Lab 1: A Simple Audio Pipeline

4541.763 Laboratory 1

Assigned: September 10, 2009
Due: September 24, 2009

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 power and performance (our design must have high throughput and be low power), especially if we are dealing with real-time applications, running on mobile platforms.

Figure 1 shows a high-level picture of the audio pipeline and the driving software infrastructure. 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. We can also add hooks to check the correctness of the output when running tests.

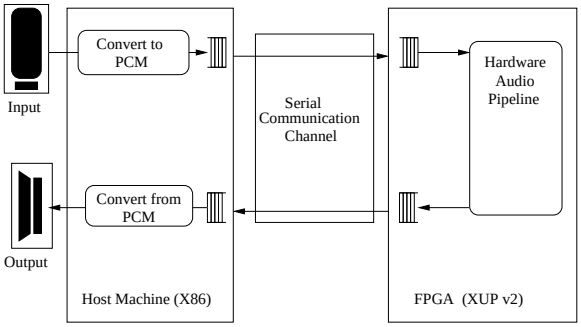

Figure 1: High-level Audio Pipeline Diagram

The box marked Hardware Audio Pipeline contains the hardware which performs the digital signal processing. Hardware of this type usually consists of a series of blocks. 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. The final blocks would include an IFFT (Inverse FFT) to convert it back to the time domain, and possibly a windowing function. Initially, this box just pass through values; essentially a loop-back device. However, by the end of this lab, we will add a FIR (Finite Impulse Response) filter which can be used to attenuate specified frequency ranges. Over the next few weeks, we will augment it further.

1.1 HW/SW Interaction

Getting the HW and SW to talk to each other is no trivial matter, and there are some interesting subcomponents which make all this work when running on the FPGA. While conceptually, the hardware and software components communicate directly to each other through a pair of unidirectional FIFOs, the actual implementation of these FIFOs requires a bit of magic. We have chosen to implement these FIFOs by multiplexing a serial connection between the host machine and the FPGA. Using a communication stack built on top of this serial connection, we are able to provide the appropriate interfaces to both the hardware and software components. The software stack is

relatively straight-forward, and builds upon the Linux kernel's serial port abstraction. The FPGA side required substantially more effort. First we instantiate a microblaze core (a simple in-order processor) onto the FPGA fabric. We have written some "firmware" to drive the serial UART, transferring data to and from FPGA block RAMs. From these block RAMs, we have additional code which passes the data over the clock boundary to and from the domain in which the hardware component of your audio pipeline is running. Some of these details are shown in Figure 2.

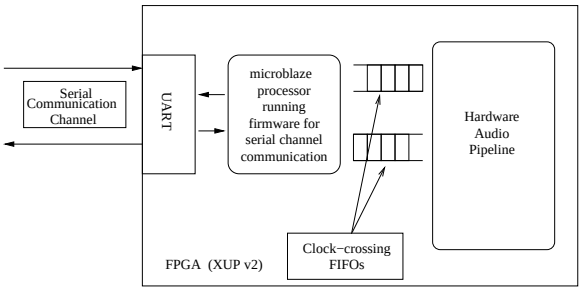

Figure 2: Details of Serial Communication

Luckily for you, many of these details are in place, and while we don't expect you to touch this infrastructure code, it is important that you understand the actual details of the system as this might influence how you eventually use it.

1.2 Lab Organization

This lab begins with an explanation of the tools used to compile and run this system. Next, we give you concrete instructions on how to set up the virtual machine environment in which all these tools can work. That is followed by a detailed tutorial on how to use the tools to configure a null audio pipeline which you will execute in multiple environments. Lastly you will write a Bluespec module which implements a simple audio filter and insert it into the pipeline created in the tutorial.

While the focus of this class is designing hardware, significant systems hacking is required to get a design to execute, either in simulation or on the FPGA, and to verify its correctness. Often the same design must be run in different environments, be it by simulation, on the FPGA platform, or using some hybrid of the two. Each environment or platform has its own tools and the collection of tools needed to be invoked to build an executable image is usually quite complicated. Experience has shown that even for very simple hardware design projects the complexity of building the executable environment can overwhelm even experienced designers. We will use **AWB** (the Architect's Work Bench), an open source tool from Intel, to shield you from the complexity of the build process. AWB also has many other useful features to facilitate code reuse between projects. For each lab, we will create a number of AWB "models", all sharing many of their components.

This lab is primarily about understanding how to use AWB the actual hardware design and the Bluespec code involved is quite trivial. By the end of this lab, you will understand the tool infrastructure and have written your first Bluespec module; a very simple audio processing pipeline. Do not be intimidated by the size of this handout, which consists mostly of a tutorial. You should have to write no more than twenty lines of code!

2 Tool Flow

Figure 3 shows the different tools we use in this lab to compile and execute your designs, as well as their dependencies. Note that we will be adding tools to this picture as the semester progresses.

To compile Bluespec SystemVerilog, we use the Bluespec Compiler, BSC. BSC can generate C for simulation, or Verilog, which is then synthesized for the FPGA using a tool known as XST. In

