The Multiplier Lab

This lab will lead you through basic aspects of the BSV (Bluespec SystemVerilog) language and the usage of the Bluespec workstation 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 the development workstation.

The lab is divided into multiple directories (mult0, mult1, mult2, and mult3) each containing the lab and the solution for each .bsv file. Copy <file>.bsv.lab to <file>.bsv before you start each section. That way if the lab has been worked on previously, you will have a clean copy to work on.

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.*

You may want to refer to the syntax cheat-sheet at the end of this document.

Part 1: Mult0

Part 1a: Compile to Verilog

- Start the development workstation and open project Mult0
  `bluespec mult0.bspec`
- Examine the Project Options (Project -> Options)
  On the Compile tab verify the following options:
    Compile to: **Verilog**
    Compile **via bsc**
  On the Link/Simulate tab:
    Select a Verilog simulator
  On the Editor tab:
    Set the editor to your text editor
  Save and Close the Options window
- Examine the multiplier code in Mult0.bsv.
  Double click on the file name in the Project Options window to open the file in your selected editor
- Compile (Build -> Compile or from the task bar)

Part 1b: Compile to Bluesim and Simulate

- Modify the Project Options
  On the Compile tab change the following option:
    Compile to: **Bluesim**
  On the Link/Simulate tab:
    Run Options: `-m 30`
    This tells Bluesim to run the executable for 30 clock cycles
- Full Rebuild
  Full rebuild will run the following tasks:
    - Clean
    - Compile

- Link
- Simulate
  The output should be:
  Product = 45 (repeated 30 times)

Part 1c: Add a $finish to the design

- Modify the design so that the simulation finishes after displaying the result
- Rebuild

Part 1d: Produce a vcd file from a Verilog Simulation

- Modify the Project Options
  Change compile to back to Verilog
  On the Link/Simulate tab add the following flag to the simulate options field:
  `+bscvcd`
  Select a verilog simulator
- Rebuild
- Open the **Module Browser** window
- Load the top module (`mkMult0`)
  Module -> Load Top Module
- View the waveforms
  - Start the waveform viewer (Wave Viewer -> Start)
  - Load Dump File (Wave Viewer -> Load Dump File)
  - Select `dump.vcd`

Part 1e: Examine the Verilog file mkMult0.v

- In the **Project Files** window, open the file `mkMult0.v`
- Review the following statements:
  - the CLK and RST_N (active-low reset) signals introduced by bsc
  - register instantiations
  - assignments to register inputs (...$D_IN)
  - assignments to register enables (...$EN)

Part 2: Mult1 Adding a Testbench

This is a version of the multiplier design with a separate testbench. Use whichever of the techniques and simulators you prefer from the previous lab to examine and test this design.

Part 2.A

- Using the Bluespec development workstation open the project file `mult1`
- Examine the files in the project. Notice that the design has been divided into three files:
  - `Mult1.bsv:` the multipler (the dut)
  - `Mult.bsv:` the multiplier interface
  - `Mult1Tb.bsv:` the testbench
- As before, arrange that the simulation terminates when it has displayed all the results. This time, though, do so by adding a **rule** to the testbench (`Mult1Tb.bsv`) to invoke `$finish(0).`
- As written, the design of the dut does not ensure that a result has been read before the next calculation is allowed to begin. Amend the design to provide for this.
  - Add a third method to the interface definition (in Mult.bsv)

    ```
    method Action acknowledge();
    ```
  - Place a call of this method at the appropriate place in the testbench (`Mult1Tb.bsv`).
  - Implement the method in the dut (`Mult1.bsv`)
  - Add an extra register, `available`, with contents of type `Bool`
  - Arrange that the appropriate methods set it to `True` and `False` respectively.
- (Optional) Make the following improvement to the architecture of the dut. We can economize on registers by storing the mplr value in the lower half of the product register. Then, instead of adding mcand into the lower end of the product and shifting mcand left after each iteration, we add it into the upper end of the product and shift the product to the right after each iteration. This means that mcand need no longer be a double-length register; and the single right-shift of the product register places both mplr and the partial result into their new positions. Implement and test this micro-architecture change.

