Lab 2: Fast Fourier Transforms - Extending the Audio Pipeline

6.375 Laboratory 2
Assigned: February 11, 2011
Due: February 18, 2011

1 Introduction

In this lab you will build on the work you did in Lab 1, and augment your audio pipeline with an FFT (Fast Fourier Transform) and an IFFT (Inverse FFT). The FFT transforms the audio signal from the time domain to the frequency domain, and the IFFT transforms it in the opposite direction. While you will not be asked to further modify the audio stream at this point, the FFT is an important component of any signal processor and we will make full use of it in the next lab.

We provide code for a combinational FFT microarchitecture. You will be asked to pipeline the microarchitecture using a linear pipeline and then a circular pipeline. You will then be asked to make your pipelines polymorphic. Finally we ask you how you might test your FFT implementations to verify they are correct.

1.1 Background: Fourier Transform

The `.pcm` files we use in this lab represent sound using PCM (Pulse-code Modulation). PCM is a digital representation of an analog signal created by sampling the analog signal at regular intervals (44 KHz, in our case), and storing those values as a series of signed integers. We refer to this format as being in the "time domain", since we are describing how the amplitude of the signal changes over time. Waveforms can also be represented as a summation of sinusoids of different frequencies, called a Fourier Series. We refer to this representation as being in the "frequency domain", since our encoding need only record the magnitude for each sinusoid frequency.

Often, the algorithms required to implement a particular audio manipulation on signals in the time domain are very complex, while the corresponding algorithms implementing the same manipulations in the frequency domain are far simpler. For example, consider distinguishing between pitches in a song: In the time domain, a cross-correlation between the input signal and a set of known pitch templates would be required. Computationally this is quite expensive; at the very least a linear comparison with each template is required. When implemented in hardware, it might also require a substantial amount of memory. If the waveform is converted into its frequency representation, a constant time comparison can be performed to detect a particular pitch. To decide in which domain to perform the audio manipulation, we must consider not only the cost of the audio manipulations, but also the cost of transforming between representations. For the audio manipulations you will implement in Lab 3, transforming to the frequency domain is almost certainly appropriate. Consequently, this lab focuses on designing an efficient means of converting signals to the frequency domain.

The Discrete Fourier Transform converts the time domain representation into the frequency domain. The DFT is best described as a basis transform. Recall from linear algebra that vectors in space may be represented by a linear combination of a set of orthogonal bases. We convert a vector to a different base by way of a matrix multiplication with a matrix of the new basis expressed in terms of the old basis. In the case of the DFT we simply use sinusoids of different frequencies as the basis matrix. This multiplication can be expressed by the well-known formula shown in Figure 1

The representation in Figure 1 is commonly presented in introductory signal processing texts. It should be clear that the complexity of this formula and the corresponding matrix multiplication is $O(n^2)$. This complexity can be reduced by noticing that many terms in the matrix may be represented by various combinations of other terms in the same matrix. This observation leads to the construction of the Fast Fourier Transform, an algorithm with a time complexity of $O(nlog(n))$.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i 2\pi k \frac{n}{N}} \qquad k = 0, \ldots, N-1 \tag{1}$$

Figure 1: Common DFT Form

Due to the commutativity of addition and multiplication, the terms of the matrix may be combined in many ways, leading to several important variations on the FFT. In this lab we will explore the Pease FFT, an algorithm which exhibits good parallelism while having a relatively simple construction. To allow for finer granularity in choosing the number of points for the FFT, we use a **"radix 2"** implementation instead of the **"radix 4"** implementation discussed in the lectures, but otherwise the implementation is the same as what you have already seen.

The Pease transform shown in Figure 2 permutes the signal at each step in order to coalesce values which need to be multiplied. In software, this permutation is not cheap, though in hardware, the permutation is represented as a simple rewiring of the circuit, making it essentially free to implement, except for the routing complexity of the wires.

