Lab 1: A Simple Audio Pipeline

6.375 Laboratory 1
Revision 1.2

Assigned: February 3, 2010
Due: February 12 2010

1 Introduction

This lab is the beginning of a series of labs in which we will design the hardware for a Digital Signal Processor (DSP) for audio signals and run it on an FPGA. Audio processing applications are good candidates for hardware implementations since we are interested both in low power and high throughput, especially if we are dealing with real-time applications or running on mobile platforms.

Figure 1 shows a high-level picture of the audio pipeline and the driving software infrastructure we will use in a later lab when we run the audio pipeline on an FPGA for real. The software component runs on the host processor, and is responsible for opening the audio file, streaming its contents across the serial communication channel to the hardware, and also for retrieving the output of the pipeline and storing the results. This lab will not use FPGAs, saving the challenges of using FPGAs for later labs.

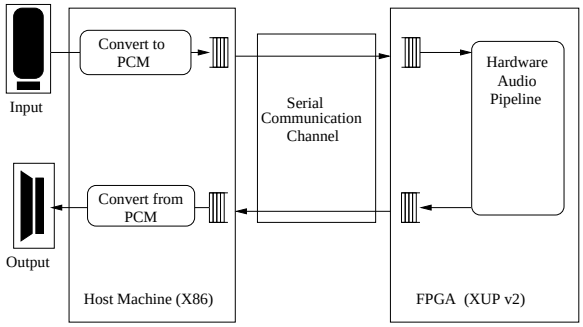

Figure 1: High-level Audio Pipeline Diagram

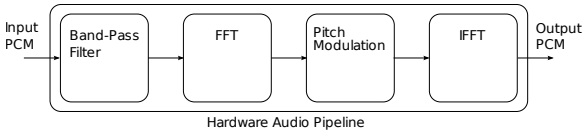

Figure 2: Audio Pipeline Made of Blocks

The box marked Hardware Audio Pipeline in figure 1 contains the digital signal processing hardware which will be our focus in this and following labs. Hardware of this type usually consists of a series of blocks, as depicted in figure 2. For example, it might begin with a band-pass filter to remove unwanted frequencies. After that, there could be an FFT (Fast Fourier Transform) module which converts the signal from the time domain to the frequency domain. Once in the frequency domain the signal can be modified by any number of hardware functions such as pitch modulation. The final blocks would include an IFFT (Inverse FFT) to convert it back to the time domain, and possibly a windowing function.

We begin our audio processor with the hardware audio pipeline empty, essentially a loop-back device. However, by the end of this lab, we will add a FIR (Finite Impulse Response) filter, serving

in place of the band pass filter, which can be used to attenuate specified frequency ranges. Over the next few weeks, we will augment it further.

1.1 Lab Organization

The lab begins by describing how to use the Bluespec Workstation to compile and simulate your design. Bluespec code for a trivial audio pipeline which passes through values unchanged is provided.

We then walk through an implementation of a simple 8 tap FIR filter written in Bluespec, describing in detail what each part of the code is for. We will ask you some questions to guide your exploration of some of the features of the Bluespec language.

Once you understand the FIR filter and have it running in simulation we provide an opportunity to write your own Bluespec code by challenging you to implement the FIR filter using a special multiplier, which may require a slightly different microarchitecture than the original FIR filter we present.

2 Getting Started

All of the 6.375 laboratory assignments should be completed on an Athena/Linux workstation. Please see the course website for more information on the computing resources available for 6.375 students. Once you have logged into an Athena/Linux workstation you will need to setup the 6.375 toolflow with the following commands.

```
% add 6.375
% source /mit/6.375/setup.csh
```

You will be using subversion to manage your 6.375 laboratory assignments. Every student has their own repository which is not accessible to other students. You can checkout your personal subversion directory using the following command.

```
% svn co $SVNROOT
```

To begin the lab you will need to make use of the lab harness located in `/mit/6.375/lab-harnesses`. The lab harness provides code used in this lab. The following commands extract the lab harness into your subversion directory and add the new project to subversion.

```
% cd $USER
% tar -xzvf /mit/6.375/lab-harnesses/lab1-harness.tar.gz
% svn add lab1
% svn ci -m "Initial checkin"
```

3 Compiling and Simulating with the Bluespec Workbench

The resulting lab1 project contains Bluespec code we will be using for this lab. The directory layout is shown in figure 3.

