Lab 3: Pitch Shifting - Completing the Audio Pipeline

6.375 Fall 2019
Assigned: September 23, 2019
Due: September 30, 2019 11:59PM

1 Introduction

In this lab we will complete the audio pipeline's functionality by implementing a pitch shifting transformation in the frequency domain. Your job will be to implement the pitch shifting based on reference C code, implement tests of your pitch adjuster to verify it works correctly, install the transformation in the audio pipeline, and then verify the functionality of the entire audio pipeline.

1.1 Background: Pitch Shifting

Pitch shifting is the process of adjusting the frequencies in an audio stream so they sound higher or lower in pitch. It is possible to change the pitch of an audio stream simply by changing the sampling rate, but that also changes the speed of the audio.

For example, to lower the pitch of `mitrib.pcm` by an octave, we could play it back at a different sample rate.

```
ffmpeg -f s16le -ar 44100 -i mitrib.pcm mitrib.wav
ffmpeg -f s16le -ar 22050 -i mitrib.pcm mitrib_halfrate.wav
```

If you listen to `mitrib.wav` and `mitrib_halfrate.wav`, you'll hear the second one is an octave lower, but it also plays back at half the speed of the original. To change the pitch of an audio stream without also changing the speed requires more work, and is what you will implement in this lab.

Once you can adjust the pitch of an audio stream without affecting the speed at which it is played back, you can couple that with a sample rate change to change the speed at which the audio is played without affecting the pitch of the audio. For example, if you raise the pitch by an octave, then resample at half the rate, the combination ends up leaving the pitch unchanged from the original while halving the speed of the audio.

Pitch shifting has many interesting applications. Pitch shifting can serve as a useful tool for musicians. For example, a clarinet player working on the opening gliss in Gershwin's *Rhapsody in Blue* may wish to record himself and play back the recording slowly to get a better idea of how the gliss sounds. The best jazz musicians learn by transcribing records of other performers. Transcription, which is the process of learning a song on a recording by listening to it, can be made easier if the recording is played at a slower speed.

The idea of pitch shifting is fairly straight forward. If you have a sine wave with a frequency of 440 Hz in the input audio stream, you want to replace that with an equivalent sine wave of, say 880 Hz in the output audio stream to raise the pitch by an octave. The Fourier Transform we implemented in the previous lab decomposes the audio signal into a bunch of sine waves. All we need to do, then, is change the frequency of each sine wave in the signal by shifting it into a different bin. If the magnitude of the sine wave captured by the `i`th bin is `M`, to double the frequency, set the magnitude of the output bin `2*i` to be `M`. This process is illustrated in figure 1. The upper graph shows the magnitude of the frequency components calculated by FFT. The lower graph shows how the spectrum can be changed to adjust the pitch.

Unfortunately, this method does not work well in practice because it assumes the frequencies of each sine wave in the signal are placed in the center of their bin. If we overlap the frames, however, we can make use of the phase of the frequency components as well as their magnitude to estimate the true frequency of the components and do proper pitch shifting.

Figure 2 shows an overlapped frame of an audio signal in the time domain with a single frequency component that is not centered on one of the FFT bins. The first window is framed by a mostly