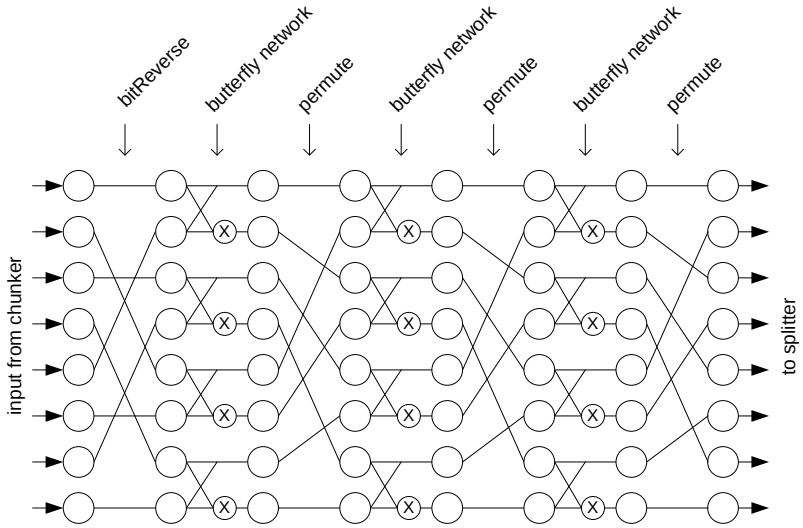

Figure 2: Dataflow of Pease transform (packets flow from left to right)

1.2 The New Pipeline

Lab 2 will build on Lab 1, so if you need to reference any material on the high-level Audio pipeline, the Lab 1 handout is still valid. We have augmented the pipeline to include an FFT, and as such are required to add a few additional modules to provide infrastructural support.

It is too expensive to compute the coefficients over the entire stream and not worth it since the additional work only captures frequencies that are too low for us to care about. Instead, we can get away with breaking the temporal stream into chunks of short-term sequences, or "audio frames", and perform an FFT on each of the frames individually. Once back in the time domain, we can reassemble these frames into a serial stream. Some of the infrastructure we have added in this lab is to support this chunking of audio samples into frames and splitting of frames back into individual audio samples.

The logical structure of our new pipeline is shown in Figure 3. The FIR filter you constructed

in Lab 1 operates on a serialized stream, after which it passes into the Chunker where the audio frames are assembled. The FFT then transforms each frame separately into the frequency domain as discussed in Section 1.1. In this lab, we will transform the stream immediately back into the time domain, and serialize the frames. There are many interesting transformations which can occur in the frequency domain, some of which we will implement in Lab 3.

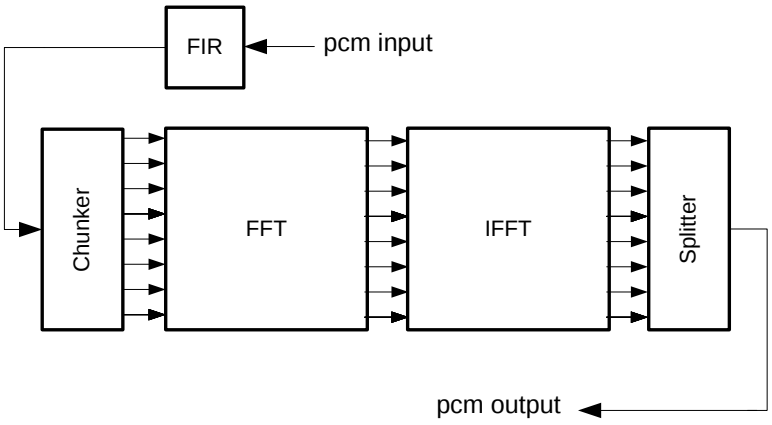

Figure 3: Lab 2 Logical Pipeline

Bluespec code for the pipeline in Figure 3 is available from the Lab 2 harness on the course locker (or website).

