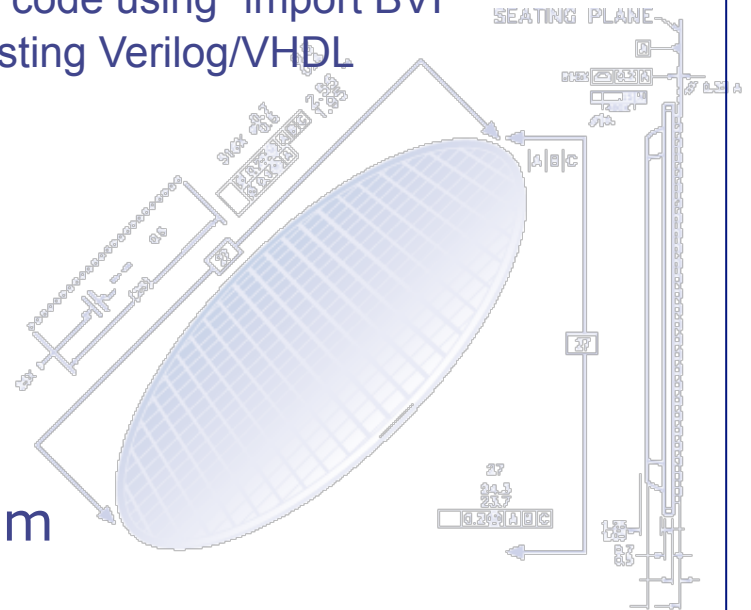
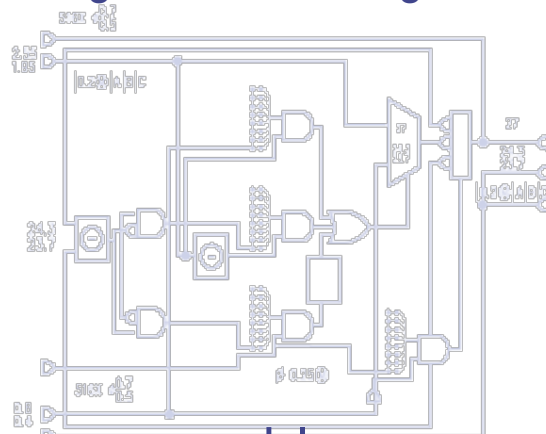




# BSV Training

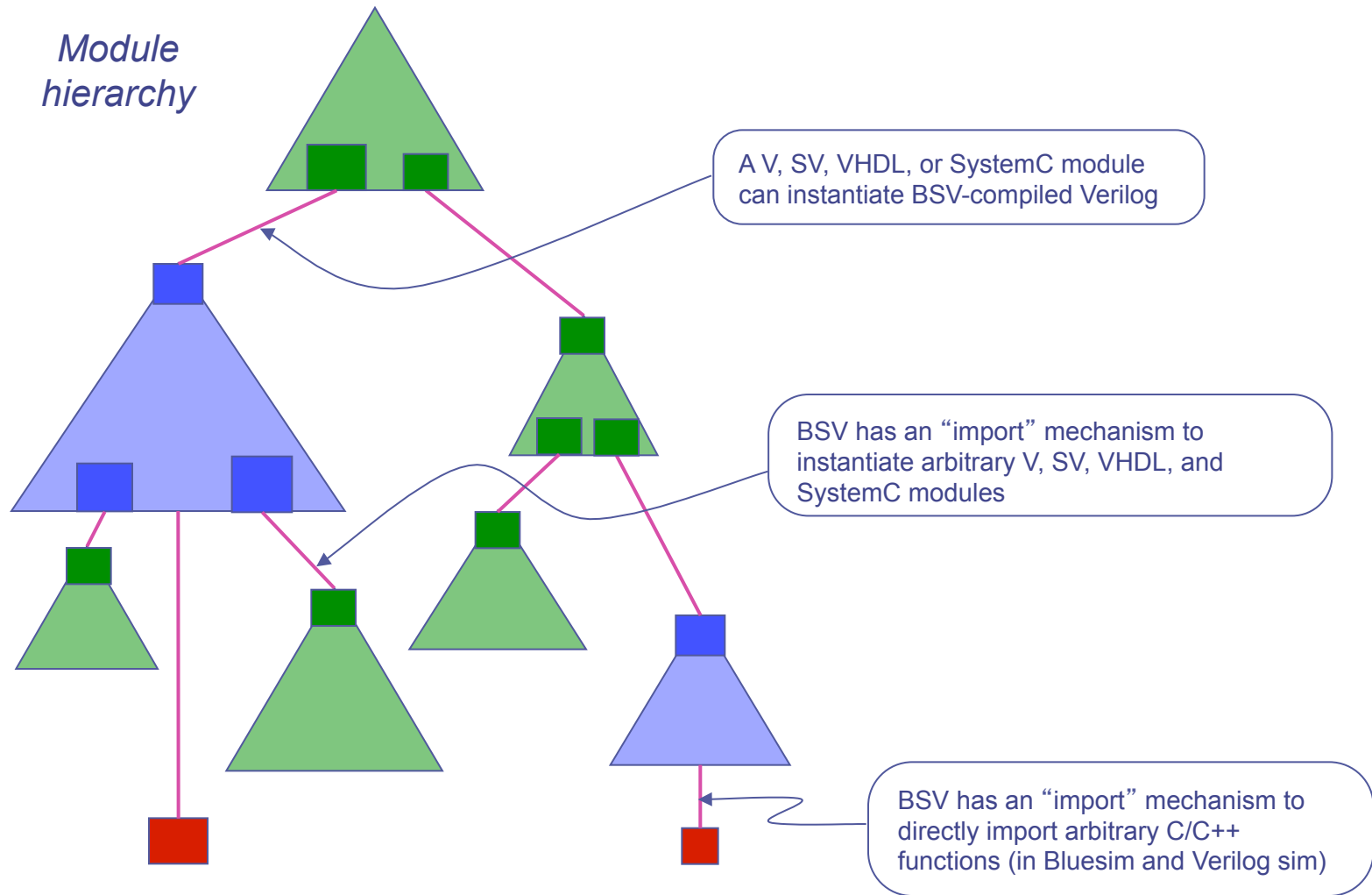
# Lec\_Interop\_RTL

## Importing existing Verilog and VHDL into BSV code using “import BVI”

[illegible]