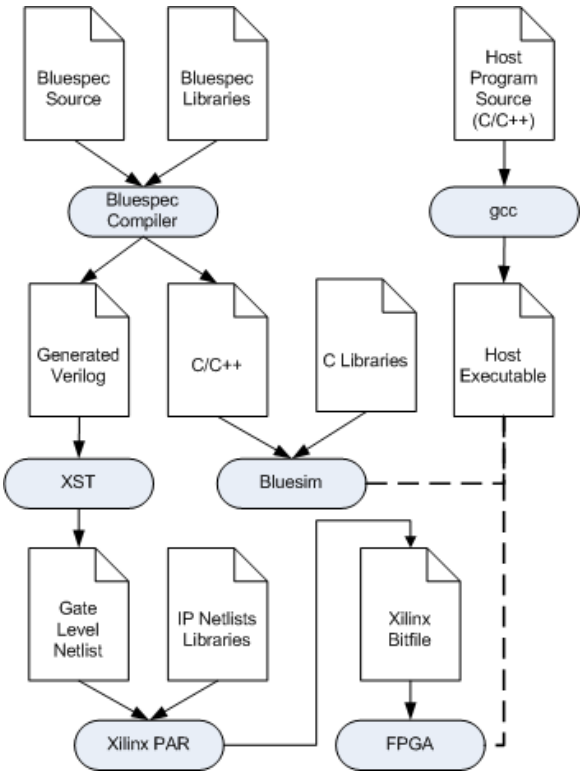

Figure 3: Toolflow for Lab 1

addition, we use gcc/g++ to compile our software, and just about every scripting language you can imagine!

Another tool which we use extensively is AFS (Andrew File System), a distributed file system developed at CMU. AFS is tightly integrated with Kerberos, an authentication mechanism developed at MIT. AFS is extremely useful for distributed computing, and in this course, we use it to store the tools, course infrastructure, and the lab code (including the code you will write).

AFS leverages Kerberos' secure authentication mechanism to allow for fast and secure file system transactions with the user. To interact with AFS first, the user obtains a ticket from the Kerberos server using the command `kinit`. Similarly the AFS server has its own ticket which it gets from Kerberos when it starts. When the user wants to start accessing files on AFS, they invoke the command `aklog` orchestrates the passing of both the user's and the AFS server's tickets to the trusted Kerberos server. This allows Kerberos to authenticate both the user's and the AFS server's identities. Kerberos then hands a token back to the user with which it can directly authenticate to AFS without checking again with Kerberos. For security reason, these tokens expire after a number of hours (in our setup, 24 hours), which means you will need to grab new tokens periodically. Both AFS and Kerberos are extremely powerful tools which you might find useful in your own work. Of course, there is a lot of documentation all over the Internet; some of it good, and much of it mediocre.

Most of these tools will be managed by AWB, but occasionally we will ask you to use them directly. In that case, it's good to understand how they all fit together.

2.1 Setting Up Your Machine

In order to run the tools, you will need to run Linux. To standardize the environment, we have chosen to preconfigure a VMWare virtual machine for you to use. You will also need a computer with USB and serial ports. The USB is used to program the FPGA board, and the serial to communicate

with the designs running on the chip.

To setup your machine do the following:

1. Download the newest version of VMWare Player from http://www.vmware.com/products/player/ and install it on your PC. Keep in mind that these tools take lots of memory. We recommend using a machine with as much RAM as possible, so that you spend less time swapping while your projects build. When installing VMWare Player, make sure that you configure it with as much memory as you can.

2. Now, go to http://davinci.snu.ac.kr/courses/emb/2009-2/handouts.html for instructions on downloading the virtual machine. Untar the file and open it using the VMWare Player. Boot your virtual machine by selecting the `.vmx` file.

3. Log onto the virtual machine with the username ‘‘`user`’’ and password ‘‘`password`’’. Logged in as ‘‘`user`’’, you have administrative access to change settings on the virtual machine, so once installed, you could change the account password to protect your work. Be aware that when working on the labs, you will need to have access to the Internet as the tools must access resources such as license servers etc. and cannot work offline.

4. As the last step in setting up your machine. There are two possible options for how do this. The simplest is to create a local directory on your virtual machine. This will be faster but will not allow you save intermediate data in AFS, only your explicit commits. Alternatively, you can put your workspace on AFS. This will give you the ability to more easily move from virtual machine to virtual machine (this may be very useful if you wish to work on multiple computers) and will prevent you from having to do some recompilations but will make compilation slower as you will make more use of network storage.

   To do this, you need to obtain Kerberos tokens and register them with AFS. To do this, type `kinit <username>@HYEWON.SNU.AC.KR` and then type `aklog -cell HYEWON.SNU.AC.KR -k HYEWON.SNU.AC.KR`. Next execute the command `source setup.sh`, which is located in your home directory.

   Remember that you will have to source the setup script every time you start up a new shell. Kerberos tickets will need to be renewed every 24 hours.

5. Change the password on your AFS account. This will prevent people from being able to type in your initial password and change with your class repository. To do this, type `kpasswd <username>@HYEWON.SNU.AC.KR`. you will be prompted for your old password and then your new password. Please select a strong password (at least 8 characters with a mix of lowercase and capital letters, numbers, and special characters,*i.e.,*$,#,%,,).

6. Create your workspace directory. If you want to build a local workspace, run the command `mkdir ~/workspaces`.

   If you want to store it over AFS, create a local link to your AFS workspace by running the command:
   `ln -s /afs/hyewon.snu.ac.kr/user/6.375/workspaces/<username> ~/workspaces`.

   These two options have tradeoffs: Storing your workspaces on AFS has the advantage of preserving your intermediate files as you move between machines. For example, you could run the synthesis tools on your laptop, and download the design to the FPGA using one of the Lab desktops. Be aware that we have limited the size of your workspace directories on AFS to two Gigabytes, so as your projects grow, you may run out of space (though we might consider raising this limit). Additionally, we have configured the AFS client running on the virtual machine with a two Gigabyte cache. If the size of the cached tools plus your workspace exceeds this limit, performance will degrade pathalogically. This particular symptom can be relieved by reconfiguring your AFS client using dpkg-reconfigure and increasing the size of the local cache. Remember that the size of your local cache and the size of your workspaces directory on AFS are two seperate issues. Creating a local workspace directories bypasses all

of these issues, but makes your work less portable. Remember, use of the Lab desktops is only necessary if your laptop lacks either `serial` or `usb` ports.

