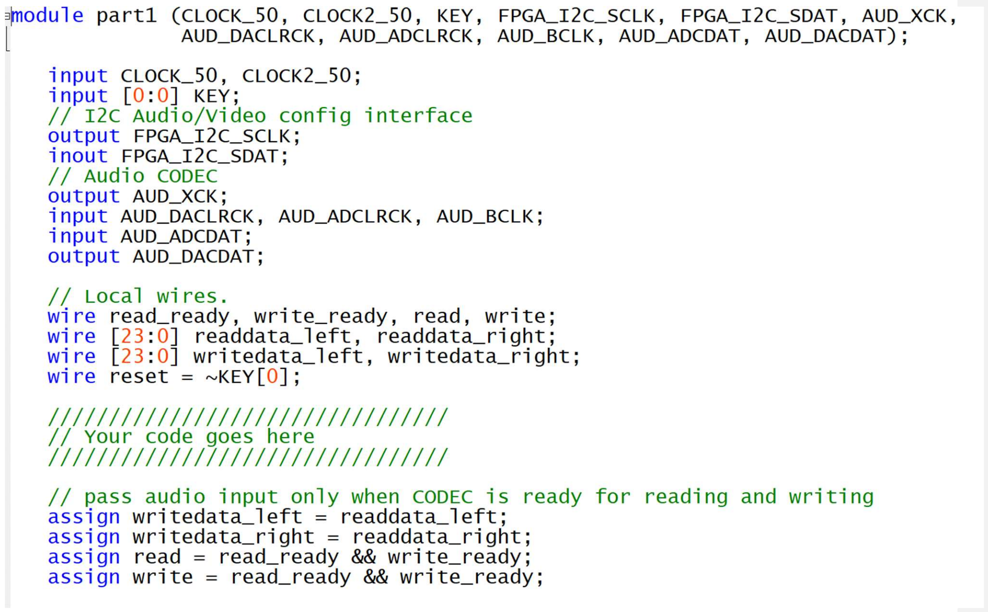


Figure 1:Top-level block diagram of our system, which interacts with the Audio CODEC and filters audio data.

Task #1:

Task 1 involved setting up the basic code that we needed for passing data with the CODEC (Figure 2). To pass data, both the audio input and audio output status signals (read_ready and write_ready) must be asserted. Only then could we assert our circuit's read and write signals and pass data to the left and right channels for the speaker (writedata_left and writedata_right, respectively). We then verified with the TA that our code is correct and accurate.



```
module part1 (CLOCK_50, CLOCK2_50, KEY, FPGA_I2C_SCLK, FPGA_I2C_SDAT, AUD_XCK,
              AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK, AUD_ADCDAT, AUD_DACDAT);

   input CLOCK_50, CLOCK2_50;
   input [0:0] KEY;
   // 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;

   // Local wires.
   wire read_ready, write_ready, read, write;
   wire [23:0] readdata_left, readdata_right;
   wire [23:0] writedata_left, writedata_right;
   wire reset = ~KEY[0];

   /////////////////////////////////
   // Your code goes here
   /////////////////////////////////

   // pass audio input only when CODEC is ready for reading and writing
   assign writedata_left = readdata_left;
   assign writedata_right = readdata_right;
   assign read = read_ready && write_ready;
   assign write = read_ready && write_ready;
```

Figure 2: A snippet of code from Task 1 that sets up data-passing logic.

Task #2:

For Task 2, we needed to initialize a ROM block with samples of a static tone generated from a Python script. We chose to generate a tone that contained about 48,000 samples, each 24 bits. We reused all the code from task 1 and added an always_ff block for an address variable to cycle through each sample of our generated tone and passed the address to the ROM block to be read to the CODEC, as seen in Figure 3.

