Lab 4: Audio Pipeline on the FPGA

6.375 Laboratory 4
Assigned: March 1, 2013
Due: March 8, 2013

1 Introduction

In this lab we will run the audio pipeline constructed in the previous labs on an FPGA. We introduce Sce-Mi as a means for communicating between a host processor and FPGA and show how Sce-Mi integrates with the Bluespec Workstation, allowing us to simulate the processor-FPGA link. We will then synthesize the audio pipeline for the FPGA, look at area and timing results, and run the audio pipeline on an actual FPGA. Following that we ask you to make the pitch factor a dynamic parameter passed in at run time over the Sce-Mi link.

1.1 Sce-Mi

The Standard Co-Emulation Modeling Interface (Sce-Mi) is an Accellera standard which was designed to aid in verification of hardware designs. The standard specifies a transaction-based modeling interface used to pass messages between an un-timed software test bench and a design under test (DUT) described in register transfer language (RTL). The DUT can be emulated on an FPGA to achieve better performance than software RTL simulators are capable of.

Bluespec provides a complete implementation of the Sce-Mi standard, which makes Sce-Mi attractive as a means for us to interact with the designs we run on FPGAs.

Figure 1 shows how the audio pipeline looks using Sce-Mi to pass the samples from the host processor to the FPGA and back. The `mkAudioPipeline` module becomes our design under test (DUT). The SceMiLayer is a wrapper around our DUT which hooks it up to the Sce-Mi ports `inport` and `outport` and the reset transactor which enables soft reset of the FPGA by software. Samples will arrive on `inport` and be passed directly to the audio pipeline. Samples coming out of the audio pipeline will go to `outport`.

We use PCI Express as our bridge from host processor to FPGA. On the host processor we have port proxies, corresponding to the Sce-Mi ports on the FPGA, which the software test bench can interact with. The test bench is implemented in c++ and runs on the host processor.

For our audio pipeline to work with Sce-Mi, we do not need to change the audio pipeline code from the previous lab. What we do is wrap the audio pipeline in the `SceMiLayer` and replace `mkTestDriver` with a new test bench written in c++ running on the host processor which sends and receives samples through the Sce-Mi port proxies to the audio pipeline on the FPGA.

1.2 Getting Started

The additional infrastructure needed to use Sce-Mi for communication between the host computer and FPGA is provided in the lab4 harness.

Update your local repository with this new infrastructure. Add the 6.375 course locker, source the setup script, navigate to the directory which contains the `audio/` folder from the previous labs, download `lab4-harness.tar.gz` from the course website, and run:

```
$ tar xf lab4-harness.tar.gz
```

This will create a directory called `audio/scemi` with some new files for using Sce-Mi in both simulation and on the FPGA. To save the changes to your local repository, run

```
$ git add audio/scemi
$ git commit -m "lab4 initial checkin"
```

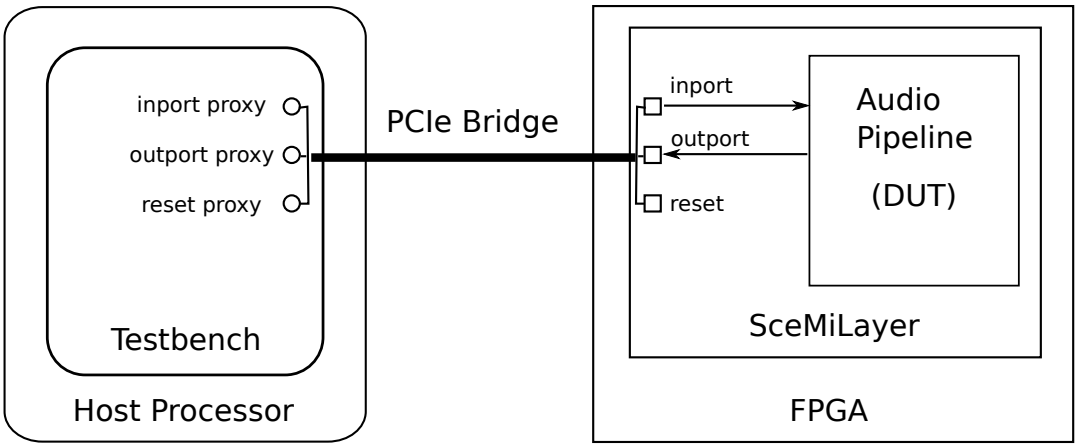

Figure 1: Audio Pipeline with Sce-Mi

The `scemi/` directory contains a number of files. The most important are

`SceMiLayer.bsv` implements the Sce-Mi wrapper around `mkAudioPipeline`. It instantiates `inport` and `outport` and hooks them up to `mkAudioPipeline`. It also instantiates the reset transactor.

