EE/CSE 371:
Design of Digital Circuits and Systems

Lab5: Digital Signal Processing[1]

Lab Objectives

In this lab, you will use the audio coder/decoder (CODEC) on the DE1-SoC board to generate and filter noise from both an external source and an internal memory.

Note: *Using audio with the DE1-SoC via LabsLand involves some special instructions so make sure to reference the `Audio_Guide.pdf` document before you do any testing on the hardware!*

Background

Sounds, such as speech and music, are signals that change over time. The amplitude of a signal determines the volume at which we hear it. The way the signal changes over time determines the type of sounds we hear. For example, an "ah" sound is represented by a waveform shown in Figure 1. The waveform is an *analog* signal, which can be stored in a *digital* form by using a relatively small number of samples that represent the analog values at certain points in time. The process of producing such digital signals is called **sampling**. The points in Figure 2 provide a sampled waveform. All points are spaced equally in time and they trace the original waveform.

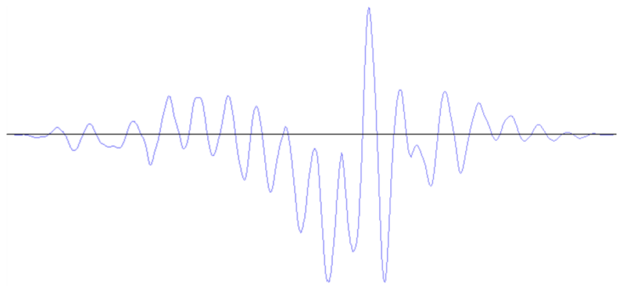
Figure 1: *An example waveform for an "ah" sound.*

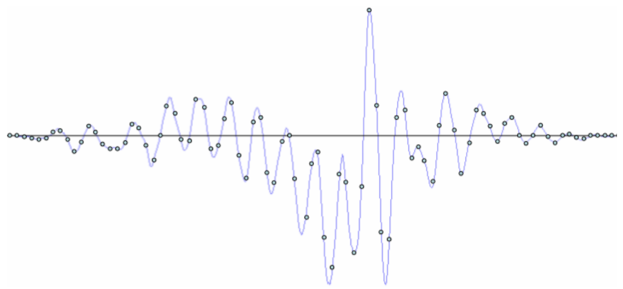
Figure 2: *Sampled waveform of Figure 1.*

The DE1-SoC board is equipped with an audio CODEC capable of sampling sound from a microphone and providing it as an input to a circuit. By default, the CODEC provides 48000 samples per second (48 kHz), which is sufficient to accurately represent audible sounds (by the Nyquist-Shannon sampling theorem).

To simplify this lab, a system that can record and playback sounds on the board is provided as a "starter kit." The system comprises a Clock Generator, an Audio CODEC Interface, and an Audio/Video Configuration module (Figure 3). For this lab, we will assume that our audio will be split into two *channels*, left and right, that are intended to be played from multiple speakers (*e.g.*, left and right earbuds/headphones).

The left column of signals in Figure 3 are the inputs and outputs of the system. These I/O ports supply the clock inputs and connect the Audio CODEC and Audio/Video Configuration modules to the corresponding peripheral devices on the DE1-SoC board. The right column of signals connects the Audio

[1] This lab is adopted from Intel's University Program

CODEC Interface module to your circuit and allows your circuit to record sounds from a microphone and play them back via speakers.

