Lab 2: Fast Fourier Transforms - Extending the Audio Pipeline

4541.763 Laboratory 2
Assigned: September 24, 2009
Due: October 8, 2009

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.

Lab 2 will follow a form you are familiar with from Lab 1: you will be given the code for an FFT microarchitecture, along with the corresponding diagrams and a detailed explanation. From this starting point, we will then guide you through various architectural modifications, similar to those discussed in Lecture four. The last exercise will begin to teach you how to write Bluespec in a more functional style.

1.1 Background: Fourier Transform

The `.wav` files we are using in this lab represent the 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 transformation on signals in the time domain are very complex, while the corresponding algorithms implementing the same transformation 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 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. In order to decide in which domain to perform the transformation, we must compare not only the corresponding transformations, but also the cost of transforming between representations. For the transformations you will implement in Lab 3, the answer is almost certainly "yes". For this reason, it is worth our time to design 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 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 $O(nlog(n))$.

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

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. It should be noted that the Pease FFT we will be working with is a **"radix 2"** implementation, unlike the example discussed in the lecture, which was **"radix 4"**.

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.

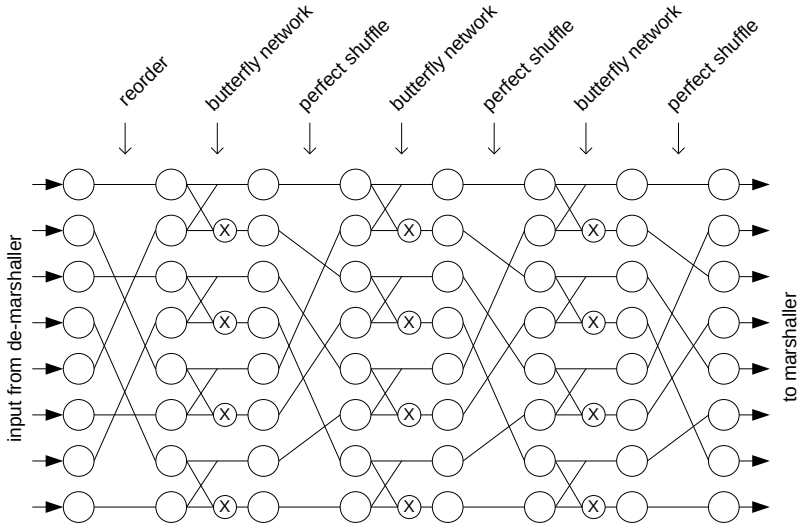

Figure 2: Dataflow of Pease transform (packets flow from left to right)

1.2 The New Pipeline

Lab 2 will build on Lab 1, so if you need to reference any material on the high-level Hardware/Software interaction, 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.

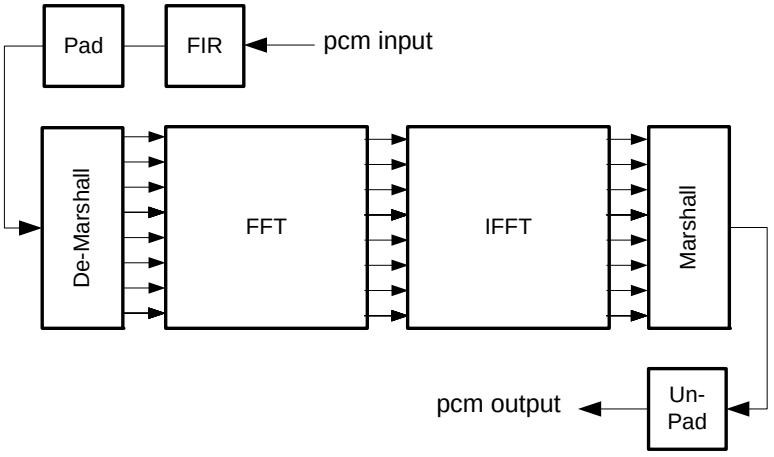

Figure 3: Lab 2 Logical Pipeline

While implementing the pipeline in Figure 3 would be ideal, the XUPv2 is not 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, allowing us to fit this design on the XUPv2.

We have provided all the Bluespec code for the pipeline in Figure 4 and you should take a moment to familiarize yourself with the general organization. The pipeline is implemented in a module called `mkAudioPipeline` in the file `FFTPipeline.bsv`. This file has been replicated in Section 5, with additional commentary.

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.bsv`. Once again, a sanitized version of this file appears in Section 6. **Before proceeding with this lab, make sure that you have read and understood this code**.