Update your local repository with the additional Bluespec code needed for Lab 2. Just as we did for Lab 1, you will need to add the 6.375 course locker and source the `setup.sh` script. Navigate to the directory which contains the `audio/` folder from Lab 1 and run

```
$ tar xf /mit/6.375/lab-harnesses/lab2-harness.tar.gz
```

This will add some new files to the `audio/common/` directory, update some of the existing files there, and create a new directory called `audio/fft/` containing the combinational FFT implementation. To save the changes to your local repository, run

```
$ git add audio/common/*.bsv audio/fft
$ git commit -m "lab2 initial checkin"
```

Take a moment to familiarize yourself with the general organization of the code handed out. The top level audio pipeline is implemented in a module called `mkAudioPipeline` in the file `common/AudioPipeline.bsv`. The new packages, Chunker and Splitter, make use of the Server, Get, and Put interfaces from the Bluespec library, which are essentially the same as the AudioProcessor interface we used for the FIRFilter in the previous lab. The new modules also are polymorphic, something which you will be familiar with by the end of this lab.

2 The Original FFT

Now that you have an overview of what this lab's audio pipeline looks like, we can concentrate on the module which you will be modifying, namely the FFT. Once again, we have provided you with the complete code which you will need to understand and then modify. The microarchitecture of the FFT we will begin with is shown in Figure 2, and is implemented by the module `mkCombinationalFFT` in file `fft/FFT.bsv`. Read and understand the FFT code.

We have provided a Bluespec Workstation project file, `fft/fft.bspec`, which is set up to run the full pipeline using the FFT code provided.

Problem 1: Compile, link, and simulate the pipeline using the `fft/fft.bspec` project.

1. `cd` to `fft/` and open up fft.bspec using `bluespec`.
2. Select `Build->Compile` to build the project.
3. Select `Build->Link` to link the project.
4. Copy the new sample `PCM` file `data/mitrib.pcm` to `fft/in.pcm`.
5. Select `Build->Simulate` in the Bluespec Workstation.
6. Verify the pipeline output the expected `out.pcm` by comparing that to `data/mitrib_fft8.pcm`.

3 Modifying the FFT

One major problem with the original FFT microarchitecture is the length of its critical path. In order to increase the throughput of this design, we need to shorten these wires so we can run it at higher frequencies. In this section of the lab, we will look at two different ways of optimizing the design.

The first microarchitectural modification, shown in Figure 4, will shorten the critical path substantially through the use of pipeline registers.

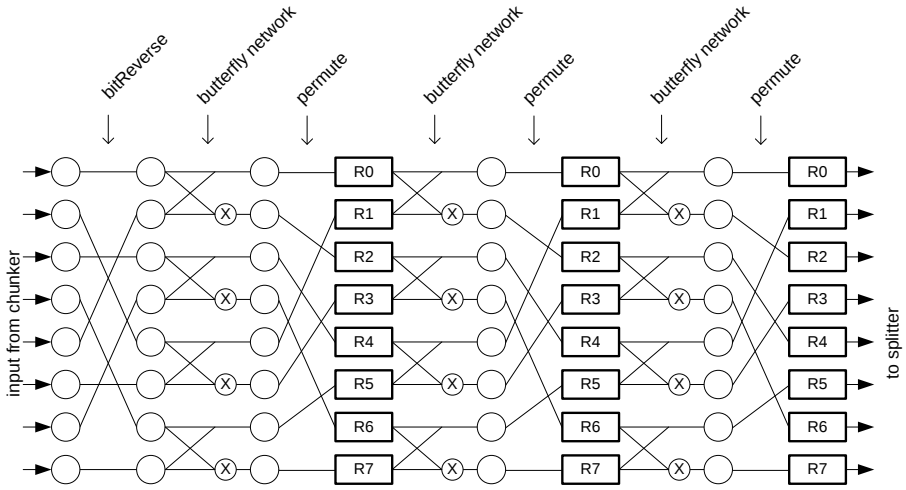

Figure 4: Linear Pipelined FFT

Problem 2: Create a new FFT module called `mkLinearFFT` which implements the microarchitecture shown in Figure 4. To have the audio pipeline use this new FFT implementation, change the implementation of the `mkFFT` module to instantiate the `mkLinearFFT` module instead of the `mkCombinationalFFT` module. Simulate the new model and verify the output is correct.

