

The Pipeline Lab

"Hello Bluespec!"

This lab will lead you through basic aspects of the BSV (Bluespec SystemVerilog) language and the usage of the Bluespec compiler (bsc) for Bluesim and Verilog simulations. In each part you will find descriptions of the actual code to write, and how to use and invoke each of these tools with and without the development workstation.

Note: *The directory $BLUESPECDIR/doc/BSV/ contains useful documentation, including the language Reference Guide, the tool User Guide, a KPNS (Known Problems and Solutions) guide, a Style Guide, and others. The directory $BLUESPECDIR/training/BSV/examples/ contains some useful examples for future reference. The full set of training materials, documentation and examples can be accessed from the file $BLUESPECDIR/index.html.*

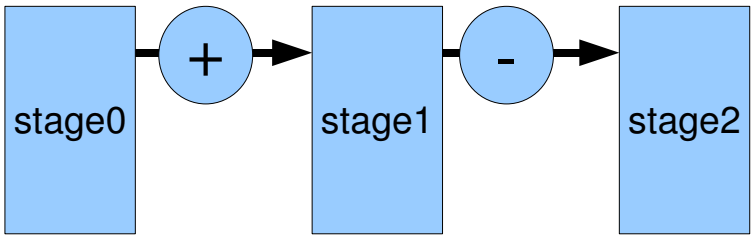
Part 1

In this part you will learn the following:

- How to create a very simple module with no interface or arguments
- How to instantiate registers
- How to write simple functions and rules
- How to compile and simulate

Part 1.A: The basic pipeline

We will a design a simple pipeline with three registers—stage0, stage1 and stage2—and increment and decrement functions as shown in the diagram below. The design will not have any external interface (just like a top-level testbench). It will just print out information as it runs.



First create a directory for our design. Since we're going to work on this design in multiple parts, let's create a directory named Part1.

The basic unit of work within the development workstation is the project (*projectname*.bspec). A project file is a named collection of project definitions and options, which are modified through the Project Options windows. Projects are only used in the development workstation. Create the project as follows only if you are using the development workstation:

- Create a new project named Pipeline in the Part1 directory created above. A project file is a named collection of settings and options. The Project Options window opens when you create a new project. Enter the following settings:

    - On the Files tab, enter Testbench.bsv for the Top file and mkTestbench for the Top module.
    - On the Compile tab, select Bluesim as the Compile to target.
    - On the Link/Simulate tab change the output file to testbench.exe and on the Run Options add the flag -m 20.
    - On the Editor tab, verify that the text editor is set to your preferred package.
- Create a file Testbench.bsv in a new directory Pipeline/Part1. In the workstation you can create a new file from the Project Files window.
- Define a package called "Testbench" using the keywords package and endpackage.
- Create two functions at the package level:
    - "increment()": given a value of type Bit#(16), it increments it by 1 and returns the result
    - "decrement()": given a value of type Bit#(16), it decrements it by 1 and returns the result
- Define a module mkTestbench with no interface and no parameters.
- Inside the module, instantiate three registers of type Bit#(16) with reset value 0 for the stages, called stage0, stage1 and stage2. Remember that in BSV a register is created by instantiating a register module (e.g., mkReg) with a register interface Reg#(<type>)[1].
- Create a simple rule[2] called "shift" that fires on each cycle and creates the connections of the pipeline in such a way that stage0's output gets incremented by one and stage1's output gets decremented by one, as in the diagram above.
- Create a simple rule called "show" that fires every cycle that uses $display to show the stage0, stage1 and stage2 values. Use %0d to format it in decimal notation.

Note: The following sections will explain how to compile, link and simulate using either the workstation or the command line. Detailed information on commands, flags and options may be found in the User Guide. When using the command line, you may wish to place these commands into a Makefile to avoid having to retype them repeatedly. The same commands can be used for all the remaining parts of this lab.

Create and run a Bluesim simulation

From the workstation:

The options to compile to Bluesim should have been set when you created your project. If not, set them now.

- Run the Compile task, from either the task bar or the Build menu.

You may get syntax or type errors that need to be fixed. You can double click on the errors in the command window to open the .bsv file to the line with the error. Once everything is correct, this will create the necessary objects (.ba files) for Bluesim.

[1]Reading and writing a register can be done using the same notation as in Verilog, i.e., just mention the register name in an expression to read it, and use the non-blocking assignment "<=" to update it. This is a special case—in general, interaction is through *interface method invocations*. Registers also have interface methods; the standard Verilog syntax can be seen as a special shorthand for invocation of the _read() and _write() methods..