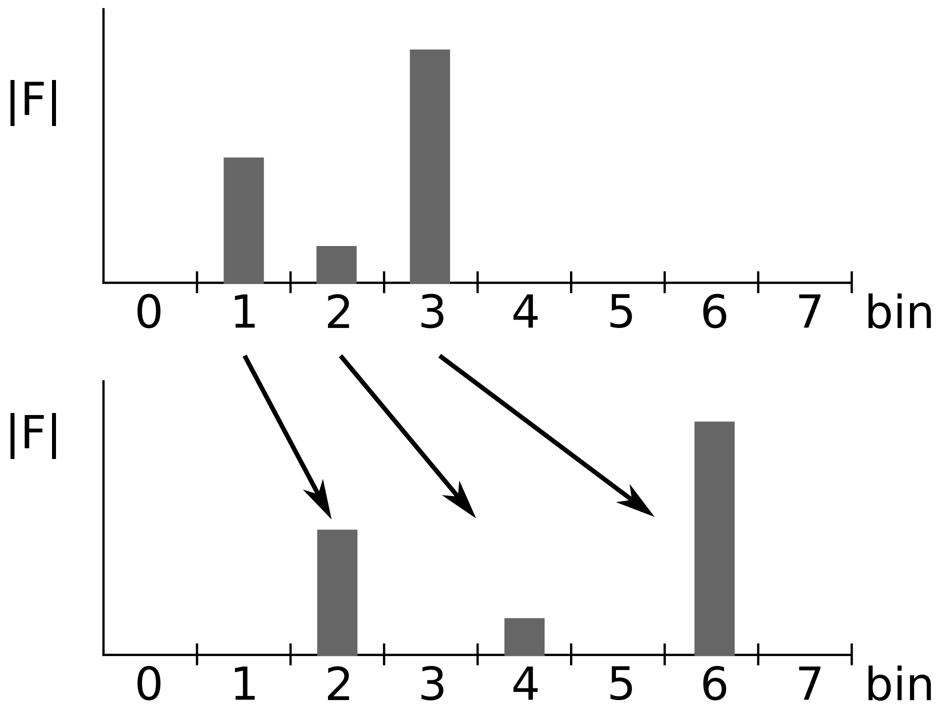

Figure 1: Illustration of naive pitch shifting

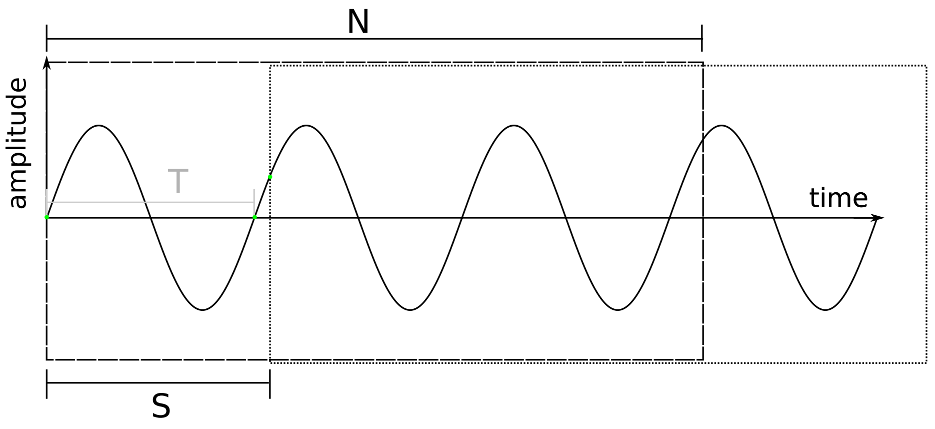

Figure 2: Overlapping Windows for Pitch Shifting

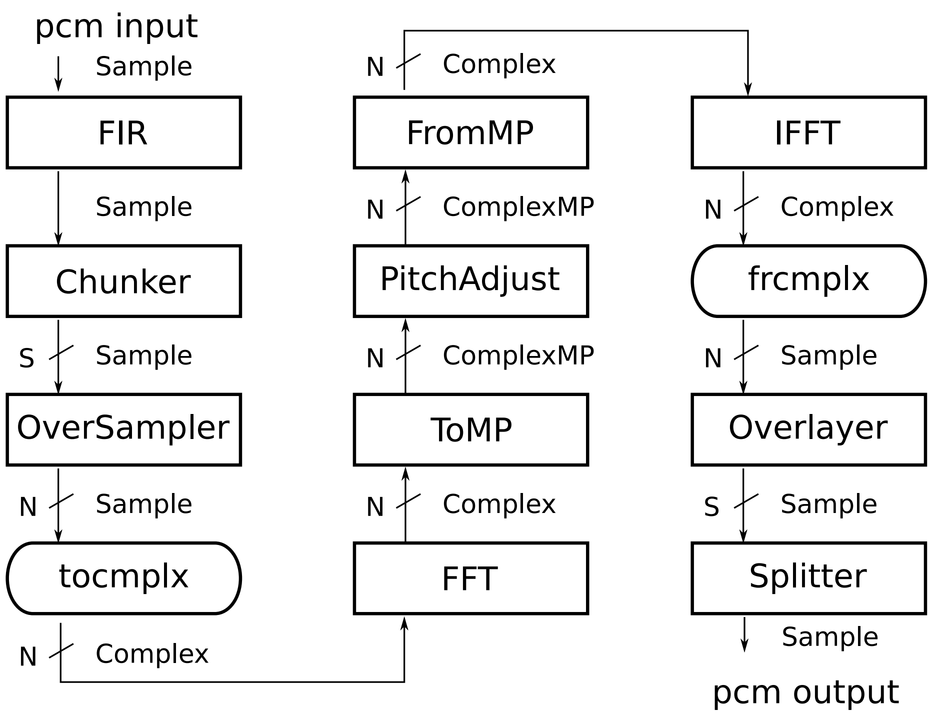

Figure 3: Pitch Shifting Audio Pipeline

solid rectangle. The next window is framed by a dotted rectangle and overlaps the first window. The phase of the frequency component in the first window is zero, because the sine wave starts at the origin. The phase of that same frequency component in the second window is a little greater than 2π. We can calculate the actual frequency of the component based on this change in phase, $d\phi$, by observing $\frac{T}{S} = \frac{2\pi}{d\phi}$, where T is the period of the sine wave in units of the sampling period. Rearranging, we arrive at:

$$f = \frac{1}{T} = \frac{d\phi}{2\pi S} \tag{1}$$

This says the actual frequency is directly proportional to the change in phase of the component from one window to the next. To accurately shift the frequencies, then, we should not only change which bin they belong to, but also shift the phase. This is the idea behind the reference C code implementation. This works well if at most a single frequency component shows up in a given bin. For this reason, using a small number of FFT points, such as only 8, will not lead to high quality output. The more points of the FFT, the better the output will sound. For example, using $N = 1024$ and $S = 64$ in the reference code leads to a good quality output stream. You'll probably run into some difficulties if you try using such a large FFT in your hardware pipeline.

1.2 The New Pipeline

Figure 3 shows the pipeline you will implement for this lab. Bluespec code for some supporting modules is available from the Lab 3 harness in the course locker and the course website.

