EE/CSE 371:
Design of Digital Circuits and Systems

Lab5:Digital Signal Processing[1]

Lab Objectives

This is an exercise in using the audio coder/decoder (CODEC) on the DE1-SoC board. The lab involves sending audio signals through a microphone to the audio CODEC to provide input sound, altering the received sound by filtering out the noise and then outputting the resulting sound from the FPGA.

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.

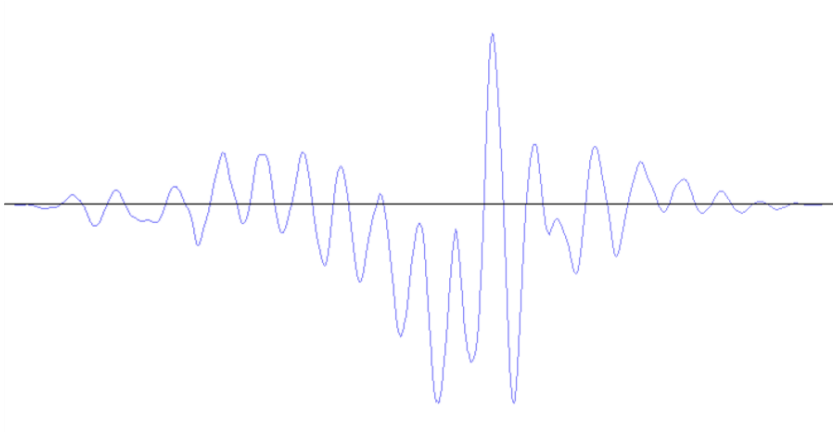

Figure 1. A waveform for an 'ah' sound

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*.

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

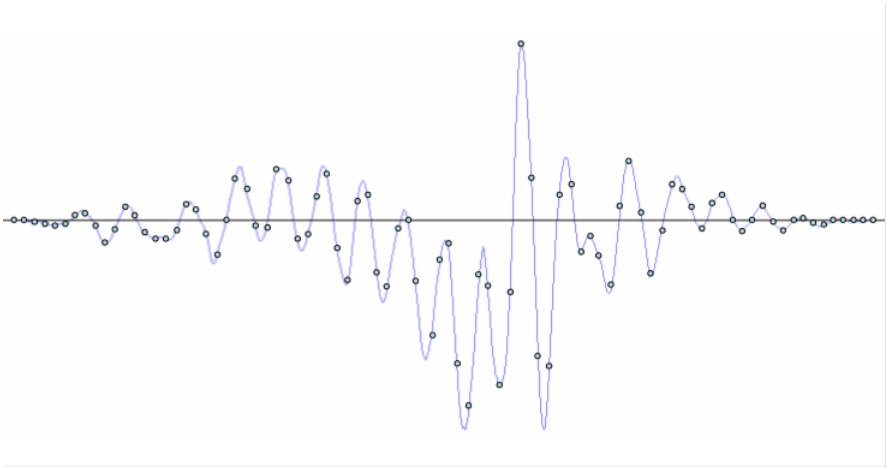

Figure 2. A sampled waveform for an ’ah’ sound.

The points in Figure 2 provide a sampled waveform. All points are spaced equally in time, and they trace the original waveform.

The DE1-SoC board is equipped with an audio CODEC capable of sampling sound from an audio input. By default, the CODEC provides 48,000 samples per second, which is sufficient to accurately represent audible sounds.

This lab involves the design of several circuits that take audio input from the CODEC, record, process this sound data, and then play it back. To simplify the task, a simple system that can record and playback sounds on the board is provided as a "starter kit". The system, shown in Figure 3, comprises a Clock Generator, an Audio CODEC Interface, and an Audio/Video Configuration module.

