Lab 2: Fast Fourier Transforms - Extending the Audio Pipeline

6.375 Laboratory 2
Assigned: February 12, 2010
Due: February 19, 2010

1 Introduction

In this lab you will build on the work you did in Lab1, and augment your audio pipeline with an FFT (Fast Fourier Transform) and an IFFT (Inverse FFT). The FFT transforms the signal from the time domain to the frequency domain, and the IFFT transforms it in the opposite direction. While you won't 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. We then ask you to parameterize the circular pipeline by the number of points in the FFT and the data type of the samples.

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 $O(nlog(n))$.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}} \qquad k = 0, \dots, N-1.$$

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.

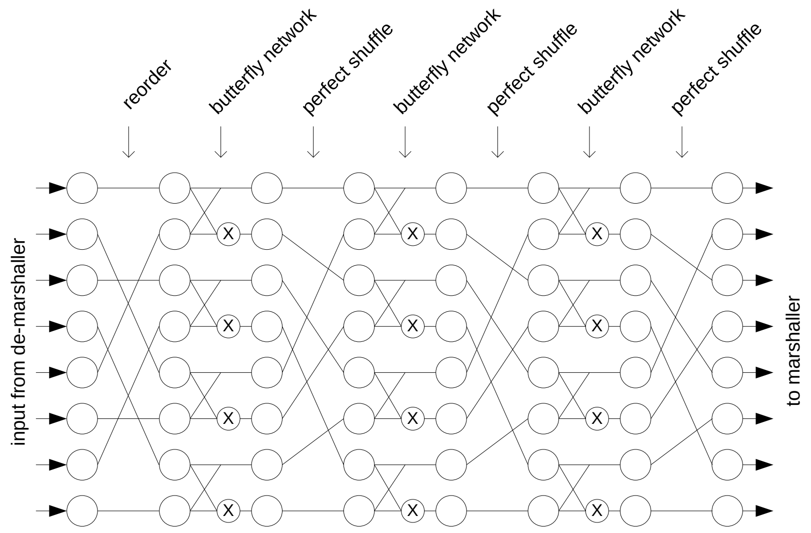

Figure 2: Dataflow of Pease transform (packets flow from left to right)

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 a series of short-term sequences, or "audio frames" (a process known as de-marshalling), and perform an FFT on each of the blocks individually. Once back in the time domain, we can reassemble, or marshall, these frames into a serial stream. Some of the infrastructure we have added in this lab is to support this de-marshalling and marshalling.

Because we assemble the serial stream into audio frames, there is always the danger that the number of PCM packets in the stream is not an exact multiple of the frame size. To handle this case, we may need to pad the stream with null tokens to fill out the last frame, and remove these tokens once the frame has been marshalled. The new pipeline also contains modules to perform this work.

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 through the padding module, into the de-marshaller 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, marshall the frames, and remove buffered tokens where they have been added. There are many interesting transformations which can occur in the frequency domain, some of which we will implement in Lab 3.

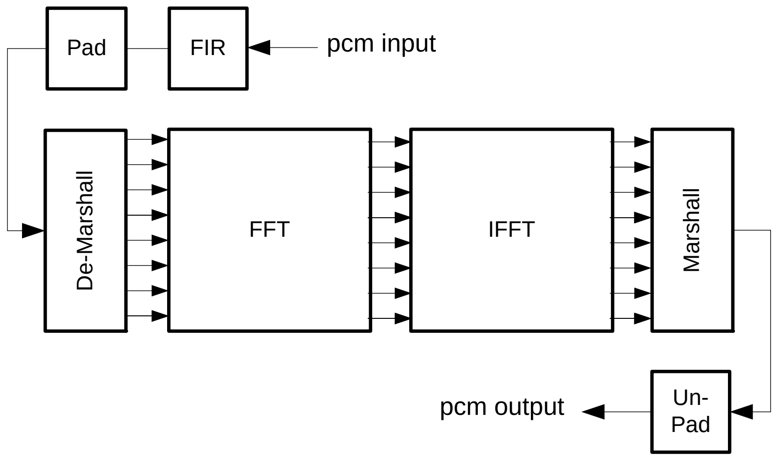

Figure 3: Lab 2 Logical Pipeline

While implementing the pipeline in Figure 3 would be ideal, the FPGA we use starting in Lab 4 may not be large enough to accommodate two FFT blocks. Luckily the exact same function which transforms the frames form the time to the frequency domain can be used to transform frames in the opposite direction. We can exploit this fact to multiplex the use of a single FFT module to perform both transformations. This gives the circular pipeline shown in Figure 4. One MUX and one de-MUX take far less area than one FFT block, which should allow us to fit this design on the FPGA in the later lab.