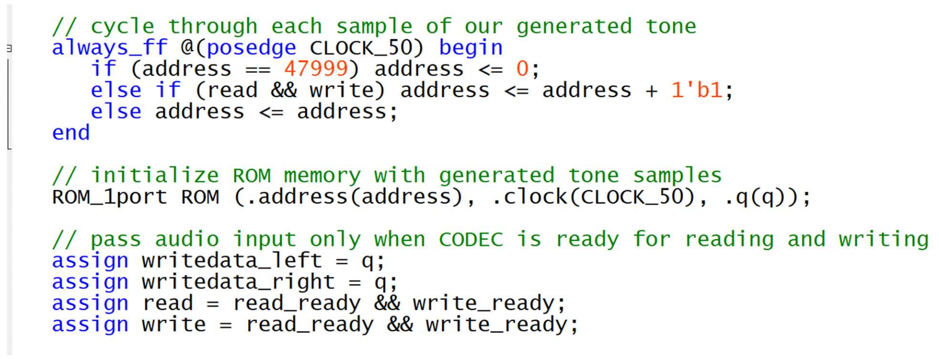
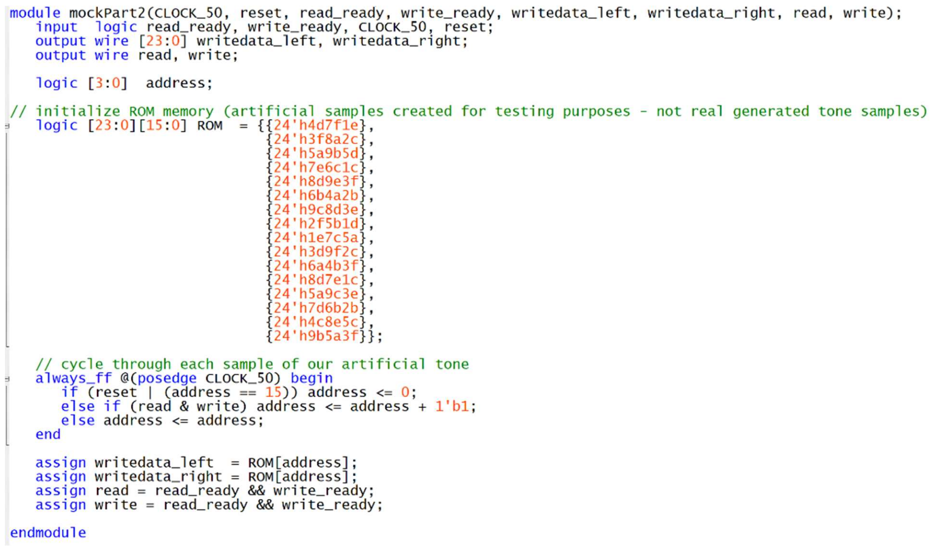

```
// cycle through each sample of our generated tone
always_ff @(posedge CLOCK_50) begin
   if (address == 47999) address <= 0;
   else if (read && write) address <= address + 1'b1;
   else address <= address;
end

// initialize ROM memory with generated tone samples
ROM_1port ROM (.address(address), .clock(CLOCK_50), .q(q));

// pass audio input only when CODEC is ready for reading and writing
assign writedata_left = q;
assign writedata_right = q;
assign read = read_ready && write_ready;
assign write = read_ready && write_ready;
```

Figure 3: A snippet of code from Task 2 that initializes a static tone stored in ROM and cycles through each audio sample to be read.