Problem 1: Create a copy of the original FFT pipeline, configured to run in simulation. Compile this model, run the benchmarks, and check it into your personal repository:

1. `cd` to `workspaces/labs/src/mit-6.375-<yourname>/config/pm/mit-6.375-<yourname>` and create the directory `lab2`. This is where you will store all the models we will create in this lab.

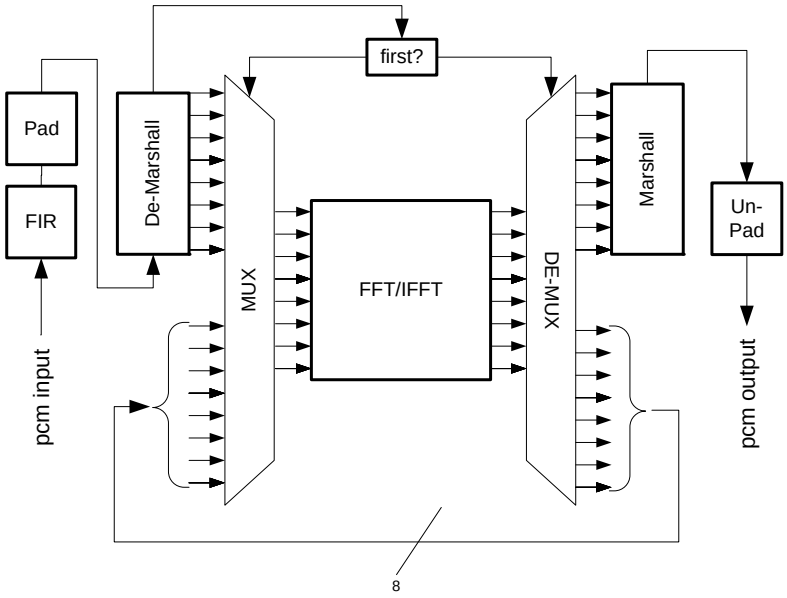

Figure 4: Lab2 Hardware Pipeline

2. Start awb from the command line (remember to run `kinit, aklog`, and `source setup.sh`) and select the "Models" tab.

3. Navigate down the Model Directories hierarchy as follows: `Models` ⇒ `mit-6.375` ⇒ `fft`, and double-click on the model named "fft_exe".

4. Once the apm editor has loaded, save the model in the directory you created in the first step as `fft_exe`

5. Configure and build the model (you should be familiar with this process after completing Lab1)

Since an FFT followed by an IFFT is effectively a nop, the only transformation we are performing on the audio stream is the FIR filter, allowing us to reuse `fir_benchmark.cfx` from the `audio_processor_test` introduced in Lab 1. You should be able to run both `null.wav.cfg` and `reuben_james_1sec.wav.cfg` and see them pass.

Problem 2: Create another copy of the original FFT pipeline, this time configured to run on the FPGA. Compile this model, run the benchmarks, and check it into your personal repository:

1. Open the model which you just created (`Models` ⇒ `mit-6.375-<yourname>` ⇒ `lab2 : fft_exe`), and save a copy as "fft_xup" in the same directory.

2. Change the fpgaenv to use the "Hybrid XUPv2 Serial FPGA Environment" alternate module (be sure to select "yes" in the dialog box asking if you want to use default parameters).

3. Due to the length of the critical path, this FFT can only be run at 10 MHz. Set the parameters of the "Hybrid XUPv2 Serial FPGA Environment" you selected in the previous step to reflect this target frequency (review the Lab 1 handout if you forget how this is done).

4. Configure and build the model you have just created, and run the fir_benchmark.cfx tests, making sure that you are connected to the FPGA board with both the USB and Serial cables.

Remember to add the two new models you just created to source control and to check them in as well.

Errors related to setting the timing parameters are sometimes a bit hard to interpret, but over the years, FPGA synthesis tools have been getting better at producing a meaningful message. In general, if you set a target frequency which the tool cannot reach, it will do one of two things: either it will tell you that it couldn't place and route your design at the desired frequency, or it will fail with a cryptic message.

3 Modifying the FFT

One major problem with the original FFT microarchitecture is the length of its critical path, which can only be clocked at 10 MHz. 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. Both these techniques were discussed in Lecture 4; **which you should review before continuing**.

The first microarchitectural modification, shown in Figure 5, will shorten the critical path substantially through the use of pipeline registers.

