# Lab 3

Ishan Dane NetID: idane
Ismail Memon NetID: ishmemon

## Design Procedure

In this lab, we were required to perform tasks using the Audio CODEC interface to generate and filter our audio. Our first task was implementing a given audio sound system to produce an audio output when the CODEC was ready that matched the given input audio. The second task involved creating a static tone based off of a chosen note through a ROM created by Quartus's libraries. The note was stored in the ROM through a MIF file and outputted through the Audio CODEC on a loop to play a static tone. The third task consisted of creating an averaging Finite Impulse Response filter. This was done through the use of a Fifo buffer and accumulator. Task 3 required us to create an FIR (finite impulse response) filter to filter out inputted sound based on our sample size N value of 256.

### Task #1

For task 1, we are asked to input an audio file into the DE1_SoC through LabsLand and receive an audio output that we can record and listen to. This process was done by modifying the Audio CODEC starter kit provided to us through part1.sv file. We had to ensure that the audio would only play when the CODEC was "ready" , which was when the ready_ready and write_ready signals were active high every 48000th of a second. To implement this logic circuit, we defined the inputs read, write, writedata_left and writedata_right in the part1.sv module. We assigned read and write to be dependent on read_ready and write_ready where both must be active high to read and write to DE1_SoC.

**Task #2**

For task 2, we used the specification's instructions to create a single port ROM from the Quartus library. The ROM is then used to store data from a MIF file note_data.mif created through the terminal to store the note C4 played at volume 5 for 1 second. From the specification, we used SW[9] to toggle between task 1 and task 2. When task 2 is active (SW[9] is high), the stored values in the ROM are outputted into the Audio CODEC on a loop causing the tone to be repeatedly outputted (loops to the start when it ends). When task 1 is active (SW[9] is low), the audio file is used as the input into Audio CODEC. Since we toggle between both task 1 and 2 with SW[9], we decided to place both of them in the top level module DE1_SoC for a clearer design and further efficiency.