`common/AudioProcessorTypes.bsv` defines the Bluespec interface we will be using for our audio pipeline throughout this series of labs. It defines an audio sample as a 16 bit signed integer, a union type which is either a sample or end of file marker, and the `AudioProcessor` interface, which contains two methods. The `putSampleInput` method is called with the next input sample data. The `getSampleOutput` method gets the sample data provided by your audio pipeline which is some transformation of the input sample data.

`common/TestDriver.bsv` contains Bluespec code to drive the audio pipeline module in simulation.

`empty/AudioPipeline.bsv` defines an initial implementation of an empty audio pipeline which simply passes the input samples as is to the output without any transformation.

```
lab1/
    common/
        AudioProcessorTypes.bsv
        FilterCoefficients.bsv
        Multiplier.bsv
        TestDriver.bsv
    data/
        foo.pcm
        foo_filtered.pcm
    empty/
        AudioPipeline.bsv
        empty.bspec
```

Figure 3: Lab1 code directory structure.

We can simulate the hardware described by the Bluespec code to get an idea of how it works without having to deal with the complications of real hardware which are many, even for such a simple design as we are working with.

- Navigate to the `empty/` directory and start up Bluespec Workstation.

  ```
  lab1% cd empty
  lab1/empty% bluespec empty.bspec&
  ```

- The Bluespec Workstation GUI will appear. Select `Build->Compile` This compiles the Bluespec source code.

- Select `Build->Link`. This creates the executable `out` which can be used to simulate the hardware described by the top level module `mkTestdriver`. The executable file can then be run like any program. The test driver code provided reads input audio file from a PCM (Pulse Coded Modulation) file called `in.pcm`, passes it to your audio pipeline, and writes the output from your pipeline to `out.pcm`. We have included a sample PCM audio file in the `data/` folder called `foo.pcm`, along with the expected output PCM file for the filtered audio using the filters we write later in the lab.

- To simulate the audio pipeline on `foo.pcm`, copy `../data/foo.pcm` to `in.pcm`, then from the `Build` menu in the Workstation, select `Simulate`.

  If everything worked correctly, you should have seen a long list of the samples processed in hex, and the audio file `out.pcm` has been created with the output of your transformations.

- Verify whether your processor output the expected data. You can do this by comparing the output `out.pcm` with the original file `foo.pcm` provided, because the empty pipeline should not be changing anything.

  ```
  lab1/empty% cmp out.pcm ../data/foo.pcm
  ```

  If the `cmp` command completes silently, the two PCM files match.

We can also direct the Bluespec Workstation to generate Verilog code, which can then be used by a whole slew of tools to build the hardware as an ASIC or programed into an FPGA. Under the `Project` menu select `Options`. In the window that popped up go to the `Compile` tab and select the compile to Verilog option. If you compile the project now, it will create the file `empty/.bsc/mkTestDriver.v`.

4 Writing FIR Filters in Bluespec

4.1 Background

A little background information on FIR filters is useful to justify the subsequent utility of this exercise. The following webpage gives a reasonable introduction to FIR digital filters:

http://www.netrino.com/Embedded-Systems/How-To/Digital-Filters-FIR-IIR

The basic idea is that by modifying the FIR filter's constant coefficients, you can change the frequencies which the filter will attenuate. By increasing the number of taps, *i.e.*,registers, you can improve the quality that a FIR filter can imitate any desired frequency response. You will create a band-pass filter, with eight taps using a predefined set of coefficients, and will use this filter in subsequent labs to create more complex audio processing applications.