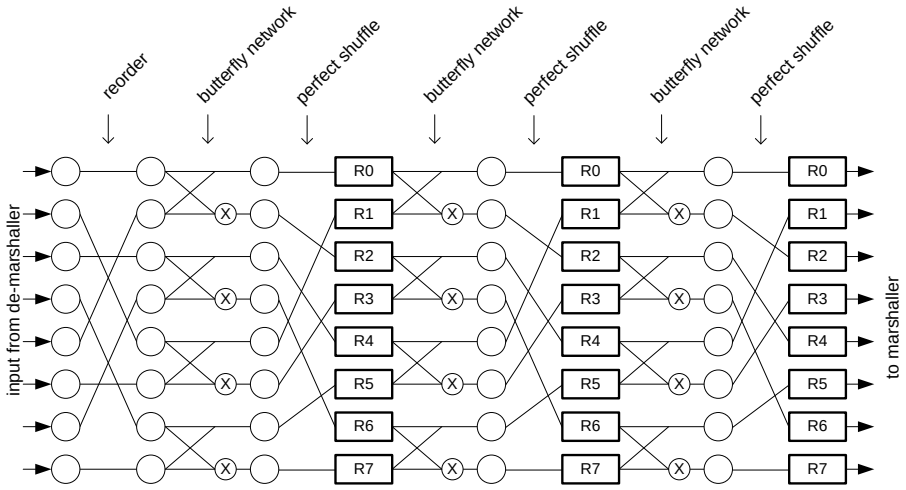

Figure 5: Linear Pipelined FFT

Problem 3: Modify the original FFT module, from the microarchitecture shown in Figure 2 to that shown in Figure 5. Using this new module, create two new FFT Pipeline models, one to run in simulation, and one for the FPGA. After testing them, check the new models (`.apm` files), and new modules (`.awb` and `.bsv` files) into your personal repository.

1. First create the directory ∼`/workspaces/labs/src/mit-6.375-<yourname>/modules/mit-6.375-<yourname>/lab2`. We will use this directory to store all the new source files we create in this lab.

2. Copy the original FFT Module files (∼`/workspaces/labs/src/mit-6.375/modules/bluespec/mit-6.375/lab2/FFT.bsv, fft.awb`) into the new `lab2` source directory and rename then `FFTLinearPipeline.bsv`, and `fft_linear_pipeline.awb` respectively.

3. Edit the contents of `fft_linear_pipeline.awb` to point to the correct `.bsv` file (`%public` field) and to contain a descriptive name (`%name`) which will be displayed in the "Alternative Modules" window.

4. Create copies of `fft_exe` and `fft_xup` (from Section 2), naming them `fft_linear_exe` and `fft_linear_xup`. Edit these new models to use the new FFT module you created in the previous step, and save them in your lab2 models directory.

5. Modify the BSV code in `FFTLinearPipeline.bsv` to reflect the microarchitecture shown in Figure 5.

6. Debug the code in `FFTLinearPipeline.bsv` in simulation using `fft_linear_exe`, and then run it on the FPGA using `fft_linear_xup`. Play with the target clock frequency for `fft_linear_xup`. You should be able to achieve a much higher clock frequency than 10 MHz.

7. Record the clock frequencies you were able to achieve. As well as reported area

Remember that although the diagrams all show an FFT with eight points, the number of points in the FFT is really a parameter of the module, and consequently you MUST write parametric code. To make sure that your BSV is correctly parametrized, change the `FFT_POINTS` parameter of the `audio_pipeline_types` sub-module of the `audio_pipeline` module and verify that your design runs correctly at a few different settings (they must be powers of two). When grading your labs, we will compile and run your solutions with a few different settings. Bear in mind that due to area restrictions, you might not be able to fit a large FFT on the FPGA, but for the purposes of this lab, you should get at least an 8 point FFT to fit.

Another aspect to keep in mind when pipelining your design is that you cannot make any assumptions about the number of audio-frames "in flight" at a particular time. The temptation when designing the linear pipeline is to use the incoming frames to "push" previous frames through the pipeline. A design like this will break if the high-level control expects to extract the transformed frame before submitting another. One might argue that using the linear pipeline in this manner is a waste, since you could, instead, have created a multicycle implementation with far fewer registers. This is a valid argument, but due to the external buffering requirments imposed by this (possibly better) usage scenario, and the ramifications on FPGA use, we have decided against it.

`FFTLinearPipeline.bsv` has a far greater throughput than the original microarchitecture, but that quantity of 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.