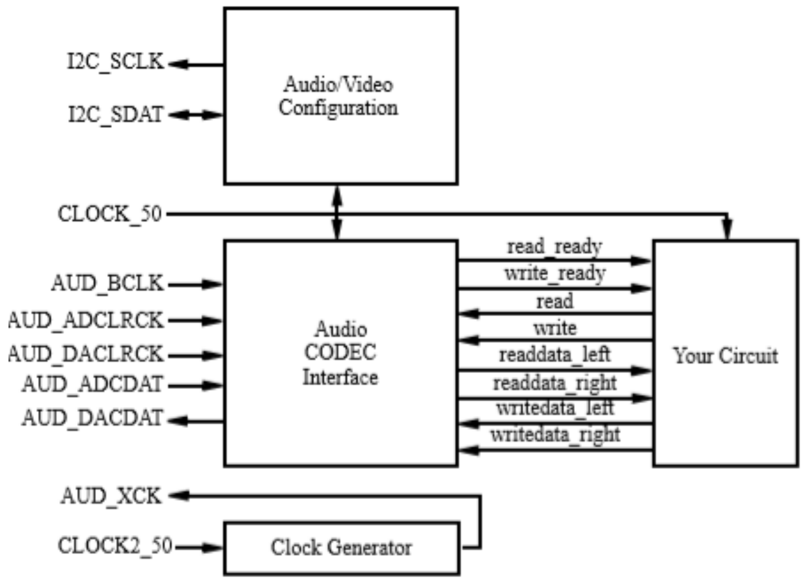

Figure 3: *The audio system used in Lab 3.*

The system works as follows (this interface is also detailed in the starter kit files):

- Upon reset, the Audio/Video Configuration begins an auto-initialization sequence. The sequence sets up the audio device to sample microphone input at a rate of 48 kHz and produce output through the speakers at the same rate.
- Once the auto-initialization is complete, the Audio CODEC begins reading the data from the microphone once every 48,000-th of a second and sends it to the Audio CODEC Interface core in the system.
- Once received, a sample is stored in a 128-element buffer in the Audio CODEC Interface core. The first element of the buffer is always visible on the `readdata_left` and `readdata_right` outputs (*i.e.*, 2 channels for 1 sample), but the data is only valid when the `read_ready` signal is asserted. When you assert the `read` signal, the current sample is replaced by the next element one or more clock cycles later, indicated by `read_ready` being reasserted.
- The procedure to output sound through the speakers is similar. Your circuit should monitor the `write_ready` signal. When the Audio CODEC is ready for a write operation, then your circuit can write a sample to the `writedata_left` and `writedata_right` inputs and assert the `write` signal. This operation stores a sample in a buffer inside of the Audio CODEC Interface, which will then send the sample to the speakers at the right time.

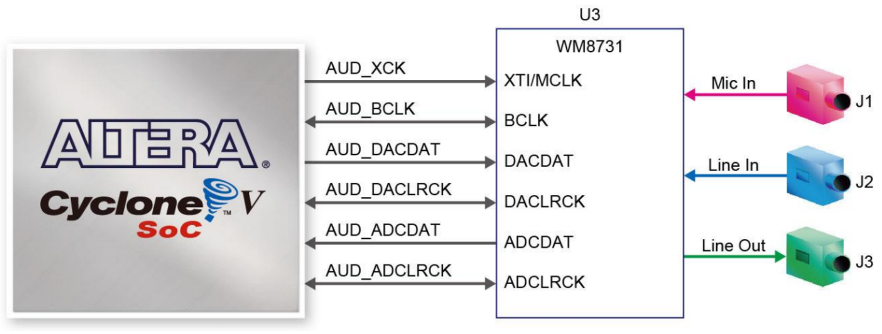

Figure 4: *Connections between the FPGA and the Audio CODEC.*

Task 1

In this task, you will play a music file into the audio input of the DE1-SoC, which will simultaneously be played through the audio output of the device.

1) Modify the provided starter kit circuit to pass the audio input to audio output *only when the CODEC is ready*.

2) In LabsLand, upload an audio file (either the provided *`piano_noisy.mp3`* or your own – see *`Audio_Guide.pdf`* for restrictions) alongside your synthesized circuit and verify that the preview of the audio file matches your recorded output sample. Recall that you must record while playing the audio file into the DE1-SoC.

Task 2

In this task, you will produce a static tone from memory. By storing the samples from a sound's waveform, you can play back those samples later to re-produce the sound. We have provided a Python script which generates MIF files for a desired tone, which can then be loaded into memory. Information on how to use this python script can be found in the tutorial document *`note_gen_tutorial.pdf`*.

1) Follow the tutorial on using the python script to create a MIF file with your desired tone. Create a ROM that initializes to the values stored in the MIF file using Quartus' built in library modules:
   a. Open the IP Catalog in the Quartus menu by clicking on `Tools` → "`IP Catalog`". In the IP Catalog window, expand "`Library`", then "`Basic Functions`", then "`On Chip Memory`". Then double-click "**`ROM: 1-PORT`**".
   b. Give the file an appropriate name, select "`Verilog`" as the file type, then click "`OK`" to open the configuration wizard.
   c. In the first window specify 24-bits as the word width. You will need to check your generated MIF file to find the number of words, as this will vary. select "`M10K`" for memory block type and "`Single clock`" for the clocking method, then click "`Next >`".
   d. Click "`Next >`" past the "`Regs/Clken/Aclrs`" window keeping the default selections.
   e. In the next window, under "`Do you want to specify the initial contents of memory?`", select "`Yes, use this file for the memory content data`" and specify your MIF file. Click "`Finish`" to use the rest of the default settings, then click "`Finish`" again at the summary page to exit the wizard. If prompted, add the new Quartus Prime IP File to your project by clicking "`Yes`".

