Using RTL Models in Bluespec SystemVerilog

October 11, 2009

Existing RTL modules can be used as part of a Bluespec SystemVerilog (BSV) design. The BSV `import "BVI"` statement defines a Bluespec module wrapper for a Verilog or VHDL module, transcribing inputs and outputs into Bluespec interfaces and methods, along with scheduling annotations to fully describe the constraints of the module.

There are multiple ways to wrap any discrete RTL file depending on the model desired. This tutorial discusses how to wrap a Verilog or VHDL module for use in a BSV design and presents some design considerations along with options available to the designer. Working Bluespec source code files (`.bsv`), along with workstation project files (`.bspec`) are available for the sizedFIFO and adder examples.

Contents

1 BSV support for existing RTL

Bluespec provides the `import "BVI"` statement to wrap RTL (Verilog or VHDL) modules for inclusion in BSV designs. This allows reuse of RTL components from previous designs or generated by other tools. This technique also enables you to write custom Verilog primitives to use in multiple designs as are many of the primitives provided by Bluespec in the Azure IP Foundation libraries.

This paper describes the process and design considerations for wrapping RTL modules using the `import "BVI"` statement. The technique is the same for both Verilog and VHDL [1]; the input and output ports are transcribed to methods and interfaces. The `import "BVI"` statement does not read the RTL file; it describes how to wrap the file. Throughout this paper Verilog will be used to refer to either Verilog or VHDL. In the examples provided here the Verilog and RTL signal names are in all CAPS and start with `V_`, while the Bluespec names are in mixed case.

During simulation, the RTL file is read by the simulator. Simulating a VHDL design requires using a mixed language simulator which supports both VHDL and Verilog. The features supported will depend on the simulator. For example, not all mixed language simulators support passing parameters between languages.

2 Overview of importBVI

The `import "BVI"` statement is used to wrap a Verilog module so that it looks like a BSV module and can be used in a BSV design. The syntax of the `import "BVI"` statement is described in detail in the section *Embedding RTL in a BSV design* in the BSV Reference Guide. Instead of Verilog ports, the wrapped module will have methods and interfaces. Scheduling annotations are added to fully describe the behavior of the methods.

Below is a brief sample of the structure of a BSV wrapper containing the more common statements; not all available `import "BVI"` statements are included.

```
//module definition
import "BVI" example =
module mkExample (Integer start, Integer period)(ClockGenIfc);

//Parameters are compile-time constants into the module
     parameter initDelay = start;
     parameter v1width = period;

//Defaults are the implicit clocks and resets for the module
     default_clock clk;
     default_reset rst;

//Verilog input port connections for clocks and resets
     input_clock clk (V_CLK) <- exposeCurrentClock;
     input_reset rst (V_RST_N) clocked_by(clk)  <- exposeCurrentReset;

//Verilog output port connections for clocks provided in the module's interface
     output_clock gen_clk(V_CLK_OUT);
```

[1] One limitation for VHDL is that BSV does not support two dimensional ports.

```
//Connection of Bluespec interface methods to Verilog wires
     method write (V_D_IN) enable(V_ENQ);
     method V_D_OUT read ();

//Scheduling constraints between methods
     schedule (read_y) SB (write, clear);
     schedule (write) C (clear);
endmodule
```

2.1 Translation of Verilog

A Verilog module encompasses a set of inputs, outputs, and parameters. In BSV, these are represented by methods and interfaces. The `import "BVI"` statement defines a BSV module providing an interface whose methods encapsulate the possible transactions clients can perform and connects the Verilog input and output ports to BSV objects.

In BSV, the declaration of an interface is separate from the module definition, allowing the development of common interfaces that can be provided by several modules without having to repeat the interface declaration with each module. When translated into Verilog, each method becomes a bundle of wires, thus requiring all interface arguments and return values to be types which are convertible to wires.

2.1.1 Parameters

Verilog parameters are compile-time constants into the module. Verilog parameters translate into BSV parameters. There is no inherent relationship between the BSV module's parameters and the Verilog module's parameters; the designer specifies in the `parameter` statement the BSV expressions for each Verilog parameter.

Verilog parameters translate to:

- BSV module parameters - arguments to the module
- An expression using or containing a module parameter
- An expression using or containing type information from the interface

2.1.2 Inputs

Verilog inputs translate to:

- Clock inputs
- Reset inputs
- Enables for `Action` or `ActionValue` methods
- Arguments for any type method - `Action`, `ActionValue`, or `Value`
- Ports

The BSV `port` statement is analogous to arguments passed in to a BSV module, but it is rarely used since the BSV style is to interact and pass arguments through methods. Verilog ports are usually connected to BSV methods. When used, BSV ports may be dynamic, differentiating them from `parameter` statements, which must be compile-time constants.

2.1.3 Outputs

Verilog outputs translate to:

- Return values from `ActionValue` or `Value` methods
- Ready signals for methods
- Clock outputs
- Reset outputs

2.1.4 Inouts

Verilog Inouts are translated to BSV Inouts.

2.2 importBVI Wizard

The Bluespec development workstation provides the **importBVI Wizard** to help write the `import "BVI"` statement. This option is accessed from the **Tools** menu. The wizard proceeds through the following six steps:

1. Verilog Module Overview: Review the Verilog parameters, inputs, outputs, and inouts
2. Bluespec Module Definition: Create the module header for the `import BVI` statement
3. Method Port Binding: Bind methods, ports, and subinterfaces to Verilog inputs and outputs
4. Combinational Paths: Add the `path` statements
5. Scheduling Annotations: Add the `schedule` statements
6. Finish: Review, compile, and save the BSV wrapper.

With each step, as details are added to the `import "BVI"` statement, you can check the information entered, as well as display the BSV statement. The importBVI Wizard is described in the *Workstation Tools* section of the BSV User Guide.

The workstation can only read in Verilog files; it cannot read in VHDL files. You can still use the **importBVI Wizard** for VHDL, but you cannot use the **Read Verilog** task button in the first step. You can use **Read From File** or manually enter the VHDL inputs and outputs.

2.3 Verilog sizedFIFO Examples

This section contains two examples of import statements for the same sizedFIFO Verilog file. The first FIFO example provides the `FIFOF` interface while the second defines a new interface, based on `Get` and `Put` interfaces. By comparing the two examples, you can see some of the different options you have for implementing the Verilog representation in BSV. For example, the FIFO example connects the `V_CLR` pin to a `clear` method, while the `GetPut` example ties off the `V_CLR` pin.