Bluespec code for the pipeline in Figure 4 is provided with the lab2 harness, with the exception of the FIR filter, which we will take from lab1.

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

   ```
   % tar -xzvf /mit/6.375/lab-harnesses/lab2-harness.tar.gz
   % svn add lab2
   ```

   This will create a directory called `lab2/` with some new Bluespec code added to the `common/` directory as well as an `fft/` directory containing the combinational FFT implementation.

2. Copy your `multfir` filter from lab1 to `fir/FIRFilter.bsv`, renaming the module `mkFIRFilter`. You can do this all with the following commands.

   ```
   % cd lab2
   lab2% cat ../lab1/multfir/AudioPipeline.bsv | \
           sed -e 's=mkAudioPipeline=mkFIRFilter=' > fir/FIRFilter.bsv
   ```

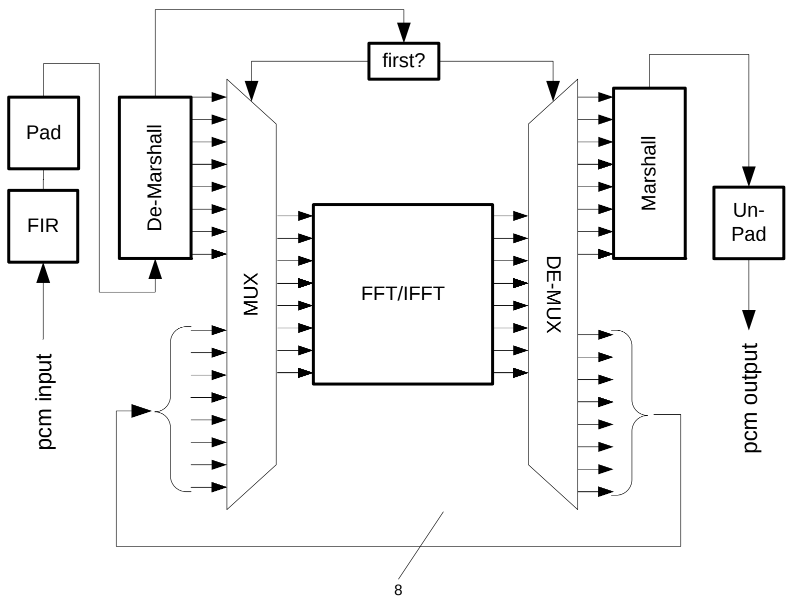

Figure 4: Lab2 Hardware Pipeline

Add your FIR filter into subversion too.

```
lab2% svn add fir/FIRFilter.bsv
```