To verify that this this code did not pass data unless the read_ready and write_ready signals were asserted from the CODEC, we needed to create a new, simplified version of the code that did not contain most of the Audio CODEC signals, as those are difficult to replicate (Figure 4). Our replicated module thus only consisted of input ports for clock and reset to control and initialize the address variable and read_ready and write_ready signals to simulate CODEC inputs. Output ports consisted of writedata_left and writedata_right, read, and write signals that go into the CODEC. For more simplicity, we initialized a much smaller ROM block with 16 samples to cycle through, where each sample is sent to both the left and right channels.



```
module mockPart2(CLOCK_50, reset, read_ready, write_ready, writedata_left, writedata_right, read, write);
   input  logic read_ready, write_ready, CLOCK_50, reset;
   output wire [23:0] writedata_left, writedata_right;
   output wire read, write;

   logic [3:0]  address;

// initialize ROM memory (artificial samples created for testing purposes - not real generated tone samples)
   logic [23:0][15:0] ROM  = {{24'h4d7f1e},
                              {24'h3f8a2c},
                              {24'h5a9b5d},
                              {24'h7e6c1c},
                              {24'h8d9e3f},
                              {24'h6b4a2b},
                              {24'h9c8d3e},
                              {24'h2f5b1d},
                              {24'h1e7c5a},
                              {24'h3d9f2c},
                              {24'h6a4b3f},
                              {24'h8d7e1c},
                              {24'h5a9c3e},
                              {24'h7d6b2b},
                              {24'h4c8e5c},
                              {24'h9b5a3f}};

   // cycle through each sample of our artificial tone
   always_ff @(posedge CLOCK_50) begin
      if (reset | (address == 15)) address <= 0;
      else if (read & write) address <= address + 1'b1;
      else address <= address;
   end

   assign writedata_left  = ROM[address];
   assign writedata_right = ROM[address];
   assign read = read_ready && write_ready;
   assign write = read_ready && write_ready;

endmodule
```

Figure 4: A snippet of code showing the simplified Task 2 code for testing purposes.

In the simulation below (Figure 5), we cycle the read_ready and write_ready signals through all possible combinations. We can see that the address variable does not increment unless the read and write signals are asserted (one cycle later, after 112 ps in the waveform), thus not updating the writedata_left and writdedata_right signals with data to pass to the CODEC. Note that the last 6-digit hex value in the ROM waveform is the value at address 0, with the next hex value to the left is at address 1, and so on. When the read_ready and write_ready signals are de-asserted once more, the address variable stops updating and data is no longer passed.

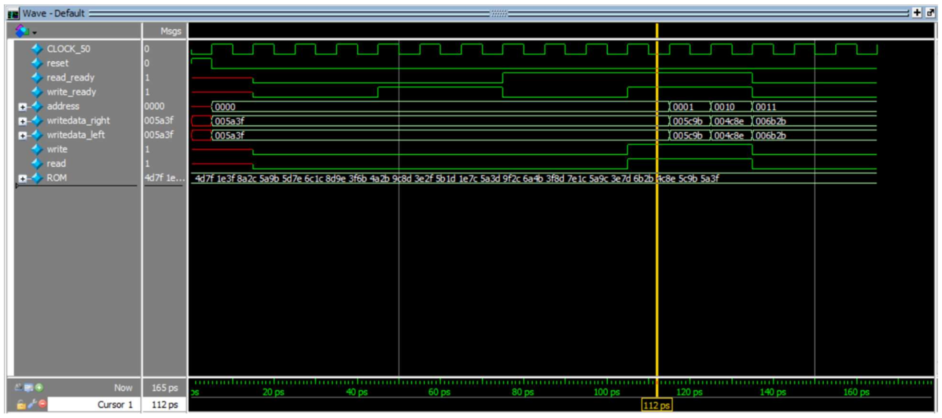
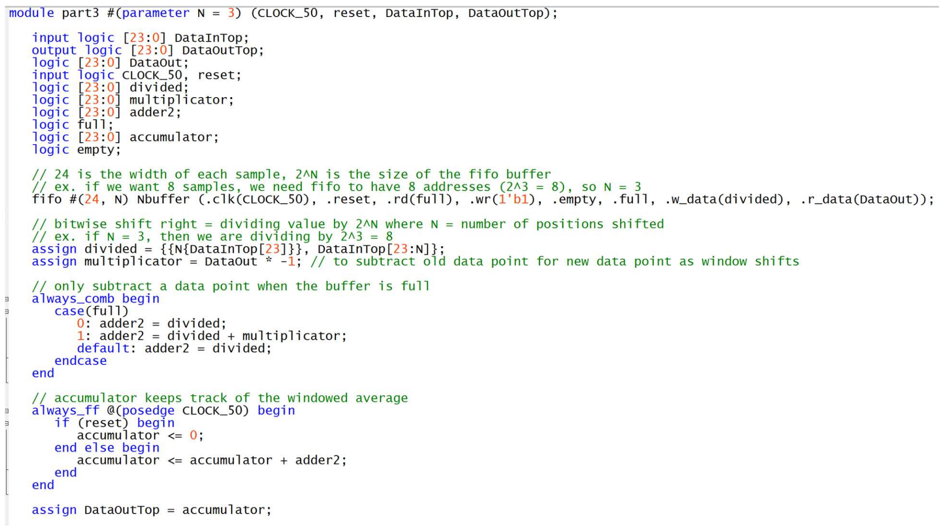