7. Finally, copy the package file `/afs/hyewon.snu.ac.kr/user/6.375/workspaces/<username>/mit-6.375-<username>.pack` to the directory `~/.asim/repositories.d/`. This makes AWB aware of your personal repository.

2.2 Learning Bluespec

We are following a two-part strategy to teach Bluespec. In the lectures, you are exposed to the concepts underlying Bluespec and how Bluespec should be used to design hardware; Architecture ideas dominate these lectures. In the labs, you will be writing actual Bluespec code requiring you to master the Bluespec SystemVerilog (BSV) syntax. In most modern languages, it is generally quite difficult to write a new program entirely from scratch. Our approach will be to provide you with an initial BSV design which you will have to read, understand, and then modify appropriately. For example in this lab, Section 3 guides you through the creation of a working Bluespec program which you are supposed to compile and run using the class tools. Section 4 gives you working code for a FIR filter, which you must read and understand, before compiling it and executing it. Furthermore, Section 4 then defines two variants of the original FIR filter which are to express in Bluespec by modifying the original code. This approach will let us deal with much more interesting examples than would be possible were we to ask you to write them from scratch. We hope that this will be as a more natural and interesting way to master the language.

Unfortunately there is not yet a textbook on Bluespec, but the following resources collectively provide a lot of useful information.

- **Running the Compiler yourself:** While we hide most of the Bluespec Compiler details with AWB, to learn more about the language, it is still worthwhile to be able to directly call the compiler yourself. After you have run `setup.sh`, you should be able to invoke the Bluespec Compiler(`bsc`) from the command line. Running `bsc` gives a good initial listing for common tasks, *e.g.,*compiling to Verilog, or building a Bluesim object and subsequently linking objects into an executable. You can get more detailed information on running the compiler from the Bluespec wiki.

- **The Bluespec Wiki:** The Bluespec community has spend a lot of time working on a wiki (http://sites.google.com/a/bluespec.com/learning-bluespec/Home) to help new and experienced users learn more about the language. We haven't looked at the site for very long, but it seems like it has almost everything you could want. This site has links to all of the other references listed here, but we are giving them to you explicitly because they worth extra notice.

- **Bluespec System Reference Manual - Bluespec Inc:** This document describes all of the syntax and common libraries. As named it is a great reference but is not meant to be read like a textbook. As search is main value of the document, this is best used as an online tool, not a printed copy. (http://asim.csail. mit.edu/redmine/attachments/61/reference-guide.pdf or on AFS at `/afs/hyewon.snu.ac.kr/user/6.375/tools/bluespec/Bluespec-2008.11.C/doc/BSV/user-guide.pdf`)

- **Bluespec Video Lectures - R.S. Nikhil, CTO of Bluespec:** These lectures are 3-4 years old. They provide a slightly different perspective than the style followed in the class. Many practicing engineers have found them useful when they start learning Bluespec (http://www.demosondemand.com/dod/proddemos/vendors/pd_bluespec.aspx)

- **Bluespec Papers:** There are a number of paper written on Bluespec. Many of the ones written by MIT can be found on Arvind's site, though it is a bit out of date: http://csg.csail.mit.edu/pubs/publications.html. Bluespec also has a number of technical papers and documents at

http://bluespec.com/forum/viewforum.php?f=15&sid=052026a10b38af822615270c3020d33c as well as a listing of technology and industrial references at http://www.bluespec.com/why-bluespec/technology-references.htm

2.3 Other Useful References for the Class

We have compiled a (by no means exhaustive) list of resources which you may find helpful as you accustom yourself with the work environment and learn to use the different languages and tools.

1. **AWB**: If you are interested in some background information on AWB, a good place to start is: http://asim.csail.mit.edu/redmine/attachments/52/EmerAWB.pdf. There is also a Wiki (with an associated discussion forum) associated with AWB located at http://asim.csail.mit.edu/redmine/wiki/awb. This discussion forum is monitored, so questions posted there will receive attention from the good people at Intel.

2. **Wiki**: There is also a course Wiki http://asim.csail.mit.edu/redmine/projects/show/mit-6375, which includes some helpful topics. This also has an associated discussion form monitored by the course TA's.

3. **Bluespec**: The Bluespec Language Reference is located on the web at http://asim.csail.mit.edu/redmine/attachments/61/reference-guide.pdf or on AFS at `/afs/hyewon.snu.ac.kr/user/6.375/tools/bluespec/Bluespec-2008.11.C/doc/BSV/reference-guide.pdf`. There is a user guide with more information on the using the Bluespec compiler which you can download at http://asim.csail.mit.edu/redmine/attachments/61/user-guide.pdf

4. **Getting Help**: **You are encouraged to use the course discussion forum (located on the course Wiki) to record your experiences, complain about the course, and to pose questions to the T.A.'s**. If we answer your questions on the forum, everyone else can benefit from your trailblazing! There is the added (and not insignificant) benefit of reducing the number of emails the T.A.'s need to write, since problems or misunderstandings experienced by one student will almost invariably be encountered by others. Of course, if the question is very narrow, you can mail the staff mailing list.

3 Tutorial

Before you write any Bluespec, we will guide you through the creation of a null audio pipeline (one which just takes the audio data and returns it), and the steps required to run it both in simulation and on the FPGA. In Section 4 you will augment this pipeline with a FIR filter to modify the audio signal.

3.1 Audio Pipeline In Simulation

All paths referenced in this lab will be relative to the workspaces directory. In your `workspaces` directory start awb [“`awb &`”]. Once the awb GUI appears, click on the “Admin” tab. In the top box, select the workspaces directory you just created. Then, in the “New workspace name” box, type “`labs`”. Finally click “Create”.