4.2 First FIR Filter

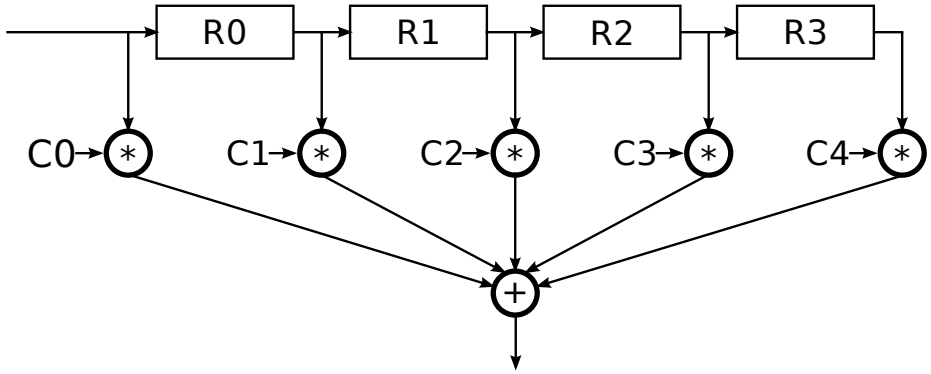

Figure 4: Simple Fir Filter

We will construct three alternate FIR pipelines. The first pipeline has a microarchitecture similar to that shown in Figure 4, the only difference being the number of registers. Create a new directory in the top level directory called `firstfir/`. We will implement our first FIR filter in this directory in the file `AudioPipeline.bsv`. Add the following lines to that file (you may wish to use empty AudioPipeline implementation as a starting base for the code).

```
import FIFO::*;
import FixedPoint::*;

import AudioProcessorTypes::*;
import FilterCoefficients::*;
```

These lines import definitions from other Bluespec files. FIFO and FixedPoint are provided as part of the Bluespec library. AudioProcessorTypes and FilterCoefficients are defined in the `common/` directory.

```
module mkAudioPipeline
    // interface:
        (AudioProcessor);
```

This begins a definition of a new module called `mkAudioPipeline`. The module implements the `AudioProcessor` interface defined in `AudioProcessorTypes.bsv`. We have included a comment, which starts with two forward slashes and runs to the end of the line, to make it clear to those reading the code what the interface is. The text in the comment is ignored by the compiler.

```
FIFO#(AudioProcessorUnit) infifo <- mkFIFO();
FIFO#(AudioProcessorUnit) outfifo <- mkFIFO();
```

These two lines instantiate two `mkFIFO` modules, which implement the `FIFO` interface. The type of object we put on the fifos is `AudioProcessorUnit`. We will place incoming samples in the `infifo` and outgoing samples on the `outfifo`.

```
Reg#(Sample) r0 <- mkReg(0);
Reg#(Sample) r1 <- mkReg(0);
Reg#(Sample) r2 <- mkReg(0);
Reg#(Sample) r3 <- mkReg(0);
Reg#(Sample) r4 <- mkReg(0);
Reg#(Sample) r5 <- mkReg(0);
Reg#(Sample) r6 <- mkReg(0);
Reg#(Sample) r7 <- mkReg(0);
```

These lines instantiate the eight registers in our design. The `mkReg` module takes a single argument, which is the initial value for the register to contain.

We have now instantiated all the state elements in our design. Next we will write a rule to describe how those state elements are updated.

```
rule process (True);

    if (infifo.first() matches tagged Sample .sample)
    begin
        r0 <= sample;
        r1 <= r0;
        r2 <= r1;
        r3 <= r2;
        r4 <= r3;
        r5 <= r4;
        r6 <= r5;
        r7 <= r6;

        FixedPoint#(16,16) accumulate =
              c[0] * fromInt(sample)
            + c[1] * fromInt(r0)
            + c[2] * fromInt(r1)
            + c[3] * fromInt(r2)
            + c[4] * fromInt(r3)
            + c[5] * fromInt(r4)
            + c[6] * fromInt(r5)
            + c[7] * fromInt(r6)
            + c[8] * fromInt(r7);

        outfifo.enq(tagged Sample fxptGetInt(accumulate));
    end
    else
    begin
        $display("FIR got end of file");
        outfifo.enq(tagged EndOfFile);
    end

    infifo.deq;
endrule
```

This rule processes a sample. It will run whenever we have an incoming sample. If the sample is a normal sample, the rule writes the registers with their appropriate new values, calculates the output sample from the existing register values, places the sample on the outgoing FIFO, and removes the input sample from the input FIFO. All of these operations occur together, *atomically*, in a single cycle.