Below is an excerpt from the Verilog file `SizedFIFO.v` provided with this tutorial. Note that in all examples the Verilog and RTL signal names are in all CAPS, while the Bluespec names are in mixed case.

```
module SizedFIFO(V_CLK, V_RST_N, V_D_IN, V_ENQ, V_FULL_N, V_D_OUT, V_DEQ, V_EMPTY_N, V_CLR);
   parameter                  V_P1WIDTH = 1; // data width
   parameter                  V_P2DEPTH = 3;
   parameter                  V_P3CNTR_WIDTH = 1; // log(V_P2DEPTH-1)
   // The -1 is allowed since this model has a fast output register
   parameter                  V_GUARDED = 1;

   input                      V_CLK;
   input                      V_RST_N;
   input                      V_CLR;
   input [V_P1WIDTH - 1 : 0]  V_D_IN;
   input                      V_ENQ;
   input                      V_DEQ;

   output                     V_FULL_N;
   output                     V_EMPTY_N;
   output [V_P1WIDTH - 1 : 0] V_D_OUT;
```

2.3.1 Existing `FIFOF` interface

This first example wraps the Verilog file `SizedFIFO.v` with a BSV module providing the `FIFOF` interface. The definition of the `FIFOF`, from the `FIFOF` package, is:

```
interface FIFOF #(type a) ;
    method Action enq(a x1) ;
    method Action deq() ;
    method a first() ;
    method Action clear() ;
    method Bool notFull() ;
    method Bool notEmpty() ;
endinterface: FIFOF
```

The following table shows the Verilog parameters, inputs, and outputs along with their BSV translations to the `FIFOF` methods:

Translation of Verilog into BSV SizedFIFO module providing FIFOF interface			
Verilog	Bluespec		
Statement	Type	Value	Description
parameter V_P1WIDTH = 1	parameter	valueOf(size_a)	function based on data type
parameter V_P2DEPTH = 3	parameter	depth	argument to module
parameter V_P3CNTR_WIDTH = 1	parameter	log2(depth+1)	expression containing parameter
parameter V_GUARDED=1	parameter	Bit#(1)'(pack(g))	argument to module
input V_CLK	Clock	clk	default clock
input V_RST_N	Reset	rst_RST_N	default reset
input V_CLR	Action	clear	enable for clear
input [V_P1WIDTH - 1:0] D_IN	Action	enq	argument for enq
input V_ENQ			enable for enq
output V_FULL_N	Value	notFull	ready for enq
output [V_P1WIDTH - 1:0] V_D_OUT	Value	first	output value
input V_DEQ	Action	deq	enable for deq
output V_EMPTY_N	Value	notEmpty	ready for deq and first

The import "BVI" statement defines a module named mkSizedFIFO, which has two arguments, depth and g, and provides the FIFOF interface. The method statements shown below connects the methods in the FIFOF interface to the appropriate Verilog wires. The *inhigh* attribute on an enable indicates that the method is always_enabled. See Section 3 for a more detailed discussion of the clock and reset statements. The source code for this example is in the file SizedFIFO1.bsv.

```
import "BVI" SizedFIFO =
module mkSizedFIFO #(Integer depth, Bool guard) (FIFOF#(a))
        provisos(Bits#(a,size_a));

        parameter V_P1WIDTH = valueOf(size_a);
        parameter V_P2DEPTH = depth;
        parameter V_P3CNTR_WIDTH = log2(depth+1);
        parameter V_GUARDED = Bit#(1)'(pack(guard));

        default_clock clk;
        default_reset rst_RST_N;

        input_clock clk (V_CLK)  <- exposeCurrentClock;
        input_reset rst_RST_N (V_RST_N) clocked_by(clk)  <- exposeCurrentReset;

        method enq (V_D_IN) enable(V_ENQ) ready(V_FULL_N);
        method deq () enable(V_DEQ) ready(V_EMPTY_N);
        method V_D_OUT first () ready(V_EMPTY_N);
        method V_FULL_N notFull ();
        method V_EMPTY_N notEmpty ();
        method clear () enable (V_CLR);
```

2.3.2 User-defined GetPut interface

In this example, instead of using the `FIFOF` interface as above, we're defining a new interface, `MyGetPut`. The interface has two subinterfaces: a `Get#(a)` interface and a `Put#(a)` interface. The only methods defined will be a `get` method and a `put` method. In this example the Verilog `V_CLR` is tied off, so there is no `clear` method defined.

The `MyGetPut` interface declaration is:

```
interface MyGetPut#(type t);
   interface Get#(t) g;
   interface Put#(t) p;
endinterface
```

Translation of Verilog into BSV SizedFIFO module using GetPut Interface			
Verilog	Bluespec		
Statement	Type	Value	Description
`parameter V_P1WIDTH = 1`	parameter	`valueOf(size_a)`	function based on data type
`parameter V_P2DEPTH = 3`	parameter	`depth`	argument to module
`parameter V_P3CNTR_WIDTH = 1`	parameter	`log2(depth+1)`	expression containing parameter
`parameter V_GUARDED=1`	parameter	`Bit#(1)'(pack(guard))`	argument to module
`input V_CLK`	Clock	`clk`	default clock
`input V_RST_N`	Reset	`rst_RST_N`	default reset
`input V_CLR`	port	`0`	tied off
`input [V_P1WIDTH - 1:0] D_IN`	Action	`put`	argument for `p_put`
`input V_ENQ`			enable for `p_put`
`output V_FULL_N`			ready for `p_put`
`output [V_P1WIDTH - 1:0] V_D_OUT`	Value	`g_get`	output value
`input V_DEQ`			enable for `g_get`
`output V_EMPTY_N`			ready for `g_get`

The source file for this example is contained in `SizedFIFO2.bsv`.

```
import GETPUT :: * ;

import "BVI" SizedFIFO =
module mkSizedFIFO #(Integer depth, Bool guard) (MyGetPut#(a))
        provisos(Bits#(a, size_a));

        parameter V_P1WIDTH = valueOf(size_a);
        parameter V_P2DEPTH = depth;
        parameter V_P3CNTR_WIDTH = log2(depth+1);
        parameter V_GUARDED = Bit#(1)'(pack(guard));
```

```
        port V_CLR = 0 ;

        default_clock clk;
        default_reset rst_RST_N;

        input_clock clk (V_CLK)  <- exposeCurrentClock;
        input_reset rst_RST_N (V_RST_N) clocked_by(clk)  <- exposeCurrentReset;

        interface Get g;
            method V_D_OUT get () enable(V_DEQ) ready(V_EMPTY_N);
        endinterface

        interface Put p;
            method put(V_D_IN) enable(V_ENQ) ready(V_FULL_N);
        endinterface
endmodule
```

2.4 VHDL Examples

This section describes two examples of import statements for VHDL files. The first example is for a sized FIFO using the `FIFOF` interface. This example is similar to the Verilog example in Section 2.3.1 . The second example describes importing a VHDL file which contains multi-dimensional arrays in the interface, a structure not supported by BSV, Verilog, or mixed-mode simulators.

2.4.1 VHDL FIFO Example

The technique for importing VHDL is the same as that for Verilog; a BSV wrapper is written using the `import "BVI"` statement. The ports are transcribed to methods and interfaces and the generic parameters translate into the BSV module's parameters. One restriction on the VHDL file is that BSV does not support multi-dimensional ports. See 2.4.2 for an example importing a VHDL file with multi-dimensional ports.