`Bridge.bsv` selects from among a number of bridges supplied by Bluespec. Each bridge is specific to the FPGA we are running on. We will use the TCP bridge for simulating the hardware with Sce-Mi and the XUPV5 bridge when running on the FPGA.

`Tb.cpp` implements the software test bench. It has the same functionality as the test bench we have been using so far. It reads `in.pcm`, sends the samples through the audio pipeline and records the transformed samples to `out.pcm`.

The other new files are, briefly,

`ResetXactor.bsv` Bluespec code for a reset transactor which allows resetting the FPGA from software.

`ResetXactor.h` Header file for software interface to reset transactor.

`ResetXactor.cpp` Implementation of software interface to reset transactor.

`sim/project.bld` Project file for building audio pipeline with Sce-Mi for simulation.

`sim/sim.bspec` Workstation project file for building audio pipeline with Sce-Mi for simulation. (Makes use of the project.bld file).

`fpga/project.bld` Project file for building audio pipeline with Sce-Mi for the FPGA.

`fpga/fpga.bspec` Workstation project file for building audio pipeline with Sce-Mi for the FPGA. (Makes use of the project.bld file).

`fpga/bluespec_init.tcl` Script to add buttons to the bluespec workstation for programming the FPGA and running the test bench on the FPGA design.

2 Simulating with Sce-Mi

We introduced Sce-Mi so we can run our design on an FPGA instead of just simulating it in software, but it is still extremely useful to *simulate* the design using the same Sce-Mi infrastructure. This

allows us to see the output of `$display` calls in our design and lets us avoid the long synthesis times needed to synthesize the design for the FPGA when we are still debugging the functionality of the design.

To use Sce-Mi we have split our system into two distinct pieces, the software test bench and the hardware audio pipeline. The only interaction between the test bench and audio pipeline is through the Sce-Mi bridge. In simulation we will still have those two distinct pieces, only instead of running on an actual FPGA, we simulate the audio pipeline using a Bluesim process on our computer separate from the test bench software. The bridge between those two processes is TCP instead of PCI Express.

Building a project with Sce-Mi is a little more complicated than the simple projects we have been using up to this point. The test bench is compiled separately from the audio pipeline because the test bench is now implemented in c++. We also produce extra information for the test bench to communicate with the audio pipeline, including things such as which bridge to use and what the Sce-Mi port names are.

Bluespec provides a build process which knows how to build everything for Sce-Mi and is specifically designed to make builds repeatable. We have set up a Workstation project which uses this build process.

Navigate to the `scemi/sim/` directory. The file `project.bld` describes what we want Bluespec's build process to build for us.

Problem 1: Simulate the audio pipeline with the Sce-Mi infrastructure by taking the following steps.

1. Copy over our test input to the local directory. We use the short test input because simulation on the full one takes around 20 minutes.

   ```
   sim$ cp ../../data/mitrib_short.pcm in.pcm
   ```

2. Open up the `sim.bspec` workstation project in `bluespec`.

   ```
   sim$ bluespec sim.bspec&
   ```

3. Compile, link and simulate the design. This will build two executables, a simulator for the audio pipeline called `bsim_dut`, and the software test bench `tb`. When the simulate command is run in the Workstation, the Workstation first starts up the `bsim_dut` program, giving it a little time to get ready, then runs `tb`. The two programs communicate over TCP.

4. Verify the output from simulation is correct by comparing it to the result from the `mkTestDriver` test bench in the previous lab.

3 Running on the FPGA

In the `fpga` directory we have set up the Workstation project `fpga.bspec` for synthesizing and running the audio pipeline on an FPGA. Open the project in `bluespec`, compile and link it, but do not "simulate".

Compiling and linking the audio pipeline calls the Bluespec compiler to generate Verilog code. It then uses the Xilinx tools to map and place-and-route the design for our FPGAs. This takes about 70 minutes to complete on the vlsifarm machines.