The circular pipeline produced by multiplexing the use of the "phase" logic will have a much lower throughput, even though we may be able to clock it at the same frequency as its linear counterpart. On the other hand, the dynamic selection of the butterfly logic may remove optimization opportunities for the synthesis tools. It should be noted that the the example given in the Lecture four compared the improvement in area due to this style of folding in the context of ASIC synthesis (about 10%). Typically, FPGA synthesis tools are slightly less aggressive in types of optimizations (constant propogation, boolean optimization, etc.) which suffer when dynamic selection is instroduced. Another difference is that the relative costs of routing in FPGAs is much higher while the cost of MUXing in FPGAs is much lower. Consequently, we may see a more substantial improvement in area when folding pipelines on FPGAs.

Problem 4: Modify the original FFT module, from the microarchitecture shown in Figure 2 to that shown in Figure 6. Using this new module, create two new FFT Pipeline models, one to run in simulation, and one for the FPGA. After testing them, check the new models (`.apm` files), and new modules (`.awb` and `.bsv` files) into your personal repository.

Follow the instructions used to create the linear pipeline to create a new circular pipeline. Name the source file `FFTCircularPipeline.bsv`, the module file `fft_circular.awb`, and the two models `fft_circular_exe` and `fft_circular_xup`. Be sure to add the new files you create to source control. Edit the contents of `FFTCircularPipeline.bsv` to implement the microarchitecture shown in Figure 6.

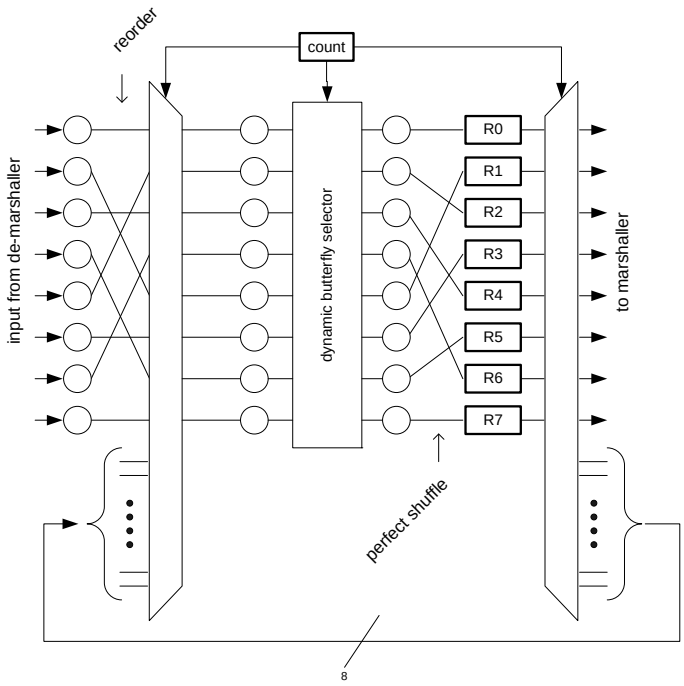

Figure 6: Circular Pipelined FFT

4 Functionalizing Common Patterns

In the first lab, we had you write down a FIR filter in Bluespec. You may have noticed that there were some simple patterns reflected in the design. For instance, you have a set of registers representing the history of samples. Each time we get a new sample, we shift each value into the next register in the line. Further, we take this series of sample history and multiply it point-wise with another series of FIR coefficient values.

In many imperative languages like C or Java, these sorts of patterns are commonly represented using for-loops. In fact the initial representation of our FIR filter did exactly this. However, in functional languages it is often more natural to represent these using higher-order functions (often called *combinators*), or functions which take functions as arguments. These functions represent just the "shape" of the computation, not the details of the computation which is represented in the function argument.

This may seem like a needless abstraction, but in practice this sort of representation gives the designer cleaner code and the compiler a better starting point for optimizations as combinators are an explicit representation of common data patterns which the compiler may want to optimize. To give you a better understanding we are going to rewrite the initial FIR filter from the previous lab in this style.

4.1 Common Combinators

This section lists some common combinators used frequently in well-written BSV code, along with small examples of their use.

- **map**: Map applies a function to each value in a vector, producing a new vector.

```
function Vector#(n,b) map(function b f(a x), Vector#(n, a) in);
    Vector#(n, b) out;
    for(Integer i =0; i < valueOf(n); i = i + 1)
      begin
        out[i] = f(in[i]);
      end
    return out;
endfunction
```

Suppose you have defined a function "`fn`" which takes some type `a` to some other type `b`, and vector "`vec`" of values of type `a` (`Vector#(n,a)`), the expression `map(fn,vec)` is a vector of type `Vector#(n,b)`) where each element `v[i] = fn(vec[i])`.