[2] A rule without conditions will be fired in each clock cycle, i.e. the Actions enclosed between "rule" and "endrule" are executed in parallel in each clock cycle. Due to the concurrent character, BSV always uses the "non-blocking" assignment notation "<=".

- Run the Link task, from either the task bar or the Build menu.

This will link all the necessary libraries and create the executable testbench.exe.

- Run the Simulate task. The flag -m 20, set on the Link/Simulate tab in the Project Options will make the simulator run for just 20 cycles[3].

From the command line:

- Run "bsc -sim -g mkTestbench Testbench.bsv".

Once everything is correct, this will create the necessary objects (.ba files) for Bluesim. (you may get syntax or type errors that need to be fixed). The -sim flag directs bsc to create Bluesim objects, and the -g flag identifies a module for which to create a simulation object.

- Run "bsc -sim -e mkTestbench -o testbench.exe *.ba".

This will link all the necessary libraries and create the executable testbench.exe. The -e flag identifies the top-level module in the executable, and the -o flag specifies the name for the executable file.

- Execute the simulation by running "./testbench.exe -m 20". The flag -m 20 will make the simulator run for just 20 clock cycles.

Create a Verilog implementation, and run it in a Verilog simulation

From the workstation:

- Open the Project Options -> Compile tab and set the Compile to target to Verilog
- Run the Full Rebuild task from the task bar or the Build menu. This will run the tasks: full clean, compile, link and simulate.
- From the task bar or the Build menu stop the simulation (it will run forever until you kill it.)

From the command line:

- Run the command "bsc -verilog -g mkTestbench Testbench.bsv".

This will create mkTestbench.v. The -verilog flag directs bsc to generate Verilog (instead of the earlier -sim flag).

- Run the command "bsc -verilog -e mkTestbench -o testbench.exe *.v".

This will create[4] the executable testbench.exe.

- Execute the simulation by running "./testbench.exe" (it will run forever until you kill it).

Part 1.B: Adding some stimulus

Modify the design so that the reset strategy for the stage registers is different, by instantiating a different register module.

- Instantiate another register holding Bit#(16) values, called "counter", initial value 0.
- Create a rule "stimuli" that increments the counter each cycle. Modify rule "shift" to feed the counter's value into stage0.
- Rebuild and re-execute Bluesim and Verilog simulations, as in Part 1.A.

---

[3] The -m flag is one of many available in Bluesim simulation. For a complete list please refer to the User Guide.

[4] bsc automatically uses any of the following standard Verilog simulators: ncVerilog, vcs, modelsim or iverilog. You can specify a particular simulator with an environment variable, e.g., BSC_VERILOG_SIM = iverilog. Or, you can give bsc the equivalent -vsim flag. Please refer to the User Guide for details.

Part 2

In this part you will learn the following:

- How to organize the design in multiple files, and use of the "import" directive
- Changing a type from Bit#(16) to Int#(16)
- Separate and incremental compiling, and compilation dependencies
- Using the Int#(n) type, i.e., an integer type that is polymorphic in its width.
- Passing multiple arguments to a function
- The Integer type, and use of the *fromInteger()* function

Part 2.A: Separate the increment/decrement functions into their own package

Modify the Design

- Create a new directory Pipeline/Part2. Copy the Pipeline/Part1 files into the new directory.
- Modify the design so that it uses Int#(16) instead of Bit#(16). From the viewpoint of strong-typing, it is better to use an Int#() type if you are doing "integer-like" operations (such as + and -), and use Bit#() type only if you use it like a true bit vector. This separates out the logical intent (integers) from particular bit representations. For example integers can be represented using different bit representations.
- Create a file Functions.bsv. Use package/endpackage keywords for defining a package called "Functions". Cut and paste to move the increment and decrement functions from Testbench.bsv into this package.

Typecheck the file Functions.bsv

Here, we are just doing a separate compilation of this file for syntax checking, type checking, etc., The compiler records intermediate information in the files Functions.bi and Functions.bo.

From the workstation:

- Run the Typecheck task.

From the command line:

- Run the command: "bsc Functions.bsv".
  Note that we do not specify the bsc options -verilog or -sim–these are only needed when a Bluesim object or Verilog output is desired.

Modify Testbench.bsv to use Functions.bsv

- In Testbench.bsv, add "import Functions :: *" at the top of the "Testbench" package in order to make the functions visible.