The Xilinx tools output reports to the `xilinx/` directory from which we can learn the area and critical path of our design. The file `xilinx/mkBridge.srp` is the synthesis report. It contains summary information about how many slices our design takes up and the critical paths in our design. The *Device utilization summary* near the end of the file shows the resources our design uses. The *Timing summary* lists the critical path and its length for each clock in the design. The clock for our audio pipeline is called `scemi_clk_port_clkgen/current_clk1`.

The file `xilinx/mkBridge.par` is the place-and-route report. If the design fails to meet timing constraints, it is reported in this file. You should always verify near the end of `xilinx/mkBridge.par` that it says "All constraints were met". It is not always obvious if not all timing constraints are met, so look carefully.

If place-and-route fails to meet all the timing constraints, you can look at the synthesis report `xilinx/mkBridge.srp` as described above to understand what the failing critical path is. You can also get more detailed information about the failing timing constraint by reading `xilinx/mkBridge.twr`, which gives a detailed description of what went wrong for any timing constraints not met.

Finally, we can get additional information about which parts of our design take which resources from the file `xilinx/mkBridge_map.mrp`, which shows a detailed report of resource utilization by hierarchy in the section *Utilization by Hierarchy*. This shows the resources used for each synthesized module in the design, hierarchically.

3.1 Adding Synthesis Boundaries

If you synthesized your design and looked at the utilization by hierarchy in the `xilinx/mkBridge_map.mrp` report, you'll notice the modules we are most interested in: the FIR filter, the FFT, IFFT, and PitchAdjust, are not listed. The reason is because when Bluespec compiles your audio pipeline, it inlines all Bluespec modules, generating a single Verilog module unless explicitly told not to. The synthesis tools determine hierarchy based on the Verilog modules, so they never see the original Bluespec module hierarchy.

To tell the compiler to not inline a module, use a synthesis pragma. For example, the multiplier module used by the FIR filter is defined in `common/Multiplier.bsv` and is annotated with the synthesis pragma.

```
(* synthesize *)
module mkMultiplier (Multiplier);
```

The synthesis pragma tells the Bluespec compiler to generate a separate Verilog file for the implementation of the module thus annotated. The synthesis tools then recognize it as a distinct module and will report resources specific to that module. For example, the utilization by hierarchy lists resources used by `fir_m`, `fir_m2` and so on, which are the `mkMultiplier` instantiations. The report also lists resources used by FIFOs, because `mkFIFO` is implemented in a separate Verilog file.

3.2 Synthesis Boundaries on Polymorphic Modules

We would like to see the FIR filter, FFT, IFFT, and PitchAdjust modules in the hierarchy report to know how much of the total resources each module takes, so we should add the synthesis pragma to those modules. The problem is, you can't add a synthesis boundary to a polymorphic module, because the underlying Verilog language doesn't support polymorphism the way Bluespec does. Instead of placing a synthesis boundary on a polymorphic module, we must wrap the polymorphic module in a non-polymorphic module specific to the instantiation we need and put the synthesis boundary around that.

For example, in our audio pipeline we instantiate the mkFFT module with something like:

```
FFT#(N, ComplexData) fft <- mkFFT();
```

We can make a specialized FFT in its own synthesis boundary by creating a new module for it.

```
(* synthesize *)
module mkAudioPipelineFFT(FFT#(N, ComplexData));
    FFT#(N, ComplexData) fft <- mkFFT();
    return fft;
endmodule
```

In our audio pipeline we will instantiate this specialized FFT module instead of the polymorphic `mkFFT`.

```
FFT#(N, ComplexData) fft <- mkAudioPipelineFFT();
```

Now when we synthesize for the FPGA, `fft` will be listed in the utilization by hierarchy report.

Problem 2: Synthesize your audio pipeline with N = 8, S = 2, and a pitch factor of 2.0 for the FPGA and review the synthesis reports. If your design does not meet the timing requirement of 50Mhz you need to reduce the critical path in your design by breaking up that combinational path across multiple cycles. For example, the combinational FFT implementation likely will not meet timing, so you should use one of your pipelined implementations for the FFT instead.

3.3 Running on the FPGA

Now that you have synthesized your design for the FPGA and the design meets timing constraints, all that remains is to run it!

Problem 3: Run your design on an FPGA.

