The Pipeline Lab

"Hello Bluespec!"

This lab will lead you through basic aspects of the Bluespec System Verilog language and the usage of the Bluespec compiler, Bluesim and Blueview tools.
In each part you will find a description of the actual code you will have to generate and how to use and invoke each of the previously mentioned tools.

Note 1: *The directory $BLUESPECDIR/../doc contains very useful documentation that includes the Language Reference Guide, User Guide, Known Problems and Solutions and Style Guide among others. Under $BLUESPECDIR/../BSV_Examples directory you will find some useful examples for future reference.*

Note 2: *you might want to run simulations of the generated Verilog code using your favorite Verilog simulator. For this and at this basic stage you will need to use the provided Verilog driver which can be found in $BLUESPECDIR/../Verilog/main.v with the next command lines:*

- *For vcs: vcs +v2k +define+TOP=<topName> $BLUESPECDIR/../Verilog/main.v <verilogFile>*
- *For Ncverilog: ncverilog +define+TOP=<topName> $BLUESPECDIR/../Verilog/main.v <verilogFile>*
- *For iverilog: iverilog -DTOP=<topName> $BLUESPECDIR/../Verilog/main.v <verilogFile>*

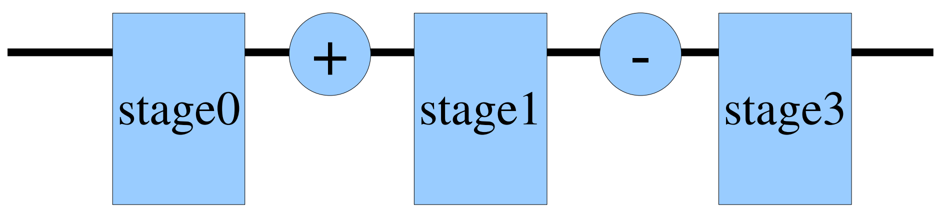
In the case the design uses other Verilog modules from the Bluespec library, these has to be included via the following option to the simulators: +libext+.v -y . -y $BLUESPECDIR/Verilog/

Part 1

In this lab you will learn:

- How to create a very simple module without interface and arguments
- How to instantiate registers
- How to write simple functions and rules
- How to compile and simulate

Background:
Our first design is going to be a pipeline with three registers: stage0, stage1 and stage2 and using an increment and decrement function. The design will not have interface and therefore there will no way to communicate with the external world, (it could be seen as a testbench that incorporates a design). Next figure shows a block diagram of our design:



Actions:

- Create the file pipeline.bsv in a new directory Pipeline/Lab1.
- Define the package pipeline using keywords package/endpackage.
- Define the module mkPipeline which will not have any interface and no arguments.
- Instantiate 3 registers, one for each stage, of type Bit#(16) with reset value 0. Remember that in Bluespec System Verilog a register is created by instantiating a register module, (mkReg, mkRegA, ...) with a register interface Reg#(<type>).
- Create two functions:
  - "increment", such that given a value of type Bit#(16), it increments it by 1 and return the result.
  - "decrement", such that given a value of type Bit#(16), it decrements it by 1 and returns the result

  You might want to define these functions outside of the module in order to reuse them in other designs.
- Create a simple rule[1] that will be fired each cycle and creates the connections of the

[1] A rule without conditions will be fired each clock cycle, i.e. all the statements enclosed between the keywords: "rule;" and "endrule" will be executed in parallel each clock cycle. Due to the concurrent character the "non-blocking" asisgnment notation "<=" has to be used.

pipeline in such a way that stage0's output gets incremented by one and stage1's output gets decremented by one, as in the figure.

- Create a simple rule that will be fired every cycle that uses $display in order to show the stage0, stage1 and stage2 values. Use %d for better visualization.

Detailed information on the next commands, flags and options may be found under *$BLUESPECDIR/../doc.*

A) Obtaining and running the verilog design:
  - run the command “bsc -verilog -g mkPipeline pipeline.bsv”, this will create mkPipeline.v if everything is correct.
  - run “bsc -e mkPipeline *.v” to create a verilog executable (bsc will determine which simulator you have, and use it)
  - write “./a.out” to execute

B) Obtaining and running a Bluesim executable:
  - run “bsc -sim -g mkPipeline pipeline.bsv”. This will create the necessary objects for the simulation.
  - run “bsc -e mkPipeline -o pipeline.exe *.o” This will link all the necessary libraries and create the executable pipeline.exe.
  - execute the simulation by writing “./pipeline.exe -m 20”. The flag -m 20 will make the simulator running for just 20 clock cycles.

- Modify the design so that the reset strategy for the register is different. What kind of mkReg will be needed to instantiate?
- Create a rule that increments a counter each cycle. Feed the counter's value in the first stage of the pipeline.
- Repeat steps A, B and C.