As in Part 1, compile, create and run a Bluesim executable

From the workstation:

- Set the Compile to target to Bluesim

- Run the Full Rebuild task

From the command line:

- Compile " bsc -sim -g mkTestbench Testbench.bsv" (creates .ba files)
- Create: "bsc -sim -e mkTestbench -o testbench.exe *.ba"
- Run: "./testbench.exe -m 20"

As in Part 1, you can also compile, create and run a Verilog executable

From the workstation:

Note: You could create 2 separate project (.bspec) files in the same directory, using the same .bsv files; one for Verilog options and one for Bluesim options.

- Set the Compile to target to Verilog
- Run the Full Rebuild task

From the command line:

- Compile "bsc -verilog -g mkTestbench Testbench.bsv" (creates .v files)
- Create: "bsc -verilog -e mkTestbench -o testbench.exe *.v"
- Run: "./testbench.exe"

Compilation dependencies

- Above, when you compiled Functions.bsv, it created two intermediate files Functions.bi and Functions.bo. These were then used by the compiler when it encountered the "import Functions::*" statement while compiling Testbench.bsv. Try deleting all .bi and .bo files, and recompiling Testbench.bsv. You should get an error message that it cannot find the Functions.bi intermediate file.
- Instead of explicitly compiling Functions.bsv as you did above, bsc has a built-in mechanism, the -u flag, to chase such dependencies and compile any imported packages as necessary. Try the command: "bsc **-u** -sim -g mkTestbench testbench.bsv".
  You should no longer get the error message about Functions.bi (in fact, you should see a messages saying that bsc is checking package dependencies and is compiling Functions.bsv). **When compiling a project from the workstation, the -u flag is automatically used.**

Dumping VCDs

The Bluesim execution flag -V will dump a standard vcd file, which you can then view in any waveform viewer

From the workstation:

- Add the -V flag to the Simulate options field on the Link/Simulate tab in the Project Options. You can access the waveform viewer from the Module Browser window.

From the command line:

- Make the Bluesim executable again, and run it: "./testbench.exe -m 20 -V testbench.vcd"

Part 2.B: Polymorphic width integers

- Modify the functions in Functions.bsv so that they work with values of type Int#(n), i.e., using a type variable *n* instead of the fixed width 16.
  Note that in your rule in module mkTestbench which invokes the "increment()" and "decrement()" functions, their polymorphic width *n* is automatically matched to 16.
- Rebuild and re-execute (either Bluesim or Verilog sim) to confirm that it works.
- Modify module mkTestbench in Testbench.bsv so that the registers all hold Int#(83) values instead of Int#(16), i.e., 83-bit integers instead of 16-bit integers.
  Note that when invoking the "increment()" and "decrement()" functions, their polymorphic width *n* is automatically matched to 83 now, instead of 16.
- Rebuild and re-execute to confirm that it works.

Part 2.C: Adding an extra Integer argument to the functions

- In Functions.bsv, modify the functions so that, instead of always incrementing and decrementing by 1, they each take an extra argument of type "Integer", the amount by which to increment and decrement, respectively. For example, the increment function takes an argument x of type Int#(n) and an argument delta of type Integer, and returns the value of x+delta.
- Typecheck Functions.bsv

  You will likely observe type-checking errors in the expressions where you try to add/subtract the Int#(n) value to the Integer value. The reason is that these are different types—the Int#(n) type is of bounded width, whereas the Integer type is a true unbounded mathematical integer. This type error can be fixed by replacing the use of Integer value "delta" by "fromInteger(delta)". This performs the appropriate type and representation conversion. Please see the Reference Manual and training slides for more information about the overloaded function "fromInteger()". In particular, Integers (true unbounded integers) and the function fromInteger() are only used during static elaboration. Further, during static elaboration, the compiler will check that the actual given Integer argument fits in the bounded width of the desired result.

- With this fix, you should now be able to compile Functions.bsv
- In Testbench.bsv, modify the invocations of the increment function to provide the additional argument "5" as the increment amount, and "3" as the decrement amount.
- In Testbench.bsv, in module mkTestbench, introduce explicit termination by adding another rule called "stop" that invokes $finish() when the counter reaches a limit value (such as 100). Now, in Bluesim, you will no longer need to use the -m flag.
- Rebuild and re-execute to confirm that it works.