If the sample is the end of file marker, the rule prints a notice that end of file was reached, and forwards that to the output.

Now we have implemented all the rules of our design. All that remains is to implement the `AudioProcessor` interface, which requires two methods.

```
    method Action putSampleInput(AudioProcessorUnit in);
        infifo.enq(in);
    endmethod
```

The first method takes a sample input, and places it on the `infifo` queue. It is an `Action` method because it modifies internal state of the module `infifo`.

```
    method ActionValue#(AudioProcessorUnit) getSampleOutput();
        outfifo.deq();
        return outfifo.first();
    endmethod
endmodule
```

Finally we implement the `getSampleOutput` method. This removes the first outgoing sample from the `outfifo` and returns it. The method is marked `ActionValue` with type parameter `AudioProcessorUnit` to indicate it both modifies internal state and return an object of type `AudioProcessorUnit`.

The `endmodule` keyword completes the definition of our module. Before you can compile and simulate the module you need to create a Bluespec Workstation project like the one we provided you with for the empty pipeline. Open the Bluespec Workstation from the `firstfir` directory.

```
lab1/firstfir% bluespec&
```

Go to `Project -> New...`. Name the project `firstfir`. This will open up the project options dialog. The `Top file` is `../common/TestDriver.bsv`, `Top module` is `mkTestDriver`. We like to put all intermediate files into a directory `.bsc` just to keep them out of the way. Specify `.bsc` for `.bo/.bi/.ba`, `Bluesim`, `Verilog`, and `Info` files locations. Our common source needs to be added to the search path. Click the `Add` button and specify `../common`.

The rest of the defaults are appropriate for us. Save and close the project options window. You should now be able to compile and simulate the module. The output PCM of this filter and the remaining filters we implement in this lab should match that in `data/foo_filtered.pcm`. Remember to copy `data/foo.pcm` to `in.pcm` in the firstfir directory before simulating. After simulating verify the output is correct.

```
lab1/firstfir% cmp out.pcm ../data/foo_filtered.pcm
```

4.3 Bluespec and Static Elaboration

If you typed in the above Bluespec lines by hand, you may have noticed it was annoying to have to copy 8 lines to instantiate all 8 registers, then again in the processing of registers. If you did not type in the above lines by hand, you anticipated the annoyance, which assuredly was there.

Fortunately, Bluespec provides constructs for powerful *static elaboration* which we can use to make writing the code easier. Static elaboration refers to the process by which the Bluespec compiler evaluates expressions at compile time, using the results to generate the hardware rather than generating hardware to evaluate the expressions dynamically.

Vectors in Bluespec are a nice way to describe sequences of things, whether they be values, registers, or even rules. We can replace our 8 separate registers with a single vector of registers instantiated with something like

```
Vector#(8, Reg#(Sample)) r <- replicateM(mkReg(0));
```

Now we can refer to the individual registers with bracket notation. For example `r[0]`, `r[1]`, and so on to `r[7]`.

We can use a for loop to copy many lines of code which have the same form. For example, to advance we could do something like

```
for (Integer i = 0; i < 7; i = i+1) begin
    r[i+1] <= r[i];
end
```

The Bluespec compiler, during its static elaboration phase, will replace this for loop with its fully unrolled version.

```
r[1] <= r[0];
r[2] <= r[1];
r[3] <= r[2];
r[4] <= r[3];
r[5] <= r[4];
r[6] <= r[5];
r[7] <= r[6];
```

Discussion questions 1 and 2 at the end of this lab handout ask you to compare the `firstfir` filter with and without using Vectors and for loops. Go answer them now.

4.4 Using a Better Multiplier

In our first FIR filter we used the `*` operator for multiplication. This generates a combinational multiplier in hardware. Because multiplication is a complex operation, the combinational multiplier potentially limits the frequency we can run our FIR filter at. We can improve the frequency by using a pipelined multiplier, breaking the multiplication across multiple cycles.