- **zipWith**: Another common operation is to point-wise merge two vectors into a single vector. This function (zipWith) can be written as:

```
function Vector#(n,b) zipWith(function c f(a x, b y),
                              Vector#(n, a) in1, Vector#(n,b) in2);
    Vector#(n, b) out;
    for(Integer i =0; i < valueOf(n); i = i + 1)
      begin
        out[i] = f(in1[i],in2[i]);
      end
    return out;
endfunction
```

Suppose you have defined a function "`fn`" which takes two arguments (one of type `a`, one of type `b`) and returns something of type `c`, and two vectors `veca` of type `Vector#(n,a)`, and `vecb` of type `Vector#(n,b)`, then the result will be of type `Vector#(n,c)` and the ith element will be equal to `fn(veca[i],vecb[i])`.

- **fold**: The fold function is used when you want to merge all the values in a vector into a single result. The natural way to write this in the for-loop style would be something like:

```
function a fold(function a join(a x, a y), Vector#(n, a) in);
  a accum = in[0];
  for(Integer i = 0; i < valueOf(n); i = i + 1)
    begin
      accum = join(accum, in[i]);
    end
  return a;
endfunction
```

However, this is a linear operation. You each loop iteration depends on the result of the previous iteration. Thus when we unroll the computation to generate circuitry we end up building a linear circuit. If know that the join function associative like addition, *i.e.*,$a+(b+c) = (a+b)+c$. Therefore there's we can't do the addition in parallel.

But for any associative join function we can use a divide and conquer approach, allowing us to get $\log(n)$ chaining instead of n. In BSV this looks more like:

```
function a fold(function a join(a x, a y), Vector#(n, a) in);
   if(n == 1)
     return in[0];
   else
     begin
       a v1 = fold(takefirsthalf(in));
       a v2 = fold(dropfirsthalf(in));
       return join(v1,v2);
     end
endfunction
```

Note that due to typing issues, this does not quite work. We need to cast to the compile-time dynamic list implementation. However, this is roughly equivalent to the scheme in the actual Bluespec library.

4.2 Rewriting the FIR Filter in a Functional Form

Problem 5: Rewrite the original FIR microarchitecture presented in Lab 1 in a more functional style. Create a new module, naming the `.bsv` file `FIRFunctional.bsv` and the `.awb` file `fir_functional.awb`. Check this module into the `lab2` modules directory you created in Section 3

- You have already seen a vector, namely the `samples`, whose elements are individual registers holding a `FixedPoint#(16,16)`. We are going to change this to a register which holds a single value: a register of `FixedPoint#(16,16)`s. This makes no significant change, but it does make operation a bit easier.

  Now instead of initializing each register to 0, we need to initialize the single register to a vector of zeros. we do this with the replicate function, which takes a value and returns a vector populated with just that value. Notice that we don't need to say how long the vector is, because the length is stored in the type of the vector and so there's only one possibility.

```
Reg#(Vector#(Taps,Sample)) samples <- mkReg(replicate(0));
```

- As part of our changes we need have the `fir_coefs` in a vector. The easiest way of going this is taking the fix length array (which should only ever be used for initializing constants and passing it to the arrayToVector function.

```
FixedPoint#(16,16) firCoefs_array [9] =  {fromReal(-0.0124),
                                          fromReal(0.0),
                                          fromReal(-0.0133),
                                          fromReal(0.0),
                                          fromReal(0.8181),
                                          fromReal(0.0),
                                          fromReal(-0.0133),
                                          fromReal(0.0),
                                          fromReal(-0.0124)};
Vector#(9, FixedPoint#(16,16) firCoefs = arrayToVector(firCoefs_array);
```

  Ideally one should be able to build a vector using something like this list syntax, but nice syntax to construct a constant vector doesn't exist.