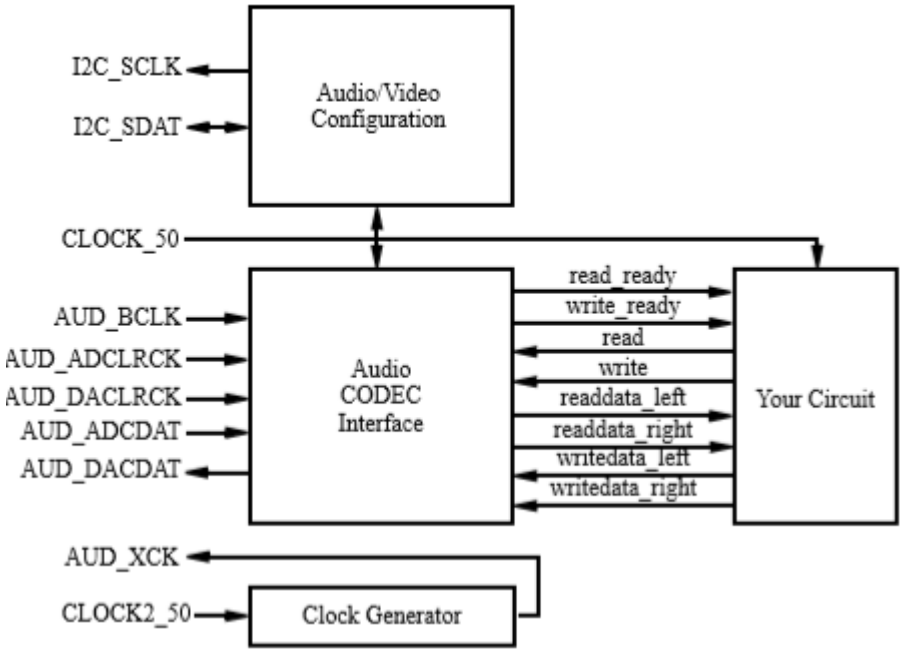

Figure 3. Audio System for this lab

The left-hand side of Figure 3 shows 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. In the middle of the figure, a set of signals to and from the Audio CODEC Interface module is shown. These signals allow the circuit depicted on the right-hand side to record sounds from a microphone and play them back via speakers.

The system works as follows. Upon reset, the Audio/Video Configuration begins an initialization sequence. The sequence sets up the audio device to sample input at a rate of 48kHz and produce output at the same rate. Once the initialization is complete, the Audio CODEC begins reading the data from the audio input at a rate of 48,000 times per second, and sends it to the Audio CODEC Interface core in the system. Once received, the 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 when the read_ready signal is asserted. The next element can be read by asserting the read signal, which ejects the current sample and a new one appears one or more clock cycles later, if the read_ready signal is asserted.

To output sound through the speakers, a similar procedure is followed. Your circuit should observe the write_ready signal and, if it is asserted, write a sample to the writedata_left and writedata_right inputs while asserting 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 audio ouput at the right time.

A starter kit that contains this design is provided as part of this lab.

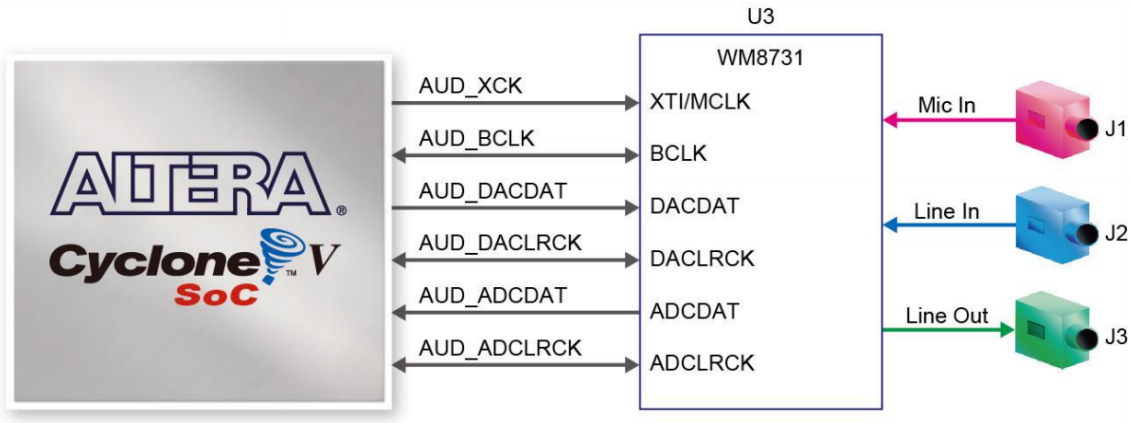

Figure 4. Connections between FPGA and Audio CODEC

Task 1

You are provided with a starter code that provides much of the functionality described above. By default, this code passes the input sound (readdata_left and readdata_right) to the Audio CODEC. It also handles the assignment of the read and write signals with the following code:

```
// only read or write when both are possible
 assign read = read_ready & write_ready;
 assign write = read_ready & write_ready;
```

These assignments mean that the Audio CODEC will only read samples from its buffer when it is ready to both read and write. Similarly, it means that your code will only write the data on writedata_left and writedata_right to the Audio CODEC under the same conditions.

