Lab 1: A Simple Audio Pipeline

6.375 Fall 2019
Assigned: September 6, 2019
Due: September 16, 2019 11:59PM

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, retrieving the output of the pipeline, and storing the results. This first lab will not use FPGAs, saving the challenges of using FPGAs for later labs.

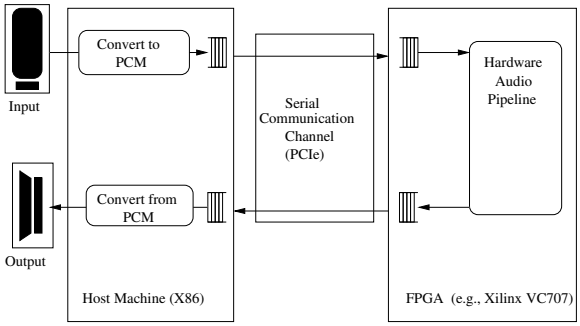

Figure 1: High-level Audio Pipeline Diagram

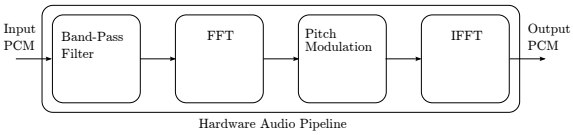

Figure 2: Audio Pipeline Made of Blocks

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 the pipeline further.

1.1 Lab Organization

The lab begins by describing how to compile and simulate your design using Bluespec Compiler (`bsc`). 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 multistage multiplier, which will require a slightly different microarchitecture than the original FIR filter we present.

2 Getting Started

The 6.375 laboratory assignments are designed be completed on an Athena/Linux workstation. We have also installed two dedicated Athena build machines for this course. Please see the course website for more information on the computing resources available for 6.375 students.

`http://csg.csail.mit.edu/6.375/6_375_2019_www/lab-machines.html`

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.sh
```

We will be using git to manage your 6.375 laboratory assignments. Every student has their own bare repository on the course locker to use when submitting code. To work on your code start by making a local clone of the bare repository using the following command.

```
$ git clone $GITROOT {DirectoryNameYouWant}
```

The repository includes code provided for this lab. After running the clone command you should see this code in a newly created directory with the same name as your username.

While we will present you with all the git commands you need to submit your code when you are done with the lab, git has many more valuable features you are welcome to take advantage of. The internet is a good place to learn more about git.

3 Compiling and Simulating Bluespec Code

```
audio/
    common/
        AudioProcessorTypes.bsv
        FilterCoefficients.bsv
        Multiplier.bsv
        TestDriver.bsv
    data/
        mitrib_short_filtered.pcm
        mitrib_short.pcm
    fir/
        FIRFilter.bsv
        Makefile
```

Figure 3: Lab1 code directory structure.

The directory layout for the code provided 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, 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.

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

`fir/Makefile` defines a set of tasks to compile Bluespec code with `bsc`, the Bluespec compiler.

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 `fir/` directory and make a build directory.

  ```
  audio$ cd fir
  ```

- Build the Bluespec simulation executable.

  ```
  audio/fir$ make simulation
  ```

  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 `mitrib_short.pcm`, along with the expected output PCM file for the filtered audio using the filters we write later in the lab.

  You can convert your own audio files to and from PCM format using ffmpeg. For example, to convert an mp3 file to PCM:

  ```
  ffmpeg -i foo.mp3 -f s16le -ac 1 foo.pcm
  ```

  And to convert a pcm file to a wav file, which can be played by your favorite media player:

  ```
  ffmpeg -f s16le -i foo.pcm foo.wav
  ```

- To simulate the audio pipeline on `mitrib_short.pcm`, copy `../data/mitrib_short.pcm` to `in.pcm`, then run `./out`.

  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 `mitrib_short.pcm` provided, because the empty pipeline should not be changing anything.

  ```
  audio/fir$ cmp out.pcm ../data/mitrib_short.pcm
  ```

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

  Alternatively, if both files have the same md5sum, they almost certainly have the same contents.