Update your local repository with the additional Bluespec code needed for Lab 3. As with the previous labs, add the 6.375 course locker, source the setup script, navigate to the directory which contains the `audio/` folder from the previous labs, and run:

```
$ tar xf /mit/6.375/lab2019f/lab3-harness.tar.gz
```

This will add some new files to the `audio/common/` directory, create a new directory called `audio/pitch` with a stub for your pitch shifting implementation, and add a new directory called `audio/ref` with a reference C implementation of the audio pipeline. To save the changes to your local repository, run

```
$ git add audio/common/*.bsv audio/pitch audio/ref
$ git commit -m "lab3 initial checkin"
```

The new files are

`common/ComplexMP.bsv` Type for representing complex numbers in polar coordinates.

`common/Cordic.bsv` An implementation of the CORDIC algorithm to convert complex numbers between rectangular and polar representations.

`common/OverSampler.bsv` Module which implements sampling of overlapping windows.

`common/Overlayer.bsv` Module which combines together overlapping windows to produce a single audio stream.

`data/mitrib_pa8_2_2.pcm` Expected output for whole system test.

`pitch/PitchAdjust.bsv` Stub where you will implement the pitch shifting transformation.

`pitch/PitchAdjustTest.bsv` A unit test for the PitchAdjust module.

`pitch/Makefile` Makefile for building simulation executables of PitchAdjust unit test and the entire audio pipeline.

`ref/pitch.c` Reference implementation in C of the entire pitch shifting audio pipeline.

`ref/Makefile` Makefile that can be used to build the reference code.