1. Log into one of vlsifarm servers 03, 04, 05, 06, or 08, which have FPGAs connected to them. Because there are a limited number of FPGAs and we don't have in place anything to manage who gets to use which FPGA, you should be careful and considerate of the other people who want to use the FPGAs. If it seems someone else is using the FPGA, try a different machine. You can see who else is logged in on the computer with the `w` command.

   Source the class setup script, navigate to the lab 4 `scemi/fpga/` directory, and open up the fpga.bspec project file in bluespec if you haven't already done so.

2. We have added two new buttons to the workstation, one for programming the FPGA and another for running the test bench. Click the new button to the left of the gear icon to program the FPGA with the synthesized design.

3. Run your test bench by clicking on the other new button in the workstation with the gear icon. Make sure you have some sample audio `in.pcm` in the `scemi/fpga` directory when you do this.

4. Verify the output is correct by comparing it to the results you got in simulation.

Because the test bench sends a soft reset to the FPGA every time the test bench start, you should be able to rerun the test bench repeatedly without reprogramming the board and still get the correct results.

4 Making the Pitch Factor Dynamic

As your design currently is, the pitch factor used in the audio pipeline is specified statically at compile time. With the Sce-Mi infrastructure we can ask the user for the desired pitch factor at runtime via software running on the host computer and pass that to our hardware dynamically so the hardware does not have to be resynthesized to be used with different pitch factors.

This part of the lab asks you to make the pitch factor a dynamic parameter.

Add a register to the `mkPitchAdjust` module which holds the FixedPoint pitch factor. Initialize the pitch factor to be Invalid. Don't allow computation in the `mkPitchAdjust` module to proceed until the pitch factor has been set externally. Only allow the pitch factor to be set once, before any computation is performed, because it is not clear what it means to change the pitch factor in the middle of a computation.

Figure 2 shows what the Sce-Mi setup should look like by the end of the lab. We will start by exposing a new `setfactor` subinterface of the `mkPitchAdjust` module to the world outside the audio pipeline. We'll then instantiate a Sce-Mi port in the `SceMiLayer` and a port proxy in the test bench used to pass the pitch factor through the audio pipeline to the `mkPitchAdjust` module.

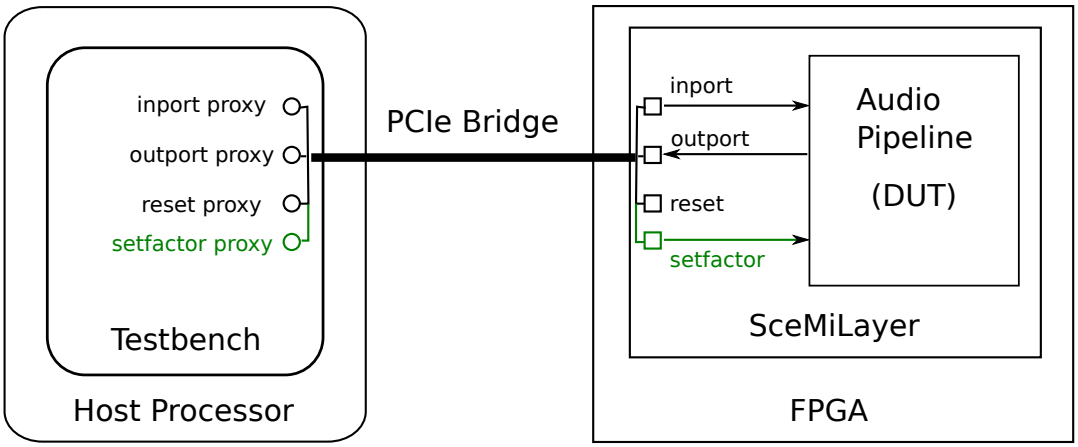

Figure 2: Sce-Mi setup with Dynamic Pitch Factor

4.1 Subinterfaces in Bluespec

Subinterfaces are a nice feature in Bluespec for assembling large designs. Subinterfaces in Bluespec let you have not only methods in a Bluespec interface, but also other interfaces. These sub interfaces can then be implemented and accessed separately in a convenient way. For example, we can change the interface of `mkPitchAdjust` to a new interface, call it `SettablePitchAdjust`, which has two subinterfaces. The first subinterface will be exactly our `PitchAdjust` interface from the previous lab. The second interface will be a `Put#(FixedPoint)` interface which allows us to set the pitch factor.

```
interface SettablePitchAdjust#(
        numeric type nbins, numeric type isize,
        numeric type fsize, numeric type psize
    );

    interface PitchAdjust#(nbins, isize, fsize, psize) adjust;
    interface Put#(FixedPoint#(isize, fsize)) setfactor;
endinterface
```

Each subinterface is given a name used when implementing the subinterface and accessing the subinterface.

