 

# The Fibonacci 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 three directories (fib1, fib2, and fib3) . Each directory contains the solution for each .bsv file. Each directory also contains a .bspec file, the project file used by 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.*

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

---

# Part 1: Fib1

## *Part 1a: Compile to Verilog*

- Start the development workstation and open project fib1
  ```
  bluespec fib1.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
  Simulate options: +bscvcd +bsccycle
  On the Editor tab:
  Set the editor to your text editor
  Save and Close the Options window
- Implement the Fibonacci circuit in FibOne.bsv. A module outline has been provided for you. The circuit should contain two registers, starting at 0 and 1, and a rule to set them so that, on the n'th clock cycle, one of the registers contains fib(n).
  *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)
- Refresh (Project -> Refresh) the Project Files window
- Examine the Verilog file. Note:
  - 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)

  *The file mkFibOne.v is displayed in the Project Options window because it is specified on the* **Files** *tab of the* **Options** *window. Use the* Display include pattern *and* Display exclude pattern *fields to control which files are displayed in the Project Options window.*

 

### *Part 1b: Produce a vcd file from a Verilog Simulation*

The flag bscvcd set in the simulate options will generate a vcd file during the simulation.

- Link (Build -> Link)
- Simulate (Build -> Simulate)
- Open the **Module Browser** window
- Load the top module (`mkFibOne`)
  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`
  - *Note the instantiation hierarchy and the enables and outputs of the registers. Also note the rule firing signals,* `WILL_FIRE_RL_rulename`.

### *Part 1c: 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:
  *Note that the* Link to *field was automatically changed to Bluesim*
  Run Options: `-m 45`
  *This tells Bluesim to run the executable for 45 clock cycles*
  The -V flag can also be added, which will have Bluesim generate a .vcd file
- Full Rebuild
  *Full rebuild will run the following tasks:*
  - Clean
  - Compile
  - Link
  - Simulate
    Compare the output with the expected results in the file mkFibOne.out.expected.

---

# Part 2: Fib2

In this lab we add a testbench to the circuit. The new module, mkFibTwo, provides an interface and has interface methods to communicate with the testbench.

### *Part 2a: Implement the Fibonacci circuit*

- Implement the Fibonacci circuit in FibTwo.bsv. A module outline (in FibTwo.bsv) and a testbench (in FibTwoTb.bsv) have been provided for you. The circuit, like FibOne, should contain two registers, starting at 0 and 1, and a rule to compute fib(n). The new bits are the interface methods, putRequest() and getReply(), which allow the module to communicate with the outside world.

 

### *Part 2b: Compile to Bluesim and Simulate*

- Compile the circuit to Bluesim
- Make sure the -V option in set in the run options field on the Link/Simulate tab
- Run the resulting executable for 1050 clock cycles
  *use the -m 1050 flag in the run options field on the Link/Simulate tab*
- Compare the output with the expected results in the file mkFibTwoTb.out.expected. Why does it take 1050 cycles to produce what took 45 cycles in FibOne?

### *Part 2c: Examine the VCD file*

*The* -V *flag tells Bluesim to generate a vcd file*

- Open the vcd file in the waveform viewer and examine the signals.

  Note, in addition to the WILL_FIRE_ruleName signals previously mentioned in FibOne, a RDY_methodName signal for each method, and an EN_methodName signal for each Action-method. Examine the WILL_FIRE signals for the rules and EN signals for the methods. Does each method/rule fire when you expect?

### *Part 2d: Compile the circuit to Verilog*

- Compile the circuit to Verilog
  *Change the Compile to target in the Options window*
  *Make sure the +bscvcd and +bsccycle flags are set*

- Examine the Verilog file mkFibTwo.v. Note:
  - how the methods are represented in the port list
  - how the method RDY_ signals are implemented
  - what the method EN_ signals control
  - what the enable and data signals for the registers are

- Examine the Verilog file mkFibTwoTb.v. Note:
  - how the device under test, mkFibTwo, is instantiated and connected
  - how the RDY_ signals from mkFibTwo's methods are used
  - how the EN_ signal to mkFibTwo's putRequest() method is connected

---

# Part 3:Fib3

In this version the Fib interface is changed slightly: the getReply() method now requires the caller to acknowledge that it is consuming the value, and getReply()'s return type becomes ActionValue#(int).

The code should implement a double-issue Fib circuit, and contain, for each of the two execution units, four registers:

- a register storing the current Fibonacci value
- a register indicating whether there is an unread answer there
- a register storing the next Fibonacci value
- a register for the count-down counter

 

The circuit should also contain two FIFOs (see cheat-sheet below for syntax) to buffer requests and responses.
Conceptually, each fib execution unit can be divided into three stages:

- load registers with request from request FIFO
- compute the Fibonacci sequence until the count has reached 0
- commit reply from registers to replies FIFO

Each stage is independent and can be implemented as a separate rule; remember to have two sets of rules to handle both execution units.

### *Part 3a: Implement the Fibonacci circuit in FibThree.bsv*

A module outline and a testbench are provided for you. The circuit is described above.

### *Part 3b: Compile and Simulate the circuit with Bluesim*

- Simulate for 575 cycles In the Options verify:
  - Top file is mkFibThreeTb
  - Top module is mkFibThreeTb
  - Target is Bluesim
  - Run options: -m 575, -V

- Compare the output with the expected results in the file mkFibThreeTb.out.expected.

- Examine the waveforms. Examine the WILL_FIRE signals for the two loading rules. What happens when both rules can fire (see CAN_FIRE signals) at the same time? Is it what you expected?
- Also note that getReply() now features an enable signal for its Action component

### *Part 3c: Compile the circuit to Verilog*

- Examine the Verilog file mkFibThree.v. Note the WILL_FIRE signals for the two load rules: how do their definitions differ? How does that relate to the compiler warning and what you saw in the VCD?

### *Part 3d: Further exploration*

In this naive implementation, the Fibonacci numbers complete in request OB order because each consecutive request takes more time. If the testbench made non-increasing requests, the replies could appear out of order with respect to the requests. What else do you need to keep track of to address this?

 

# Syntax Cheat Sheet

## *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
```

## *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
```

## *Part 3*

**Import the FIFO library at top of file:**

```
import FIFO::*;
```

**Instantiating a FIFO containing `int`s inside a module:**

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

**Instantiating a five-deep FIFO of `int`s 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();
endinterface: FIFO
```

**Instantiating a register containing a Bool:**

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

**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
```