1.3 Complex Magnitude and Phase

To perform pitch shifting you need to work with the magnitude and phase of the complex numbers output by the FFT. The Complex type in Bluespec we have been using so far represents complex numbers in rectangular coordinates, as real and imaginary components. We would prefer to work with a polar representation, which expresses a complex number in terms of its magnitude and phase. Converting from a rectangular representation to a polar representation uses arc tangent and square root computations which require specialized hardware. We have provided an implementation of the conversion between rectangular and polar representations for complex numbers using the well known CORDIC algorithm. For more information on CORDIC, see:

http://www.dspguru.com/dsp/faqs/cordic

One interesting question is how we should represent phase in hardware. In our ComplexMP type and in our implementation of CORDIC, we represent phase as a m bit integer p, such that the angle a in radians represented by the integer p is

$$a = \frac{p\pi}{2^{m-1}} \tag{2}$$

This representation forces the phase to be in the interval $[-\pi, \pi)$. Understanding this representation of phase is important for implementing the pitch shifting algorithm. For example, if m is 16, the angle $\frac{3\pi}{4}$ is represented as the integer 24576, and the angle $\frac{5\pi}{4}$ is represented as the integer -24576.

2 Implementing Pitch Shifting

In the `pitch/` directory is a file called `PitchAdjust.bsv` which contains the stub for your pitch shifting implementation. The interface for the `PitchAdjust` module is given as:

```
typedef Server#(
    Vector#(nbins, ComplexMP#(isize, fsize, psize)),
    Vector#(nbins, ComplexMP#(isize, fsize, psize))
) PitchAdjust#(
    numeric type nbins, numeric type isize,
    numeric type fsize, numeric type psize);
```

The `PitchAdjust` interface is a Server interface from the Bluespec library. Requests are a vector of `nbins` complex numbers in a polar representation. The numeric type `nbins` corresponds to the number of points of the FFT used to produce the complex numbers. The numeric type `isize` is the number of bits to use for the integer part of the magnitude of the complex numbers, `fsize` is the number of bits to use for the fractional part of the magnitude of the complex numbers, and `psize` is the number of bits to use for the phase of the complex numbers.

The `mkPitchAdjust` module starts with:

```
module mkPitchAdjust(
    Integer s,
    FixedPoint#(isize, fsize) factor,
    PitchAdjust#(nbins, isize, fsize, psize) ifc);
```

The module is polymorphic. It takes two parameters, the Integer `s` says how many samples each window is shifted from the previous window, and the parameter `factor` is what to multiply the pitch by.

The `mkPitchAdjust` module should compute the same function as the `pitchadjust` procedure in the reference C code in `ref/pitch.c`.

Problem 1: Implement the `mkPitchAdjust` module based on the reference C code. Before you start writing code, you should think about your design. Carefully consider what types to use for each operation. Should your implementation be entirely combinational, or some other kind of pipeline?

To test your pitch implementation by running the unit test provided in `pitch/PitchAdjustTest.bsv`. You can build the unit test by running

```
pitch$ make PitchAdjust
```

This will create a simulation executable `PitchAdjust` which you can execute by running

```
pitch$ ./PitchAdjust
```

The test harness will print "PASSED" if all the test cases succeed, and "FAILED" otherwise.

The `mkPitchAdjustTest` test harness works by feeding sample inputs to your pitch adjust module and comparing the result against known expected values. We have provided you with a sequence of 3 sample test vectors in `mkPitchAdjustTest`. Being able to produce test vectors and expected results yourself is a valuable skill to have when building complex digital systems.

One way to generate sample test vectors is by leveraging the reference C code, which is, by definition, correct. You can insert print statements in the reference code to print sample vectors input to the pitchadjust function and their corresponding output vectors.

Problem 2: Run the reference C code on `mitrib.pcm` to find a different sequence of 3 sample test vectors for pitch adjust. Replace the 3 test vectors in the `mkPitchAdjustTest` test harness with your new sequence, and rerun the test harness on your pitch adjust implementation. Note that the C code uses floating point numbers and your implementation uses fixed point numbers, so even if correct, your `mkPitchAdjust` module may produce results slightly different from the reference code.