Where before we implemented the `PitchAdjust` interface in the `mkPitchAdjust` module with something like

```
    interface Put request = toPut(infifo);
    interface Get response = toGet(outfifo);
```

we must now distinguish between the PitchAdjust subinterface and the setfactor subinterface. To implement the `SettablePitchAdjust` interface, we would instead write something like

```
    interface PitchAdjust adjust;
        interface Put request = toPut(infifo);
        interface Get response = toGet(outfifo);
    endinterface

    interface Put setfactor;
        method Action put(FixedPoint#(isize, fsize) x) if (...);
            ...
```

```
        endmethod
    endinterface
```

We use the `interface` keyword, followed by the type of the interface without type parameters, followed by the name of the subinterface. Between that and the `endinterface` keyword we put the implementation for that specific subinterface. Notice we already do this for `request` and `response`, which are just subinterfaces of the `Server` interface in Bluespec.

To access subinterfaces, you specify the name of the subinterface you want to use after a dot. For example, in our pitch adjust test bench, if before we instantiated a module as

```
PitchAdjust#(8, 16, 16, 16) pitch <- mkPitchAdjust(2, 2.0);
```

and call its methods as

```
pitch.request.put(v);
```

Now we will instantiate it as

```
SettablePitchAdjust#(8, 16, 16, 16) pitch <- mkPitchAdjust(2);
```

and call its methods as

```
pitch.adjust.request.put(v);
```

You can also name a subinterface.

```
SettablePitchAdjust#(8, 16, 16, 16) pitch <- mkPitchAdjust(2);
PitchAdjust#(8, 16, 16, 16) adjust = pitch.adjust;
```

and call its methods as before

```
adjust.request.put(v);
```

Notice we instantiate the module first, then use the equality operator in the next line to give a name to one of its interfaces without instantiating more hardware.

Problem 4: Change your `mkPitchAdjust` module to implement the `SettablePitchAdjust` interface as described above. Update the test bench for the `mkPitchAdjust` module to use this new interface. The `mkPitchAdjust` module should no longer take the factor as a module parameter. You will have to set the factor when the test bench starts using the `setfactor` subinterface for things to work.

We want to expose the `setfactor` subinterface inside the audio pipeline to the software test bench, which means the audio processor interface needs to be expanded too with a `setfactor` interface.

Problem 5: Change the `mkAudioPipeline` module to use an interface which exposes the `setfactor` of the `mkPitchAdjust` module to the outside world as shown in figure 3. Before we can try out the new `mkAudioPipeline` module we need to add a new Sce-Mi port for setting the pitch factor from software.

4.2 Adding a Sce-Mi Port

Because we chose to use the `Put` interface for setting the pitch factor, it is very easy to instantiate a Sce-Mi port. Bluespec provides a library of Sce-Mi ports, including `mkPutXactor`, which works with the `Put` interface. Other useful SceMi transactors provided by the Bluespec library are `mkGetXactor`, `mkServerXactor` and `mkClientXactor`. For more information on both the hardware and software libraries for Sce-Mi provided by Bluespec, see the document `$BLUESPECDIR/../doc/BSV/emVM.pdf`.

The first step in adding a Sce-Mi port is to instantiate it in the `SceMiLayer`.