- Now we start to see the advantage of thinking of things as vectors and functions on them. The first half of what the process rule does is shift the values the vector by one, adding in the new sample. We can do this by adding an element to the head of the vector (making samples[i], now new_samples[i+1]) and then dropping the right most one (by taking all the elements but the last).

```
Vector#(9, Sample) new_samples = cons(sample,samples);
samples <= take(new_samples);
```

- The remainder of the process rule takes all the new samples, multiplies by the associated FIR coefficient and sums them together. This corresponds nicely to the zipWith and fold functions that we discussed previously. Note that the infix operators `+` and `*` are escaped and surrounded by spaces.

```
Vector#(9,FixedPoint#(16,16)) tmp = map(fromInt,new_samples);
Vector#(9,FixedPoint#(16,16)) accumVector = zipWith( \* , tmp, firCoefs);
FixedPoint#(16,16) accumulate = fold( \+ , accumVector);
```

- Lastly, the endOfFile rule is still treating samples as if it were a vector of registers. We need to change it to reset things all in one go.

```
samples <= replicate(0);
```

This is all that was necessary. Note how much simpler this representation is compared to what we had originally. All of the complexity has been pushed into the higher-order library functions, leaving the user with a high-level understanding of the data patterns.

5 Appendix A (mkAudioPipeline)

For readability, most comments have been removed from the BSV and appended below the code, along with additional commentary; use the line numbers to correlate comments with the actual code.

The following code implements the microarchitecture shown in Figure 4. This module does nothing more than instantiate all the sub-modules you see in the diagram, and form the connection logic between them.

```
 0:  module [Connected_Module] mkAudioPipeline (AudioPipeline);
 1:
 2:    // Pipeline processing elements
 3:    FIRFilter filter <- mkFIRFilter;
 4:    FFT#(ŕFFT_POINTS,Complex#(FixedPoint#(16,16))) fft <- mkFFT;
 5:
 6:    // We need a marshaller/demarshaller to expand and contract the wider fft input type.
 7:    DEMARSHALLER#(Complex#(FixedPoint#(16,16)),Vector#(ŕFFT_POINTS,Complex#(FixedPoint#(16,16))))
 8:                    demarshaller  <- mkDeMarshaller;
 9:    MARSHALLER#(Vector#(ŕFFT_POINTS,Complex#(FixedPoint#(16,16))), Complex#(FixedPoint#(16,16)))
10:                     marshaller  <- mkMarshaller;
11:    Padder#(ŕFFT_POINTS) padder <- mkPadder;
12:    Depadder#(ŕFFT_POINTS) depadder <- mkDepadder;
13:
14:    // Local state, mostly needed for padding and keeping track of EOF
15:    FIFO#(Bool) endOfFile <- mkSizedFIFO(2*ŕFFT_POINTS);
16:    FIFO#(AudioProcessorUnit) outfifo <- mkFIFO();
17:    FIFO#(Bool) fftOutputsFrequency <- mkSizedFIFO(1);
18:    FIFO#(Bool) fftOutputsTime      <- mkSizedFIFO(1);
19:
20:    function data getRel(Complex#(data) val);
21:      return val.rel;
22:    endfunction
23:
24:    // Due to padding, the FFT will always deal with FFT_POINTS
25:    rule startDemarshaller;
26:      demarshaller.start(ŕFFT_POINTS);
27:    endrule
28:
29:    mkConnection(filter.sampleOutput,padder.pipeline.sampleInput);
30:    mkConnection(padder.padding,depadder.padding);
31:
32:    rule padderToDemarshaller;
33:      let padderData <- padder.pipeline.sampleOutput.get;
34:      case (padderData) matches
35:        tagged Sample .sample:
36:          begin
37:            endOfFile.enq(False);
38:            demarshaller.insert(cmplx(fromInt(sample)*fromReal(1/ŕFFT_POINTS),0));
39:          end
40:        tagged EndOfFile:
41:          begin
42:            endOfFile.enq(True);
43:          end
44:      endcase
45:    endrule
46:
47:    rule demarshallerToFFT;
48:      let demarshalledData <- demarshaller.readAndDelete;
49:      fft.fftInput(demarshalledData);
```

```
50:      fftOutputsFrequency.enq(?);
51:    endrule
52:
53:    rule fftToIFFT;
54:      let fftData <- fft.fftOutput();
55:      fft.fftInput(fftData);
56:      fftOutputsFrequency.deq();
57:      fftOutputsTime.enq(?);
58:    endrule
59:
60:    rule fftToMarshaller;
61:      let ifftData <- fft.fftOutput();
62:      fftOutputsTime.deq();
63:      marshaller.enq(append(tail(ifftData),cons(head(ifftData),nil)),ḞFT_POINTS);
64:    endrule
65:
66:    rule marshallerToOutData (!endOfFile.first);
67:      marshaller.deq();
68:      depadder.pipeline.sampleInput.put(tagged Sample fxptGetInt(marshaller.first.rel));
69:      endOfFile.deq();
70:    endrule
71:
72:    rule marshallerToOutEndOfFile (endOfFile.first);
73:      depadder.pipeline.sampleInput.put(tagged EndOfFile);
74:      endOfFile.deq();
75:    endrule
76:
77:    interface sampleInput = filter.sampleInput;
78:    interface sampleOutput = depadder.pipeline.sampleOutput;
79:
80:  endmodule
```