You will also notice a block of combinational logic used to assign the value of writedata_left and writedata_right. The four cases described in this block are as follows:

- **CASE A:** When none of the KEYs are pressed, the microphone audio is written directly to the Audio CODEC.
- **CASE B:** When KEY0 is pressed, noise is added to the microphone audio, and noisy_left and noisy_right are written to the Audio CODEC.
- **CASE C:** When KEY1 is pressed, task2_left and task2_right are written to the Audio CODEC. These will be the outputs of the filter you design in Task 2.
- **CASE D:** When KEY2 is pressed, task3_left and task3_right are written to the Audio CODEC. These will be the outputs of the filter you design in Task 3.

You do not need to make any modifications to the provided starter code for Task 1. **Refer to the *LabsLand Audio Interface Guide* on Canvas for instructions on how to send input audios to the microphone and record the output audios from the speaker.** Holding KEY0, a high-pitched noise will be added to the output audio.

Task 1 Demonstration

Play the provided **piano.mp3** into your FPGA. Record the output of your FPGA's speaker into a 20-second audio file called **task1.mp3**. The first half of the audio should be the output audio of **CASE A**, whereas the second half of the audio file should be the output audio of **CASE B**. I.e., no **KEY** should be pressed during the first half of the recording, and **KEY0** should be pressed during the second half of the recording.

Task 2

In this task, you will implement a filter to remove the high-pitched noise you heard in task 1. 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. The schematic diagram of the filter is shown in Figure 5.

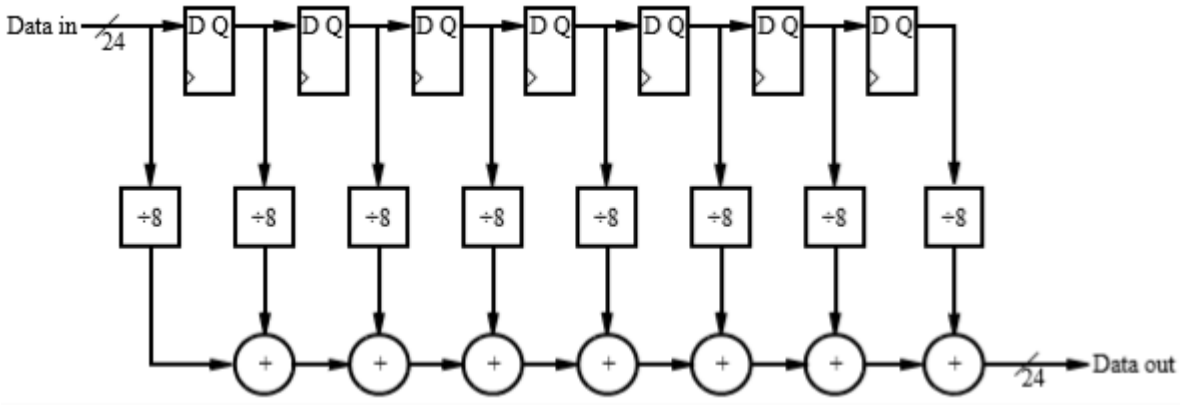

Figure 5. A simple averaging FIR filter

An averaging filter like the one shown in Figure 5 removes noise from a sound by averaging the values of multiple samples. In this particular case, it removes small deviations in sound by averaging the adjacent 8 samples.

Your task is to design and test a module that implements the circuit shown in Figure 5. Then, use your module to process the noisy audio (noisy_left and noisy_right) and output the filtered audio to task2_left and task2_right. Compile and upload your code to your board, and observe the effect of your filter by comparing the audio output from pressing KEY0 and KEY1.

Note: As described above, the read signal is asserted true whenever the Audio CODEC both has a sample that's able to be read and is ready to accept a new sample to be written. As such, **the registers in your filter should be enabled only when read is true.**

Note: Division can cause strange behaviors when synthesized onto your board. The most robust way to divide by a power of two is using shifting. The following code snippet uses the replication operator and concatenation to divide the *N*-bit signal *data* by 2^n:

```
assign divided = {{n{data[N-1]}}, data [N-1:n]};
```

Task 2 Demonstration

Play the provided **piano.mp3** into your FPGA. Record the output of your FPGA's speaker into a 20-second audio file called **task2.mp3**. The first half of the audio should be the output audio of **CASE B**, whereas the second half of the audio file should be the output audio of **CASE C**. I.e., **KEY0** should be pressed during the first half of the recording, and **KEY1** should be pressed during the second half of the recording.

