Massachusetts Institute of Technology – Laboratory for Computer Science

IAPBlue A Mini Course on Bluespec

http://www.csg.lcs.mit.edu/IAPBlue

Lab Exercises 1 13 Jan 2003

The purpose of this lab is to become familiar with using the Bluespec compiler. You will explore the TRS and scheduling information that the compiler generates for some of the examples from lecture. You will use the compiler to generate Verilog or C descriptions of these examples and simulate them. You will also begin to write a few small designs in Bluespec, based on the examples we have given.

This handout will refer to the "Lab How-To" document, also handed out today, which briefly explains how to run the Bluespec compiler, use the Bluespec C simulator, and run various Verilog tools.

More complete documention on the Bluespec compiler is available in the "Bluespec Compiler User Manual" on the IAPBlue website. The Bluespec language and standard libraries are documented in the "Bluespec Language Manual," also on the website.

Exercise 1 The GCD example

The file `GCD.bs` contains the Bluespec code for the GCD example presented in Lecture 1.

Part a:

Examine the Bluespec code for mkGCD. What state does the module contain? How many rules does the module have? Do the rules conflict? What interface methods does the module have? Do the interface methods conflict with each other or with the rules?

As explained in the "Lab How-To" handout, generate the ATS and Verilog for the GCD example by running the following command:

```
bsc -verilog -ats -g mkGCD GCD.bs
```

The ATS will be dumped to the screen and the Verilog will appear in the file `mkGCD.v`. Look over the ATS and make sure that you can read it. Do the state and rules correspond to your understanding of the Bluespec source code?

Now have the compiler give rule and interface conflict information by running the following command:

```
bsc -verilog -squery \* \* -g mkGCD GCD.bs
```

You can also view the conlfict information for just the interface methods by looking at the header of the generated Verilog file, `mkGCD.v`:

```
// Method conflict free info:
// [result_ < start_]
```

Does the conflict information determined by the compiler match your guesses?

Part b:

Now let's simulate the GCD design and watch the firing of rules.

If you are familiar with simulating Verilog, you can compile to Verilog and use a Verilog simulator and wave viewer to explore the design. This is described in the "Lab How-To" handout.

However, we can also compile a Bluespec design to C and run a C simulation of the design. The C simulator will let us view which rules are enabled, which rules the scheduler actually picks to fire, and what the state of the system is on every clock cycle. Besides dumping this information to the screen, the C simulator can also dump signals to a file to be viewed in a waveform viewer, if you find that more comfortable.

For this exercise, try simulating in both C and Verilog. If you find one way more convenient for debugging, feel free to use that one in future exercises.

The GCD module does nothing on its own. It simply responds to inputs. So whether you compile to Verilog or C, you will need a test wrapper which stimulates the GCD module. We have provided a Bluespec wrapper called `GCDTest.bs` which you can compile to either C or Verilog and use in either simulation environment. We have also provided a native Verilog wrapper, called `GCDTest.v`. You can modify the Verilog example to work with Verilog modules in later exercises. However, it might be more convenient to write your test code in Bluespec and compile it to Verilog (or C) for simulation.

To compile the C simulation, give the following commands:

```
bsc -c -g mkGCD GCD.bs
bsc -c -g mkGCDTest GCDTest.bs
bsc -c -e mkGCDTest -o GCDTest mkGCD.o mkGCDTest.o
```

This will produce a binary called `GCDTest` which you can execute to simulate the design. To see what the simulation is doing, you should run it with the flags `-v -s`. Other flags are described in the "Lab How-To" handout.

Does the execution of rules match your understanding of the Bluespec source code? Why does the "`Get result`" rule in the Bluespec test wrapper fire on the first cycle?

Exercise 2 — CPU with 2-stage pipeline

The GCD module was an overly simple example. Let's now consider the two-stage CPU presented in Lectures 1 and 2. The file `TwoStageCPU.bs` contains the Bluespec code from lecture.

Read over the Bluespec source code for the CPU and make sure you understand all the state and rules in the system. Note that the rules for `Load` and `Store` commands have not been added yet.

Part a:

On slide 13 of Lecture 2, you were asked whether the "`Fetch`" and "`Add`" rules can be executed simultaneously. Let's find out what the compiler determined.

To do that, we need to know the names of the rules. Rule names are given in the ATS dump, so generate the ATS for the CPU with the following command:

```
bsc -verilog -g mkTwoStageCPU -ats TwoStageCPU.bs
```

Locate the names of the "`Fetch`" and "`Add`" rules and ask the compiler to print out the conflict information for this pair of rules:

```
bsc -verilog -g mkTwoStageCPU -squery rule1 rule2 TwoStageCPU.bs
```

The compiler will tell you if the rules are mutually exclusive, conflict free, or sequentially composable. Did the compiler determine the correct relationship?

Remember to compare the rules in both directions! The compiler will only report a sequential composability relationship in the order that you specify the rules with `-squery`. Often rules can be composed in one direction but not the other. Don't forget to check the other direction:

```
bsc -verilog -g mkSimpleCPU -squery rule2 rule1 TwoStageCPU.bs
```

Part b:

On slide 14 of Lecture 2, you were asked whether the "`Fetch`" and "`Bz Taken`" rules can be executed simultaneously. Ask the compiler for the conflict information for this pair of rules. Did the compiler determine the correct relationship?

Part c:

On slide 21 of Lecture 2, you were presented with four rules from the two-stage CPU pipeline and were asked to determine which pairs of rules are mutually exclusive, which are conflict free, and which are sequentially composable. Use the `-squery` flag to dump the conflict information for pairs of these rules. (You may wish to use `-squery \* \*` to dump information for all pairs of rules.) Do the relationships determined by the compiler match our answers from lecture?

Exercise 3 — Writing Bluespec designs

If you have time left in the lab, here are some suggestions of simple Bluespec designs that you can write.

Part a:

Currently, the Bluespec test code in `GCDTest.bs` sends inputs to the GCD module forever. The only way to stop the simulation is to interrupt it. Can you modify `GCDTest.bs` to send only one pair of inputs to the GCD module, get a result, and then stop? Remember that a Bluespec C simulation stops when no rules can fire.

Part b:

Can you write a Bluespec module which receives a hexidecimal number one digit at a time and keeps track of whether the number it has seen so far is divisible by 3? Remember that a number is divisible by three if the sum of its digits is divisble by 3. You should be able to construct a finite state machine to keep track of this computation for an arbitrary number of digits. Use the following interface for your module:

```
interface Divisible =
    nextDigit   :: Bit 4 -> Action
    isDivisible :: Bool

mkDivThree :: Module Divisible
mkDivThree = module ...
```

Part c:

Add rules for the `Load` and `Store` to 2-stage CPU.

Part d:

The interface for the 2-stage CPU is `Empty`. This means that the module has no contact with the outside. And in fact, if you tried to synthesize the Verilog, the tools might notice this and optimize away the entire design!

Let's add an interface to the CPU that allows us to read and write into instruction and data memory. Change the CPU example to have the following interface:

```
interface CPU =
    imem_ifc :: MemIF
    dmem_ifc :: MemIF
```

Now write a test wrapper for the CPU which loads a program into memory, activates the CPU, waits a certain number of cycles, and then reads the result from memory.

If you're feeling very adventurous, you can add an instruction to the instruction set which causes the CPU to halt and to signal to the outside world that it has finished computation. You will need to add the start and finish signals to the CPU's interface:

```
interface CPU =
    ...
    start :: Action
    done  :: Bool
```