Lab 4: Audio Pipeline on an FPGA

6.375 Fall 2019
Assigned: September 30, 2019
Due: October 7, 2019 11:59PM ET

1 Introduction

In this lab we will run the audio pipeline constructed in the previous labs on an FPGA. Emulating a hardware design or design-under-test (DUT) on an FPGA achieves much better performance than software simulation of the design can provide. However, running a non-simulation test bench that controls the hardware design on an FPGA is another challenge. We introduce ***connectal*** as a means for communication between a host processor (software) and FPGA. Connectal integrates Bluespec designs with software, allowing us to easily run and test our designs on FPGAs. We first test the audio pipeline by simulating the processor-FPGA link using connectal. We will then synthesize the audio pipeline for the FPGA, look at area and timing results, and run the audio pipeline on an FPGA. Following that we will ask you to make the pitch factor a dynamic parameter passed to the audio pipeline at run time using the connectal request interface.

1.1 Connectal

Connectal[1] consists of a fully-scripted tool-chain and a collection of libraries which can be used to develop applications composed of software components running on CPUs communicating with hardware components in FPGAs. Connectal takes in your software written in C/C++ and hardware written in Bluespec and automatically generates components required for hardware-software communication. In connectal, a virtual communication link between hardware and software is called a *portal*, and by convention, we call a portal from SW to HW a *request* and a portal from HW to SW an *indication*. Connectal can build your design for simulation (HW and SW-HW links are simulated) and for various target FPGAs and physical links (e.g., PCIe, AXI).

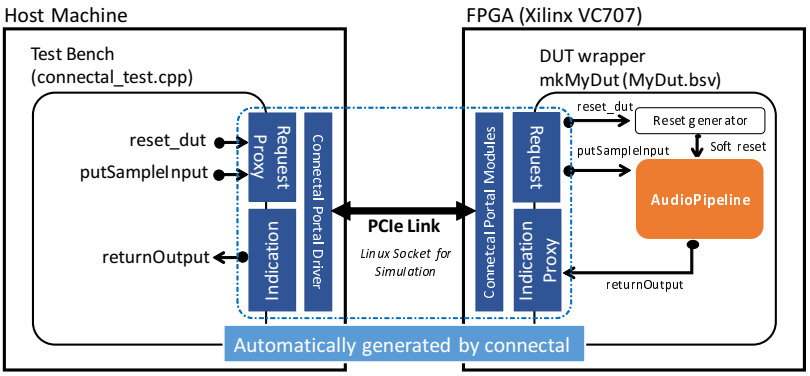

Figure 1: Audio Pipeline with Connectal

Figure 1 shows how the test environment looks using connectal to pass the samples from the host processor to the FPGA and back. The `mkAudioPipeline` module becomes our design under test (DUT). `mkMyDut` is a wrapper around our DUT which hooks it up to the auto-generated connectal *request* and *indication* interfaces. `mkMyDUT` also instantiates a reset generator to soft-reset `mkAudioPipeline`. Samples are sent by a C++ function `putSampleInput` of the generated request proxy that is translated to a hardware method of the request sub-interface of `MyDut`. Samples coming out of the audio pipeline will be sent back to software by calling an indication method `returnOutput` in hardware which eventually calls a call-back function `returnOutput` defined in software.

[1] `https://github.com/cambridgehackers/connectal`, `http://www.connectal.org`

In this lab, we use Xilinx VC707 FPGA evaluation boards[2] and they are connected to our course machines via PCIe links. Connectal automatically generates all the components for PCIe communication and package them with `myMyDut` to build an FPGA image for VC707.

For our audio pipeline to work with connectal, we do not need to change the audio pipeline code from the previous lab. What we do is wrap the audio pipeline in `mkMyDut` (`MyDut.bsv`) and replace `mkTestDriver` with a new test bench written in C++ (`connectal_test.cpp`) running on the host processor which sends and receives samples through connectal portals.

1.2 Getting Started

The additional infrastructure needed to use connectal for communication between the host computer and FPGA is provided in the lab4 harness. Also, the details on course machines have been updated, please refer to the course website – Resources.