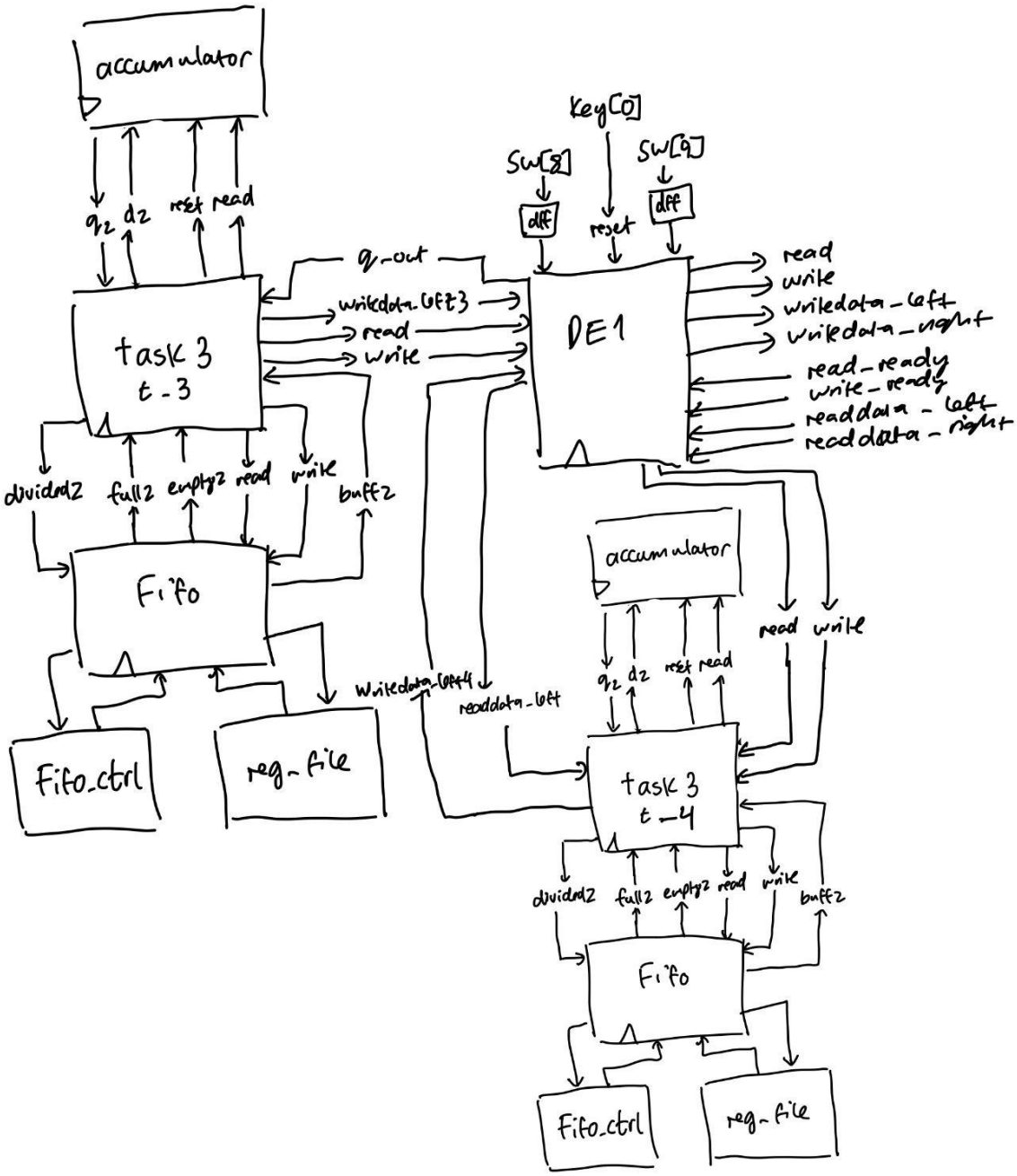
## Task #3

Task 3 had us create an FIR (finite impulse response) filter. Our design process was creating a first in first out buffer of a specified smoothing length, N, that we used to store N values. Using an accumulating register that was synchronized to the clock, we were able to sum together those N samples. Each time we summed we took out one sample that the buffer expelled so that our sample size would remain N. We divided each of the N samples by N before placing them in the buffer, so we could get the average value of those N samples. We then used that averaged, smoothed, output and made it work with switch 8 of the board. SW[8] was used to turn on or off the filter. SW[8] is high = On, SW[8] is low = Off.

