Lab 3: Simple Audio Signal Manipulation

6.375 Laboratory 3
Assigned: February 19, 2010
Due: February 26, 2010

1 Introduction

In this lab we will implement a couple of audio manipulations using the infrastructure built up in the previous two labs. We will implement pitch modulation and equalization of the audio signal in the frequency domain.

Figure 1 shows the pipeline we will be working with in this lab, which adds a `Transform` module between the `FFT` and `IFFT` blocks.

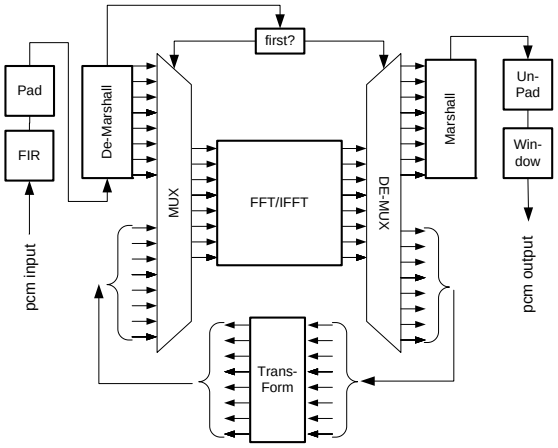

Figure 1: Lab 3 Audio Pipeline

1.1 Background: Frequency Domain Transformations

When an audio frame is transformed into the frequency domain, the representation is merely the relative amplitudes of each constituent frequency. The number of constituent frequencies is referred to as the *resolution* of the FFT, and corresponds directly to the number of points. Our current implementation has eight points, which is relatively low granularity, but if we decided to double the number of points, it would double our resolution and give us far greater accuracy when modifying the signal.

The transformation required to modulate the pitch in the frequency domain is very simple, since all we are doing is shifting the pitches either up or down. The intervals we can shift will be multiples of the resolution of the FFT, or the distance in frequency between two adjacent points (the size of the bins). This number can be calculated by dividing the sampling rate by half the number of points on the FFT. In our case, we are using CD quality audio, which is sampled at a rate of 44.1 KHz. If we configure our FFT with 8 points, we can achieve a resolution of 11.02 KHz.

Equalization is a process by which specific frequencies in the audio signal are emphasized or deemphasized. We can achieve this in the frequency domain simply by multiplying the frequency in each bin by some constant factor to increase or decrease it.

1.2 The New Pipeline

By now you should be familiar with the overall pipeline organization. We will reuse the high-level organization from the previous lab. This lab will involve the addition of the transform module as shown in Figure 1. The transform module we provide does not do any transformation. It will be your job to make the transform module perform pitch modulation for the first part of the lab and equalization for the second part of the lab.

As before, you should provide your own implementation of the FIR filter and FFT from the previous labs.

1. Extract the code from the lab3 harness and add it to your subversion repository. Just as we did for the previous labs, you will need to add the 6.375 course locker and source the `setup.csh` script. Navigate to the directory which contains the lab1 and lab2 folders and run

   ```
   % tar -xzvf /mit/6.375/lab-harnesses/lab3-harness.tar.gz
   % svn add lab3
   ```

   This will create a directory called `lab3/` with some new Bluespec code in the `common/` directory as well as `pitch/` and `equalizer/` directories containing the boiler plate code for the audio manipulations you will implement in this lab.

2. Copy your FIR filter and parameterized FFT from lab 2 and add them to subversion.

   ```
   % cd lab3
   lab3 % cp ../lab2/fir/FIRFilter.bsv fir/FIRFilter.bsv
   lab3 % cp ../lab2/cpfft/FFT.bsv fft/FFT.bsv
   lab3 % svn add fir/FIRFilter.bsv fft/FFT.bsv
   ```