Part 3: Mult2 BOOTH MULTIPLICATION

This is a technique for speeding up naive multiplication by a constant factor, based on the fact that shifting is quicker than adding (or subtracting). The key observation is the following. A run of consecutive 1s in a binary number

```
          m...   n... 210
 0000000000001111110000000
```

is equal to ($2^m - 2^n$). We use this to construct a *Booth encoding* of the multiplier, considering one or more bits at a time. We'll actually be using a two-bit encoding, but let's consider the one-bit encoding first as a simpler example. Whenever we consider a bit we also look at the bit to its right.

The encoding is as follows:

The_bit	**Adj_bit**		**Contribution**
0	0	zero on its own:	0
0	1	high end of a run of ones:	1
1	0	low end of a run of ones:	-1
1	1	middle of a run of ones:	0

The actual contribution to the value is of course the value shown times 2^n, where n is its bit-position.

The two-bit version is as follows (here the value will be multiplied by 2^n where n is the position of the less significant bit):

The_bits	**Adj_bit**		**Contribution**
00	0	zero on its own:	0
00	1	high end of a run of ones:	1
01	0	an isolated one:	1
01	1	high end of a run of ones:	2
10	0	low end of a run of ones:	-2
10	1	end of a run, start of another:	-1 (i.e. -2 + 1)
11	0	low end of a run of ones:	-1
11	1	middle of a run of ones:	0

When each pair of bits is encoded in this way, its contribution to the partial product may be found by multiplying and adding the multiplicand by the appropriate factor (i.e. shifting and adding-or-subtracting).

This encoding is done in the Mult2 design by the function `boothenc`. It takes two arguments: the first is a three-bit quantity (note that the least significant of these is the "adjacent bit" above). The second argument will be the current value of the multiplier; the function produces the contribution of the two bits concerned to the partial product. The "cycle" rule can thus deal with the bits of the multiplier two at a time.

TASKS

- Complete the `cycle` rule according to the description given above.
- Write the body of the `result` method.
- Test and debug the completed design.
- (optional) Arrange for the result to be acknowledged, as in the previous lab. Again you will have to amend the interface definition, the dut and the testbench; but this time, instead of adding an extra method, change the `result` method to be of type `ActionValue(Tout)`. Then calling this method will automatically acknowledge its receipt, and the body of the method should record the acknowledgment.

Part 4: Mult3 Adding a Server Farm

This is a considerably more complicated design than any of the previous ones, and uses some of BSV's more advanced features. However, the tasks you are asked to do are quite self-contained, and do not require familiarity with all the advanced stuff. So we suggest you approach this lab at two levels. For the tasks themselves you will have to understand what you are doing, and the syntax involved, in detail; but we suggest you also look at the rest of the design to understand in general terms how it is put together, but not trying to learn all the syntactic and other details -- they are all described in the documentation, and will be there for you to study (maybe looking at this example again) if you need to use them later.

Server Farms

A server farm is a set of identical servers, which can each perform the same task, together with a controller. The controller allocates incoming tasks to any server which happens to be available (free), and sends results back to its caller. This design includes two server farms, one for each of the two multiplication modules (mkMult1 and mkMult2) of the previous two labs. The testbench sends identical random tasks to each farm, and checks to see whether corresponding results returned from each are the same.

For both kinds of server, the time needed to complete each task depends on the value of the multiplier argument; there is therefore no guarantee that results will become available in the order the tasks were started. It is required, however, that the controller return results to its caller in the order the tasks were received. The controller accordingly must instantiate a special mechanism for this purpose. Fortunately, the appropriate mechanism is available in a library package.

Completion Buffers

The `CompletionBuffer` package may be imported from the library. The `CompletionBuffer` interface provides three methods. The `reserve` method allows the caller to reserve a slot in the buffer; the method returns a token holding the identity of the slot. When a task completes, the result may be stored in the buffer using the `complete` method; this takes a pair of values as its argument -- the token identifying its slot, and the result itself. Finally, results may be retrieved from the buffer using the `drain` method, which returns results in the order in which the tokens were originally allocated; thus the results of quick tasks might have to wait in the buffer while a lengthy task ahead of them completes.