3.1.1 Configuring the Pipeline

One of AWB's strengths is how it facilitates IP reuse. Conceptually, IP is stored in Repositories, which you can treat like source control repositories. In fact, AWB manages multiple repositories and is aware of many source control protocols, (*e.g.,*CVS, SVN, GIT). In these repositories are collections of packages, which themselves contain Models, Modules, and Submodels. For a complete disambiguation of these terms, look at the terminology listed on the AWB wiki page http://asim.csail.mit.edu/redmine/wiki/awb. We will create a model corresponding to an audio pipeline and fill in all its constituent pieces.

To begin with, we must perform an initial checkout for each of the repositories containing the components we need. Checking out a repository is as simple as going to the "Admin" tab, selecting the name in the "Repositories" menu and clicking the "Checkout" button. Checkout the following repositories, and make sure the "build" box has been checked (you can imagine what that does).

Throughout this document, we refer to ``<yourname>''. We have created AFS and Kerberos accounts for each student in the class. The names of these accounts are your first name concatenated with your last name (in Latin characters) as you entered them when signing up for the course.

```
asimcore
awb
hasim
mit-6.375
platforms
mit-6.375-<yourname>
```

When performing a task in AWB, you will generally see a window pop up titled "RunLog". Keep an eye on this output to make sure the operations complete successfully. While an operation is running, you can terminate it using the "Kill" button on the RunLog window. When an operation has terminated, you can click the "OK" button to close the RunLog window.

Now that you have checked out the necessary repositories, you will need to restart AWB. Ideally you should be able to just select the "Refresh" command from the "Edit" menu command, but this does not always work. We are working on getting this fixed, but you should remember this **bug** in the future.

Once you've restarted AWB, return to the "Admin" tab, and look at the packages list at the bottom of the panel. Since all these are backed by active source control repositories, packages may be updated after you checked. We may explicitly ask you do update your copies of the packages in the event of a bug fix, but we recommend that you update your packages periodically as it is a good practice.

To update your checkouts, click the "Refresh" button below the packages list, then either select a particular package to update or click the "All" box to update all of your checkouts. Select the "Update" radio button and click "Execute". After updating it is advisable to build your packages by performing the same steps and selecting the "Build" radio button. Alternately, you can do the same thing by running the command `awb-shell update package all` in your workspaces directory.

Now that all the repositories are checked out and built, it is time to actually construct the pipeline. Click on the "Models" tab, and expand the tree in the "Model Directories" panel. Select the model directory `Models` ⇒ `mit-6.375` ⇒ `audio_processor_test` and double click on the model named "`audio_processor_exe`" in the "Models" panel. The choice of this particular model is not important, but it is helpful to base new models on an existing one.

Double clicking will open up a new configuration tool, called the apm editor. Each model `model_name`) is described using an apm file (`model_name.apm`). While it is possible to edit these manually, we recommend using the GUI tool. Each apm file describes which modules and submodels are used to compose the corresponding model and also how these components are linked together. In addition to specifying which components are used, this tool is also used to specify how the various components are run. For example, we are developing a hybrid model (designed to run components on the FPGA as well as the host processor), but we may want to run the hardware in simulation, rather than on the FPGA. Using this tool, we can toggle this behavior easily.

Select the high-level item "model" in the "Type" hierarchy. Listed in the "Alternative Modules" window are the alternative module types. Depending on which one you select, the submodel requirements will change. Double click on the "Default Model Foundation" and notice how the submodule requirements change in the top panel. Since we are developing a Hybrid Model, make sure that "HW/SW Hybrid Project Foundation" is selected as the implementation for the model type. In AWB, a model is composed of a number of components. You must specify an application environment (`application_env`), an FPGA environment (`fpgaenv`), and `project_common`, which is an implementation of the standard component library.

Let's begin by specifying the `application_environment`. Select that submodel, and look at the alternative modules in the lower left panel. Make sure that "Soft Connections Hybrid Application Environment" is selected. There are many other submodules available, and we will explain more about them in the future. Choose "Audio Processor Application" to be the "connected_application" submodel of the `application_env`. For the `audio_processor_types` submodel, select the only alternative; "Audio Processor Types" module. For the `audio_pipeline` submodel, select the "Default Audio Pipeline" from the Alternative Modules. Select "Standard Platform Services" as the alternative module for `platform_services` (this contains utilities such as memory models etc.), and "Standard Soft Connections Lib" as the Alternative Module for `soft_connections_lib`.

Next, we need to specify the `fpgaenv` submodel. This represents the execution environment of the "Hardware" component of your hybrid design. Select the "Hybrid Simulation FPGA Environment" as the Alternative Module. This indicates that you will be running your hardware components in simulation (to begin with). Lastly, you will need to specify an Alternative Module for `project_common`: "Default Hybrid Project Common".

3.1.2 Executing the Pipeline

Before execution, we must save the model. To save the fully specified model, you will first need to create the appropriate directory in your personal repository in which to save it. In the terminal and `cd` to `workspaces/labs/src/mit-6.375-<yourname>/config/pm/mit-6.375-<yourname>` and create the directory `lab1` (`mkdir lab1`). Back in the AWB GUI, go to File ⇒ Save As. Navigate to the directory you just created, and save the model as "`lab1_audio_pipeline_sim`" in that directory.

Now we are going to run this pipeline in simulation. To do that, we must first configure and build the model. Close the apm editor as well as AWB, then restart AWB and select your newly created model in the "Models" tab (`Models` ⇒ `mit-6.375-<yourname>` ⇒ `lab1` and select "lab1_audio_pipeline_sim" in the right pane). Click on the configure button at the bottom of the "Models" panel in the AWB GUI. Once it has configured, select the "Next" button, which will bring you to the "Build Options" tab, where all you need to do is click "Build". Depending on the machine load, this could take a few minutes. Once this has completed, close the RunLog and click "Next". This brings up the "Benchmarks" panel, where we choose a test to run. In AWB, a benchmark consists of input data, as well as meta data specifying what makes an execution correct and hooks for timing and statistics gathering etc. In subsequent labs, you will be asked to create your own benchmarks, but more details on their structure will be provided before then.