`mkLinearFFT` has a far greater throughput than the original microarchitecture, but the pipeline registers we added are quite expensive in terms of area. Figure 5 shows an alternative which will run at similar frequencies, but require far less area due to the reduced number of registers.

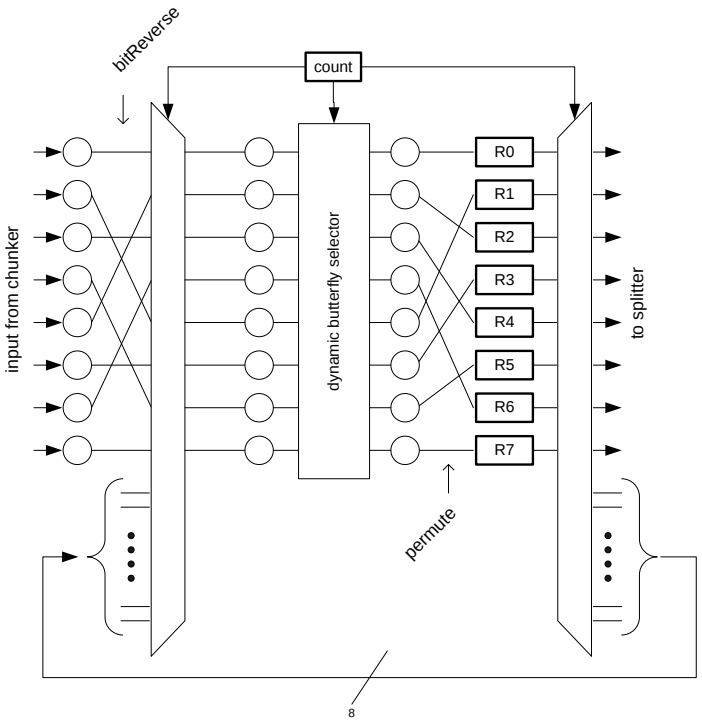

Figure 5: Circular Pipelined FFT

Problem 3: Create a new FFT module called `mkCircularFFT` which implements the microarchitecture shown in Figure 5. Update the `mkFFT` module to use the circular pipelined FFT, simulate the new model and verify the output is correct.

After you have this working, you can make a commit to your local repository to save the code so you can go back to it later in case you break something.

```
audio$ git add fft/FFT.bsv
audio$ git commit -m "Working FFT pipelines"
```

4 Making the FFT Pipelines Polymorphic

Bluespec's powerful static elaboration lets us generate a wide variety of different hardware implementations from the same source code at no cost. Modules which can be instantiated into many different hardware structures are called polymorphic. There are many ways in which a module can be polymorphic.

4.1 Module Parameters

Modules can take as input static parameters. For example, consider your FIR filter from Lab 1. The behavior of the FIR filter, and hence the hardware generated for it, depends on the values of the filter coefficients. The source code needed to describe the FIR filter, however, does not depend on the values of the filter coefficients. We can make the FIR filter work with any set of coefficients by passing the coefficients vector as a parameter to the module, rather than referencing a global vector of coefficients.

Module parameters are listed before the interface to the module. To pass the vector of coefficients to the FIR filter, the module definition would begin

```
mkFIRFilter(Vector#(9, FixedPoint#(16, 16)) coeffs, AudioProcessor ifc);
```

The vector `coeffs` can now be referenced anywhere in the module. Note, when we add module parameters this way, we have to give an explicit name for the interface, in this case `ifc`. The name will not be used anywhere, but is needed by the compiler for whatever reason.