Update your local repository with this new infrastructure. Add the 6.375 course locker, source the setup script, navigate to the directory containing the `audio/` from the previous labs, and run:

```
$ tar xf /mit/6.375/lab2019f/lab4-harness.tar.gz
```

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

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

The `connectal/` directory contains a number of files. Read the description below and the content of the files carefully before moving on:

`Makefile` describes a connectal project. It defines the names of the SW-HW interfaces for auto-generation. The clock speed for the project is specified using `--mainclockperiod` flag in nanoseconds. The default Makefile will try to build your design to run at 66.67 MHz (T=15ns).

`MyDut.bsv` implements the connectal wrapper around `mkAudioPipeline`. The wrapper module is `mkMyDut` whose interface type is `MyDut`. In the interface `MyDut`, it has a sub-interface named `request` whose type is `MyDutRequest`.

The `MyDutRequest` interface defines a list of methods that can be called by software. These method cannot use custom types for arguments but only `Bit#(n)` types. `n` is usually 16 and/or 32 (mapped to `uint16_t/uint32_t` data types in C++) and connectal parses the bit-widths to generate the proxy functions (each has corresponding request method) for software.

`MyDut.bsv` also defines an interface `MyDutIndication`. This interface defines a list of methods, each of which has a corresponding call-back function in software. Indication methods can only use `Bit#(n)` types for arguments. Our wrapper module `mkMyDut` takes in a `MyDutIndication` inferface as an argument and name it `indication`. Connectal automatically generates and provides this interface to `mkMyDut` during compilation. In `mkMyDut`, one can invoke the software call-back functions using `indication` and its methods.

`connectal_test.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`.

All the methods that are defined in the Bluespec `MyDutIndication` interface must be defined as members of C++ `MyDutIndication` class. If methods have arguments, connectal maps it to `uintXX_t` (e.g., `uint16_t/uint32_t`) types depending on bit-widths defined in Bluespec. Hardware request methods can be called using `device`, an instance of `MyDutRequestProxy`.

When running software, a separate thread is generated to invoke indication functions. Thus, shared variables between the main thread (the one sends requests) and the indication thread must be properly protected using mutexes.

[2] `https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html`

2 Simulating with Connectal

We introduced connectal 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 connectal 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 connectal 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 connectal 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 a linux socket instead of PCIe.

Problem 1: Simulate the audio pipeline with the connectal infrastructure by taking the following steps in the `connectal/` directory.

1. Copy over our test input to the local directory.

   ```
   connectal$ cp ../data/mitrib.pcm in.pcm
   ```

2. Build the simulation.

   ```
   connectal$ make -j8 simulation
   ```

3. In the generated `bluesim/bin/` directory, two files `ubuntu.exe` and `bsim`, which represent the software and hardware, respectively, are generated. The software executable `ubuntu.exe` automatically launches `bsim` so you don't need to run it explicitly. In the `connectal/` directory, run the simulation using either of the followings.

   ```
   connectal$ make run_simulation
   --- or ---
   connectal$ bluesim/bin/ubuntu.exe
   ```

   Both the print out from software and hardware will show up. This takes about 3-4 minutes since connectal adds more complexity to the simulation.

4. Verify the output from simulation by comparing it to the result provided for Lab 3.

   ```
   connectal$ cmp out.pcm ../data/mitrib_pa_2_2.pcm
   ```

3 Running on an FPGA

We will now prepare our project to run on an FPGA, Xilinx VC707.

3.1 Adding Synthesis Boundaries

Prior to synthesizing the hardware, we should tell the Bluespec compiler how we want to break up our design into Verilog modules. By default, the compiler inlines all Bluespec modules, generating a *single flat* Verilog module. The synthesis tools determine hierarchy based on the *Verilog* modules, so they never see the original *Bluespec* module hierarchy. This makes debugging timing errors and anlayzing resource utilizations difficult since we are unable to see the module hierarchy we expect (e.g., FFT, IFFT, FIR, PitchAdjust) in the synthesis tool reports.