The **ImportBVI Wizard** provided in the Bluespec development workstation cannot read in VHDL files, therefore you cannot use the **Read Verilog** task. You can write an input file describing the VHDL or manually enter the VHDL inputs and outputs, and then proceed through the remainder of the wizard.

To simulate a VHDL design you must use a mixed-mode simulator which supports both VHDL and Verilog. The features supported will depend on the simulator.

The following VHDL excerpt provides the same functionality as the Verilog `SizedFIFO.v` example above.

```
ENTITY SizedFIFO IS
   GENERIC(V_P1WIDTH      : NATURAL RANGE 1 TO 32 := 16;
           V_P2DEPTH      : NATURAL RANGE 1 TO 32 := 4;
           V_P3CNTR_WIDTH : NATURAL RANGE 1 TO 32 := 1;
           V_GUARDED      : BOOLEAN := TRUE);

   PORT(V_CLK                  : IN  std_logic;
```

```
        V_RST_N                 : IN  std_logic;
        V_CLR                   : IN  std_logic;
        V_D_IN                  : IN  std_logic_vector(V_P1WIDTH-1 DOWNTO 0);
        V_ENQ                   : IN  std_logic;
        V_DEQ                   : IN  std_logic;
        V_FULL_N                : OUT std_logic;
        V_EMPTY_N               : OUT std_logic;
        V_D_OUT_N               : OUT std_logic_vector(V_P1WIDTH-1 DOWNTO 0));
END SizedFIFO;
```

The following table shows the VHDL parameters, inputs, and outputs along with their BSV translations using the `FIFOF` interface and methods, as shown in Section 2.3.1:

Translation of VHDL into BSV SizedFIFO module providing `FIFOF` interface			
VHDL	Bluespec		
Statement	Type	Value	Description
`Generic V_P1WIDTH`	parameter	`valueOf(size_a)`	function based on data type
`Generic V_P2DEPTH = 3`	parameter	`depth`	argument to module
`Generic V_P3CNTR_WIDTH = 1`	parameter	`log2(depth+1)`	expression containing parameter
`Generic V_GUARDED=TRUE`	parameter	`Bit#(1)'(pack(g))`	argument to module
`Port V_CLK`	Clock	`clk`	default clock
`Port V_RST_N`	Reset	`rst_RST_N`	default reset
`Port V_CLR`	Action	`clear`	enable for `clear`
`Port D_IN`	Action	`enq`	argument for `enq`
`Port V_ENQ`			enable for `enq`
`Port V_DEQ`	Action	`deq`	enable for `deq`
`Port V_FULL_N`	Value	`notFull`	ready for `enq`
`Port V_EMPTY_N`	Value	`notEmpty`	ready for `deq` and `first`
`Port V_D_OUT_N`	Value	`first`	output value

The `import "BVI"` statement defines a module named `mkSizedFIFO`, which has two argument, `depth` and `g`, and provides the `FIFOF` interface. The `method` statements shown below connects the methods in the `FIFOF` interface to the appropriate VHDL wires. The `*inhigh*` attribute on an enable indicates that the method is `always_enabled`. The `import "BVI"` wrapper for the VHDL file shown below is identical to the one created in Section 2.3.1 for the Verilog file.

```
import "BVI" SizedFIFO =
module mkSizedFIFO #(Integer depth, Bool g) (FIFOF#(a))
        provisos(Bits#(a,size_a));

        parameter V_P1WIDTH = valueOf(size_a);
        parameter V_P2DEPTH = depth;
        parameter V_P3CNTR_WIDTH = log2(depth+1);
        parameter V_GUARDED = Bit#(1)(pack(g));

        default_clock clk;
```

```
default_reset rst_RST_N;

input_clock clk (V_CLK)  <- exposeCurrentClock;
input_reset rst_RST_N (V_RST_N) clocked_by(clk)  <- exposeCurrentReset;

method enq (V_D_IN) enable(V_ENQ) ready(V_FULL_N);
method deq () enable(V_DEQ) ready(V_EMPTY_N);
method V_D_OUT first () ready(V_EMPTY_N);
method V_FULL_N notFull ();
method V_EMPTY_N notEmpty ();
method clear () enable (V_CLR);
```

2.4.2 VHDL Array Example

VHDL allows multi-dimensional arrays in interfaces; BSV, Verilog, and mixed-mode simulators do not. To import a VHDL file containing interface arrays you create a wrapper VHDL module that eliminates the arrays by converting them to vectors of standard bits (`STD_LOGIC_VECTOR`). The `import "BVI"` statement imports the VHDL wrapper. Both VHDL files (the original and the wrapper) are required to simulate the design.

The following example contains 2 arrays; `V_ARRAY_BIT2` is an array of 2-bit elements, `V_ARRAY_BIT4` is an array of 4-bit elements. Each array has `V_COUNT` number of elements.

These statements are included in the VHDL wrapper to convert the arrays into vectors of bits. The complete VHDL entity definition, VHDL wrapper, and `import "BVI"` statements follow.

1. The VHDL arrays in the interface (the `PORT` statements) are defined as vectors of standard logic bits (`STD_LOGIC_VECTOR`). Example:

   ```
   V_INPUT_2BIT  :   IN  V_ARRAY_BIT2(V_COUNT-1 DOWNTO 0)
   ```

   is redefined in the wrapper as:

   ```
    V_INPUT_2BIT  :  IN  STD_LOGIC_VECTOR(2*V_COUNT-1 DOWNTO 0);
   ```

2. The internal signals are defined as arrays. Example:

   ```
   SIGNAL V_INPUT_2BIT_ARRAY   : V_ARRAY_2BIT(V_COUNT-1 DOWNTO 0);
   ```

3. The wrapper instantiates the component from the original VHDL file. Example:

   ```
   V_U_EXAMPLE1 : V_EXAMPLE1
   ```

4. The array signals are connected to the vector inputs and outputs. Example:

   ```
   PORT MAP (
       V_INPUT_2BIT   =>  V_INPUT_2BIT_ARRAY,
   ```

5. The arrays are processed through generate statements. Example:

```
         g1:  FOR i IN 0 TO (V_COUNT-1) GENERATE
            V_INPUT_2BIT_ARRAY(i)    <=  V_INPUT_2BIT ( 2*(i+1)-1 DOWNTO  2*i);
```

VHDL Entity Definition

The interface of the entity `EXAMPLE1` is defined as:

```
ENTITY V_EXAMPLE1 IS
    GENERIC (
        V_COUNT : NATURAL RANGE 1 TO 16 := 8 );
    PORT(
        V_INPUT1         : IN  STD_LOGIC;
        V_INPUT2         : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        V_INPUT_2BIT     : IN  V_ARRAY_BIT2(V_COUNT-1 DOWNTO 0);
        V_INPUT_4BIT     : IN  V_ARRAY_BIT4(V_COUNT-1 DOWNTO 0);
        V_OUTPUT1        : OUT STD_LOGIC;
        );
END V_EXAMPLE1;
```