- Observe the values of the signal WILL_FIRE_RL_shift and WILL_FIRE_RL_stop (where "shift" is the name of the rule that does the shifting and "stop" is the name of the termination rule you just created):
  - If you are running Bluesim, you can observe the signals by creating the following Tcl script

```
while {true} {
    if [catch { sim step }] { break }
    puts "Clock at [sim time]"
    puts [format "WILL_FIRE_RL_shift  %s" [sim get [sim lookup WILL_FIRE_RL_shift]]]
    puts [format "WILL_FIRE_RL_stop   %s [sim get [sim lookup WILL_FIRE_RL_stop]]]
}
```

in a file "show_rules.tcl", and then running "./testbench.exe -f show_rules.tcl".
[In general, Bluesim can be controlled by a Tcl script. Other Tcl commands allow you to single step, dump state, and so on. Details may be found in the User Guide.]

- If you are running Verilog sim, you can dump VCDs and look for the signals WILL_FIRE_RL_shift and WILL_FIRE_RL_stop.

Observe that the "shift" rule fires on every clock (value 1'h1), whereas the "stop" rule does not fire (value 1'h0) until last clock before termination.

Part 2.D: Towards an elastic pipeline instead of a rigid pipeline

- In Testbench.bsv, modify the module mkTestbench so that each pipeline stage register is replaced by a FIFO of depth 4 (keep the names stage0, stage1 and stage2 for the FIFOs). Remember that in BSV a FIFO is instantiated using a FIFO module, (mkFIFO, mkSizedFIFO, mkFIFOF, ...) with an interface FIFO#(<type>) or FIFOF#(type)[5]. The FIFO modules and interfaces are declared in the library package FIFO; therefore you will need to import them with "import FIFO :: *" at the top of the package.
- In the "shift" rule that moves the pipeline stage data, and the "show" rule that displays pipeline stage data,
  - A stage register write (such as "stage0 <= ...") must be replaced by "stage0.enq(...)", i.e., a method invocation that enqueues a value into a FIFO. (And similarly for stage0 and stage2 writes).
  - A stage register read (such as "stage1") must be replaced by "stage1.first", i.e., a method invocation that returns the value of the head of the FIFO.
    (And similarly for stage0 and stage2 reads.)
  - Since the ".first()" methods are non-destructive, i.e., they do not actually remove an element from a FIFO, we must also add Actions such as "stage0.deq" in rule "shift" to remove old elements from stage registers.
- Rebuild, and re-execute (in Bluesim or Verilog sim).
  You will find that when you simulate, there is no output—the simulation exits immediately!
- Observe the WILL_FIRE_RL_shift and WILL_FIRE_RL_stop signals as you did in part 2.C. You should observe that the "shift" rule never fires. Why?

[5] Remember that the only way to interact with a module is to invoke the methods of its interface. Please see the Reference Guide for details on standard FIFO modules and interfaces.

View the schedule

From the workstation:

- Use the Schedule Analysis window

From the command line:

- Recompile with the -show-schedule flag:
  "bsc -sim -u -g mkTestbench **-show-schedule** Testbench.bsv"
  This should create a text file "mkTestbench.sched". View the contents of this file

Observe the predicate of the "shift" rule. What is the value of this predicate at the start of the simulation, and why? Given this, can you infer the value of the predicate after 1 clock cycle? After n clock cycles?

Part 2.E: Completing the transformation to an elastic pipeline

- Split the "shift" rule into four rules: Rule "shift0" just enqueues the counter value into the stage0 FIFO. Rule "shift1" reads the head of the stage0 FIFO and dequeues it, increments the value, and enqueues into the stage1 FIFO. Rule "shift2" reads the head of the stage1 FIFO and dequeues it, decrements the value, and enqueues into the stage2 FIFO. Finally, rule "shift3" dequeues the stage2 FIFO.
- The "show" rule has a $display statement to display the first elements of the stage0, stage1 and stage2 FIFOs. What happens in a clock cycle where some of the FIFOs are non-empty and some are empty? Fix this by splitting the "show" rule into three separate rules "show0", "show1" and "show2", that display the first elements of the stage0, stage1 and stage2 FIFOs respectively.
- Rebuild and re-execute to confirm that it works. When building, view use the Schedule Analysis window or use the -show-schedule flag to create the "mkTestbench.sched" file. Make sure that you understand its contents. When executing, observe the WILL_FIRE_RL_shift0, WILL_FIRE_RL_shift1, WILL_FIRE_RL_shift2 and WILL_FIRE_RL_shift3 signals, particularly in the first few clocks of the simulation.
- What would happen if you deleted rule "shift3" (the one that dequeue the stage2 FIFO)? Try it, to confirm your analysis. (You may want to observe the WILL_FIRE signals for this as well.) What would happen if you increased the capacity of the FIFOs from 4 to 10?