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 *)`.

```
(* 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 produce reports specific to that module.

3.2 Synthesis Boundaries on Polymorphic Modules

We would like to create synthesis boundaries for all of the components of our pipeline to understand their individual resource utilizations and timings. The problem is that 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 `mkAudioPipelineFFT` 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: Create synthesis boundaries for your audio pipiline for the following modules: AudioPipeline, FIR, FFT, ToMP, PitchAdjust, FromMP, and IFFT. Use parameters N=8, S=2, and a pitch factor of 2.0.

3.3 Building Project for FPGAs

We are ready to build our design to run on a Xilinx VC707 FPGA board. Build the project using:

```
connectal$ make -j8 fpga
```

This creates a `vc707g2` directory[3] to build software and hardware for your design. Once the build completes, the directory size is about 130 MB for this project, so make sure you have enough space in your Athena account. Never try to add/push contents in this directory to git. Connectal first reads your design and generates software and hardware components to facilitate the HW-SW communication. Once all components are generated, connectal builds a software executable. At the same time, connectal calls the Bluespec compiler to generate Verilog files. Generated verilog files are located in `vc707g2/verilog/`. You should see a separate Verilog file for each of your modules of interest if synthesis boundaries were added correctly. Then, connectal uses the Xilinx tool, Vivado, to synthesize and place-and-route the design for our FPGAs (VC707). This takes about 30-40 minutes to complete on the course machines.

Once the build is done, connectal may print out timing violation if it could detect it from the generated report. Sometimes it misses violation so make sure you read the report described in the following section. All the generated files and reports are located in `vc707g2/bin/`.

After the entire build is done, if you want to update software only, you can run (no HW build):

```
connectal$ make -j8 fpgaUpdateSW
```

[3] meaning VC707 connected via PCIe Gen 2

3.4 Timing and Resource Analysis

In `vc707g2/bin/`, the resource utilization is reported in `top-post-route-util.txt`. The report hierarchically shows how much of each FPGA resource type is used for each module that Vivado identifies. You can search for modules that you had synthesis boundaries to see their utilization. If you see two lines for the same module, use the first one. The second one with parentheses is the remainder that does not belong to any of the sub-modules (the second belongs to the first).

The `top-post-route-timing-summary.txt` reports the clocks and timing summary. Our audio pipeline is clocked by `connectal_main_clock` configured to run at 66.67 MHz (set in Makefile). If you search for `connectal_main_clock` in the file, the first occurrence shows you its period (15 ns) and frequency (66.67 MHz). The second occurrence is in the Intra Clock Table. The second column reports the total negative slack (TNS) and the third column reports the number of failing endpoints (TNS Failing Endpoints). **If your design has met timing, both columns should be 0.** If your design did not meet timing, search for VIOLATED in the same file. If there are multiple occurrences, find the one whose preceding lines have `connectal_main_clock` as both From and To Clocks. If your design met timing, search for `connectal_main_clock` to find a place where it reports a max deplay path that uses `connectal_main_clock` as both From and To Clocks. Regardless of being MET or VIOLATED, this part reports the critical path within our audio pipeline design.

If your design does not meet the timing requirement, you need to break up the identified critical path across multiple cycles. For example, the combinational FFT implementation likely will not meet timing, so you should use one of your pipelined implementations instead. Repeat the build process until your design meets timing. **Please do not attempt to run the design that did not meet the timing since bad hardware can cause the course machines to crash.**

While connectal generates text-based reports for our convenience, it is possible to open the final FPGA implementation of your design using Xilinx Vivado GUI to see the reports and implemented design graphically. If you want to use the GUI tool to see your final design, you can run (optional):

```
connectal$ vivado vc707g2/bin/top-post-route.dcp &
```

3.5 Running on an FPGA

Now that you have synthesized your design for the FPGA and the design meets timing constraints, all that remains is to run it! Generated in the `vc707g2/bin/` directory, `mkTop.bin.gz` is a VC707 FPGA binary image and `ubuntu.exe` is an executable of software. `ubuntu.exe` by default loads the FPGA image to the FPGA when executing, or requires a NOPROGRAM=1 environment variable to suppress automatic FPGA programming. All the commands needed to program the FPGA and run software are integrated into `Makefile`, so you can take a look if interested.

Problem 3: Run your design on an FPGA.

1. After the build is done, log into one of the FPGA servers listed on the course website under Resources. Because there are a limited number of FPGA servers, you should be careful and considerate of the other people who want to use the FPGAs. Please log into the FPGA servers only when you are using the FPGA and log out as soon as you are done using it. Simulation and FPGA builds should be done on build servers, not on FPGA servers. We have implemented a lock so that you can use the FPGA exclusively for 2 minutes or until the commands below complete, whichever is shorter. Note that FPGA programming takes about 20 seconds and test bench software takes around 10 seconds. If you see a message: "lockfile: Sorry, giving up on /tmplock/fpgalock", please retry in a few minutes as another student is holding the lock.

2. In `connectal/`, program the FPGA and then run software. The first command programs the FPGA and the second command runs the software without programming the FPGA. Try to run software multiple times without re-programming the FPGA and see if you get the same results every time you run. Make sure you have `in.pcm` in the same directory.