Go to `Benchmarks` ⇒ `MIT-6.375` ⇒ `audio_processor_test` ⇒ `null_benchmark.cfx` ⇒ `benchmarks` in the "Benchmark Directories" hierarchy. Select the `null.wav.cfg` benchmark (This is an empty file, but a good edge case to test). Click the "Setup" button, and finally "Next". We won't be using the Parameters options yet, so just proceed to the "Run Options" tab by clicking "Next". As you can see, you can input various options to constrain the simulation of your model, but for we'll work with the default parameters specified by the benchmark. Select "Run", and watch the output in the RunLog window. If you see the line "`*** Output comparison passed ***`" in the RunLog, you have successfully executed the default audio pipeline in simulation. Now, repeat this step with each of the other benchmarks in the `benchmarks` directory.

The last step in this section is to add your newly created model to source control. There is no way to do this through the GUI, so exit AWB `cd` to `workspaces/labs/src/mit-6.375-<yourname>/config/pm/mit-6.375-<yourname>`. Add the newly created `lab1` directory and all of its contents to the source control. (execute `svn add lab1`. Then commit the files using the command `asim-shell commit package mit-6.375-<yourname>`. When asked to enter comments, give a short meaningful message (such as "original check-in"). After that, an emacs buffer appears with list of svn changes. If it looks OK, close the emacs buffer. You will be prompted once again for confirmation, enter 'yes'.

3.1.3 Listening to the Output

If you want to analyze to the .wav files to observe the effects of your filter, here are directions to locate the important files (I will use the model name "lab1_audio_pipeline_sim", but you can substitute any of the alternative models and the directory structure will be the same):

1. Each compiled module gets it's own directory in `workspaces/labs/build/default`, so `cd` to `workspaces/labs/build/default/lab1_audio_pipeline_sim`.

2. You will see two directories, one titled `bm` and the other `pm`. `pm` stands for "Performance Model" and contains the files pertaining to the execution of your test. If you are running in simulation, these will be executable files, and if you are running on the FPGA, you will see scripts which invoke the FPGA functionality. `bm` stands for "Benchmark", and contains information pertaining to the benchmark tests, and this is the one we are currently interested in.

3. When selecting a benchmark to execute, you are in effect selecting the input file. In the `bm` directory, you will see a directory corresponding to each benchmark you have set up and executed

4. Supposing you want to listen to `reuben_james_1sec.wav`, cd into the directory by that name.

5. In that directory, you will see a number of files, three of which are of interest to us now: `input.wav`, `out_gold.wav` and `out_hw.wav`. `input.wav`, as you might imagine, is the input file, while `out_gold.wav` is the file generated by our reference code, and `out_hw.wav` is the file generated by the code you executed. You can compare the input and output files to see how your transformations have changed them.

3.2 Audio Pipeline on FPGA

Getting this model to run on the FPGA is deceptively simple. Restart the AWB GUI and return to the "Models" tab and double click on `lab1_audio_pipeline_sim` you created in the previous section. We need to create a new model with a different `fpgaenv`, so highlight `fpgaenv` it in the Type hierarchy, and select "Hybrid XUPv2 Serial FPGA Environment" by double clicking on it in the "Alternative Modules" window. A pop-up will appear for which you must select yes. There is one last step, and that is to set the target clock frequency for the FPGA synthesis tools. The XUPv2 FPGA has a default clock frequency of 100 Mhz. but you can change that by specifying a rational multiplier. To set this value, click on the "Parameters" tab in the lower right hand corner of the apm editor. There you will see a few parameters, among which are "MODEL_CLOCK_MULTIPLIER" and "MODEL_CLOCK_DIVIDER". The default FIR filter will run at 50 Mhz, so specify a value of "1" for the "MODEL_CLOCK_MULTIPLIER", and a value of "2" for the "MODEL_CLOCK_DIVIDER".

One slight inconvenience with our toolchain is that there are certain restrictions for the values you can assign to the clock divider and clock multiplier. For various reasons, you can specify a clock multiplier of **1** and a clock divider of **2**, but in general, these values should range from **2** to **32** inclusive. Therefore, if you want to specify a frequency of 10 HMz, you must use the fraction `2/20` and not `1/10`.

Most often, if the FPGA synthesis tools are unable to compile your design under the specified timing constraints, they will exit with an error to that effect, which will appear in your Run Log. If this happens, you should loosen the timing constraints (specify a lower target frequency) and reattempt synthesis.

You will want to save this as a different model, so go to `File` ⇒ `Save As`, and give it the name `lab1_audio_pipeline_fpga`. Follow the same steps as in the previous section to configure, build, and run your new model. Be aware that synthesizing for an FPGA is far more complex than compiling a simulator and will therefore take much longer. Keep this in mind when developing your modules: get things working in simulation where you can more quickly recompile and change your code first. Finally add the new apm to source control using `svn`, and commit it using `asim-shell`.

Running the design on the FPGA requires that you have correctly connected your board to your development machine. You will need to make sure the USB cable is connected as this will be used to program the board, and the Serial cable is connected, as that one is used to communicate with the design. Be aware that other processes running on your operating system which access the Serial ports may interfere with the Virtual Machine's use of these ports. Make sure that you shut these applications down before attempting this step. Remember too that these devices can time out,

so if you don't see a message indicating either success or failure when programming the board or executing the design, kill the attempt and retry. On some older machines and operating systems, we have noticed that the serial connection between the Software component (running on the host) and the Hardware component (running on the FPGA) is not always completely reliable. We are working on this issue but for the time being, you may have to retry a few times before a successful execution. We have found that running VMWare from Windows is more reliable than running it on Linux (in spite of the fact that the Virtual machine itself is Linux!).