VHDL Wrapper

```
ENTITY V_EXAMPLE1_WRAP IS
    GENERIC (
        V_COUNT : NATURAL RANGE 1 TO 16 := 8 );
    PORT(
        V_INPUT1              : IN  STD_LOGIC;
        V_INPUT2              : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        -- 1. Arrays are redefined as vectors
        V_INPUT_2BIT          : IN  STD_LOGIC_VECTOR(2*V_COUNT-1 DOWNTO 0);
        V_INPUT_4BIT          : IN  STD_LOGIC_VECTOR(4*V_COUNT-1 DOWNTO 0);
        V_OUTPUT1             : OUT STD_LOGIC;
        );
END V_EXAMPLE1_WRAP;

ARCHITECTURE synth OF V_EXAMPLE1_WRAP IS
  -- 2. Internal signals defined as arrays defined in original file
  SIGNAL V_INPUT_2BIT_ARRAY   : V_ARRAY_BIT2(V_COUNT-1 DOWNTO 0);
  SIGNAL V_INPUT_4BIT_ARRAY   : V_ARRAY_BIT4(V_COUNT-1 DOWNTO 0);

BEGIN
  -- 3. Instantiate the component from the original file
  V_U_EXAMPLE1 : V_EXAMPLE1
    GENERIC MAP (
      V_COUNT        => V_COUNT)

    PORT MAP (
        V_INPUT1           => V_INPUT1,
        V_INPUT2           => V_INPUT2,
  -- 4. Connect the arrays to the input ports
        V_INPUT_2BIT       => V_INPUT_2BIT_ARRAY,
        V_INPUT_4BIT       => V_INPUT_4BIT_ARRAY,
        V_OUTPUT1          => V_OUTPUT1)
  -- 5. Assign the correct values in the array to the inputs
   g1:  FOR i IN 0 TO (V_COUNT-1) GENERATE
```

```
      V_INPUT_2BIT_ARRAY(i)    <= V_INPUT_2BIT( 2*(i+1)-1 DOWNTO  2*i);
      V_INPUT_4BIT_ARRAY(i)    <= V_INPUT_4BIT( 4*(i+1)-1 DOWNTO  4*i);
  END GENERATE;
END synth;
```

BSV Wrapper

The BSV wrapper includes both the interface definition and the `import "BVI"` statement.

```
interface ExampleBVIIfc#(numeric type number);
   method Action input1 (Bool value);
   method Action input2 (Bit#(3) value);
   method Action input_2bit (Vector#(number, Bit#(2)) value);
   method Action input_4bit (Vector#(number, Bit#(4)) value);
   method Bool output1;
endinterface

import "BVI" V_EXAMPLE1_WRAP =
module mkExampleBVI (ExampleBVIIfc#(number));
   let i = valueOf(number);

   parameter V_COUNT        = i;

   method input1           (V_INPUT1)     enable((*inhigh*)ignore00);
   method input2           (V_INPUT2)     enable((*inhigh*)ignore01);
   method input_2bit       (V_INPUT_2BIT) enable((*inhigh*)ignore02);
   method input_4bit       (V_INPUT_4BIT) enable((*inhigh*)ignore03);
   method V_OUTPUT1 output1;
 endmodule
```

3 Clocks and Resets

The following statements are used to connect clock and reset ports to module clocks and resets, and to associate methods with clock and reset signals:

- `input_clock`
- `default_clock`
- `output_clock`
- `input_reset`
- `default_reset`
- `output_reset`

The `input_clock` and `input_reset` statements specify the port connections for the clocks and resets into the module; they correspond to Verilog input ports. The `output_clock` and `output_reset` specify the port connections for clocks and resets provided by the module; they correspond to the Verilog outputs. The default clock and reset specify the implicit clock and reset used by methods and submodules without

`clocked_by` and `reset_by` statements. Refer to the section *Embedding RTL in a BSV design* in the BSV Reference Guide for the complete explanation of these statements.

A module with multiple input clocks defines a clock domain crossing; the methods in different domains are clocked by different clock, These modules allow the safe crossing of signals and data from one clock domain (the source domain) to another (the destination domain).

Some rules to remember when writing `import "BVI"` clock statements:

- Every `import "BVI"` wrapper must specify a default clock and a default reset. They do not have to be connected to Verilog ports and can be defined as `no_clock` or `no_reset`.
- An input clock/input reset statement is required for each RTL input clock/reset port which is not the default.
- For default clocks and resets which are connected to RTL ports, the input and default statements can be combined into a single statement, which is a `default_clock` statement.
- Every `Action` or `ActionValue` method must be associated with a clock. The default clock will be used unless there is a `clocked_by` clause in the method statement. If the default clock is defined as `no_clock`, the method statement must include a `clocked_by` clause specifying an input clock.

3.1 Clocks

Let's look at the clock statements from the SizedFIFO example (Section 2.3.1):

```
default_clock clk;
input_clock clk (V_CLK)  <- exposeCurrentClock;
```

The first statement declares the BSV clock named `clk` to be the default clock, the clock used by methods that don't contain an explicit `clocked_by` clause. The second statement defines the input clock named `clk`, connecting the Verilog port (`V_CLK`) to the the current clock of the module.[2] The two statements can be combined into a single statement defining the clock and declaring it to be the default:

```
default_clock clk (V_CLK) <- exposeCurrentClock;
```

The name `clk` is the BSV name of the clock, used in `clocked_by` statements. This name applies only within the `import "BVI"` context.

A clock can be connected to two Verilog ports, the first port name within the parentheses is the oscillator, the second port name is the gate. If there is only one port name, it is the oscillator and the gate port is unconnected and defaults to `unused`. Since in this example there is only one Verilog port (`V_CLK`), it is the name of the oscillator and there is no gate port.

If you wanted to set the value of the clock `clk` to something other than the current clock (for example a module argument of type clock, named `dclock`) the statement above would be:

```
default_clock clk (V_CLK) = dclock;
```

[2]The function `exposeCurrentClock` returns the current clock of the module

The default clock does not have to be connected to a Verilog port, in which case the parentheses after the clock name is left empty. For example, the `default_clock` statement for a BSV clock named `clk` that does not have a Verilog port connection would be:

```
default_clock clk();
```

If you do not want the module to have a default clock, forcing the designer to explicitly state which clock each method is clocked by, the default clock statement is still required, and would be:

```
default_clock no_clock;
```

3.2 Resets

Let's look at the reset statements from the SizedFIFO example (Section 2.3.1):

```
default_reset rst_RST_N;
input_reset rst_RST_N (V_RST_N) clocked_by(clk)  <- exposeCurrentReset;
```

The first statement declares the BSV clock named `rst_RST_N` to be the default reset for the module. The default is the reset used by methods that don't have an explicit `reset_by` clause. The second statement defines the clock named `rst_RST_N`, connecting the Verilog reset pin (`V_RST_N`) to the current reset of the module.[3] The `clocked_by` statement indicates that the reset is in the `clk` domain. Since this is the default clock for the module, the `clocked_by` statement could be omitted.