Figure 5: Task 2 waveform to verify data-passing logic.

Task #3:

For Task 3, we needed to implement an averaging FIR filter by using a FIFO buffer and accumulator instead of a series of D-Flip Flops (Figure 6). We accomplished this by first dividing the value of the input signal sample by the number of samples we wished to average, then writing the divided data to the buffer and to an adder unit. Once the buffer becomes full (its size the same as the number of samples we are averaging), the output of the buffer gets multiplied by -1, which is equivalent to subtracting the samples already accumulated in the buffer to make room for new samples as the averaging window “moves”. For the last few samples that are, in total, smaller than the windowing size, we simply read them from the buffer and subtract them from the accumulator. The result of this implementation smooths out signals by dismissing any fast-changing signals, a characteristic of noise signals. The larger the value of the FIFO buffer (2^N), the more filtered the signal will be, which will sound more muted because of the reduced amplitudes.



```
module part3 #(parameter N = 3) (CLOCK_50, reset, DataInTop, DataOutTop);

   input logic [23:0] DataInTop;
   output logic [23:0] DataOutTop;
   logic [23:0] DataOut;
   input logic CLOCK_50, reset;
   logic [23:0] divided;
   logic [23:0] multiplicator;
   logic [23:0] adder2;
   logic full;
   logic [23:0] accumulator;
   logic empty;

   // 24 is the width of each sample, 2^N is the size of the fifo buffer
   // ex. if we want 8 samples, we need fifo to have 8 addresses (2^3 = 8), so N = 3
   fifo #(24, N) Nbuffer (.clk(CLOCK_50), .reset, .rd(full), .wr(1'b1), .empty, .full, .w_data(divided), .r_data(DataOut));

   // bitwise shift right = dividing value by 2^N where N = number of positions shifted
   // ex. if N = 3, then we are dividing by 2^3 = 8
   assign divided = {{N{DataInTop[23]}}, DataInTop[23:N]};
   assign multiplicator = DataOut * -1; // to subtract old data point for new data point as window shifts

   // only subtract a data point when the buffer is full
   always_comb begin
      case(full)
         0: adder2 = divided;
         1: adder2 = divided + multiplicator;
         default: adder2 = divided;
      endcase
   end

   // accumulator keeps track of the windowed average
   always_ff @(posedge CLOCK_50) begin
      if (reset) begin
         accumulator <= 0;
      end else begin
         accumulator <= accumulator + adder2;
      end
   end

   assign DataOutTop = accumulator;
```

Figure 6: A snippet of code from Task 3 with the FIR filtering logic.

In ModelSim, we verified the functionality of our averaging FIR filter on the ROM static tone with a window size of 8 (for easier testing and viewing in ModelSim, a smaller window size was desired). The multiplicator waveform confirms for us that the first 8 samples of the ROM static tone were read into the buffer but were not yet read and multiplied by –1 until the buffer became full. While the buffer is filling up with the first 8 samples, adder2 helps keep track of incoming divided samples (same as the divded waveform), and accumulator continuously adds the previous values of adder2. When the buffer is full and data is being read and written simultaneously, adder2 sums divided and multiplicator and passes this value to the accumulator.