We pass the vector of coefficients we want to use when we instantiate the module. For example,

```
AudioProcessor fir <- mkFIRFilter(c);
```

Problem 4: Change your FIR filter to take the coefficients as a module parameter. If you have done it correctly, `AudioPipeline.bsv` should import the `FilterCoefficients` and `FIRFilter.bsv` should not.

4.2 Type-Polymorphic Module Parameters

Your FIR filter can now be used to instantiate any 8 tap FIR filter, but as we saw in Lab 1, very little code needs to change if we want our FIR filter to be 16 taps, or 256 taps. By making the coefficients parameter to the module polymorphic in the length of the vector, we can use our FIR filter module to generate hardware for fir filters with different numbers of taps.

```
mkFIRFilter(Vector#(tnp1, FixedPoint#(16, 16)) coeffs, AudioProcessor ifc);
```

This says our FIR filter takes as input a vector of coefficients of length `tnp1`. That `tnp1` is a numeric type variable which will be bound to the actual length of the coefficients vector supplied when the FIR filter is instantiated. Note, `tnp1` is one more than the number of taps in the FIR filter, because we have one more coefficient than the number of taps in the FIR filter.

Working with numeric types in Bluespec can be a little confusing. A numeric type is a `type` in Bluespec, not an `Integer`. This means we can not use a numeric type where an `Integer` is expected without an explicit conversion. The `valueof` function will make that conversion for us.

For example, if we have a for loop iterating over an `Integer` value, we will need the `valueof` function to compare that `Integer` to the numeric type.

```
for (Integer i = 0; i < valueof(n); i = i+1)
    ...
```

There is no way to convert an `Integer` to a numeric type. A consequence of this is we need special functions to do things like addition and subtraction of numeric types. The function `TSub` can be used to perform subtraction of two numeric types. For example,

```
Integer numtaps = valueof(TSub#(tnp1, 1));
```

Other useful operations on numeric types are `TAdd`, `TDiv`, and `TLog`.

Problem 5: Change your FIR filter to take a variable length vector of coefficients as a module parameter. What do you have to do now to instantiate a 4 tap FIR filter? A 256 tap FIR filter? Will your FIR filter behave correctly in those cases?

4.3 Polymorphic Interfaces

Another way we can make modules polymorphic is by making their interfaces polymorphic. Consider now, for example, the FFT interface.

```
typedef Server#(
    Vector#(FFT_POINTS, ComplexSample),
    Vector#(FFT_POINTS, ComplexSample)
) FFT;
```

It is another name for the `Server` interface from the Bluespec library with a specific number of points. We can make the number of points in the FFT interface a parameter so that FFTs with different numbers of points can reuse the same interface definition. To do this we would change the FFT interface to

```
typedef Server#(
    Vector#(fft_points, ComplexSample),
    Vector#(fft_points, ComplexSample)
) FFT#(numeric type fft_points);
```

The number of points is now a parameter to the FFT interface. When we define our FFT module, we specify the number of points it implements.

```
module mkCombinationalFFT (FFT#(FFT_POINTS));
```

To instantiate the module:

```
FFT#(FFT_POINTS) fft <- mkCombinationalFFT();
```

But, of course, we don't want our FFT modules to only work for a single number of points. We can make our module polymorphic, just like we made the FIR filter polymorphic in the number of taps.

```
module mkCombinationalFFT (FFT#(fft_points));
```

The numeric type variable `fft_points` is the number of points our FFT module should be instantiated with.

Functions can also be made polymorphic by using type variables. For example, the signature of the bitReverse function can be changed to

```
function Vector#(fft_points, ComplexSample)
    bitReverse(Vector#(fft_points, ComplexSample) inVector);
```

Where the lowercase `fft_points` is a numeric type variable which will be set to the appropriate number of points based on how the function is called.

Problem 6: Change all of the FFT implementations, and the IFFT implementation to be polymorphic in the number of points. You can assume the number of points will be a power of 2. Note, this requires many more changes than the FIR filter module required.