The two statements can be combined into a single statement defining the reset and declaring it to be the default:

```
default_reset rst_RST_N (V_RST_N) clocked_by (clk) <- exposeCurrentReset;
```

A reset can be defined which is not connected to a Verilog port, in which case the parentheses are left blank:

```
input_reset rst_RST_N () <- exposeCurrentReset;
```

The default reset may be unused or not connected to a port, but it must still be declared:

```
default_reset no_reset;
```

or the `default_reset` keyword can be omitted:

```
no_reset;
```

[3] The function `exposeCurrentReset` returns the current reset of the module

3.3 Examples

3.3.1 Combinational module - no clocks

The `RWire` primitive in the Prelude library is an example of a wrapper for a combinational module, one containing no state elements. There are no `input_clock` statements in this example; the clock is not connected to a Verilog port. A default clock and a default reset must still be defined and named (it cannot be `no_clock`) since there is an `Action` method in the module. `Action` methods must be have a clock to prevent random clock crossings; they cannot be clocked by `no_clock`.

```
import "BVI" RWire =
   module vMkRWire (VRWire#(a))
      provisos (Bits#(a,sa));
      parameter width = valueOf(sa);

      default_clock clk();  //() indicates no Verilog clock port
      default_reset rst();  //() indicates no Verilog reset port

      method wset(V_WVAL) enable(V_WSET);  //clocked by default, clk
      method V_WHAS whas ;
      method V_WGET wget ;

      schedule (wget, whas) CF (wget, whas) ;
      schedule wset SB (wget, whas) ;
      schedule wset C wset ;

      path (V_WSET, V_WHAS) ;
      path (V_WVAL, V_WGET) ;
   endmodule: vMkRWire
```

3.3.2 Domain crossing with 2 input clocks, no default

Let's take a look at an `import "BVI"` statement which has two clocks and two resets. In this example the default clock is defined as `no_clock`, forcing the designer to explicitly specify which clock is associated with each method. This approach helps minimize mistakes.

The Verilog clock ports into the module are `V_CLKA` and `V_CLKB`. The resets are not connected to any Verilog ports.

```
import "BVI" BRAM2 =
module vSyncBRAM2#(Integer memSize,
                   Bool hasOutputRegister,
                   Clock clkA, Reset rstNA, Clock clkB, Reset rstNB
                    ) (BRAM_DUAL_PORT#(addr, data))
   provisos(Bits#(addr, addr_sz),
            Bits#(data, data_sz));
```

```
   default_clock   no_clock;
   default_reset   no_reset;

   input_clock     clkA(V_CLKA, (*unused*) V_CLK_GATEA) = clkA;
   input_clock     clkB(V_CLKB, (*unused*) V_CLK_GATEB) = clkB;

   input_reset     rstA() = rstNA;
   input_reset     rstB() = rstNB;

   interface BRAM_PORT a;
     method put(V_WEA, (*reg*)V_ADDRA, (*reg*)V_DIA) enable(V_ENA)
           clocked_by(clkA) reset_by(rstA);
     method V_DOA read() clocked_by(clkA) reset_by(rstA);
   endinterface: a

   interface BRAM_PORT b;
     method put(V_WEB, (*reg*)V_ADDRB, (*reg*)V_DIB) enable(V_ENB)
           clocked_by(clkB) reset_by(rstB);
     method V_DOB read() clocked_by(clkB) reset_by(rstB);
   endinterface: b
endmodule: vSyncBRAM2
```

The module in this example has two BSV clocks (`clkA` and `clkB`) and two BSV resets (`rstNA` and `rstNB`). The clock oscillators are connected to Verilog ports `V_CLKA` and `V_CLKB`. The resets are not connected to any Verilog ports. The default clock and the default reset are defined as `no_clock` and `no_reset`, making it necessary to use `clocked_by` and `reset_by` clauses to associate methods with the correct clock and reset. Every `Action` method must have a clock, so if the default clock is `no_clock`, the method must be explicitly associated with a clock through a `clocked_by` clause. Since the `read` methods are value methods they do not have to be clocked, but are in this example.

Another interesting aspect of this example is the two interfaces defined within the `import "BVI"` statement. Since a module can only provide a single interface (`BRAM_DUAL_PORT` in this example), when there are multiple interfaces provided, they are written as subinterfaces in the import statement body. An `import "BVI"` statement can have a hierarchy of interfaces. The definition of the interface provided in this example is:

```
    interface BRAM_DUAL_PORT#(type addr, type data);
       interface BRAM_PORT#(addr, data) a;
       interface BRAM_PORT#(addr, data) b;
    endinterface: BRAM_DUAL_PORT
```

3.3.3 Domain crossing with 2 input clocks, one defined as default

Let's take a look at another `import "BVI"` statement with two clocks and two resets. This is the same example as shown above, but in this case one of the clocks, `clkA`, has been declared the default clock.

The method statements in interface `a` don't require `clocked_by` or `reset_by` clauses, since they are in the default clock domain. They will be clocked by `clkA` and reset by `rstA`. The method statements in interface `b` require explicit clocking statements.

```
import "BVI" BRAM2 =
module vSyncBRAM2#(Integer memSize,
                   Bool hasOutputRegister,
                   Clock clkA, Reset rstNA, Clock clkB, Reset rstNB
                    ) (BRAM_DUAL_PORT#(addr, data))
   provisos(Bits#(addr, addr_sz),
            Bits#(data, data_sz));


   input_clock     clkA(V_CLKA, (*unused*) V_CLK_GATEA) = clkA;
   input_clock     clkB(V_CLKB, (*unused*) V_CLK_GATEB) = clkB;

   input_reset     rstA() = rstNA;
   input_reset     rstB() = rstNB;

   default_clock   clkA;
   default_reset   rstA;

   interface BRAM_PORT a;
     method put(V_WEA, (*reg*)V_ADDRA, (*reg*)DIA) enable(ENA);
     method DOA read();
   endinterface: a

   interface BRAM_PORT b;
     method put(WEB, (*reg*)V_ADDRB, (*reg*)V_DIB) enable(V_ENB)
                clocked_by(clkB) reset_by(rstB);
     method V_DOB read() clocked_by(clkB) reset_by(rstB);
   endinterface: b
endmodule: vSyncBRAM2
```

You can combine the `default_clock` and `input_clock` statements into a single statement. The above clock and reset statements could be rewritten, combining the input and default statements for `clkA` and `rstA`.

```
   //definition and declarations of clkA and rstA
   default_clock   clkA(V_CLKA, (*unused*) V_CLK_GATEA) = clkA;
   default_reset   rstA() = rstNA;
   //definition of clkB and rstB
   input_clock     clkB(V_CLKB, (*unused*) V_CLK_GATEB) = clkB;
   input_reset     rstB() = rstNB;
```

3.3.4 Domain crossing with one input clock, defined as default

This next example has two clock domains, but only one clock defined in the wrapper. This example is a dual port Ram, an asynchronous RAM providing a domain crossing by having its read and write methods in separate clock domains. The default BSV clock, `clk`, is connected to Verilog input ports, `V_CLK` for the oscillator and `V_CLK_GATE` for the gate.