The type of the interface is as follows.

```
interface CompletionBuffer #(type n, type a);
    interface Get#(CBToken#(n)) reserve();
```

```
        interface Put#(Tuple2 #(CBToken#(n), a)) complete();
        interface Get#(a) drain();
    endinterface: CompletionBuffer
```

Each methods is actually in its own sub-interface -- a `Get` interface for those which get information from the buffer, and a `Put` interface for the one which sends information to the buffer. The `Get` and `Put` interfaces are described in a lecture, and are available in the `GetPut` library package. Note that the type of the items to be stored, "`a`", is a type argument for this definition, and so is the required size of the buffer, "`n`". "`n`" is also a type argument for `CBToken`, the type for the tokens issued. This allows the type-checking phase of the synthesis to ensure that the tokens are the appropriate size for the buffer, and that all the buffer's internal registers are of the correct sizes too.
The `mkCompletionBuffer` module may be used to instantiate a completion buffer. It takes no size arguments, as all that information is already contained in the type of the interface it is being asked to produce. A typical instantiation is:

```
    CompletionBuffer#(17, nat) cb <- mkCompletionBuffer;
```

which instantiates a buffer of size 17 to hold items of type `nat`. Then a token may be claimed from this buffer by:

```
   CBToken#(17) t <- cb.reserve.get();
```

or, more succinctly, by

```
    let t <- cb.reserve.get();
```

(there is no need to give the type of "`t`" explicitly -- the tool can work it out for itself). Here, "reserve" is the name of one of "`cb`"'s sub-interfaces, and `get` is the name of the method of that sub-interface. A result `res` may be stored in the buffer by the call:

```
    cb.complete.put(tuple2(t, res));
```

TheMult3 Design

Note that the versions of `Mult1` and `Mult2` used in this design use the `ActionValue` form of the `Mult_IFC` interface, as produced by you during the previous lab.

The modules "`mkMult1Farm`" and "`mkMult2Farm`" instantiate the two farms. Each declares and instantiates an array of servers, and passes it as an argument to the `mkFarm` module, which instantiates the control mechanism. `mkFarm` accepts the array of server interfaces (we actually use the word `List` for the type of an array). It constructs a FIFO to hold the input queue, and a completion buffer for results. It constructs an array of Boolean registers to tell which servers are free, and an array of token registers to hold the tokens for tasks in progress. Then, in a loop (executed at synthesis time), it instantiates the elements of the array, and declares a pair of rules to handle each server in the list. The start method of the controller itself merely enqueues incoming tasks on the fifo; its end method is simply the "drain" method of the completion buffer.

The Testbench Mult3Tb

The testbench instantiates the two farms, and sends identical tasks to each. Each task, however, requires two random numbers, and the random number generator provides a stream of numbers one at a time. A convenient way of handling this predicament is to split the stream into a pair of fifos, with rules to keep each fifo topped up; then the rule which starts a task can take one element from each fifo.

The random number generator is in a separate package. For any who are interested, its design is described in internal comments.

Advanced Features Used in this Design

The following is a checklist of some of the advanced features used in this design.

- Passing complex types (arrays of interfaces) as arguments to modules.
- Rules generated inside synthesis-time loops.
- Multiple rules (even loop-generated ones) manipulating the same resource (the same completion buffer and the same input fifo), relying on the tool to provide all the control logic.
- Polymorphic modules: for example, mkCompletionBuffer constructs completion buffers for any type of item (provided it can be stored as bits) and any size of buffer. The type-checker ensures that each particular instantiation of mkCompletionBuffer uses tokens of the appropriate size.
- Definition of an interface method as one of the methods of an internal module (e.g. the "result" method of "mkFarm").
- The use of previously prepared and verified library packages.

Tasks

- Complete the bodies of the two incomplete methods in mkFarm (the one which processes the end of a server task, retrieving the result and sending it, along with the token, to the completion buffer; and the one which accepts new tasks from the farm's caller and enqueues them on the input fifo).
- Test and debug the completed design.
- Synthesizing this design results in lots of warning messages about rule conflicts. These do not actually matter (after all, if several servers are free id doesn't really matter which one gets the job); but it is often thought good practice to eliminate warnings. Examine these warnings, and understand how they arise. Eliminate at least some of them.
- Amend the testbench so that it no longer displays arguments as tasks are dispatched, but instead displays the arguments alongside their results. For this, instantiate a fifo of the appropriate size, to hold pairs of arguments. The start_task rule should of course enqueue each pair in this fifo, and the end_task rule retrive them for display.