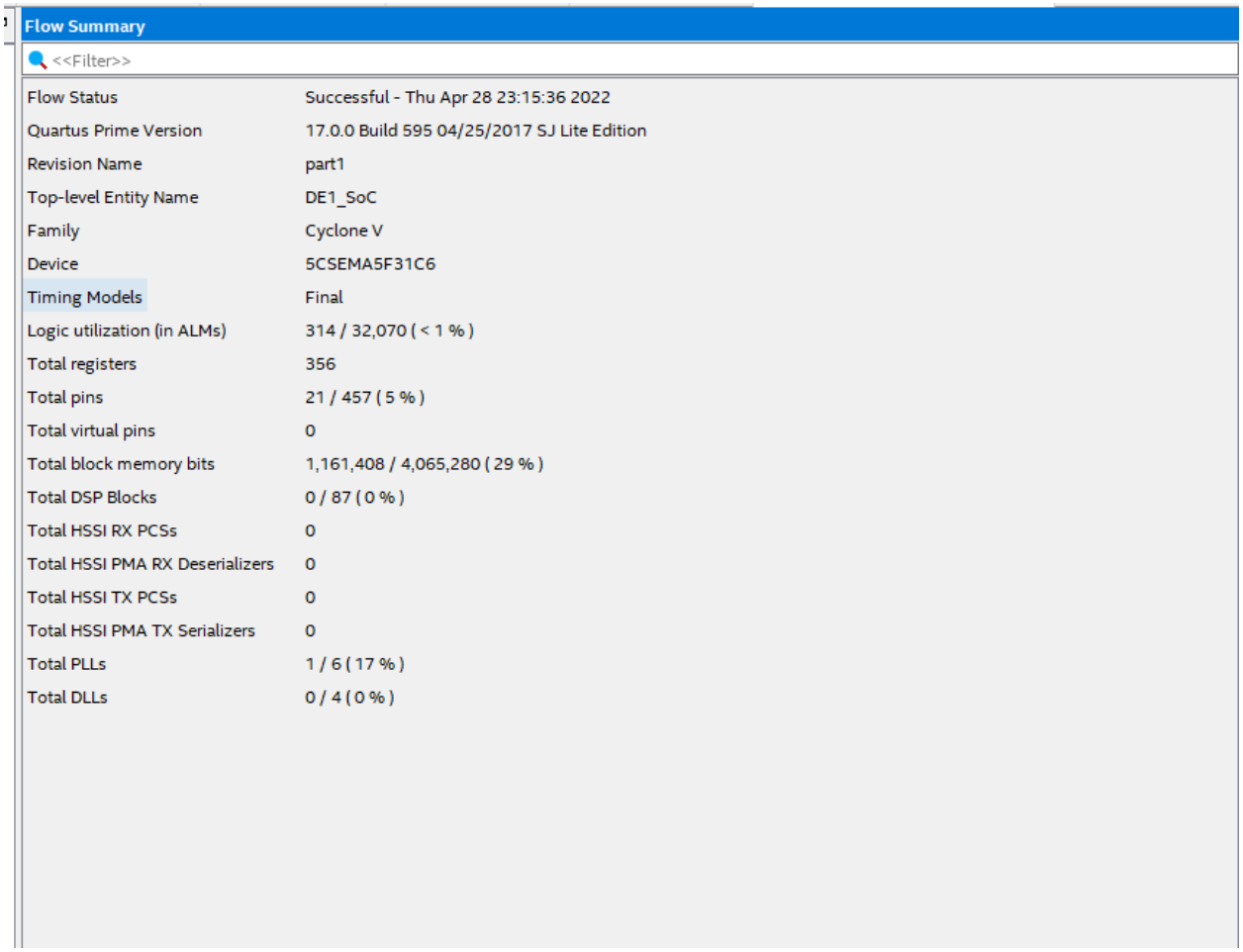
## Overall System

For our system, the DE1_SoC acted as the top level module. We used SW[9] to toggle between task 2 and task 1 written inside the top level module. SW[8] was used to toggle the filter from task 3 on or off. KEY[0] was used as our reset. Our top level module instantiates task3.sv. The top level module calls the task3 module twice for output. The logic for SW[9] and SW[8] is written within the top level module. When SW[9] is high, external audio is used for input through the DE1_SoC. When SW[9] is low, audio input from the ROM is used through the DE1_SoC. For task 3, a circular queue FIFO Buffer is implemented which only outputs when the queue is full. The FIFO buffer used an accumulator to sum the current value with the next inputted value and Fifo_ctrl and reg_file to help implement its design. All modules in the system used KEY[0] as reset and the clock to synchronize the system.

The top level module block diagram is shown below.


accumulator
q_z
d_z
reset
read
task 3
t-3
q_out
writedata_left3
read
write
DE1
Key[0]
SW[8]
SW[9]
diff
diff
reset
read
write
writedata_left
writedata_right
read_ready
write_ready
readdata_left
readdata_right
divided_z
full_z
empty_z
read
write
buff_z
Fifo
Fifo_ctrl
reg_file
writedata_left4
readdata_left
accumulator
read
write
q_z
d_z
reset
read
task 3
t-4
divided_z
full_z
empty_z
read
write
buff_z
Fifo
Fifo_ctrl
reg_file

## Flow summary



Flow Summary

<<Filter>>

| | |
|---|---|
| Flow Status | Successful - Thu Apr 28 23:15:36 2022 |
| Quartus Prime Version | 17.0.0 Build 595 04/25/2017 SJ Lite Edition |
| Revision Name | part1 |
| Top-level Entity Name | DE1_SoC |
| Family | Cyclone V |
| Device | 5CSEMA5F31C6 |
| Timing Models | Final |
| Logic utilization (in ALMs) | 314 / 32,070 ( < 1 % ) |
| Total registers | 356 |
| Total pins | 21 / 457 ( 5 % ) |
| Total virtual pins | 0 |
| Total block memory bits | 1,161,408 / 4,065,280 ( 29 % ) |
| 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 | 1 / 6 ( 17 % ) |
| Total DLLs | 0 / 4 ( 0 % ) |