The `write` method is clocked by the default clock and the default reset, making the `clocked_by` and `reset_by` clauses unnecessary. But, as shown, they can still be provided for clarity.

The `read` method is clocked by `no_clock`. The clock that the destination (`read`) domain is associated with is determined by the module where the method is used. There does not have to be a specific clock associated with the method since it is a value method.

```
import "BVI" DualPortRam =
module mkDualRam( DualPortRamIfc #(addr_t, data_t) )
   provisos ( Bits#(addr_t,sa),
              Bits#(data_t,da) ) ;

   parameter addrWidth = valueOf(sa);
   parameter dataWidth = valueOf(da);

   default_clock clk(V_CLK, (*unused*)V_CLK_GATE) ;
   no_reset ;

   method write(V_WADDR, V_DIN) enable(WE) clocked_by( clk )  reset_by( no_reset ) ;
   method V_DOUT read(V_RADDR)  clocked_by( no_clock ) reset_by( no_reset ) ;

   // read and write are independant
      schedule write CF read ;
      schedule read  CF read ;
      schedule write C write ;
endmodule
```

3.3.5 Domain crossing with 2 input clocks, 1 input reset

This example defines two input clocks, connected to the Verilog input ports `V_SCLK` and `V_DCLK` for the oscillators and `V_SCLK_GATE` and `V_DCLK_GATE` for the gates. There is a single reset connected to the Verilog input port, `V_SRST_N`, for the source domain. The destination domain reset is not connected to a Verilog port; it is reset by `no_reset`. In this example, the defaults are declared as `no_clock` and `no_reset`, requiring `clocked_by` clauses in the method statements.

```
import "BVI" SyncBit =
module vSyncBit( Clock sClkIn, Reset sRstIn,
                 Clock dClkIn,
                SyncBitIfc#(one_bit_type) ifc )
      provisos( Bits#(one_bit_type, 1)) ;

   default_clock no_clock ;
   no_reset ;

   parameter init =  0;

   input_clock clk_src( V_SCLK, (*unused*)V_SCLK_GATE ) = sClkIn ;
```

```
   input_clock clk_dst( V_DCLK, (*unused*)V_DCLK_GATE ) = dClkIn ;

   input_reset (V_SRST_N) clocked_by (clk_src) = sRstIn ;

   method          send (V_SD_IN) enable(V_SEN) clocked_by (clk_src) reset_by(sRstIn);
   method V_DD_OUT read()                       clocked_by (clk_dst) reset_by(no_reset);

      schedule read CF read ;
      schedule read CF send ;
      schedule send C  send ;
endmodule
```

4 Combinational Circuit Example - Adder

This section presents a Verilog Adder module, along with multiple ways to implement the Adder in BSV, both through the BVI wrapper and the instantiation of the wrapper in a BSV module. Both bluespec source files (`.bsv`) and bluespec workstation files (`.bspec`) are provided for these examples.

The Verilog Adder module is:

```
module adder (A, B, X);
    parameter W = 1;            // Data width
    input [W-1:0] A, B;
    output [W-1:0] X;
```

There are no clocks in the Adder module, so there are no `input_clock` statements in the module wrappers.

The same Verilog module can be wrapped in different ways in BSV, providing different interfaces as well as different clocking structures. In this section we show different BVI wrappers, providing either the interface `Adder1` or the interface `Adder2`. `Adder1`, has a single `Value` method, which takes the input arguments `a` and `b` and returns the sum, `t`. `Adder2` has two methods: an `Action` method `start` for the inputs, and a `Value` method `result` for the output. The `Addr2` implementation allows the input and the output to be in different clock domains.

This section includes the following examples:

- Adder1 Interface - single interface method
  - Method in wrapper clocked by default clock, instantiated in default clock domain (4.1.1)
  - Method in wrapper clocked by default clock, instantiated in non-default clock domain. (4.1.2)
  - Method in wrapper clocked by `no_clock`, instantiated in default clock domain (4.2.1)
  - Method in wrapper clocked by `no_clock`, instantiated in non-default clock domain (4.2.2)
- Adder2 Interface - two interface methods
  - Both methods clocked by default clock in the wrapper (4.3.1)
  - Output method clocked by `no_clock` in the wrapper (4.3.2)

Adder1 Interface

Since the interface has a single method, the inputs and outputs of the Adder will always be in the same clock domain.

Adder1 methods		
Name	Type	Description
`add`	Value	Adds together the values `a` and `b`, returning the sum, `t`.

```
interface Adder1#(type t);
   method t add(t a, t b);
endinterface
```

Adder2 Interface

With this implementation, you must be careful to be explicit with your clocks, as the methods of the Adder could be in different clock domains. This is not considered a safe clock domain crossing.

Adder2 methods		
Name	Type	Description
`start`	Action	Reads in the values `a` and `b`.
`result`	Value	Outputs the sum, `t`.

```
interface Adder2#(type t);
   method Action start(t a, t b);
   method t result();
endinterface
```

4.1 Single wrapper method clocked_by default clock

In this example, there are no clock connections and the method is clocked by the default clock, `clk`. When instantiated, the clock will be whatever the current clock of the module is.

The `add` method should only be called once in a cycle, so the schedule statement specifies that the `add` method conflicts (`C`) with itself.

```
import "BVI" adder =
module mkAdder1 (Adder1#(t))
   provisos ( Bits#(t, st) );
   // Should have other provisos to limit type which can be added directly
   parameter W = valueOf(st);

   default_clock clk();         // no clock connections
   default_reset rst();         // no reset connections

   method X add (A, B);

   // Add can only be call once in a cycle
   schedule add C add;
endmodule
```

4.1.1 Instantiated in default clock domain

To use the Adder, you can instantiate it in a BSV module and return the result. This method is in the default clock domain of the module.

```
(*synthesize*)
module sysTest1A ( Empty ifc );
   Reg#(UInt#(32)) a_c1 <- mkReg(0);
   Reg#(UInt#(32)) b_c1 <- mkReg(0);
   Reg#(UInt#(32)) x_c1 <- mkReg(0);

   Adder1#(UInt#(32))   adder <- mkAdder1;

   rule doit_c1 (True);
      x_c1 <= adder.add(a_c1, b_c1);
   endrule
endmodule
```

4.1.2 Instantiated in a different clock domain

The compiler will prevent you from instantiating across clock domains. In this example, the inputs `a_c1` and `b_c1` are in the default clock domain. But the `adder.add` method is in the `clkb` domain.

```
(*synthesize*)
module sysTest1B (Clock clkb, Reset rstb, Empty ifc );
   Reg#(UInt#(32)) a_c1 <- mkReg(0);
   Reg#(UInt#(32)) b_c1 <- mkReg(0);
   Reg#(UInt#(32)) x_c1 <- mkReg(0);

   Adder1#(UInt#(32))   adder <- mkAdder1A (clocked_by clkb);

   rule doit_c1 (True);
      x_c1 <= adder.add(a_c1, b_c1);
   endrule
endmodule
```