Adding divided and multiplicator is key to creating the averaging effect of the moving average window as samples are passed through the buffer. Details in Figure 7 below.

Commented [CM1]: add caption for figure 7 plz thx. can just say "Task 3 waveform to verify beginning behavior of averaging FIR filter logic."

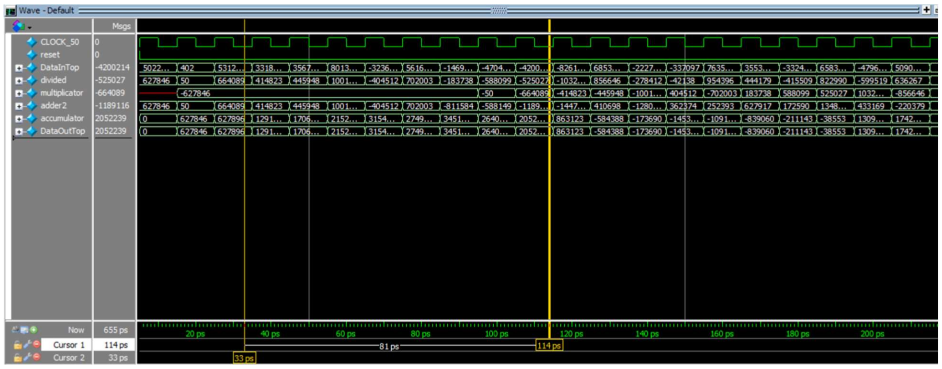


Figure 7: Task 3 waveform to verify beginning behaviour of averaging FIR filter logic.

The last few samples of the ROM static tone still pass through the buffer and are multiplied by –1 and are added to the accumulator. Eventually, the accumulator reaches 0 once the samples stop coming in (Figure 8).

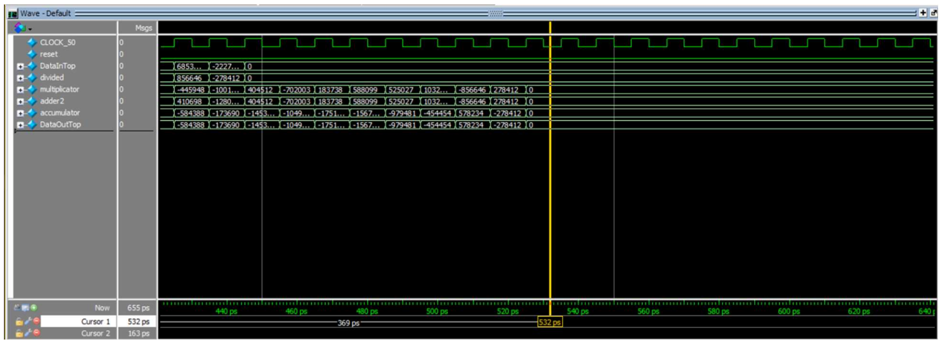


Figure 8: Task 3 waveform to verify end-behavior of averaging FIR filter logic.

Results

Our completed design of lab the digital signal processing system meets all the design requirements and is very well tuned to produce high clarity audio from either a dynamic input source such as microphone or a static input source such as a ROM. The user can select which sound source is sent to the CODEC output and whether it's filtered or unfiltered.

To verify that the digital signal processing system meets all the requirements of the specifications, we tested all the 4 possible cases that the system can experience. We tested two cases of each different input source being unfiltered, and then tested the two cases of each different input source being filtered. Additionally, we experimented with different averaging window sizes to determine what size produced the best sound. We concluded that the more samples averaged, the smoother the sound, whereas the less samples averaged, the noisier the sound.