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. You are filtering `.wav` files, which means that you can listen to the file before and after you run it through your filter, to appreciate the effect of your work!

4.2 Default 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. A simplified version of the code we have provided to you appears in Section 6. To create an audio pipeline consisting of this filter, select and double click on the `lab1_audio_pipeline_sim` you just created to open it in the apm editor, and select the `audio_pipeline` submodel. Choose the `FIR Filter Pipeline` as the alternative module. Two red submodules will appear, indicating we need to specify these subtypes to complete the system. Select `FIR Filter Pipeline Types` as the alternative module for `audio_pipeline_types` and `FIR Filter Default Implementation` for `fir_filter`. Save this model as `lab1_audio_pipeline_1_sim` in the same directory where your previous apm files are saved.

In the apm editor for `lab1_audio_pipeline_1_sim`, select the `fir_filter` submodel, and right-click on the implementation and select "edit". This will allow you to look at the source code (it opens all the files in emacs). `FIRFilterDefault.bsv` is the Bluespec corresponding to Figure 4. Take a moment to understand it and see how it relates to the diagram. Once you understand the source code, you should execute the pipeline in simulation, following the instructions given in Section 3, with the exception that you need to run a different set of benchmarks. The new benchmarks have the same input files, but differ in how the output is verified. The verifier must reflect the fact that our new pipeline now contains a FIR filter. These benchmarks can be found in `Benchmarks` ⇒ `MIT-6.375` ⇒ `audio_processor_test` ⇒ `fir_benchmarks.cfx` ⇒ `benchmarks`. Create another model titled `fir_filter_audio_pipeline_1_fpga`, (identical to `lab1_audio_pipeline_1_sim`, except for the fpgaenv, which should be configured to run the HW on the FPGA instead of in simulation) and make sure that it executes correctly on the FPGA.

4.3 Modified FIR Filters

Next, you need to create the necessary files for the two subsequent modifications of the original FIR filter.

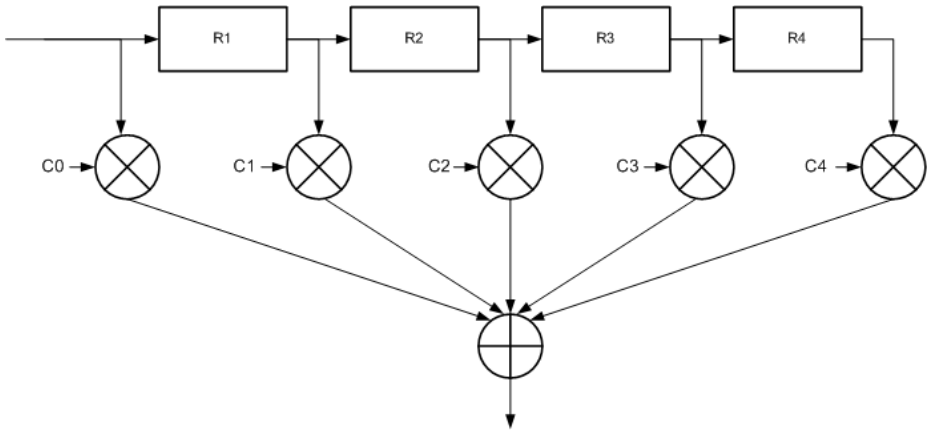

Figure 4: Simple Fir Filter

1. New source files are needed to implement the alternate microarchitectures. First create the appropriate directory for your `.bsv` files as follows: `cd` to ∼`/workspaces/labs/src/mit-6.375-<yourname>/modules/mit-6.375-<yourname>` and create the directory `lab1`. Copy the file ∼`/workspaces/labs/src/mit-6.375/modules/bluespec/mit-6.375/lab1/FIRFilterDefault.bsv` to the `lab1` directory twice; once as `FIRFilter2.bsv`, and once `FIRFilter3.bsv`. Add the `lab1` directory and its contents to source control and commit them as we did in the previous section (using ‘‘`svn add`’’ and ‘‘`svn commit`’’).

2. Files with the `.awb` extension are used to advertise modules to the AWB tool. If we create a new module in Bluespec, we need to create a corresponding `.awb` file which contains information about our new module, such as interface type, name, etc. The two alternative microarchitectures we are creating will become AWB modules. That way we can use the default pipeline as a starting point and swap the original FIR filter for a new one. In order to do this, we need create two new `.awb` files, one for each new module. In the same directory where we found `FIRFilterDefault.bsv`, you will find the AWB module `fir_filter_default.awb`, Copy `fir_filter_default.awb` as `fir_filter_2.awb` and `fir_filter_3.awb` into the lab1 directory you created in the previous step.

3. As you've created them, these two new modules (`fir_filter_2.awb`, and `fir_filter_3.awb`) are indistinguishable from the original, so we need to manually edit the files (using a text editor of your choice). If you open the `.awb` files, you will see that the first line begins with the marker ‘‘`%name`’’. You will need to change the names of these new modules to accurately reflect their contents. Change the name of `fir_filter_2.awb` to "FIR Filter Pipeline2", and the name of `fir_filter_3.awb` to "FIR Filter Pipeline3". You will also notice the line which begins with the string ‘‘`%public`’’. This is where you need to list the `.bsv` file which contains the module which you are wrapping. Change `fir_filter_2.awb` to point to "FIRFilter2.bsv", and `fir_filter_3.awb` to point to "FIRFilter3.bsv".

4. be sure to add and commit the new `.awb` files to source control.