```
audio/fir$ md5sum out.pcm ../data/mitrib_short.pcm
```

- We can use Bluespec compiler (`bsc`) 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. Run:

  ```
  audio/fir$ make verilog
  ```

  It will create the file `fir/bscdir/mkFIRFilter.v`.

- We can also analyze the performance of your circuit in terms of area and critical-path delay, using `synth`. `synth` will invoke `yosys`[1], an open-source synthesis tool, and uses `ABC`[2] for technology mapping.

  To analyze your circuit using `synth`, please run:

  ```
  audio/fir$ make synth
  ```

  It will report *area* and *critical-path* delay to terminal.

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 of the FIR filter for a specific frequency response. You will create a band-pass filter with eight taps using a predefined set of coefficients. We will use this filter in subsequent labs to create more complex audio processing applications.

4.2 First FIR Filter

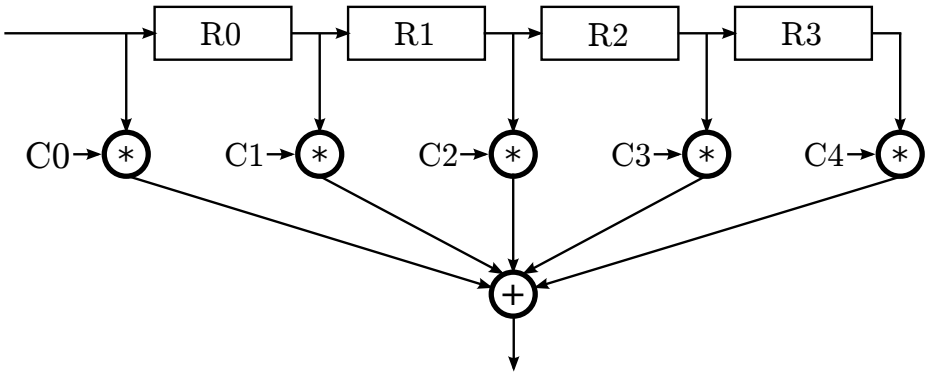

Figure 4: Simple Fir Filter

[1] `http://www.clifford.at/yosys/`
[2] `https://people.eecs.berkeley.edu/~alanmi/abc/`

We will build up our FIR filter in steps. The first pipeline has a microarchitecture similar to that shown in Figure 4, the only difference being the number of registers. Change the file `fir/FIRFilter.bsv` to have the following lines.

```
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.

```
// The FIR Filter Module Definition
module mkFIRFilter (AudioProcessor);
```

This begins a definition of a new module called `mkFIRFilter`. The module implements the `AudioProcessor` interface defined in `AudioProcessorTypes.bsv`. The line beginning with two forward slashes is a comment. The text in the comment is ignored by the compiler.

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

These two lines instantiate two `mkFIFO` modules, which implement the `FIFO` interface. The type of object we put on the fifos is `Sample`. 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);
        Sample sample = infifo.first();
        infifo.deq();

        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(fxptGetInt(accumulate));
    endrule
```

This rule processes a sample. It will run whenever we have an incoming 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.

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(Sample 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#(Sample) 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 `Sample` to indicate it both modifies internal state and returns a value of type `Sample`.

The `endmodule` keyword completes the definition of our module. 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/mitrib_short_filtered.pcm`. Remember to copy `data/mitrib_short.pcm` to `in.pcm` in the fir directory before simulating. After simulating verify the output is correct.

```
audio/fir% cmp out.pcm ../data/mitrib_short_filtered.pcm
```

Now generate the Verilog code for the current FIRFilter.bsv and save it to `fir/FIRFilter_unstatic.v`. Also synthesize your circuit to analyze its area and critical-path delay and save the output in `fir/synth_FIRFilter_unstatic.txt`.

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];
```

To see the effect static elaboration has on the hardware generated, we will compare the Verilog code generated from the existing FIRFilter.bsv with a version using Vectors and loops.