3. Check the code into subversion.

   ```
   lab3% svn ci . -m "Lab3 Initial Checkin"
   ```

2 Pitch Modulation

The following pseudocode can be used to modulate the pitch of an audio sample up by the resolution of the FFT where `N` is the number of points of the FFT, `x` is the input vector of samples, and `z` is the output vector of samples.

```
for (int i = 0; i < N; i++) {
    if (i == 0 || i == N-1) {
        z[i] = 0;
    } else if (i < N/2) {
        z[i] = x[i-1];
    } else if (i > N/2) {
        z[i] = x[i+1];
    } else {
        z[i] = x[i];
    }
}
```

We have provided for you the module `mkTransform` in the directory `pitch/` where you should implement this algorithm to perform pitch modulation. Currently the `mkTransform` module just passes the input as is to the output.

1. Change the `Transform` module in the `pitch/` directory to implement the pitch modulation algorithm shown above. The output from running the audio pipeline with pitch transformation on input `data/foo.pcm` with an 8 point FFT is provided at `data/foo_pitch8.pcm`.

3 Equalization

For equalization we scale the magnitude of the frequencies in each of the FFT bins to adjust how strong they are in the output signal. If we are given a scaling factor for each bin in the vector `mixer`, the following algorithm can be used to perform equalization.

```
for (int i = 0; i < N; i++) {
    z[i] = mixer[i] * x[i];
}
```

3.1 Module Parameters

We saw in lab 2 how modules can be made generic with the aid of Bluespec types. Here we show another way of making modules generic: module parameters.

We can implement a `mkEqualizer` module which takes as input a parameter which is the mixer values to use in equalization. The definition of the `mkEqualizer` module might look like

```
module mkEqualizer#(Vector#(points, data) mixer) (Transform#(points, data))
    provisos(Arith#(data), Bits#(data, _));
```

The `#` after `mkEqualizer` introduces module parameters. In this case we have a single parameter, the mixer vector. Inside of the `mkEqualizer` module we can refer to this vector as if it were defined locally.

When we instantiate the `mkEqualizer` we provide the specific mixer vector to use. For each instantiation we could pass different mixer vectors, each generating hardware specific to the mixer values, all using the same `mkEqualizer` Bluespec source code. Here's an example of such an instantiation.

```
Transform#(points, data) equalizer <- mkEqualizer(mixervals);
```

3.2 Implementing the Equalizer

In `equalizer/Transform.bsv` we have set up the boilerplate code for the equalizer. We provide the function `genMixer` to generate an appropriate mixer vector given two scaling factors, one for the low frequencies and one for the high frequencies. The `mkTransform` module is already set up to use this function.

Your job is to implement the equalizer algorithm listed above. We are concerned about space in this implementation, so instead of using the `*` operator we have again provided a special multiplier to use, called `mkGenericMultiplier` with interface `GenericMultiplier` just like the multiplier we used in lab 1. Implement your equalizer to use the `mkGenericMultiplier`, and specifically implement your equalizer to use a *single* instance of the `mkGenericMultiplier` for all of the multiplications you need. We have instantiated the single instance you are allowed to use for you.

1. Modify the `mkEqualizer` module to implement the equalization algorithm shown above using a single instance of the `mkGenericMultiplier` for the multiplications.

   The file `data/foo_equalizer8.pcm` contains the expected output of your audio pipeline with the equalizer when run on `data/foo.pcm`.

4 Discussion Questions

1. Describe how you implemented the equalizer algorithm using only a single instance of the multiplier.

2. How is the performance of the equalizer impacted by the choice to multiplex over a single multiplier instead of instantiating multiple multipliers like we did for the FIR filter in lab 1?

5 What to Turn In

When you have completed the lab you should check in a final version via subversion. This should include your modifications to `pitch/Transform.bsv` and `equalizer/Transform.bsv`. It should also include your answers to the discussion questions in a file called `answers` in the top level lab directory.

Make sure to add your new files before checking them in if you have not already done so. For example, you could run:

```
lab3% svn add fir/FIRFilter.bsv fft/FFT.bsv
lab3% svn add answers.pdf
lab3% svn ci -m "Lab3 final submission"
```