## Results

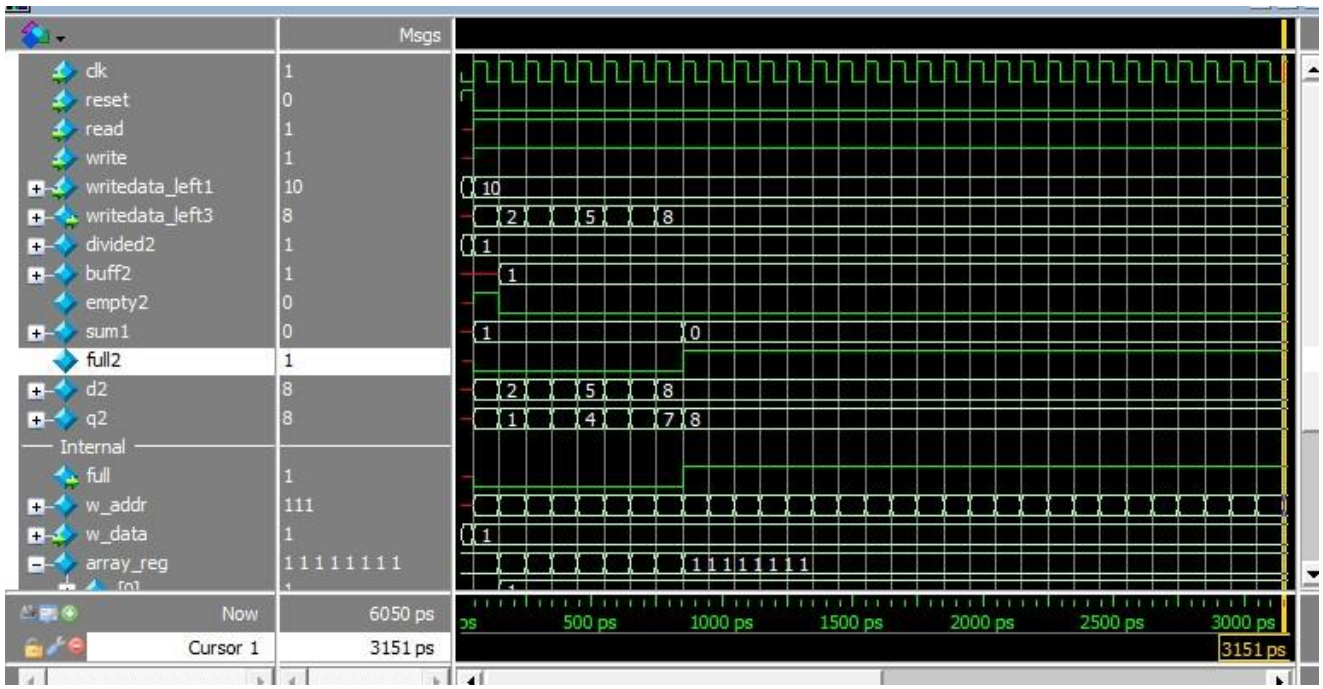
The ModelSim waveforms for each major module are presented and explained here.

### Task 2

We decided not to create a ModelSim testbench for task since our task 2 was only an incrementing flipflop (similar to the counter method demonstrated in lecture). All we did was increment the address by 1 bit through the flip flop when write was enabled so we decided to put it into our top level module in order for our Lab to work and further efficiency.

### Task 3

Task 3 was implementing a filtered output. The idea was to pass a continuous stream of output, divided by N, into a fifo buffer to allow for averaging. The testbench allowed us to verify whether this activity was correct. We passed in a steady stream of input data and observed the output. We kept read and write asserted so that the buffer would read the values after it is full. We expected to see the output be an average of the input stream and that average should discard the N+1 sample. We were able to confirm this with our Modelsim as shown in the figure below




## Experience Report

We found this lab to be very difficult compared to the previous labs. Although task 1 and task 2 were relatively quick. Creating the FIFO buffer through the specification took a while to figure out even though we were given the files as we had to implement it to the required design. Another incredibly difficult part of the lab was implementing task 3 into the top level module DE1_SoC. Ensuring that the top level module was receiving the right outputs and consequently filtering out the sound when tested required a significant amount of time to debug. What was especially the problem with our design process was that we did not name our variables efficiently. We ended up in a position where variables were named x1,x2,x3,x4 and thus in the debugging process took a significant amount of time to try and understand the paths the data was going through. It also made it very easy to accidentally confuse between different variables whilst coding and create errors. To fix this, we had to write down all the variables we had and

rename them or delete the ones we didn’t need, and thus cleaning up the code, making it a lot easier for us to understand the overall system.

The Lab took us a total of 27 hours to complete:

- Reading: 1 hour
- Planning: 1 hour
- Design:  2 hours
- Coding: 5 hours
- Testing: 6 hours
- Debugging: 12 hours