Modify FIRFilter.bsv to use a vector of registers and for-loops. You will need to import the Vector package to use vectors. After you have verified your new version of the FIRFilter works correctly, generate the Verilog code for it and compare that to `fir/FIRFilter_unstatic.v`. How has using for-loops in the Bluespec code changed the hardware the compiler generates for the FIR filter?

Now run `synth` tool on the new code, compare that to `fir/synth_FIRFilter_unstatic.txt`. How has the area and critical-path delay of your hardware change by using for-loops in Bluespec?

4.4 Using a Multi-Stage Multiplier

So far we have been using the `*` operator for multiplication in our FIR Filter. 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 may be able to improve the frequency by using a multistage multiplier, breaking the multiplication across multiple cycles.

In the common directory we provide you with an implementation of a multiplier which supports the multistage interface a high performance multiplier may have. The interface for the multistage multiplier is as follows.

```
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 (perhaps 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 `mkFIRFilter` module in the same way we instantiated registers.

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

Here is how you can get the result of the multiplication:

```
let x <- multiplier.getResult();
```

The left arrow is needed in this case instead of a simple equals sign because the `getResult` function of the multiplier is an `ActionValue` method, which can potentially change the state of the multiplier. You should think of this use of the left arrow as different from that of the left arrow you use when instantiating a module. The `let` keyword tells the compiler to figure out what type the variable `x` should be on its own. In place of the `let` keyword you could instead explicitly put the type of `x`, `FixedPoint#(16, 16)`.

Change your FIR filter implementation to use 9 instances of the multistage multiplier instead of the `*` operator. You will need to import the Multiplier package 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 an entire iteration with a single atomic action. 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.

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.

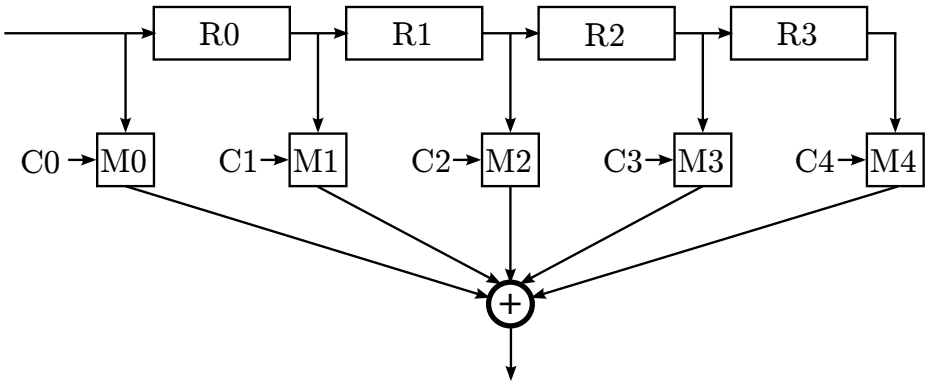

Figure 5: Fir Filter with Special Multiplier

After you successfully changed FIR filter using multi-stage multipliers, synthesize the circuit and compare the area and critical path delay with results from unstatic and statically elaborated Bluespec code in sections 4.2 and 4.3.

5 Discussion Questions

1. In section 4.3 we asked you to compare the hardware generated for the FIR filter before using for-loops, and then again after. How does using for-loops in the Bluespec source code change the hardware the compiler generates for the FIR filter? How has the area and critical-path delay of your hardware change by using for-loops in Bluespec?

2. How many lines of code would you have to change in the original filter description without a for-loop and vectors 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? 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. After switching your filter to use the multistage 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?

4. After you successfully changed FIR filter using multi-stage multipliers, how do the area and critical path delay compare with results from unstatic and statically elaborated Bluespec code in sections 4.2 and 4.3?

5.1 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 FIR filter with multistage multiplier and your answers to the discussion questions in a file called `lab1` in a new directory called `answers` at the top level of the lab directory. Text is the preferred format for the `answers/lab1` file, pdf format is also acceptable.

To submit your final version you must 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 answers/lab1
audio$ git add fir/FIRFilter.bsv
audio$ git commit -m "Lab1 submission"
audio$ git push
```