The compiler will generate these errors when you attempt to compile this example:

```
Error: "adder412.bsv", line 36, column 9: (G0007)
  Reference across clock domain in rule ‘doit_c1’.
  Method calls by clock domain:
    Clock domain 1:
      x_c1.write a_c1.read b_c1.read
    Clock domain 2:
      adder.add at "adder412.bsv", line 22, column 13,
Warning: "adder412.bsv", line 36, column 9: (G0043)
  Multiple reset signals influence rule ‘doit_c1’.
```

```
  This can lead to inconsistent, non-atomic results when not all of these
  signals are asserted.
  Method calls by reset:
    Reset 1:
      x_c1.write a_c1.read b_c1.read
    Reset 2:
      adder.add at "adder412.bsv", line 22, column 13,
```

4.2 Single wrapper method clocked by `no_clock`

In this example the method `add` is clocked by `(no_clock)` instead of the default clock, `clk`. This allows you to instantiate a method in a different clock domain, though the compiler will prevent you from calling a method across multiple clock domains. The `no_clock` approach is not recommended; it is considered good design practice to always have methods associated with specific clocks.

Since there is no clock annotation on the method, the only allowed schedule is conflict-free (`CF`).

```
import "BVI" adder =
module mkAdder1A (Adder1#(t))
   provisos ( Bits#(t, st) );
   // Should have other provisos to limit type which can be added directly
   parameter W = valueOf(st);

   default_clock clk();         // no clock connections
   default_reset rst();         // no reset connections

   // not recommended - no clock associated with method
   method X add (A, B) clocked_by(no_clock) reset_by(no_reset);

   // Add should only be call once in a cycle
   // But without a clock annotation, only CF annoations are allowed.
   schedule add CF add;
endmodule
```

The following two modules both instantiated the `mkAdder1A` module. The first example, `sysTest1C`, is in the current clock domain. The second example, `sysTest1D`, is in the `c2` clock domain, which is not the current clock domain.

4.2.1 Instantiated in default clock domain

This example calls the module `mkAdder1A`, defined in the `import "BVI"` statement above. Since there is no clock domain specified, the Adder is in the default clock domain of the module.

```
(*synthesize*)
module sysTest1C ( Empty ifc );
   Reg#(UInt#(32)) a_c1 <- mkReg(0);
   Reg#(UInt#(32)) b_c1 <- mkReg(0);
```

```
   Reg#(UInt#(32)) x_c1 <- mkReg(0);

   Adder1#(UInt#(32))   adder <- mkAdder1A;

   rule doit_c1 (True);
      x_c1 <= adder.add(a_c1, b_c1);
   endrule
endmodule
```

4.2.2 Instantiated in a different clock domain

The following example shows that the `add` method can be called from any clock domain without a `clocked_by` clause on the instantiation, because the module wrapper specifies `no_clock` for the method.

```
(*synthesize*)
module sysTest1D( Clock c2, Reset r2, Empty ifc );

   Reg#(UInt#(32)) a_c2 <- mkReg(0, clocked_by(c2), reset_by(r2) );
   Reg#(UInt#(32)) b_c2 <- mkReg(0, clocked_by(c2), reset_by(r2) );
   Reg#(UInt#(32)) x_c2 <- mkReg(0, clocked_by(c2), reset_by(r2) );

   Adder1#(UInt#(32))   adder <- mkAdder1A;  //clocked_by not required

   // Adder methods can be called from any clock domain
   rule doit_c2 (True);
      x_c2 <= adder.add(a_c2, b_c2);
   endrule
endmodule
```

4.2.3 Instantiated in multiple clock domains

The following example shows that the compiler will prevent you from calling the the `add` method from multiple clock domains.

```
(*synthesize*)
module sysTest1D( Clock c2, Reset r2, Empty ifc );
   Reg#(UInt#(32)) a_c1 <- mkReg(0);
   Reg#(UInt#(32)) b_c1 <- mkReg(0);
   Reg#(UInt#(32)) x_c1 <- mkReg(0);

   Reg#(UInt#(32)) a_c2 <- mkReg(0, clocked_by(c2), reset_by(r2) );
   Reg#(UInt#(32)) b_c2 <- mkReg(0, clocked_by(c2), reset_by(r2) );
   Reg#(UInt#(32)) x_c2 <- mkReg(0, clocked_by(c2), reset_by(r2) );

   Adder1#(UInt#(32))   adder <- mkAdder1A;
```

```
   rule doit_c1 (True);
      x_c1 <= adder.add(a_c1, b_c1);
   endrule

   rule doit_c2 (True);
      x_c2 <= adder.add(a_c2, b_c2);
   endrule
endmodule
```

The compiler generates an error such as:

```
Method ‘adder.add’ is used in multiple clock domains:
      ‘adder.add b__h420 b__h421’ at ""
      ‘adder.add b__h434 b__h435’ at ""
```

4.3 Multiple wrapper methods

This implementation uses the `Adder2` interface, which has two interface methods. You have to be careful with your clock statements with this implementation it is possible to have the methods in different clock domains. This is not considered a safe domain crossing.

```
interface Adder2#(type t);
   method Action start(t a, t b);
   method t result();
endinterface
```

4.3.1 Wrapper methods clocked by default clock

With this module wrapper (`mkAdder2`) both methods are clocked by the default clock (`clk`) of the module where it is instantiated. The inputs and the output must be in the same clock domain.

```
import "BVI" adder =
module mkAdder2 (Adder2#(t))
   provisos ( Bits#(t, st) );
   // Should have other provisos to limit type which can be added directly
   parameter W = valueOf(st);

   default_clock clk();         // no clock connections
   default_reset rst();         // no reset connections

   method start (A, B) enable((*inhigh*) unused1);
   method X result ;

   // start can only be call once in a cycle
      schedule start C start;
```

```
   // start must be call before result,  and not in the same rule (restricted)
      schedule start SBR result ;
   // result can be call multiple time, hence conflict  free
      schedule result CF result;
endmodule

(* synthesize *)
module sysTest2A ( Clock c1, Reset r1, Empty ifc );
   Reg#(UInt#(32)) a_c1 <- mkReg(0, clocked_by(c1), reset_by(r1));
   Reg#(UInt#(32)) b_c1 <- mkReg(0, clocked_by(c1), reset_by(r1));

   Reg#(UInt#(32)) x_c1 <- mkReg(0, clocked_by(c1), reset_by(r1) );

   Adder2#(UInt#(32))   adder <- mkAdder2A(clocked_by c1, reset_by r1);

   rule doit_c1 (True);
      adder.start(a_c1, b_c1);
   endrule

   rule doit_c2 (True);
      x_c1 <= adder.result();
   endrule
endmodule
```