- In the case that we are connecting two modules whose types are consistent, all that is required is the module `mkConnection`. You may be wondering why we don't give these connections a name in the following fashion:

  ```
  Empty connector <- mkConnection(foo,bar);
  ```

  This is use the interface type of mkConnection is (as shown by the instantiation of "connector") Empty. It would be perfectly valid to use the above syntax, but for cleanliness, we generally use the more concise form.

- In the case that the types are not consistent, we have chosen to connect modules with rules containing logic to perform the type conversion. Take special notice of the rules demarshallerToFFT, and fftToIFFT. Together these two rules form the two MUX you see to the left of the FFT block. The combination of rules fftToIIFFT and fftToMarshaller form the De-Mux you see to the right of the FFT block.

- One subtle point in this code is how contention for the use of the FFT module is resolved (literally, how the selector for the Mux is derived). Look closely at the two fifos "fftOutputsFrequency" and "fftOutputsTime", both of which only hold one value. In rule fftToIFFT, you can see that "fftoutputsFrequency" is dequed, while "fftOutputsTime" has a value enqued. Because of the implicit conditions on these methods, this ensures that there is only one frame "in flight" at a time. While we could have build a more explicit MUX structure, we have chosen to use this data-flow style since it is usually less error-prone.

- Though deceptively simple, there are some other subtle points to this design. Data dependencies are handled exclusively through the use of FIFOS. As explained in the previous paragraph, they are used to resolve the contention for shared resources (FFT). Additionally, we have the endOfFile FIFO which keeps track of the EOF token when it arrives, and allows the pipeline to drain before forwarding it.

6 Appendix B (mkFFT)

For readability, most comments have been removed from the BSV and appended below the code, along with additional commentary; use the line numbers to correlate comments with the actual code.

This is the source code for the original FFT implementation, shown in Figure 2. The rule `fft`, appears towards the bottom of the module definition. Look at that first, and see how the functions are invoked to build up the combinational logic. This is the first code we are seeing which makes use of provisos. Provisos are a means of constraining types (for those who know about type theory, it is a form of ad. hoc. polymorphism).

```
 0:  interface FFT#(numeric type points, type data);
 1:    method Action fftInput(Vector#(points, data) inVector);
 2:    method ActionValue#(Vector#(points, data)) fftOutput;
 3:  endinterface
 4:
 5:  module mkFFT (FFT#(points,Complex#(cmplxd)))
 6:    provisos(Log#(points, log_points),
 7:             Arith#(cmplxd),
 8:             RealLiteral#(cmplxd),
 9:             Bits#(cmplxd,cmplxd_sz));
10:
11:    FIFO#(Vector#(points,Complex#(cmplxd))) inputFIFO <- mkFIFO;
12:    FIFO#(Vector#(points,Complex#(cmplxd))) outputFIFO <- mkFIFO;
13:
14:    function Complex#(cmplxd)  twiddle(Integer i);
15:       return cmplx(fromReal(cos(fromInteger(i)*pi/fromInteger(valueof(points)))),
16:                    fromReal(-1*sin(fromInteger(i)*pi/fromInteger(valueof(points)))));
17:    endfunction
18:
19:    function Vector#(points,Complex#(cmplxd))
20:                              perfectShuffle(Vector#(points,Complex#(cmplxd)) inVector);
21:      Vector#(points,Complex#(cmplxd)) outVector = newVector();
22:      for(Integer i = 0; i < valueof(points)/2; i=i+1)
23:        begin
24:          outVector[i] =  inVector[i*2];
25:          outVector[i + valueof(points)/2 ] = inVector[i*2+1];
26:        end
27:      return outVector;
28:    endfunction
29:
30:    function Vector#(points,Complex#(cmplxd)) bitReverse(Vector#(points,Complex#(cmplxd)) inVector);
31:      Vector#(points,Complex#(cmplxd)) outVector = newVector();
32:      for(Integer i = 0; i < valueof(points); i = i+1)
33:        begin
34:          Bit#(log_points) reversal = reverseBits(fromInteger(i));
35:          outVector[reversal] = inVector[i];
36:        end
37:      return outVector;
38:    endfunction
39:
40:    function Vector#(points,Complex#(cmplxd))
```

```
41:                         butterflyNetwork(Integer stage, Vector#(points,Complex#(cmplxd)) inVector);
42:     Vector#(points,Complex#(cmplxd)) outVector = newVector();
43:     for(Integer i = 0; i < valueof(points); i = i+2)
44:       begin
45:         Complex#(cmplxd) multiplicand = inVector[i+1] * twiddle((i/exp(2,valueof(log_points)-stage))
46:                                                         * exp(2,valueof(log_points)-stage));
47:         outVector[i] = inVector[i] + multiplicand;
48:         outVector[i+1] = inVector[i] - multiplicand;
49:       end
50:     return outVector;
51:   endfunction
52:
53:   rule fft;
54:     inputFIFO.deq();
55:     Vector#(points,Complex#(cmplxd)) outputdata = bitReverse(inputFIFO.first());
56:     for(Integer i = 0; i < valueof(log_points); i=i+1)
57:       begin
58:         outputdata = perfectShuffle(butterflyNetwork(i,outputdata));
59:       end
60:      outputFIFO.enq(outputdata);
61:   endrule
62:
63:   method Action fftInput(Vector#(points, Complex#(cmplxd)) inVector);
64:     inputFIFO.enq(inVector);
65:   endmethod
66:
67:   method ActionValue#(Vector#(points, Complex#(cmplxd))) fftOutput;
68:     outputFIFO.deq;
69:     return outputFIFO.first;
70:   endmethod
71:
72: endmodule
73:
```