2) Output the values stored in your ROM sequentially to the Audio CODEC, looping back to the start when you reach the end. Make sure to increment the address only once each time you write.

3) Compile and download your overall design to the DE1-SoC board, record the generated tone in LabsLand and then listen to the generated tone played from the FPGA.
4) Combine your Task #1 and Task #2 by instantiating them in a single top-level module that plays the input audio (Task #1) when `SW9=0` and your tone from memory (Task #2) when `SW9=1`. While testing, you should switch between the tasks about halfway through your recording.
5) Save your newly created .mp3 file as "**task2.mp3**". You will need to turn this in.

Task 3: Extra Credit

In this task, you will learn a basic signal processing technique known as **filtering**. Filtering is a process of adjusting a signal (*e.g.*, removing noise). **Noise** in a sound waveform is represented by small, but frequent changes to the amplitude of the signal.

A simple logic circuit that achieves the task of noise-filtering is an averaging **Finite Impulse Response (FIR)** filter. An averaging filter removes noise from a sound by averaging the values of adjacent samples. Conceptually, a basic FIR filter uses registers as delay blocks and computes the moving average from their outputs, as shown in Figure 5.

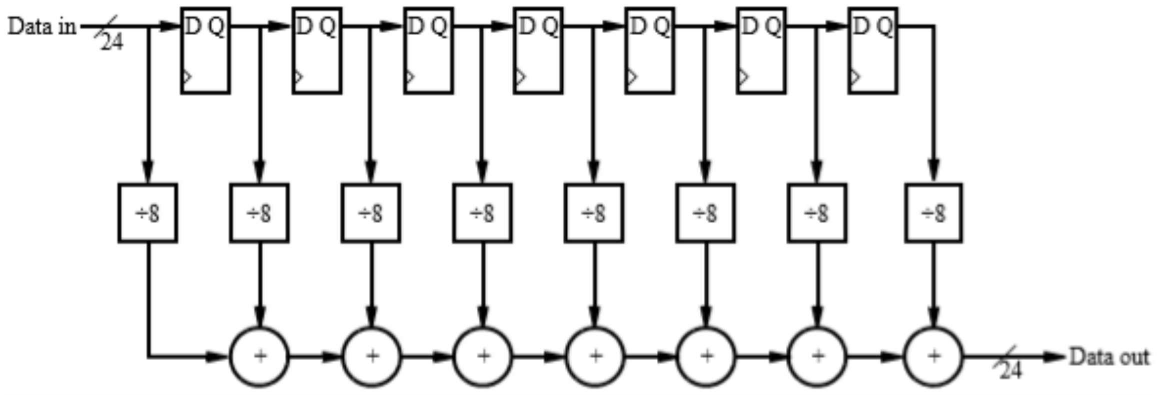

Figure 5: *A Finite Impulse Response (FIR) filter that takes a moving average of 8 samples. While this is conceptually what we want to achieve in Task #3, we will use a different implementation, as shown in Figure 6.*

However, we can achieve the same result using a FIFO buffer and accumulator (*i.e.*, a register whose input is the sum of the next incoming data value and its current stored value), as shown in Figure 6.