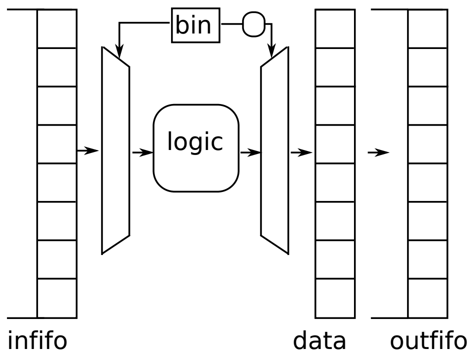

Figure 4: Possible micro-architecture for `ToMP` and `FromMP`

3 Completing the Pipeline

The last modules we need to implement for the pipeline shown in figure 3 are the `ToMP` and `FromMP` modules. The `ToMP` module converts a Vector of Complex numbers to a Vector of ComplexMP numbers, and the the `FromMP` module converts a Vector of ComplexMP numbers to a Vector of Complex numbers. The work of the conversion has already been provided in `common/Cordic.bsv`. The module `mkCordicToMagnitudePhase` converts a single Complex number to its corresponding ComplexMP. The module `mkCordicFromMagnitudePhase` converts a single ComplexMP number to its corresponding Complex number.

Problem 3: Design and implement the `ToMP` and `FromMP` modules, making use of the Cordic implementation provided. There are a couple different reasonable ways to implement these simple modules. If you are clever you may even be able to reuse the same code across both implementations, but do what is easiest for you first. The purpose of this exercise is to design your own module and write it from scratch. You may choose to use any micro-architecture for you implementation. The circular pipeline shown in figure 4 is one such example.

All that remains now is to update the audio pipeline to look like that shown in figure 3. All the individual modules have been implemented and tested (right?), it's a matter of putting the pieces together.

Problem 4: Update your audio pipeline to look like that shown in figure 3. Use $N = 8$, $S = 2$, pitch shifting factor $factor = 2$, and $psize = 16$ bits for the phase values.

Even if all of the pieces of the pipeline work correctly, the pipeline as a whole may not function correctly if there are bugs. We have provided the expected output of your pipeline for $N = 8$, $S = 2$, and $factor = 2$ when run on `mitrib.pcm` in `data/mitrib_pa8_2_2.pcm`. Note that the simulation may take a couple minutes to complete.

To build and simulate the entire audio pipeline, run the following in the `pitch/` directory:

```
pitch$ make AudioPipeline
pitch$ ./AudioPipeline
```

To analyze the area and critical path of your AudioPipeline design, run the following in the `pitch/` directory:

```
pitch$ make synthAudioPipeline
```

This might take more than 5 minutes. Which module accounts for the critical path in your pipeline?

Problem 5: Verify your pipeline as a whole works correctly by comparing its output for `mitrib.pcm` with the expected output `mitrib_pa8_2_2.pcm`.

4 Discussion Questions

1. Describe your design of the `mkPitchAdjust` module. How many cycles does it take to do a single round of pitch adjustment in your design?

2. Describe your design of the `ToMP` and `FromMP` modules. How many cycles does it take to convert a Vector of `N` Complex numbers to a Vector of `N` ComplexMP numbers assuming it takes `K` cycles to convert a single number?

3. What portion of your time in this lab was spent initially implementing the `mkPitchAdjust` module and completing the pipeline, and what portion of your time was spent designing and implementing tests and debugging the pipeline?

4. Report the synthesis result (i.e., area and critical path) of your `mkAudioPipeline` module. The report also shows you where the critical path starts and ends. Can you figure out which module accounts for the critical path?

5 What to Turn In

When you have completed the lab you should check in a final version via git. This should include the complete pipeline with your PitchAdjust, ToMP, and FromMP implementations, your updated PitchAdjustTest, and answers to the discussion questions in a file called `lab3` in the `answers/` directory.

To submit your final version first commit the new code to your local repository, then push those changes back to the git repository in the course locker. For any new files you have added that you wish to commit, you must add them to git first. For example, if you created a new file called `common/ToMP.bsv`, you should add it to git by running the command

```
audio$ git add common/ToMP.bsv
```

To commit the entire lab, do:

```
audio$ git add -u .
audio$ git add answers/lab3
-- Add all other new files with the git add command here --
audio$ git commit -m "Lab 3 submission"
audio$ git push
```