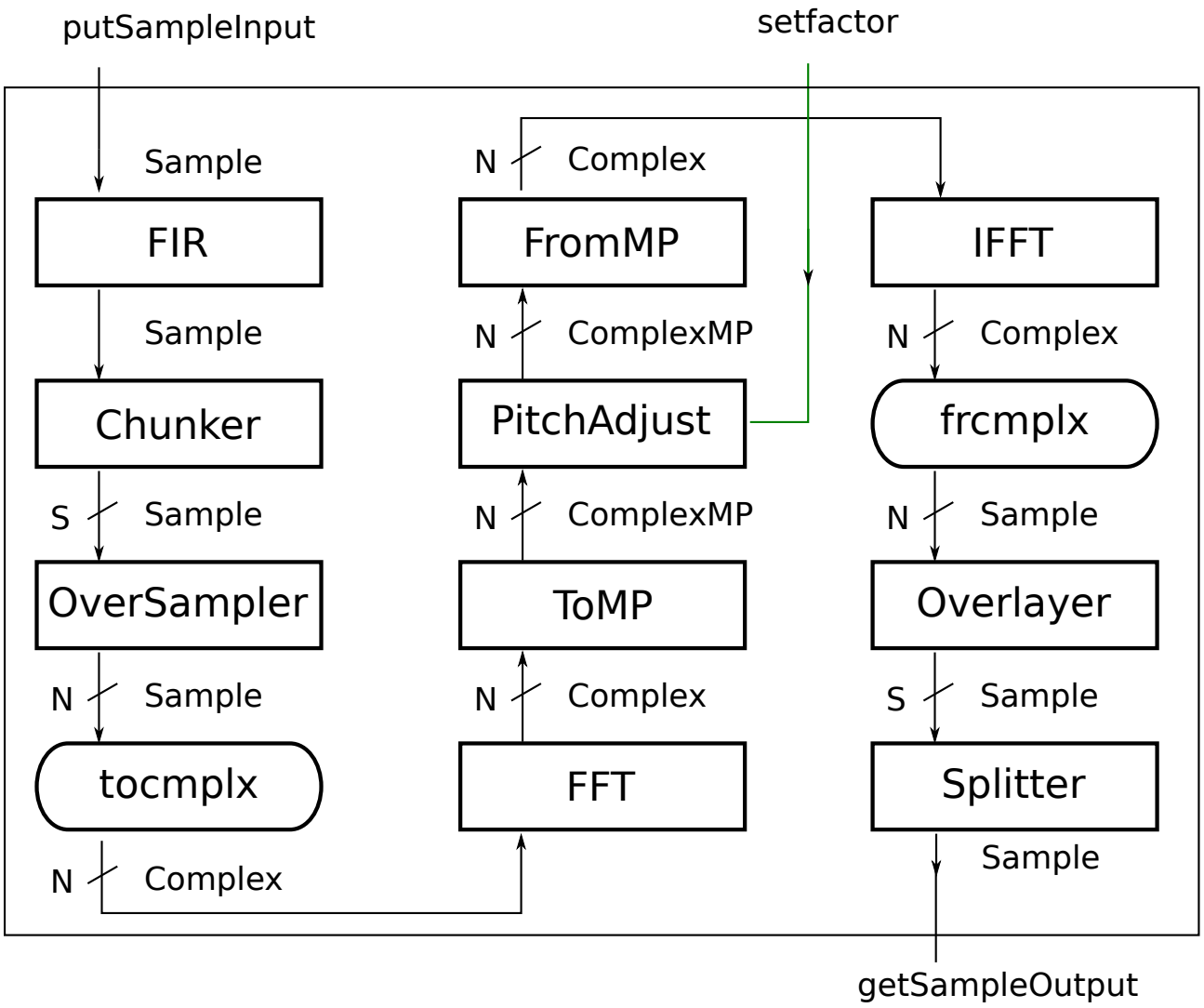

Figure 3: AudioPipeline with setfactor exposed

Problem 6: Change the `DutInterface` interface and `mkDutWrapper` module in `SceMiLayer.bsv` to expose the `setfactor` interface. Update the instantiation of `mkServerXactor` to take the audio processor subinterface of the dut as an argument, and instantiate a new port for the `setfactor` interface with something like:

```
Empty setfactor <- mkPutXactor(dut.setfactor, clk_port);
```

Now we need to update the software test bench, `Tb.cpp` to get and send the pitch factor over the Sce-Mi port to the hardware.

You will need to instantiate an InportProxyT in `Tb.cpp` which corresponds to the `mkPutXactor` you instantiated in `SceMiLayer.bsv`. When instantiating the inport proxy we need to specify the type of the data that will be going across the port and the name of the input port on the hardware side to connect the port proxy to.

The type of data our new port uses is the type of the pitch factor, a `FixedPoint#(isize, fsize)`, where `isize` and `fsize` are set by the audio pipeline code. For example, they may both be 16 bits, in which case the pitch factor is 32 bits. We can use the type `BitT<32>` with the `InportProxyT` on the software side and manually pack it to correspond with the representation used on the hardware side. This is painful and fragile however. If we change the type of the pitch factor in the audio pipeline, we would like our test bench to keep working without change.