The comments were removed from the code for clarity, but appear with their corresponding line numbers:

- **lines 0-3**: This FFT interface has two parameters. The first 'points' controls (as you might imagine) the width (number of points) of the design. The fact that is a 'numeric' type, is a subtle issue relating to the different kinds of types present in the Bluespec language. You can read more about numeric types in the reference manual. The parameter 'data' controls the type of data which will be transformed. For example, the data type could be int, fixed point, complex, etc. depending on the intended use. Through the use of Type Classes, the type parameters can be restricted, as can be seen in the definition of module mkFFT
- **lines 5-9**: mkFFT provides an implementation of the FFT interface. As you can see the parametrization of this module definition is slightly more restricted than that of the interface definition, in that we are restricting the data parameter to be a Complex number. Through the use of pattern matching, we are able to extract the type parameter of Complex, and give it a name 'cmplxd'. Provisos (indicated by the language keyword 'provisos') are used to restrict the allowable types which can be used to parametrize this module. Provisos generally fall into two major categories, the first being to relate numeric types, and the second to assert that type parameters are members of certain Type Classes. Go to link for a good explaination of type classes and provisos. Provisos can also be used to create new type variables. For example, the proviso Log#(points, log_points) is not not so much an assertion as the creation of a new variable log_points, which receives the value log(points). Arith#(cmplxd) asserts that the parameter cmplxd is a member of the Arith type Class. Members of the Arith typeclass

have arithmetic operations defined on them. For explaination of RealLiteral and Bits, see the language reference guide.

- **lines 14-15**: compute the twiddle factor. This function implements the math described in Section 1.1
- **lines 19-28**: In a perfect shuffle, an ordered list of elements is split in half, and the two resulting lists are merged by interleaving the elements. This shuffle appears three times in the FFT shown in Figureřefblah.
- **lines 30-38**: this function reorders the vector by swapping words at positions corresponding to the bit-reversal of their indices. This phase of the Pease transform can be clearly seen in Figureřefblah. The reordering can be done either as as the first or last phase of the transformation.
- **lines 40-41**: This function implements the butterfly networks, using the stage variable to differentiate between the different phases of the FFT
- **lines 43-49**: As the stages progress, the roots of unity change. They are based on the top bits of the butterfly, Later stages use "smaller" roots, i.e. 1st stage uses 1, 2nd stage uses -j, etc. An N point FFT requires log(N) stages. The numeric type log_point was introduced in the provisos and is used as a loop bound. We use a different function at each stage of the FFT, and use i to select this function in the butterflyNetwork. There rule creates one gigantic combinations circuit.
- **lines 63-65**: Interface method used to input Data to the FFT module. Examine FFTPipeline.bsv to see how this interface method is used
- **lines 67-70**: Interface method used to get the transformed data from the FFT Module. Examine FFTPipeline.bsv to see how this interface method is used