At some point in your changes, the Bluespec compiler will complain to you about provisos. The `stage_ft` function, it complains, needs an additional proviso `Add#(2, a__, fft_points)`. What is that all about?

The way the `stage_ft` function is implemented we must impose additional restrictions on the type parameters that are allowed. Specifically, we use the `takeAt` function to extract two elements from the `stage_in` vector. This only makes sense if the `stage_in` vector has two elements. We use provisos to specify explicitly these sorts of assumption our code depends on. The function signature should be changed to something like

```
function Vector#(fft_points, ComplexSample)
    stage_ft(
        TwiddleTable#(fft_points) twiddles,
        Bit#(TLog#(TLog#(fft_points))) stage,
        Vector#(fft_points, ComplexSample) stage_in)
    provisos(Add#(2, a__, fft_points));
```

The `provisos` keyword introduces a list of the assumptions our implementation makes. The proviso `Add#(2, a__, fft_points)` says the numeric type 2 plus some other numeric type, call it `a__`, must add up to `fft_points`. Because numeric types can't be less than zero, this means `fft_points` must not be less than 2.

Provisos can be applied to modules in the same way they are applied to functions. For example,

```
module mkCombinationalFFT (FFT#(fft_points))
    provisos(Add#(2, a__, fft_points));
```

You should now have enough information to finish Problem 6.

There are many different kinds of provisos you can specify. Fortunately, the compiler is very good at informing you of what provisos it expects your functions and modules to have. Often you can copy its suggestions directly to your code, but do take the effort to understand why the proviso is needed. Are you making more assumptions in your code than you expect?

To get more practice with provisos, and make the FFT modules useful in even more cases, add another type parameter to the FFT interface which is the type of Complex number it can work on.

```
typedef Server#(
    Vector#(fft_points, Complex#(cmplxd)),
    Vector#(fft_points, Complex#(cmplxd))
) FFT#(numeric type points, type cmplxd);
```

This will expose us to some different kinds of provisos.

Arith#(t) Asserts the given type `t` has `+`, `*` and other arithmetic operations defined for it.

Bits#(t, t_sz) Asserts the type `t` can be converted to a bit representation using the `pack` and `unpack` functions. The number of bits needed is `t_sz`.

Bitwise#(t) Asserts the type `t` has bitwise operations, such as bit shifting defined for it.

RealLiteral#(t) Asserts you can convert the Real type to the given type `t`.

Each of these provisos asserts the type `t` belongs to the named typeclass. You can consult the Bluespec reference guide for more information on typeclasses.

Problem 7: Update all the FFT modules to work for any kind of complex data. Many of the functions and modules will require additional provisos.

After you have made this change, the FFT code is no longer dependent on the AudioProcessor types. It can be reused in other projects as is. To verify your changes are complete, remove the import of AudioProcessorTypes from FFT.bsv, run the pipelines, and verify they still compile and work as expected.

Now that you are familiar with polymorphic modules, revisit the Chunker and Splitter modules in the `common/` directory. Do they make any more sense to you now?

5 Discussion Questions

1. Before you made your FFT implementations polymorphic, the number of points and data type was specified with global typedefs. What are the advantages of making the interface and modules polymorphic instead of just using typedefs? Are there any disadvantages to making the interfaces and modules polymorphic?

2. Now that your FFT can be fully separated from the AudioPipeline, it makes sense to test the FFT modules independently. Describe in detail (two or three paragraphs is appropriate) how you can test your FFT implementations independently from the rest of the AudioPipeline. Are your FFT implementations correct for both larger and smaller number of points than 8? Is it any easier to test the FFT implementations because they are polymorphic?

6 What to Turn In

When you have completed the lab you should check in a final version via git. This should include the Bluespec implementation of the polymorphic combinational, linear, and circular FFT pipelines, the polymorphic FIR filter, and answers to the discussion questions in a file called `lab2` 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 example, you could run:

```
audio$ git add -u .
audio$ git add answers/lab2
audio$ git commit -m "Lab 2 submission"
audio$ git push
```