Part 3

In this part you will learn the following:

- How to refine a monolithic module into two functionally separate modules
- How to use an interface for communication between modules
- How to instantiate a user-defined module
- Use of the zeroExtend() function
- Use of the pack() and unpack() functions
- Use of the (* synthesize *) compiler attribute

Part 3.A: Module hierarchies and separate compilation/synthesis

- Copy the Pipeline/Part2 files into a new directory Pipeline/Part3/. Also copy the file Testbench.bsv into another file Pipeline.bsv. We are going to separate the functionality currently in the mkTestbench module into two parts:
  1) A "design" module containing the basic 3-stage pipeline and an interface through which an outer module feeds values into it and drains values out of it.
  2) A "testbench" module containing the stimulus counter, an instance of the design module, and rules that feed values into the design and drain values out of it through the design module's interface.

  Instead of defining a new interface type for this purpose, we shall just reuse the standard FIFO interface type, since the 3-stage pipeline itself looks like a FIFO with some internal computation. In the Reference Guide, you will see that the FIFO interface is:

```
interface FIFO#(type t);
        method Action   enq (t  x);
        method t        first ();
        method Action   deq ();
        method Action   clear;
endinterface
```

  so our design module will have to implement these methods.
- In the file Pipeline.bsv, change the package name from Testbench to Pipeline. Rename the module mkTestbench to mkPipeline, and modify it so that it has a FIFO#(Int#(16)) interface, where previously it had no interface.
- Remove all the stimulus "counter" stuff used for feeding the pipeline (it'll go into the testbench):
  - Remove the "counter" register instantiation
  - Remove the "stimuli" rule that increments the counter
  - Remove the rule "shift0" that enqueues the counter value into the stage0 FIFO.
  - Remove the "stop" rule that invokes $finish() when the counter reaches 100.
- Retain the "shift1" and "shift2" rules.
- Remove the "shift3" rule that drained the stage2 FIFO.

- Add definitions for the interface methods for *enq(), first(), deq()* and *clear()*:
  - enq() will enqueue its argument into the stage0 FIFO
  - first() will return the first element of the stage2 FIFO
  - deq() will drain the stage2 FIFO
  - clear() will clear all three stage FIFOs
- In the file Testbench.bsv, to the existing import list (FIFO and Functions), add an import for the Pipeline package.
- Change the type of the existing "counter" register from Int#(16) to UInt#(8), i.e., unsigned integers of width 8.
- Replace the instantiation of the 3 stage FIFOs by an instantiation of the module mkPipeline. Let the name of the instance be "pipe".
- Retain the "stimuli" rule that increments the counter.
- Edit the "shift0" rule so that it enqueues the counter value into "pipe" instead of the earlier stage0 FIFO.
- Delete the "shift1" and "shift2" rules.
- Edit the "shift3" rule so that it drains the "pipe"module, instead of the earlier stage2 FIFO.
- Delete the "show" rules.
- Retain the "stop" rule.

Rebuild

From the workstation:

- Change the Top File to Testbench.bsv , rebuild

From the command line:

- Try rebuilding ("bsc -u <...flags...> Testbench.bsv")

You will likely encounter a type-checking error in rule "shift0", because we are invoking the "pipe.enq()" method with a UInt#(8) argument whereas the interface is for Int#(16) values. We need a conversion function.

Conversion Functions:

- A UInt#(8) value can be converted into a Bit#(8) value using the **pack()** function.
- A Bit#(8) value can be zero-extended into a Bit#(16) value using the **zeroExtend()** function.
- A Bit#(16) value can be converted into an Int#(16) value using the **unpack()** function.

These functions are described in more detail in the Reference Guide. They are all "overloaded" functions, i.e., **pack()** converts various types into bits, **zeroExtend()** can extend between various widths, and **unpack()** converts bits into various types.

Also, instead of **zeroExtend()**, you can also use the Verilog {0, counter} bit-concatenation notation.

Rebuild and Re-execute

- Rebuild and re-execute to ensure that everything is working.

