6.973 Communication System Design 03/06/06

Problem Set 3 – Bluespec Introduction

Getting Started

To use the Bluespec compiler, add the following lines to your **`.cshrc.mine`** file:

```
1. add 6.375
2. setenv BSPATH /mit/6.375/tools/bluespec/current
3. setenv BLUESPECDIR $BSPATH/lib
4. set path = ( $BSPATH/bin $path )
5. set path = ( /mit/6.973/tools/bluespec $path )
6. setenv BS_VLIB_PATH $BSPATH/lib/Verilog
7. setenv NCVLOG_ROOT_DIR `cds_root ncvlog`
```

A good place to put these lines is after your vppsim configuration commands but before your **`setenv LD_LIBRARY_PATH`** command. You'll need to launch a new terminal window for these settings to take effect. In addition to these modifications to your startup script, you'll need to edit your **`VppSim/.vppsimrc`** file and add the following lines:

```
ncverilog_command: ncverilog
ncverilog_ams_command: ncverilog +ncams
```

The code for this problem set is located in the course locker at:

```
/mit/6.973/homework3
```

I recommend that you copy this entire directory (using **`cp –R`**) to your home directory so that you can make changes to the code and schematics. To make Cadence aware of this new version of the code, edit your **`VppSim/cds/cds.lib`** file and change the **`80211a_`** lines to the following:

```
DEFINE 80211a_transmitter [your path to homework3]/homework3/80211a_transmitter
DEFINE 80211a_receiver [your path to homework3]/homework3/80211a_receiver
```

To make sure everything is still working, open up Cadence, netlist the **`transmitter_harness`**, and run **`vppsim –cpp`** and **`make`** in the appropriate directory. (If you are unfamiliar with these steps, consult the Problem Set 2 Addendum.)

Using VppSim for C++/Verilog Co-Simulation

Before writing your own Bluespec modules, it will be helpful to simulate the system using the Bluespec implementation of the Convolutional Encoder that was covered in Tutorial 4. The first thing you will need to do is compile the Bluespec description of the Encoder into Verilog so that it can be used by VppSim. To do this, first create the directory:

```
[your path to homework3]/homework3/80211a_transmitter/convolutional_encoder/verilog
```

In the same way that the C++ implementations of the modules live in the **`cppsim`** subdirectory, the Verilog implementations will live in the **`verilog`** subdirectory of each module. Next, **`cd`** into this newly created directory and copy the Bluespec description of the Convolutional Encoder there. This file can be found at:

```
/mit/6.973/homework3/ConvEncoder.bsv
```

Finally, from within the Convolutional Encoder's **`verilog`** subdirectory, issue the command **`compile_convolutional_encoder`**. This script will do two things: run the Bluespec compiler on **`ConvEncoder.bsv`** and, if the compilation is successful, call a perl script that will convert the Verilog generated by the Bluespec compiler into a format that is recognizable by VppSim. The end result of this will be a file named **`verilog.v`** in the current directory that contains the VppSim compatible Verilog implementation of the Convolutional Encoder; this is the file that VppSim will be looking for.
To run the simulation using our newly created Convolutional Encoder implementation, first cd into the directory where we make and run the simulator (**`VppSim/SimRuns/80211a_transmitter/transmitter_harness`**) and issue the command **`vppsim –vpp`** (note the change from **`-cpp`** to **`-vpp`**). Because simulating Verilog modules takes longer than simulating C++ modules, we will be running the simulator on smaller amounts of data in this problem set (the number and length of the messages have already been set for you in the tx mac module). Edit the **`test.par`** file and set **`num_sim_steps`** to **`35e4`** to take advantage of this shorter simulation time.

Within the same directory (**`VppSim/SimRuns/80211a_transmitter/transmitter_harness`**), edit the file **`test.hierarchy`**. In this file you will see the names of all the modules you are simulating preceded by either a "c" or a "v". Any module with its name preceded by a "c" will be simulated using its C++ implementation and any module preceded by a "v" will be simulated using its Verilog implementation. Find the **`convolutional_encoder`** module and switch it to "v" (note the words cppsim and verilog in parentheses next to the module's name; this means that both implementations are available for this module). Save the file, return to a command prompt in the simulation directory, run **`vppsim -vpp`** again and **`make`** to build and run the simulator. When the simulation has completed, run **`bit_errors`** on the output files to make sure the system is working.

Problem 1 - Bluespec Implementation of the Scrambler (60 points)

Use what you have learned from the Convolutional Encoder's Bluespec implementation to create a Bluespec implementation of the Scrambler module (name this file **`Scrambler.bsv`**). Simulate your implementation in the same manner as you did the Convolutional Encoder and make sure that **`bit_errors`** reports zero errors for your simulation's output. You only need to submit **`Scrambler.bsv`** containing your implementation of the Scrambler.