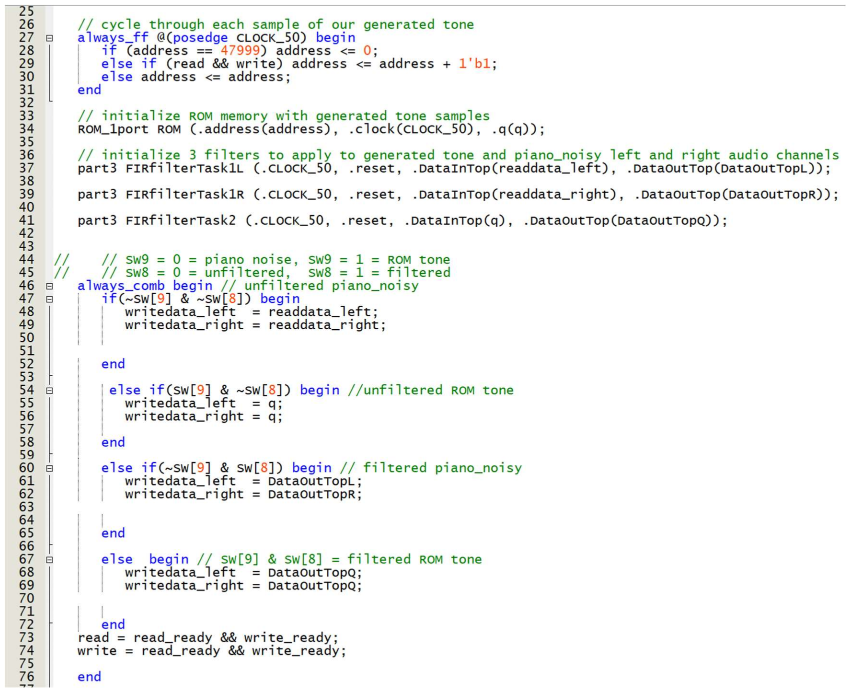
Since all functionality was tested and demonstrated in tasks 1, 2, and 3, no top-level test bench was needed.

DE1_SOC

The goal of our top-level module was to allow users to choose between an external or internal sound source, and whether that sound was filtered or unfiltered. We achieved this by setting up the given starter code for initializing the CODEC and then building case logic to assign the outputs of the CODEC to the desired case based on inputs from onboard switches SW9 and SW8. We instantiated a ROM instance to support Task 2 and initialized it with the generated .mif file from the Python script described in Task 2. We also instantiated 3 FIR filters: one for the left audio channel of Task 1, one for the right audio channel of Task 1, and finally one for the mono channel of Task 2. The code is shown in Figure 9 below.

The 4 cases for audio source and filter combinations are described as follows. The first case simply passed the raw unfiltered left and right audio channels of the external source from the CODEC input to the CODEC output. The second case passed the filtered left and right audio channels of the external source from the CODEC input to the CODEC output. The third case passed the unfiltered mono channel stored in the ROM to the output of the CODEC. The last case passed the filtered mono channel stored in the ROM to the output of the CODEC. We also had the read and write signals of the CODEC independent of the case. They are always being triggered directly by the status signals coming from the CODEC.