5. You will now create four new models to use the new modules you created in the previous steps. Each new FIR module (there are two of them) should be used in two new models one configured for simulation and one for running on the FPGA. Do this by cloning one of the two versions of the default models you have already saved in your lab1 directory (Section 3.1.2), selecting the alternative FIR implementations which should now appear in the "Alternative Modules" window. Save the new models as `lab1_audio_pipeline_[2,3]_[sim,fpga]` in your repository, and remember to add them to source control.

Now that you've created these four new pipelines, let's take a closer look at how we want to modify the microarchitectures for the two variants. As you have configured them, `lab1_audio_pipeline_2_*` will use `FIRFilter2.bsv` and `lab1_audio_pipeline_3_*` will use `FIRFilter3.bsv`. `FIRFilter2.bsv` needs to be modified to reflect the microarchitecture in Figure 5, while `FIRFilter3.bsv` needs to be modified to reflect the microarchitecture in Figure 6.

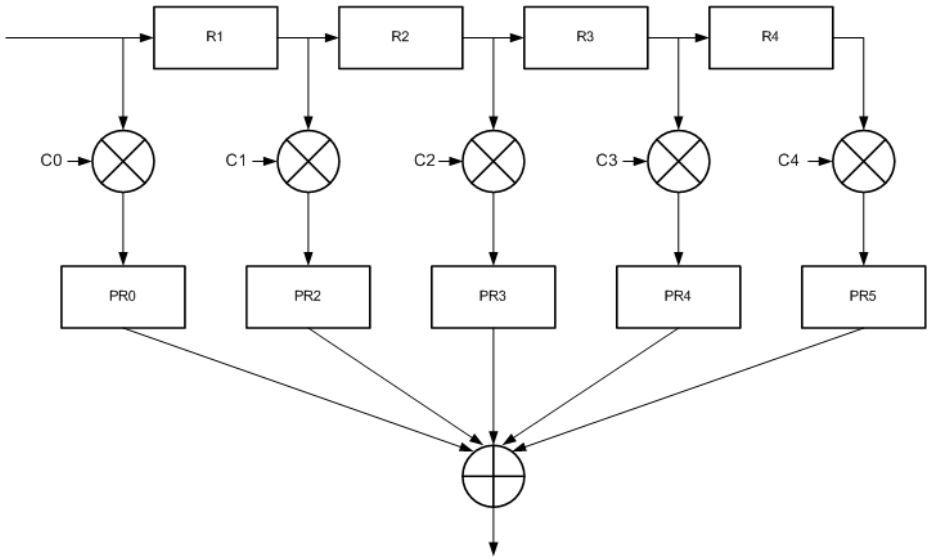

Figure 5: FIRFilter2.bsv

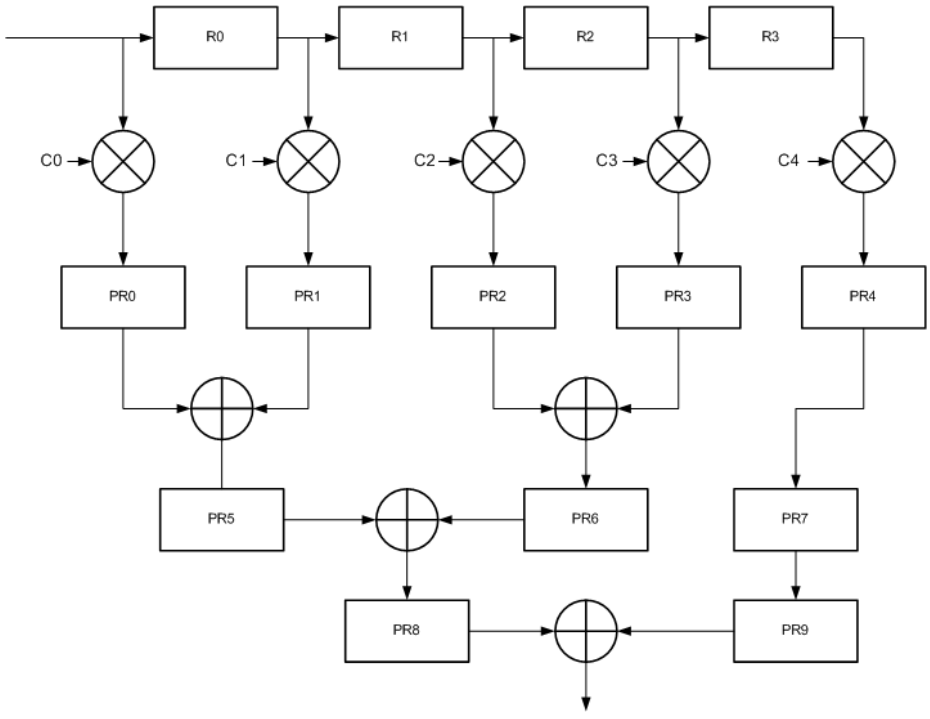

Figure 6: FIRFilter3.bsv

These two variants on the original microarchitecture demonstrate increasing degrees of pipelining. Pipelining allows us to shorten critical paths, increasing the frequency at which a design can run. Provided we can keep our new pipelines filled, an increase in frequency yields a higher throughput. (Think about it: you can consume packets faster!).

Make sure that `lab1_audio_pipeline_2_sim` and `lab1_audio_pipeline_3_sim` correctly execute the `fir_benchmarks` in simulation.

Next synthesize `lab1_audio_pipeline_2_fpga` and `lab1_audio_pipeline_3_fpga` and verify that they correctly execute the `fir_benchmarks` in on the FPGA. See how fast you can clock your modified designs. Do this by specifying a larger rational multiplier in the Parameters tab of the "fpgaenv" module. Increasing the degree to which your design is pipelined should increase the target frequency for which the FPGA tools can synthesize your design. Naturally, we expect that `pipeline_2` will be able to run at a higher frequency than `pipeline_1`, and `pipeline_3`, will be able to run faster than `pipeline_2`.