In the common directory we provide you with a special implementation of a multiplier to improve the performance of the hardware. Well, the senior grad students claim they will implement a good multiplier, but not until your architecture is setup to use it. They give you the following interface for the multiplier.

```
interface Multiplier;
    method Action putOperands(FixedPoint#(16, 16) coeff, Int#(16) samp);
    method ActionValue#(FixedPoint#(16, 16)) getResult();
endinterface
```

To multiply two operands with the multiplier you first call the `putOperands` method with the operands. Some number of cycles later (the implementers have not decided how many cycles later is best yet), the result can be taken using the `getResult` method. The multiplier is pipelined, so you can do multiple calls to `putOperand` before getting results. The results will come in the order they were put in.

We can instantiate the multiplier in our `mkAudioPipeline` module in the same way we instantiated registers.

```
Multiplier multiplier <- mkMultiplier();
```

Change your FIR filter implementation to use 9 instances of the special multiplier instead of the `*` operator. You will need to import the multiplier at the top of your bluespec code. You might wish to use a vector to make it less tedious to instantiate 9 multipliers.

Because the multiplier takes multiple cycles to perform a multiplication, you can no longer perform the entire thing in a single rule. You will need to break apart the single rule into multiple rules. One rule should initiate each of the 9 multiplication operations you need to perform for a

new sample by calling the `putOperands` method on each of the 9 multiplier instances. A second rule should be used to collect the results of each multiplication and perform the sum of those.

Make sure you process all the input samples before outputting the end of file sample. There are multiple different ways to do this, many of which require additional state elements in your design.

Figure 5 shows the new architecture, again limited to 4 taps here so it fits on the page; your filter should remain an 8 tap filter.

Write this version of the FIR filter in `multfir/AudioPipeline.bsv`.

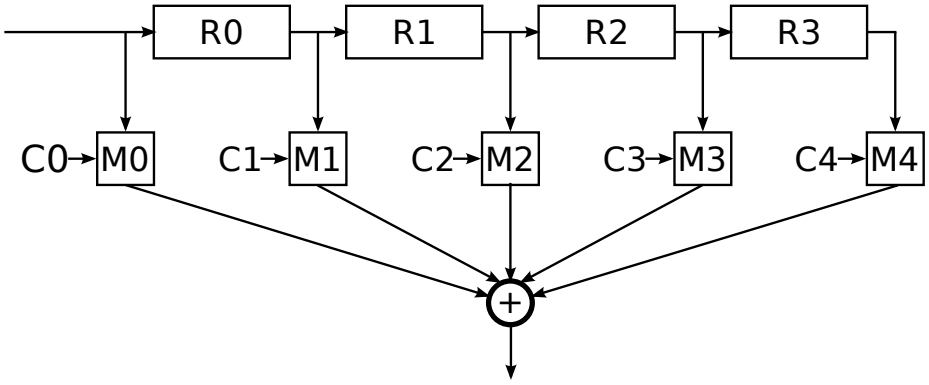

Figure 5: Fir Filter with Special Multiplier

5 Discussion Questions

1. Modify the firfirst Bluespec code to use a vector of registers and for loops. You will need to import the Vector package to use vectors. Generate Verilog code for the FIR filter both with and without for loops. Compare the different Verilog code, and comment based on what you see about how using for loops in Bluespec changes the hardware generated.

2. How many lines of code would you have to change in the original firstfir filter description if we wanted to turn it into a 16 tap FIR filter? How many lines of code have to change in the version with the for loop you implemented for question 1? A 256 tap FIR filter? Comment on how for loops can be used to write source code which is more generic and easily reusable in different situations.

3. Describe how you ensured in your multfir design that all the input samples were processed before outputting the end of file sample.

4. After switching your filter to use the special pipelined multiplier in place of the Verilog `*` operator, the builders of the multiplier discover an enhancement they can make to their implementation of the multiplier. How does your implementation of the filter have to change to accommodate the new implementation of the multiplier assuming the multiplier interface stays the same?

5.1 What to Turn In

When you have completed the lab you should check in a final version via subversion. This should include the Bluespec implementation of the firstfir and multfir filters. It should also include your answers to the discussion questions in a file called `answers` in the top level lab directory.

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

```
lab1% svn add --parents firstfir/*.{bsv,bspec}
lab1% svn add --parents multfir/*.{bsv,bspec}
lab1% svn add answers.pdf
lab1% svn ci -m "Submission"
```