HINTS:

1. Use the Convolutional Encoder's Bluespec implementation as a starting point and modify it to become the Scrambler.
2. When you are ready to compile and test your design, the command **`compile_scrambler`** should be used instead of **`compile_convolutional_encoder`** in the procedure described in the previous section.
3. The data type of your input and output ports should be identical to one of the types defined in the Convolutional Encoder module.
4. Remember that the first bit of a VppSim vector will be mapped to the most significant bit of a Verilog bit vector. This means that if you have a 24 bit VppSim vector x connected to a Verilog (or Bluespec) bit vector y, x.get_elem(0) will be mapped to y[23].

Problem 2 - Bluespec Implementation of the IFFT (40 points)

You have been given a skeleton of an 8 point IFFT implementation. Fill in the missing Bluespec code to make the module work. The code and test harness for this module is also located in the **`/mit/6.973/homework3`** directory under **`ifft_8`**. Once this directory has been copied to your local homework3 directory, you'll need to add it to Cadence in the same way you added the **`80211a_transmitter`** and **`_receiver`**. Namely, you must add the following line to your **`VppSim/cds/cds.lib`** file:

```
DEFINE ifft_8 [your path to homework3]/homework3/ifft_8
```

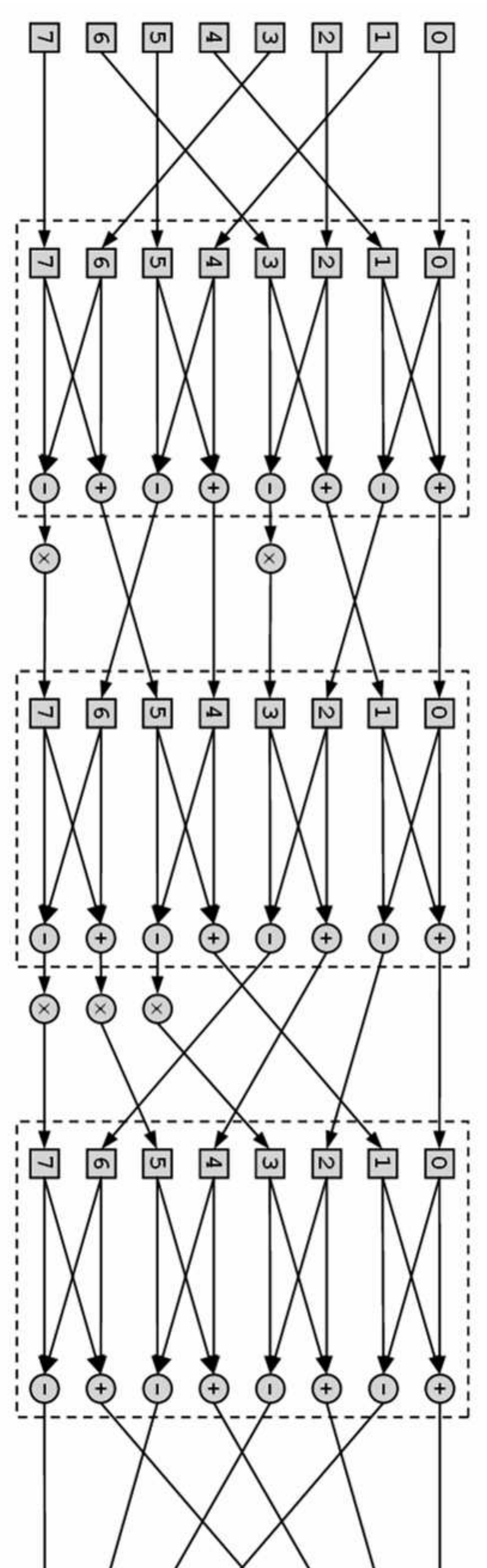
The skeleton Bluespec code is provided in **ifft_8/ifft8/verilog**. You can compile it using the command **compile_ifft**. To help you, a C version of this module has been provided in **/mit/6.973/homework3**. You can use this code as an example of how to implement the IFFT, as well as a way to check your results. To compile this code, use the following command:

```
gcc –o ifft_rtl ifft_rtl.c -lm
```

Submit the completed version of **ifft8.bsv**.

HINTS:

1. The **ifft8/verilog** directory contains 3 files: **ifft8.bsv**, **ComplexF.bsv**, and **SVector.bs**. You only need to worry about **ifft8.bsv**, the other two files are libraries that will be needed when compiling your design.
2. The **ComplexF** type has the multiplication, addition, and subtraction operators overloaded to perform complex arithmetic. Addition and subtraction will scale the result by a factor of 0.5 to avoid overflow.
3. Although **permute** and **twiddle** are functions, you can access them like arrays (for example, **twiddle[0][3]**).
4. Because the elements of a VppSim vector are mapped to Verilog in reverse order, you should access the vectors in reverse order (e.g. **inputs[7-i]**). This does not apply to the **permute** and **twiddle** arrays.
5. A diagram of the IFFT is provided on the last page of this handout.


0
1
2
3
4
5
6
7
+
−
×