3. Check the code into subversion. From the lab2 directory run

```
lab2% svn ci . -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`.

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 `mkFFT` 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 sample `PCM` file in `data/foo.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/foo_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 5, will shorten the critical path substantially through the use of pipeline registers.

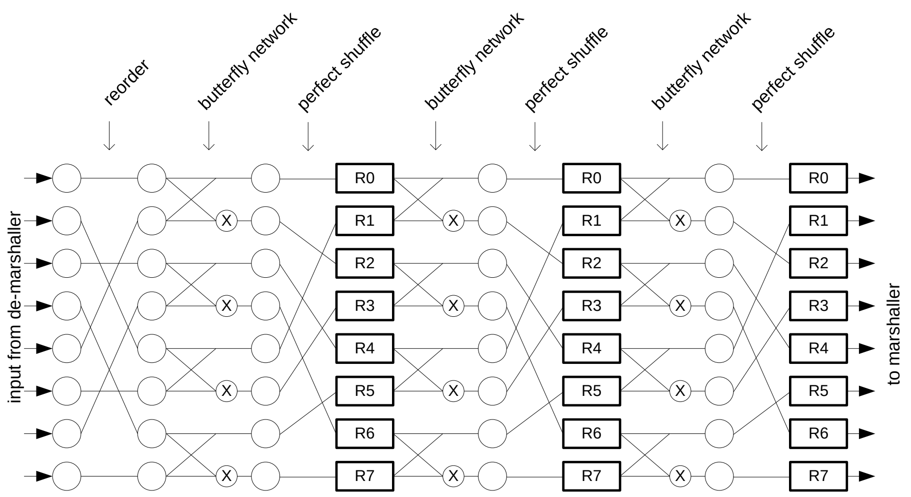

Figure 5: Linear Pipelined FFT

Problem 2: Create a new FFT module based on the original FFT module, changing the microarchitecture shown in Figure 2 to that shown in Figure 5. Simulate the new model and verify the output is correct.

1. From within the `lab2/` directory create the directory `lpfft/`.
2. Copy the original FFT Module file and Bluespec Workstation project to the directory `lpfft/`
3. Modify the BSV code in `lpfft/FFT.bsv` to reflect the microarchitecture shown in Figure 5.
4. Compile and Simulate the new FFT code using the `lpfft/fft.bspec` project. Remember to copy the `data/foo.pcm` file to `lpfft/in.pcm` before simulating.
5. Verify the output `lpfft/out.pcm` matches `data/foo_fft8.pcm`.

`FFTLinearPipeline.bsv` has a far greater throughput than the original microarchitecture, but the pipeline registers we added are quite expensive in terms of area. Figure 6 shows an alternative which will run at similar frequencies, but require far less area due to the reduced number of registers.

Problem 3: Create a new FFT module based on the original FFT module, changing the microarchitecture shown in Figure 2 to that shown in Figure 6. Simulate the new model and verify the output is correct.

Follow the instructions used to create the linear pipeline to create a new circular pipeline. Put the circular pipeline in the directory `cpfft`.

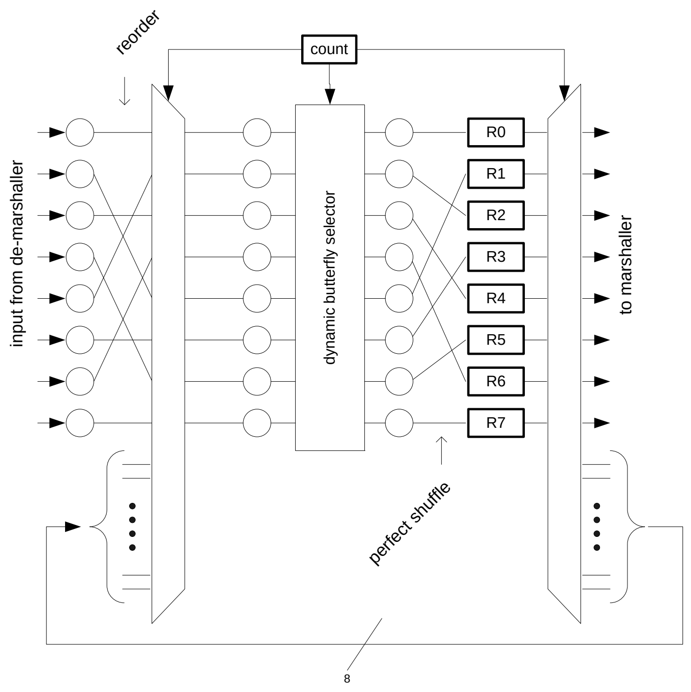

Figure 6: Circular Pipelined FFT

4 Parameterizing the FFT Circular Pipeline

The FFT code we wrote describes an 8 point FFT that operates on `Complex#(FixedPoint#(16, 16))` samples. We may wish to use a 16 point FFT, or operate on some other kind of samples. It would be nice if we did not have to completely reimplement the FFT whenever we want to use a different number of points or sample type.

Our FFT code is already written somewhat parametrically because we use the macro `NUM_POINTS` and have a typedef for the `FFT_DATA` type. We can do better by changing the `FFT` interface to take type parameters and changing the `mkFFT` to use those type parameters rather than refer to some global definition. This will allow us to instantiate multiple FFT modules in our design simultaneously with different parameters all from the same source code.

Our new `FFT` interface looks like

```
interface FFT#(numeric type points, type data);
  method Action fftInput(Vector#(points, data) inVector);
  method ActionValue#(Vector#(points, data)) fftOutput();
endinterface
```

It takes two type parameters. The parameter `points` is a numeric type indicating the number of points the FFT uses, and the parameter `data` is the type of the sample data the FFT works with. To instantiate the FFT module now we might say

```
    FFT#(8, Complex#(FixedPoint#(16, 16))) fft <- mkFFT();
```

If we wanted a 16 point FFT we could change that to

```
    FFT#(16, Complex#(FixedPoint#(16, 16))) fft <- mkFFT();
```

Our `mkFFT` module definition also changes to accept the new interface.