- If you did not build for Verilog sim in the previous step, do so now, and observe which Verilog files are created. Then, in Pipeline.bsv, add a line containing "(* synthesize *)" just before the mkPipeline module header. Rebuild, and again observe which Verilog files are created. If you are familiar with Verilog, examine the Verilog files to see how the mkTestbench module instantiates the mkPipeline module in the Verilog.

  Note: In general, bsc creates separate Verilog modules, and separate Verilog files for each module that either has the (* synthesize *) attribute or is named with the -g flag on the compiler command line. In general, we recommend using the (* synthesize *) attribute rather than the -g flag, so that this module-hierarchy decision is documented in the source file (instead of in command-line scripts or Makefiles).

Part 4

In this part you will learn the following:

- How to generalize the 3-stage pipeline to an n-stage pipeline
- How to generalize the mkPipeline module to take *n* as a parameter
- How to make a synthesizable wrapper for a non-synthesizable module

Part 4.A: generalizing from 3-stages to n stages

- Copy the Pipeline/Part3 files into a new directory Pipeline/Part4/. If using the workstation, copy the Pipeline.bspec file as well.
- In Pipeline.bsv, in module mkPipeline, replace the 3 explicit instantiations of the 3 FIFOs stage0, stage1 and stage2 with the following:

```
Integer n = 3;

FIFO#(Int#(16)) stages [n];
for (Integer j = 0; j < n; j = j + 1)
   stages [j] <- mkSizedFIFO (4);
```

  This declares an Integer n with value 3, and then an array called "stages" of size *n,* containing FIFO#(Int#(16)) interfaces. The for-loop instantiates *n* FIFOs and retains the resulting interfaces in the array.
- Replace the two rules "shift1" and "shift2" rules with a loop generating *n-1* rules:

```
for (Integer j = 1; j < n; j = j + 1)
   rule shift_j;
      stages[j].enq (increment (stages[j-1].first, j*5));
      stages[j].deq;.
   endrule
```

  (Here, we've chosen an arbitrary j*5 as the increment value for the j'th stage.)
- Similarly, replace the three rules "show0", "show1" and "show2" with a loop that generates *n* rules.
- Finally, edit the body of the "clear" method to clear all *n* FIFOs.
- Rebuild and re-execute, and ensure that it is working.
- Note: now, to convert the 3-stage pipeline to, say, 5 stages, one just needs to change the value defined for *n.* Try it—edit the definition for *n,* rebuild and re-execute.

Part 4.B: Generalizing the module to take n, the pipeline depth, as a parameter

- In Pipeline.bsv, in the module mkPipeline, remove the "Integer n=3;" declaration and add an "Integer n" parameter to the module.

```
module mkPipeline #(Integer n) (FIFO#(...));
```

- In Testbench.bsv, in module mkTestbench, replace the "mkPipeline" module instantiation by "mkPipeline(5)", i.e., instantiating a 5-stage pipeline.
- Try to rebuild.
  You are likely to see an error message saying that the module mkPipeline cannot be synthesized because it has an Integer parameter (assuming you have retained the (*synthesize*) attribute on the module from before). The reason is that the Integer type represents true unbounded integers, for which it is not possible to generate fixed hardware (a fixed number of wires) to carry this value.
- Remove or comment-out the (*synthesize*) attribute on the module. Rebuild and re-execute to ensure that your 5-stage pipeline is working correctly. Without this attribute, the module just gets inlined into the parent module (mkTestbench), where static elaboration resolves all references to Integers, i.e., it is no longer necessary to build hardware to carry an Integer value.

Part 4.C: Creating a synthesizable wrapper for a non-synthesizable module

In the previous part, we created a parameterized module, but lost a level of module hierarchy because we could not synthesize it separately. Here we'll restore that lost level of module hierarchy. In Pipeline.bsv, add a new module definition like this:

```
(* synthesize *)
module mkPipeline_5  (FIFO#(...));
   let m <- mkPipeline (5);
   return m;
endmodule
```

We call this module a "specialization" of mkPipeline, i.e, it has nailed down the unsynthesizable Integer parameter to the specific value 5. This module has the (*synthesize*) attribute, and this is indeed now synthesizable.

- In Testbench.bsv, in module mkTestbench, replace the "mkPipeline(5)" module instantiation by "mkPipeline_5", i.e, by an instantiation of the specialized module.
- Rebuild and re-execute and ensure that everything is working. If you build for Verilog sim, you will see that you indeed get separate Verilog modules for the testbench and the design.