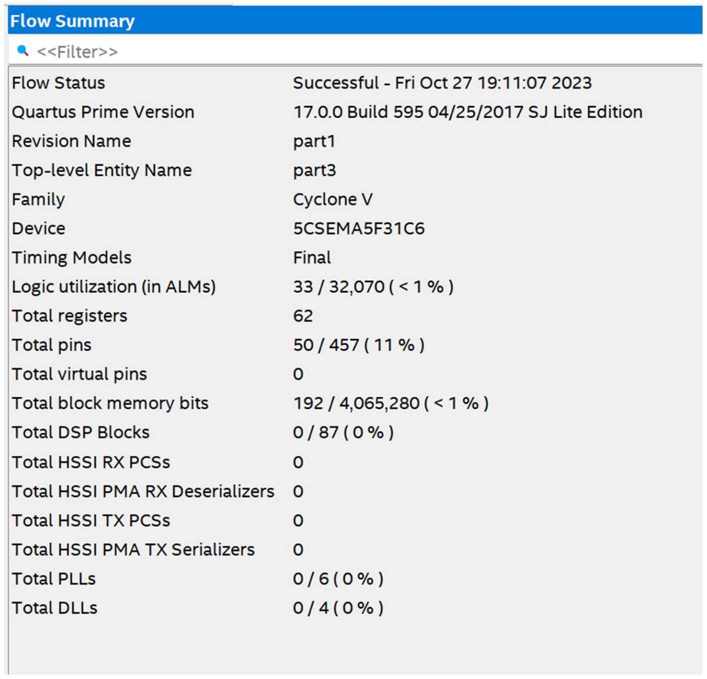
Since all functionality was tested and demonstrated in tasks 1, 2, and 3, no top-level test bench was needed.



```

    // cycle through each sample of our generated tone
    always_ff @(posedge CLOCK_50) begin
       if (address == 47999) address <= 0;
       else if (read && write) address <= address + 1'b1;
       else address <= address;
    end

    // initialize ROM memory with generated tone samples
    ROM_1port ROM (.address(address), .clock(CLOCK_50), .q(q));

    // initialize 3 filters to apply to generated tone and piano_noisy left and right audio channels
    part3 FIRfilterTask1L (.CLOCK_50, .reset, .DataInTop(readdata_left), .DataOutTop(DataOutTopL));

    part3 FIRfilterTask1R (.CLOCK_50, .reset, .DataInTop(readdata_right), .DataOutTop(DataOutTopR));

    part3 FIRfilterTask2 (.CLOCK_50, .reset, .DataInTop(q), .DataOutTop(DataOutTopQ));


//      // SW9 = 0 = piano noise, SW9 = 1 = ROM tone
//      // SW8 = 0 = unfiltered,  SW8 = 1 = filtered
    always_comb begin // unfiltered piano_noisy
       if(~SW[9] & ~SW[8]) begin
          writedata_left  = readdata_left;
          writedata_right = readdata_right;


       end

        else if(SW[9] & ~SW[8]) begin //unfiltered ROM tone
          writedata_left  = q;
          writedata_right = q;

       end

       else if(~SW[9] & SW[8]) begin // filtered piano_noisy
          writedata_left  = DataOutTopL;
          writedata_right = DataOutTopR;


       end

       else  begin // SW[9] & SW[8] = filtered ROM tone
          writedata_left  = DataOutTopQ;
          writedata_right = DataOutTopQ;


       end
    read = read_ready && write_ready;
    write = read_ready && write_ready;

    end
```

Figure 99: A snippet of code from the top-level module with user input logic.

Flow Summary



Flow Summary

<<Filter>>

Flow Status	Successful - Fri Oct 27 19:11:07 2023
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	part1
Top-level Entity Name	part3
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	33 / 32,070 (< 1 %)
Total registers	62
Total pins	50 / 457 (11 %)
Total virtual pins	0
Total block memory bits	192 / 4,065,280 (< 1 %)
Total DSP Blocks	0 / 87 (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	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 1100: Top level module flow summary

Experience Report

Creating the digital audio processing system was moderately difficult. We initially encountered issues with implementing the FIFO size correctly to ensure the filtering was done properly for task 3. We also encountered problems with not knowing if our code is working or if the boards on LabsLand are broken. We also spent a lot of time determining if modules are combinational or sequential. For teamwork, we used GitHub for easy collaboration of shared files which made code developing and debugging more efficient. It also helped to reference the FIR filter figure when implementing it into our own code. For the next lab we would like to get a head start and aim to work together live more frequently to reduce debugging time. Additionally, using GitHub's full functionality for version control may help make our code development process even smoother.

The estimated total time working on this lab was 24 hours broken as follows:

- Reading: 2 hours
- Planning: 2 hours
- Design: 1 hours
- Coding: 10 hours
- Testing: 5 hours
- Debugging: 4 hours.