4.3.2 Wrapper output method clocked by `no_clock`

In this wrapper (`mkAdder2A`) the output is clocked by `no_clock`, making this a clock domain crossing. When instantatiated, the input and outputs can be defined to be in different clock domains.

```
import "BVI" adder =
module mkAdder2A (Adder2#(t))
   provisos ( Bits#(t, st) );
   // Should have other provisos to limit type which can be added directly
   parameter W = valueOf(st);

   default_clock clk();         // no clock connections
   default_reset rst();         // no reset connections

   // A no clock onthe output make this module a clock domain crossing!
   method start (A, B) enable((*inhigh*) unused1);
   method X result clocked_by(no_clock) reset_by(no_reset) ;

   // start can only be call once in a cycle
      schedule start C start;
   // start must be call before result,  and not in the same rule (restricted)
      // With separare clock  only a CF relation is allowed.
      schedule start CF result ;
```

```
   // result can be call multiple time, hence conflict  free
      schedule result CF result;
endmodule
```

The inputs into the Adder are in clock domain `c1`, the output is in clock domain `c2`.

```
(* synthesize *)
module sysTest2B ( Clock c1, Reset r1, Clock c2, Reset r2, Empty ifc );
   Reg#(UInt#(32)) a_c1 <- mkReg(0, clocked_by(c1), reset_by(r1));
   Reg#(UInt#(32)) b_c1 <- mkReg(0, clocked_by(c1), reset_by(r1));

   Reg#(UInt#(32)) x_c2 <- mkReg(0, clocked_by(c2), reset_by(r2) );

   Adder2#(UInt#(32))   adder <- mkAdder2A(clocked_by c1, reset_by r1);

   rule doit_c1 (True);
      adder.start(a_c1, b_c1);
   endrule

   rule doit_c2 (True);
      x_c2 <= adder.result();
   endrule
endmodule
```

5 Design considerations

5.1 Design options for an input enable port

The sizedFIFO examples in Section 2.3 demonstrated two different ways of handling a Verilog input from a `V_ClEAR` pin:

- Connect it to a `clear` method, as done in the `FIFOF` example (Section 2.3.1)
- Tie it off to 0, as done in the `GetPut` example (Section 2.3.2)

Now we're going to look at another example that has multiple implementation possibilities. This example is of an input used as an enable, in this case a debug enable (`V_DBGEN`). The definition of this input states that it can be tied LOW to disable it, if debug is not required. We'll consider four different options for mapping the Verilog into BSV:

1. Treat the signal as a regular method
2. Treat the signal as a regular always_enabled method
3. Bring out the enable signal as a port
4. Use a port for an explicit tieoff

5.1.1 Signal as an Action method

This option can only be used if the enable signal is a single bit, since in BSV only single bit enables can be enabled high. In this example, the enable is mapped into an interface as an `Action` method:

```
method Action dbg_enable
```

In the `import "BVI"` statement, the method statement uses the Verilog enable, `V_DBGEN` as the enable signal:

```
method dbg_enable () enable (V_DBGEN);
```

Nothing more is required to tie off this method. If the compiler sees that the method is never called, and if will tie it off to 0.

5.1.2 Signal as an always_enabled method

If the enable signal is more than a single bit, you can't define the enable as active high. Instead, you can create a method and define it as `always_enabled`.

```
(* always_enabled *)
method Action dbg_enable (Bool enable);
```

In the `import "BVI"` statement, the method statement would be:

```
method dbg_enable (V_DBGEN) enable ((*inhigh*)  V_UNUSED1);
```

where `V_UNUSED1` is a placeholder name, required by the `*inhigh*` attribute syntax.

Since this method is always enabled, it must be called every cycle, and there must be a rule defined to call it. In the BSV design, a possible rule could be:

```
rule set_debug ();
   dbg_enable ( some boolean expression );
endrule
```

If the rule is never to be used, it must be tied off, with a module or a rule:

```
rule tieoff_debug ();
    dbg_enable (False);
endrule
```

5.1.3 Signal as a port

If the signal is used as a port, then no interface method is present. To control the signal, use a port statement on the `import "BVI"`:

```
module mkTestWrapper (Bool debug_enable, Wrapper ifc);
...

port V_DBGEN = debug_enable;
```

With this approach, the port value can be changed dynamically during runtime. This module is close to the traditional RTL style of module instantiations. The BSV instance would look like:

```
let test <-  mkTestWrapper (some_boolean_expression);
```

5.1.4 Explicit tie-off

In this final case, the value is tied to a constant value in the `import "BVI"` statement and the feature is unavailable in the BSV design. Unlike option 3 above, the pin cannot be changed dynamically during run time.

Instead of a method statement in the `import "BVI"`, the pin is represented by a port statement with a constant value:

```
port V_DBGEN = False ;
```

5.2 Defining scheduling constraints

The `schedule` statement in the `import "BVI"` statement specifies the scheduling constraints between methods in the imported module.

The operators and their meanings are:

`CF`	conflict-free
`SB`	sequences before
`SBR`	sequences before, with range conflict (that is, not composable in parallel)
`C`	conflicts

Each pair of methods must have one, and only one, relationship defined. If no scheduling constraint is defined for a pair of methods the compiler will generate a warning and insert a default annotation: `C` for methods clocked by related clocks, `CF` for methods clocked by unrelated clocks. `CF` is the only correct annotation for methods clocked by unrelated clocks.

This section provides guidelines for determining the scheduling constraints in a module. For most modules, the first-pass scheduling constraints in Section 5.2.1 will suffice. For smaller blocks, or blocks with specific constraints, Section 5.2.2 provides some guidance.

5.2.1 First-pass scheduling constraints

For large IP blocks the following conditions usually apply:

- Most outputs and inputs are registered
- No combination path exists from input to output
- The interfaces are conceptually separated

For these situations, you can start with a simple approach to defining scheduling constraints:

- `Action` methods conflict (`C`) with themselves
- `Action` methods are conflict-free (`CF`) with other `Action` methods
- `Value` methods are conflict-free (`CF`) with other methods (both `Action` and `Value`)

  or

- `Value` methods are sequenced before (`SB`) all `Action` methods. This specifies that all states can be read before any `Action` method changes the state.
- Any `Action` methods which must not be asserted at the same time conflict (`C`) with each other.

5.2.2 Refining scheduling constraints

For smaller blocks, or sub-interfaces on a larger block, defining scheduling constraints can be more complex. Start with the guidelines above and then modify them to allow for specific reasoning about a given module.

For example, the table below shows the scheduling constraints specified for the methods of a `mkFIFO` module.

Scheduling Annotations for methods of FIFO	
Annotation	Description
`enq C enq`	Only 1 enqueue can occur at a time
`clr SB clr`	Multiple clears can occur, the results are the same
`(first, deq, enq, clr) SB clr`	Nothing can occur after a clr in the same cycle
`deq C deq`	Only 1 dequeue per cycle
`enq CF deq`	Can both enq and deq at the same time
`first SB deq`	Cannot read the first after a deq even though the hardware will let you read the *old* value