Task 3

The filter in task 2 operates by averaging 8 adjacent samples. At 48 kHz, that's a very small time frame. Now, your task is to make a generalized averaging filter that accepts a parameter *N*, the number of samples over which to average. A schematic of this filter is shown in Figure 6.

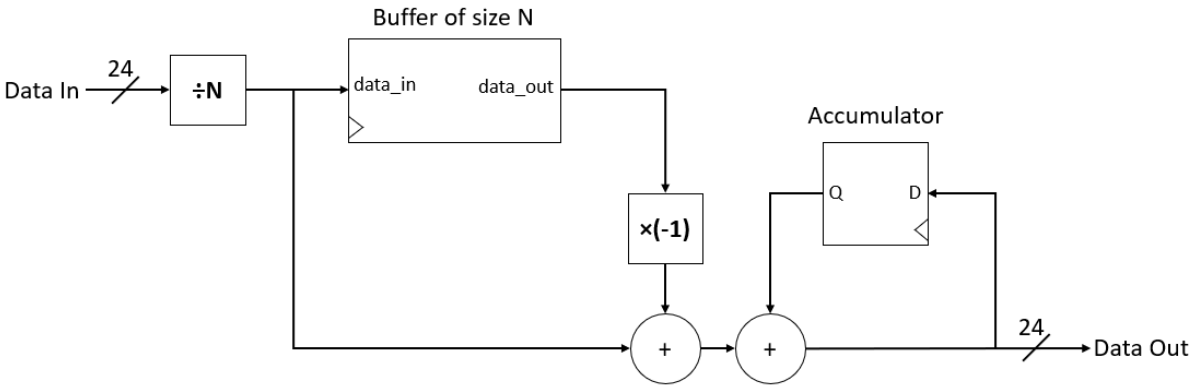

Figure 6. N-sample averaging FIR filter.

This filter uses a buffer of size *N*. When a new sample is read, it is divided by *N*. Then, two things happen with this new sample:

1. The divided sample is added to a buffer of size *N*
2. The divided sample is added to the output of the filter

At the same time, the *oldest* sample in the buffer is *subtracted* from the output. Thus, the output is always equal to the average of the previous *N* samples.

You might notice that the buffer used here is a First-In-First-Out (FIFO) buffer. However, it's probably easier to implement this buffer as a shift register that's similar to the circuit of Task 2, as opposed to the FIFOs you have seen previously in this course.

Design and test a module that implements the circuit shown in Figure 6. Then, use your module to process the noisy audio (noisy_left and noisy_right) and output the filtered audio to task3_left and task3_right. Compile and upload your code to your board, and observe the effect of your filter using the KEYs. Try different values of *N* to see how they affect the output of the system.

Note: As before, your filter should only take new samples when read is true.

Task 3 Demonstration

Use **N=16** for the demonstration. (Even though we specified N for the demonstration, we will change the N in your code to other numbers when grading your submissions. Therefore, you should still create the generalized filter and make sure we can find and change the parameter in your top-level module.)

Play the provided **piano.mp3** into your FPGA. Record the output of your FPGA's speaker into a 20-second audio file called **task3.mp3**. The first half of the audio should be the output audio of **CASE B**, whereas the second half of the audio file should be the output audio of **CASE D**. I.e., **KEY0** should be pressed during the first half of the recording, and **KEY2** should be pressed during the second half of the recording.

Lab Demonstration and Submission Requirements

- While working on the lab, on Padlet, write about a problem you had in the lab and the fix to it, and share a tip or trick you learned. You can also share an aha moment that you discovered while working on the lab. Avoid duplicating comments made by your classmates. NO videos for this Padlet task, please use textual comments. The link to the Padlet can be found in the corresponding Canvas assignment. Please note that the grace period does **not** apply to this portion of the lab.
- Submit the three audio files (task1.mp3, task2.mp3, task3.mp3) to the "Lab5: Demo" assignment on Canvas. No demo video is required for this lab. Please be sure to strictly follow the instructions in the Task 1 Demonstration, Task 2 Demonstration, and Task 3 Demonstration sections.
- 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 ket. Submit your lab report as a pdf file and all of your SystemVerilog files on Canvas. Please refer to the grading rubric on Canvas. As the grace period applies to the lab report and code assignment, you may submit it up to 2-days late without receiving a late penalty.