Fortunately, Bluespec provides a script integrated in the build system for automatically generating c++ class files for types that are used on Sce-Mi ports. If we define a typedef somewhere in the audio pipeline such as:

```
typedef FixedPoint#(16, 16) FactorType;
```

and then use that typedef in the `DutInterface` `setfactor` subinterface, then Bluespec will generate a file called `tbinclude/FactorType.h` defining a corresponding c++ type for `FactorType` which we can refer to in the test bench code.

The `FactorType`, because it is a FixedPoint, will have the same structure as FixedPoints in Bluespec, it will have a field `m_i` with the integer part and `m_f` with the fractional part. The length of these fields can be tested for in software using the `getBitSize` method. For example, to create a FactorType object in the test bench from a double `pf` which is the pitch factor, you could do

```
FactorType factor;
factor.m_i = (int)floor(pf);
factor.m_f = (int)(pow(2, factor.m_f.getBitSize()) * (pf - floor(pf)));
```

To send the FactorType object over the setfactor port proxy is now as easy as

```
setfactor.sendMessage(factor);
```

The other piece of information we need before we can instantiate our setfactor port proxy in software is the name of the corresponding port instantiated in the `SciMeLayer`. This name is determined based on the module hierarchy of the generated `Verilog` code and is not always obvious from the Bluespec code. A good way to identify the appropriate name is to first compile the Bluespec code, then look at the file `scemi.params` which the compilation process generates. The `scemi.params` lists for each port its name, width, and type. Copy the name from the `scemi.params` file into your test bench when instantiating the setfactor `InputProxyT`.

Problem 7: Update your SceMi test bench to accept the pitch factor dynamically and pass it to your audio pipeline via SceMi. Verify the change works both when simulating Sce-Mi, and when running on the FPGA. The following changes will be required in `Tb.cpp`.

1. Add code to get the pitch factor from the test bench `argv[1]` parameter to the main function. You can use the function atof to convert the string to a double in C.
2. Instantiate an `InportProxyT` of appropriate type and link it with the appropriate hardware port by specifying the right name (you may need to compile the Bluespec code first and look at `scemi.params` to figure out this name.)

3. Convert the pitch factor from a double to a FixedPoint and send it over the factor port using the sendMessage method after resetting the dut, but before running the test.

You can add an argument to the `tb` command run by the workstation by changing the project options for simulation and changing the `bluespec_init` file for the FPGA.

5 Discussion Questions

1. Report the total number of Slice Registers and LUTs used in your design after making the pitch factor dynamic. Report the Slice Reg, LUTs, and DSP48E used by each of the AudioPipeline, FIR, FFT, ToMP, PitchAdjust, FromMP, and IFFT modules.

2. Report the length of the critical path of your audio pipeline. Can you tell where in the design your critical path is?

3. Did you run into any problems using the FPGA?

4. What advantages are there to using subinterfaces in an interface instead of just using methods?

5. If you try to synthesize the audio pipeline with a 32 point FFT, you'll see it doesn't fit on the FPGA. Ideally we could use a 1024 point FFT and still maintain the target rate of processing 44100 samples per second. Using the circular pipeline for the FFT, it would take 10 cycles to calculate the FFT of 1024 samples. At 50 Mhz this means the FFT supports a throughput of $1024/10 * 50,000,000 = 5,120,000,000$ samples per second, which is much greater than the $16 * 44100 = 705600$ samples per second our application requires the FFT to support assuming we have an overlap of $N/S = 16$. Also, with a 1024 point FFT, the input vector is whopping $1024 * (2 * (2 * 16)) = 65536$ bits wide. Given our design of the FFT has a much higher throughput than we need and an excessively large input width, how could we change our implementation of the FFT to both fit on the FPGA using 1024 points and still meet our required sample rate? Would the rest of the audio pipeline have to change to support this as well?

6 What to Turn In

When you have completed the lab you should check in a final version via git. This should include the pipeline updated with dynamic pitch factor and answers to the discussion questions in a file called `lab4` in the `answers` directory. For example

```
audio$ git add -u .
audio$ git add answers/lab4
audio$ git commit -m "Lab 4 submission"
audio$ git push
```