```
module mkFFT (FFT#(points,Complex#(cmplxd)))
  provisos(Log#(points, log_points),
           Add#(2, _, points),
           Arith#(cmplxd),
           RealLiteral#(cmplxd),
           Bits#(cmplxd,cmplxd_sz));
```

This says our `mkFFT` module implements an FFT parameterized by the number of points, and using `Complex` data of some sort. The type variable `points` is a numeric type describing the number of points to instantiate which we can refer to in our implementation, and `cmplxd` is a type variable which tells us what type of `Complex` data to use.

The way our `mkFFT` module is implemented we must impose additional restrictions on the type parameters that are allowed. For example, we use the `*` and `/` operators on the complex data type, so we can only support data types which have those arithmetic operations implemented for them. We specify this using the `Arith` proviso, which asserts that `cmplxd` belongs to the `Arith` type class.

Another type of proviso is for specifying type relationships. We will want to refer to the log of the number of points in our implementation. The `Log` proviso gives us a convenient way to define `log_points` as the log of `points`. We also require the number of points to be greater than 2, so we use the `Add` proviso, which requires the first two arguments sum to the third. In this case we use the wildcard _ as the second argument to indicate we do not care how much greater `points` is than 2, just so long as there exists some nonnegative numeric type which summed with 2 is `points`.

The `RealLiteral` proviso means we can convert real literals in Bluespec to the type `cmplxd`. The `Bits` proviso asserts the `cmplxd` type can be stored in a state element using `cmplxd_sz` bits, which Bluespec will set for us based on the specifc type for `cmplxd` used.

Problem 4: Parameterize the FFT interface and module for the circular pipeline in `cpfft/FFT.bsv`

1. Change the `FFT` interface to the parametric version shown above.
2. Change the `mkFFT` module to use the parametric `FFT` interface and add provisos as shown above.
3. Remove the `` `define`` and typedef of `NUM_POINTS`, `LOG_NUM_POINTS`, `COMPLEX_DATA` and `FFT_DATA`, and replace their uses in the `mkFFT` module using the new type variables provided from the provisos and `mkFFT` interface specification.

   Section 4.1 on working with numeric types may be helpful to you in this process.
4. Try running your new pipeline with different numbers of points and verify they still work. You can specify the number of points by changine the `` `define FFT_POINTS`` in `common/AudioPipeline.bsv`. We have provided the expected output for 4, 8, and 16 points at `data/foo_fft4.pcm`, `data/foo_fft8.pcm` and `data/foo_fft16.pcm` respectively.

4.1 Working with Numeric Types

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(points); i = i+1)
    ...
```

Remember also that we can not put an `Integer` in a register in Bluespec, because `Integers` are unbounded. Instead we may want to use a `Bit#(n)` or `int`. Just as before, a `Bit#(8)` in Bluespec has a different type from `Integer`, so we need an explicit conversion to use an `Integer` where an `Bit#(8)` is expected. That conversion is done with the `fromInteger` function.

```
Integer n = 4;
Bit#(8) asbits = fromInteger(n);
int asint = fromInteger(n);
```

An interesting consequence of this is what happens when you want to convert a numeric type to bits. You must first call the `valueof` function to convert to an `Integer`, then call the `fromInteger` function to convert to the bits.

```
numeric type n = 4;
Bit#(8) asbits = fromInteger(valueof(n));
```

If you forget to make the conversion, the Bluespec compiler might give an error such as

```
Unbound variable ‘n’
```

because `n` is not a variable, it's a type.

5 What to Turn In

When you have finished you need to check your code into your subversion repository. The code should include the original `fft/FFT.bsv` and `fft/fft.bspec` which runs successfully using your FIR filter from lab1. The FIR filter from lab1 should be checked in at `fir/FIRFilter.bsv`. You should also include the linear pipeline `lpfft/FFT.bsv`, and the parametric circular pipeline `cpfft/FFT.bsv`. This can be accomplished from the `lab2/` directory by running

```
lab2% svn add --parents fir/FIRFilter.bsv {lp,cp}fft/*.{bsv,bspec}
lab2% svn ci -m "Lab2 final submission"
```