Part 2

In this Lab you will learn:

- How to divide the design in different files and the directive "import"
- Differences with Int#(n) type
- Separated and incremental compiling
- Using a defined type of undefined size
- Passing multiple arguments to a function
- Use of *fromInteger* function

Actions:

- Copy the Pipeline/Part1 files into a new directory Pipeline/Part2 directory
- Modify the design in order to be able to manipulate Int#(16) instead of Bits#(16)
- Create the file functions.bsv. Cut and paste to this file our increment and decrement functions. Use package/endpackage keywords for defining the functions package.

A) Compile the functions package by running the command: bsc functions.bsv Note that the bsc options -verilog or -sim are not specified – these are only need when a Bluesim or simulation is required, or when compiling into Verilog.

- Add "import functions :: *" on top of the mkPipeline module definition in order to make the functions visible.

B) Obtaining the verilog design:

- run the command "bsc -verilog -g mkPipeline mkPipeline.bsv", this will create mkPipeline.v if everything is correct.

It is possible to make an incremental compilation by using the -u flag. Delete all the intermediate files *.bi and *.bo and try the command: bsc -verilog -u -g mkPipeline pipeline.bsv. The result will be the same as steps A and B

- run the command "bsc -e mkPipeline *.v" to create a verilog executable.

C) Obtaining a Bluesim executable:

- run "bsc -sim -u -g mkPipeline pipeline.bsv". This will create the necessary objects for the simulation.
- run "bsc -e mkPipeline -o pipeline.exe *.o" This will link all the necessary libraries and create the executable pipeline.exe.

D) Running a Bluesim executable:

- execute the simulation by writing "./pipeline.exe -m 20 -V pipeline.vcd". The flag -V will dump an standard vcd file.

E) Running Blueview:

- Bluesim requires to get as input a cross reference file that is generated during verilog compilation time. Delete all *.bi and *.bo files and recompile the design to Verilog using the -cross-info flag. "bsc -verilog -u -cross-info -g mkPipeline pipeline.bsv". This will create the file mkPipeline.info
- execute "blueview mkPipeline.info". Use blueview, explore the correspondence between the source BSV file and the generated Verilog file.

- Modify the file functions.bsv such that increment and decrement accept arguments of type Int#(n).
- Modify the module pipeline.bsv design in order to be able to manipulate Int#(128)
- Choose two Int values for being used by the increment and decrement functions and use them in each function invokation from pipeline.bsv design.

F) Repeat steps A, B and C for obtaining the verilog code and the simulation executable.
G) Execute the simulation by writing "./pipeline.exe -m 20 –V pipeline.vcd –T 10ns/1ns –p 25". The flag -T will set the timescale and precision and the flag –p will set the clock period using the timescale value.

- Modify the file pipeline.bsv such that the pipeline uses fifos of depth 4 instead of registers. Remember that in Bluespec System Verilog a fifo is created by instantiating a fifo module, (mkFIFO, mkSizedFIFO, mkFIFOF, ...) with a fifo interface FIFO#(<type>), FIFOF#(type). The fifo modules and interfaces are declared in the library package FIFO, therefore you will need to import them with "import FIFO :: *"
- Use the counter rule in order to finish the simulation when a limit value is reached by using $finish(). This will allow you execute step G without the -m flag.

Why the simulation does not show the expected results? why the display does not work?

Part 3

In this Lab you will learn:

- How to create and use an interface
- Module instantiation
- The (* synthesize *) pragma
- Testbench creation

Actions:

- Copy the Pipeline/Part2 files into a new directory Pipeline/Part3 directory
- Create a file called pipelineInterface.bsv and create the interface IfcPipeline with methods write and read.
- Modify the design pipeline.bsv in order to provide the interface IfcPipeline. Do not forget to import the interface file.
- Remove the counter used for counting simulation cycles, this will be controlled from the testbench.
- Remove the counter used for increasing the stage0's input. Now our design will communicate with the external world through its interface.
- Add the methods *read, write, clear* and *consume* to pipeline.bsv. write will invoke the enq method of stage0, read will invoke the first method of stage2, consum will invoke the deq method of stage2 and clear will invoke the clear methods of all stages. Write *(* synthesize *)* on top of the module declaration.

A) Compile the design to Verilog using the command: bsc -verilog -u pipeline.bsv. Note that you don't need to set the flag -g in order to indicate the compiler to synthesize the module into Verilog thanks to the *(* synthesize *)* attribute.

- Create a testbench file pipelineTest.bsv. This file will contain a module mkTestbench which will instantiate our design, do not forget to import it. Write *(* synthesize *)* on top of the module declaration.
- Create a rule that will increment a counter each cycle. It is possible to use the counter value to drive the inputs in our design under test. This counter can be used also for limiting the simulation time.

B) run bsc -verilog pipelineTest.bsv for obtaining the verilog testbench.