```
connectal$ make program_fpga
connectal$ make run_fpgaSW
```

3. (Alternative to 2) The following command programs the FPGA and then runs the software automatically.

```
connectal$ make run_fpga
```

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 connectal 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. On the next run, registers will be cleared by soft reset, i.e., register values initialize back to the value specified in `mkReg(init_value)`;

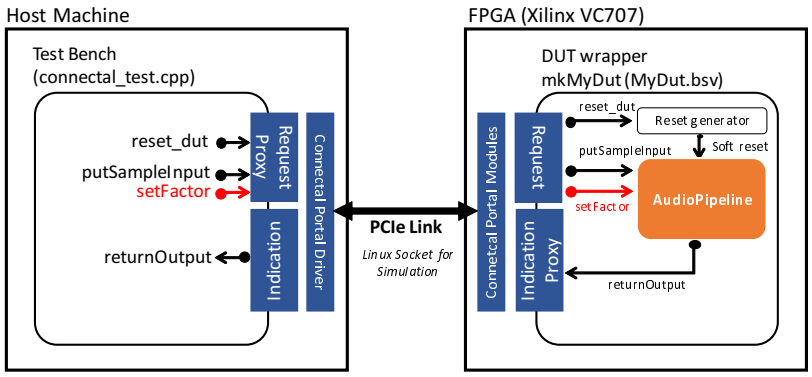

Figure 2: Connectal setup with Dynamic Pitch Factor

Figure 2 shows what the connectal 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 add a request method `setFactor` in the `MyDutRequest` so that the software test bench can pass the pitch factor through it.

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 test bench (`PitchAdjustTest.bsv`), if before we instantiated a module as

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

and call its methods as

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

Now we will instantiate it as

```
    SettablePitchAdjust#(8, 16, 16, 16) pitch <- mkPitchAdjust(2); // module name changed!
```

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 in `PitchAdjustTest.bsv` 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.

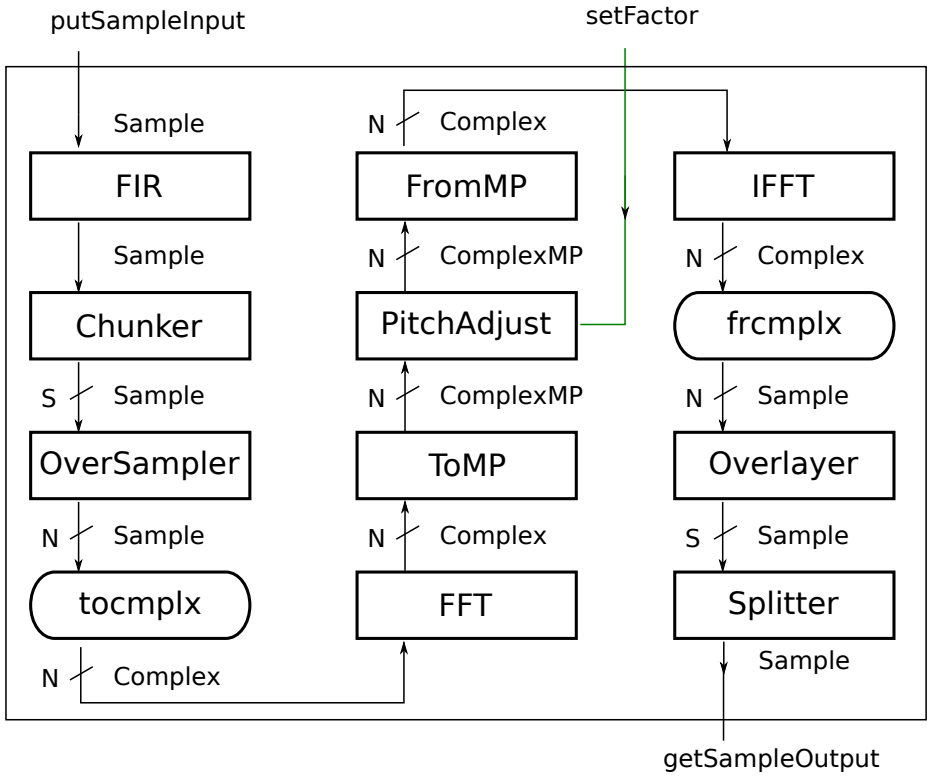

Figure 3: AudioPipeline with setfactor exposed

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. You can create a new method or a subinterface in the `AudioProcessor` interface. Before we can try out the new `mkAudioPipeline` module we need to add a new connectal request method for setting the pitch factor from software.

4.2 Adding a Connectal Request Method

Let's add a connectal request method `setFactor` that software can invoke. This should call the new method/subinterface method we created for `AudioPipeline` to dynamically set the pitch factor.

Problem 6: Change the `MyDutRequest` interface and implement the new method `setFactor`:

1. Add the `setFactor` method in `MyDutRequest` interface:

```
method Action setFactor (Bit#(32) factorPkt);
```

2. Implement the new method at the end of the `mkMyDut` module.

```
interface MyDutRequest request;
 ...
    method Action setFactor (Bit#(32) factorPkt) if (!isResetting);
    // your implementation
    //  You can use "unpack(factorPkt)" to cast factorPkt to Fixed#(16, 16)
    //  Actual bits remains the same.
    endmethod
endinterface
```

   Note that the argument type of the method is `Bit#(32)` instead of `Fixed#(16, 16)`. This is because connectal only supports Bit types for arguments used in HW-SW interfaces that are actually sent over the physical link, i.e., PCIe. The `unpack` function casts the Bit representation into other types used in Bluespec. This does not change 0s and 1s of the physical wires.

Once you add the new `setFactor` method in the connectal request interface, connectal automatically generates a new proxy function that software can invoke. Now, we have to update the software to set the pitch factor before it transfers `in.pcm`.

Before we add lines to software, we have to discuss how to generate 32-bit unsigned integer value to be used as an argument to the function. The `FixedPoint#(16, 16)` type in Bluespec expects the upper 16 bits to be the integer part and the lower 16 bits to be the fractional part. Using this, we can construct a 32-bit unsigned integer from a double `pf`:

```
double pf = ...; // holding pitch factor
uint16_t m_i = (uint16_t)floor(pf);
uint16_t m_f = (uint16_t)( pow(2, 16) * (pd - floor(pf)) );
// 32-bit unsigned integer that can be used by software
uint32_t factorPkt = (uint32_t)(m_i << 16) | m_f;
```

Problem 7: Update your connectal test bench (`connectal_test.cpp`) to accept the pitch factor dynamically and pass it to your audio pipeline via connectal. Verify the change works both when simulating connectal, and when running on the FPGA.

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++. You may want to include `stdlib.h` and `math.h` for lab 4.

2. Convert the pitch factor from a double to an unsigned 32-bit integer value (`uint32_t`) and send it to hardware using

   ```
   uint32_t factorPkt = ...;
   device->setFactor(factorPkt); // invoke HW method
   ```

3. Build the design again for simulation and fpga.

   ```
   connectal$ make -j8 simulation
   connectal$ make -j8 fpga
   ```

   Please READ your timing report and do not rely on the build command printing the violation. If you implemented your PitchAdjust in a combinational manner, it is likely that your design does not meet the timing due to the dynamic pitch factor in hardware. If this is the case, you have to update your PitchAdjust to be staged, pipelined and/or folded. The slowest clock allowed is 62.5 MHz (`--mainclockperiod 16`) if you have difficulty in meeting timings for the default speed 66.67MHz (`--mainclockperiod 15`).

4. To run the software with an argument AFTER you re-build your project for simulation or fpga, you can do:

```
connectal$ ARG1=2.0 make run_simulation # this runs simulation
--- or ---
connectal$ make program_fpga        # this re-programs the FPGA
connectal$ ARG1=2.0 make run_fpgaSW # this runs software only
--- or ---
connectal$ ARG1=2.0 make run_fpga   # this programs fpga and then runs software
```

5 Discussion Questions

1. Report the number of Total LUTs, FFs, and DSP48 used and their percentage in your entire design (`mkPcieTop`) after making the pitch factor dynamic. Report the Total LUTs, FFs, and DSP48 used and their percentage 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
```