Now that you have your new models running in simulation and on the FPGA, make sure that all the new `.apm` and `.bsv` files have been committed to SVN. If you don't, the lab graders won't be able to see them.

5 Discussion Section

Put the answers to these questions in to a text file called `answers.txt` and check it into source control in your `lab1` directory.

1. Based on what you have learned in the lectures, compare the relative areas and clock frequencies of the three design. Which one is needs the least area? the most?
2. Which design should have the highest clock speed? The lowest?
3. What is the throughput(samples/cycle) each of the three designs?
4. It is generally fairly easy to achieve a clock frequency of 50MHz on the majority of reasonably sized XUPv2 FPGA designs. What is the maximum number of cycles per sample we can sustain in order to handle CD quality sound (44.1KHz)? Music Studio sound (48KHz)?
5. How many times faster is our design than necessary? If you implemented the filter in software, instead of on FPGA could you meet the throughput requirements?
6. If instead of sound we wanted to filter WiFi signals (which have a minimal bandwidth of 20 MHz), what throughput must we be able to sustain? What possible properties can you infer about a system which can filter WiFi?
7. List the maximum frequencies for which you were able to synthesize your modified FIR filters.

6 Appendix: BSV Source code for FIR Filter

```
0:  include "asim/provides/soft_connections.bsh"
1:  include "asim/provides/common_services.bsh"
2:  include "asim/provides/audio_pipeline_types.bsh"
3:  include "asim/provides/audio_processor_types.bsh"
4:  typedef 8 Taps;
5:  module [Connected_Module] mkFIRFilter (FIRFilter);
6:   FIFO#(AudioProcessorUnit) infifo <- mkFIFO;
7:   FIFO#(AudioProcessorUnit) outfifo <- mkFIFO;
8:   Vector#(Taps,Reg#(Sample)) samples = newVector();
9:   for(Integer i = 0; i < valueof(Taps); i=i+1)
10:      samples[i] <- mkReg(0);
11:   FixedPoint#(16,16) firCoefs [9] = {fromReal(-0.0124),
12:                                     fromReal(0.0),
13:                                     fromReal(-0.0133),
14:                                     fromReal(0.0),
15:                                     fromReal(0.8181),
16:                                     fromReal(0.0),
17:                                     fromReal(-0.0133),
18:                                     fromReal(0.0),
19:                                     fromReal(-0.0124)};
20:   rule process (infifo.first matches tagged Sample .sample);
21:     samples[0] <= sample;
22:     for(Integer i = 0; i < valueof(Taps) - 1; i = i + 1)
23:        begin
24:          samples[i+1] <= samples[i];
25:        end
26:     FixedPoint#(16,16) accumulate= firCoefs[0] * fromInt(sample);
27:     for(Integer i = 0; i < valueof(Taps); i = i + 1)
28:       begin
29:         accumulate = accumulate + firCoefs[1+i] * fromInt(samples[i]);
30:       end
31:     outfifo.enq(tagged Sample fxptGetInt(accumulate));
32:     infifo.deq;
33:   endrule
34:   rule endOfFile (infifo.first matches tagged EndOfFile);
35:     for(Integer i = 0; i < valueof(Taps); i = i + 1)
36:       begin
37:         samples[i] <= 0;
38:       end
39:     outfifo.enq(infifo.first);
40:     infifo.deq;
41:   endrule
42:   interface sampleInput = fifoToPut(infifo);
43:   interface sampleOutput = fifoToGet(outfifo);
44:  endmodule
```

The following list describes in detail the FIR filter you will be modifying. If something isn't mentioned here, you can check the language reference manual for library functions and keywords.

- **lines 0-1**: AWB includes. These import the structure which allow us to communicate with the outside world, and are part of the AWB library code
- **lines 2-3**: Local includes. Look for the correspondingly named .awb files workspace/labs/src/mit-6.375/modules/bluespec/mit-6.375/common/ to find the actual Bluespec files which are used to generate these includes. These files are specific to this audio processing pipeline

- **lines 6-7**: instantiate an input FIFO and an Output FIFO mkFIFO returns a fifo of length 2 (by default) AudioProcessorUnit is the name given to the packets of DATA processed by our audio pipeline. For their definition, look in the file workspace/labs/src/mit-6.375/modules/bluespec/mit-6.375/common/AudioProcessorTypes.bsv
- **lines 8-10**: an alternate syntax for instantiating the samples vector would have been as follows:

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

  we have used an explicit loop here, to demonstrate how vectors can be instantiated during the static elaboration phase, even though replicateM is far more concise.
- **lines 11-19**: fromReal takes a Real number and converts it to a FixedPoint representation. The compiler is smart enough to infer the type (bit width) of the fixed point (in this case, we have 16 bits of integer, and 16 bits of fraction.
- **lines 20-33**: This rule implements a fir filter. We do the fir computations in 16.16 fixed point. This preserves the magnitude of the input pcm. This code was implemented using for loops so as to be more clear. Using the functions map, fold, readVReg, and writeVReg would have been more concise.
- **lines 22-25**: Advance the fir filter, by shifting all the elements down the Vector of registers (like a shift register)
- **lines 26:30**: Filter the values, using an inefficient adder chain. You will need to shorten the combinatorial path, by pipelining this logic.
- **lines 34-41** Handle the end of stream condition. Look at the two rule guards, these are obviously mutually exclusive. The definition of AudioProcessorUnit shows that it can be tagged only as a Sample, or EndOfFile; nothing else!
- **lines 35-38**: Reset state for next invocation
- **line 39**: pass the end-of-file token down the pipeline, eventually this will make it back to the software side, to notify it that the stream has been processed completely
- **lines 42-43**: this section connects the fifos instantiated internally to the externally visible interface