C) Obtaining a Bluesim executable:

- run "bsc -sim -g mkTestbench pipelineTest.bsv". This will create the necessary objects for the simulation.
- run "bsc -e mkTestbench -o pipelineTest.exe *.o" This will link all the necessary libraries and create the executable pipelineTest.exe.

D) Run the simulation by executing: ./pipelineTest.exe -cc. the -cc flag will output each cycle in your simulation.

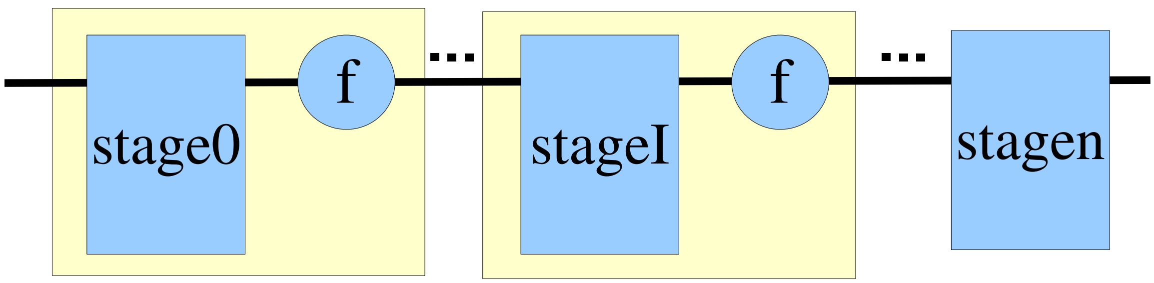
- Remove the *(* synthesize *)* attribute from pipeline.bsv

E) Delete all the *.bi, *.bo and *.v files. Run bsc -verilog -u pipelineTest.bsv. What is the effect of not having the *(* synthesize *)* attribute in a deeper hierarchical module?

Part 4

In this Lab you will learn:
- Hierarchical designs
- The (* synthesize *) attribute
- Complex recursion

Lets Consider the next figure in order to generalize the pipeline:



You will modify the design in such a way that it will instantiate five modules. Each of these modules will be composed by a FIFO and an increment function. The resulting design will have two levels of hierarchy.

Actions:
- Copy the Pipeline/Part3 files into a new directory Pipeline/Part4 directory
- Copy the file pipelineInterface.bsv to singleInterface.bsv. In singleInterface.bsv change the interface name to IfcSingle.
- Create a file single.bsv and define a single module mkSingle and uses the IfcSingle interface. Instanciate a FIFO and define the methods as declared in IfcSingle. Use the increment function from the functions package when reading a value from the FIFO.
- Modify the interface IfcPipeline in pipelineInterface.bsv such that now it will contain three methods, writePipe , readPipe and clearPipe.
  - readPipe will be now an ActionValue method that extracts a value from the last stage and brings the value out.
  - ClearPipe will clear all the FIFOs instantiated in each single module and the last stage FIFO.

Now you will need to instantiate 5 mkSingle mkSingle modules using IfcSingle interface inside of the mkPipeline module. For doing this you can use a for loop that will instantiate the modules but you will need to remember the instance name somehow.
Declare an array of 5 elements of type IfcSingle:

```
IfcSingle#(UInt#(32)) singleArray[5];
```

You will to do the instantiation of the 5 modules as follows:

```
for(Integer i = 0; i < 5; i = i + 1)
begin
    IfcSingle#(UInt#(32))  a <- mkSingle;
    singleArray[i] = a;
end
```

In order to create the connections you will proceed similarly with a for loop:

```
for(Integer i = 0; i < 4; i = i + 1)
begin
    rule
        // Make the connections here
    endrule
end
```

This will create 4 rules that interconnect the 5 single instantiations between them.

- Instantiate the FIFO corresponding to the last stage and connect the last stage.
- Create/modify the methods as necessary
- Modify the testbench as needed.

Repeat steps A, B, C and D.

- How to generalize the number of stages? and the function used in each stage?

Part 5

In this Lab you will learn:
- How to use a polymorphic type
- Needed provisos

Actions:
- Copy the Pipeline/Part4 files into a new directory Pipeline/Part5 directory
- Modify functions.bsv such that increment and decrement accept arguments of polymorphic type t.
- Modify the module pipeline.bsv design in order to be able to manipulate UInt#(32).

A) run bsc -verilog -u -g mkPipeline pipeline.bsv for obtaining the verilog design.
  - run the command "bsc -e mkPipeline *.v" to create a verilog executable.

B) Obtaining a Bluesim executable:
  - run "bsc -sim -g mkPipeline pipeline.bsv". This will create the necessary objects for the simulation.
  - run "bsc -e mkPipeline -o pipeline.exe *.o" This will link all the necessary libraries and create the executable pipeline.exe.

D) Check the file *$BLUESIMDIR/../doc/user-guide* in order to see all the available flags during execution. How can you check the final and the partials cycle times?