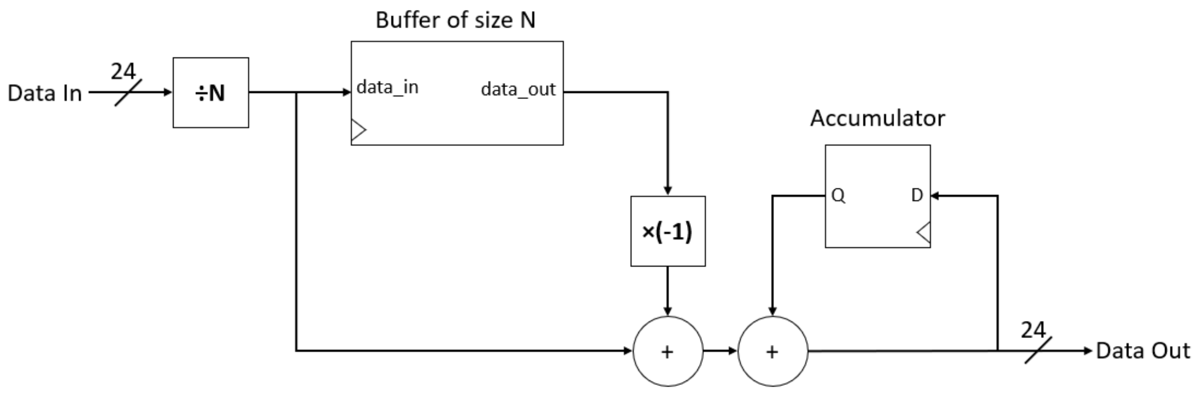

Figure 6: *N-sample averaging FIR filter using a FIFO buffer and accumulator that you will implement in Task #3.*

To compute the average of the last N samples, this circuit first divides the input sample by N. Then, the resulting value is stored in a First-In First-Out (FIFO) buffer of length N and added to the accumulator. To make sure the value in the accumulator is the average of the last N samples, the circuit subtracts the value that comes out of the FIFO, which represents the $(N+1)^{\text{th}}$ sample.

A few notes to help:

- Although the starter kit includes an Altera library FIFO buffer, we recommend using the circular queue implementation from lecture and hw3 for familiarity and ease of use.
- **Division can cause strange behaviors when synthesized onto your board.** The most robust way to divide by a power of two is using an arithmetic right shift. The following code snippet uses the replication and concatenation operators to divide the w-bit signal `data` by 2^n:
    - `assign divided = {{n{data[w-1]}}, data[w-1:n]};`
- Carefully consider how to ensure that your FIR filter updates when it is supposed to. As a single system, the FIFO buffer and accumulator should be synchronized together (*i.e.*, at any clock trigger, either both or neither should update). But when should they update based on the CODEC signals?

1) Implement the N-sample averaging FIR filter. Use `SW8` to select between filtered (`SW8=1`) and unfiltered (`SW8=0`) audio to output; this filtering should be able to interact with *both* Task #1 (`SW9=0`) and Task #2 (`SW9=1`).
2) Compile and download your overall design to a DE1-SoC board in LabsLand and use *piano_noisy.mp3* to generate your recorded output.
3) Experiment with different values of N to see what happens to the noisy audio sample. Use the switch to tell what kind of effect the filter is having on the audio.
    a) When you change N, remember to adjust *both* the FIFO buffer length and the circuit divisor. We recommend experimenting with values of N that are powers of 2 for easier division.
4) Save your .mp3 as "**task3_extra_credit.mp3**". You will need to turn this in.

Lab Demonstration and Submission Requirements

- No demo needed specifically for Task 1; Task 2 and Task 3 will have the Task 1 functionality built into the design.
- Record a demo video for Task 2 that toggles between the piano_noisy audio file and the tone from memory. You do not need to open up your recorded .mp3 and play the contents in the demo video. However, you will need to submit the audio file to the "Lab5: Demo" assignment, along with the demo video. Your demo video must be a screen recording created from software like ActivePresenter or Zoom.
- Record a demo video for Task 3 (if you are attempting the extra credit) that demonstrates filtered and unfiltered audio, both for piano_noisy and the tone from memory. You do not need to open up your recorded .mp3 and play the contents in the demo video. However, you will need to submit the audio file to the "Lab5: Demo" assignment, along with the demo video. Your demo video must be a screen recording created from software like ActivePresenter or Zoom.
- Write a Lab Report, as framed by the Lab Report Outline document on Canvas. Comment your code. Follow commenting guidelines as discussed in the Commenting Code document on Canvas. Please be sure to include block diagrams in your Lab Report. Please include the simulation results in your lab report. Please include waveforms for all modules you used unless it is explicitly stated otherwise. For this lab, you only need to include waveforms of all modules you created. You do not need to include waveforms of the modules in the provided starter set. Submit your lab report as a pdf file and all of your SystemVerilog files on Canvas. Please refer to the grading rubric on Canvas.