Syntax Cheat Sheets

Part 1

Comments:

```
  // line comment
     /* block comment */
```

Declaring a module:

```
   (* synthesize *)
     module moduleName();
       ... <module contents> ...
     endmodule: moduleName     // ": moduleName" optional
```

The (* synthesize *) attribute ensures that moduleName will be compiled into a separate Verilog module (it would be inlined otherwise).

Instantiating a register "r", containing type int (= 32-bit), and reset to 42, inside a module:

```
  Reg#(int) r <- mkReg(42); // shortcut for register instantiation
   alternately,
  // long form of register instantiation
  Reg#(int) r();            // instantiate interface; don't forget parens!
  mkReg#(42) r_instance(r); // instantiate module and connect to interface
```

Adding a rule to a module:

```
  rule ruleName(predicate); // (predicate) may be omitted (= True)
       ... <actions> ...
  endrule: ruleName         // ": ruleName" optional
```

Incrementing a register "r" in a rule:

```
     r <= r + 1;               // right-hand-side can be any expression
   (Note that "r = r + 1" will not work; why?)
```

Displaying the value of a register "r" (formatted as decimal) in a rule:

```
     $display("%d", r);        // %-formatting as in Verilog
```

Ending a simulation:

```
     $finish(0);
```

This ends silently; if the argument is 1 or 2 instead, some closing information will be displayed too (depending on which simulator is being used). Several "system tasks" (e.g. a $display and a $finish) may occur in the same action block: although from the Bluespec point of view these happen in parallel, the simulators will give effect to them in their order in the source text.

Part 2

Interface declaration:

```
interface IfcName;
     // value method
     method resultType methodName(argType argName, ...);
     // action method
    method Action methodName(argType argName, ...);
     // actionvalue method
     method ActionValue#(resultType) methodName(argType argName, ...);
   endinterface: IfcName
```

Implementing a value-method inside a module:

```
   method returnType methodName(argType argName, ...);
     ... <computations> ...
     return ... <expression> ...
   endmethod: methodName
```

Implementing an action-method inside a module:

```
 method Action methodName(argType argName, ...);
     action
       ... <actions> ...
     endaction
  endmethod: methodName
```

Implementing a method with implicit conditions inside a module:

```
method returnType methodName(argType argName, ...) if (expression);
     ...
endmethod
```

Importing a package within another:

```
import <packagename>::*;
```

Note: the "*" indicates that all the definitions in <packagename> are to be imported; we do not support selective importing.

Instantiating a register containing a Bool:

```
Reg#(Bool) regName <- mkReg(True);  // Bool = True or False
```

Part 3

Implementing an ActionValue method inside a module:
(note that ActionValue method cannot take arguments.)

```
method ActionValue#(returnType) methodName()  if (expression);
     actionvalue
       ... <actions> ...
     return ... <expression> ...
     endactionvalue
endmethod
```

Invoking an Actionvalue method inside a rule (or inside another method):

variableName <- interfaceName.methodName();
or
typeName variableName <- interfaceName.methodName();
or
let variableName <- interfaceName.methodName();

Part 4

Instantiating a FIFO containing ints inside a module:

```
 FIFO#(int) fifoName <- mkFIFO();
```

Instantiating a five-deep FIFO of ints inside a module:

```
FIFO#(int) fifoName <- mkSizedFIFO(5);
```

The FIFO interface (from FIFO library):

```
interface FIFO#(int);
     method Action enq(int value);
     method int    first();
     method Action deq();
     method Action clear() ;
 endinterface: FIFO
```

As a convention, we recommend an alternative syntax for defining modules with port-like arguments (e.g. the list of interfaces for mkFarm).

The original syntax

```
module mkFarm#(module#(Mult_IFC) mkM) (Mult_IFC);
```

continues to work; but it is preferable to move the port-like argument into the second argument list:

```
   module mkFarm(module#(Mult_IFC) mkM, Mult_IFC ifc);
```

The meaning is exactly the same: any extra elements in the second list are treated as if they are appended onto the end of the first list. Note that in the alternative syntax, the final element (the interface provided by the module) must have an identifier as well as the type.