[www.bluespec.com](http://www.bluespec.com)

# BSV interoperates with V, SV, VHDL, SystemC, and C/C++



## Legend

- V (Verilog), SV (SystemVerilog), VHDL, or SystemC (event-driven)
- BSV
- C, C++

# Plugging RTL (or RTL-like SystemC) into BSV

# When does BSV interface to RTL (V, SV, VHDL)?

## *Plugging RTL into BSV:*

- Many BSV designs are used in projects where there is already some existing verified RTL IP, which we simply wish to re-use
- Users may wish to add a new “primitive” to BSV that is important for an application domain
  - In fact, all BSV “primitives” are imported this way—knowledge about primitives is not built into the *bsc* compiler

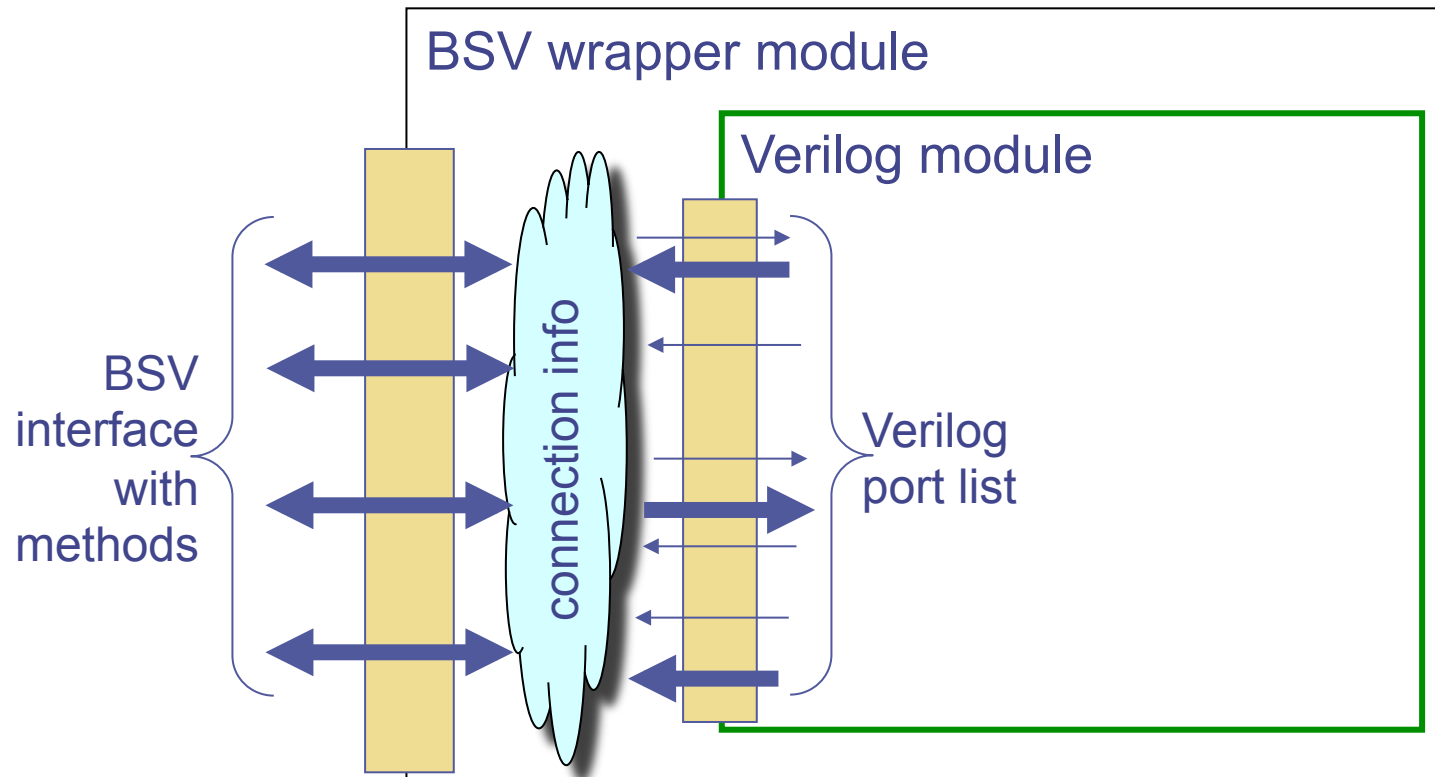
*This is done using BSV’s “import BVI” capability*

## *Plugging BSV into RTL:*

- A BSV design may fit into a larger RTL design
- A BSV design may be verified in an existing verification environment that is based on SystemVerilog (e.g., VMM), ‘e’, etc.

*This is done using by scheduling and naming control on the top-level interface, to be compatible with the RTL environment into which it fits*

# Basic structure of importing a Verilog component



- Define the BSV interface type that the module should provide to its environment
- Use the “import BVI” mechanism to define a wrapper module, which will:
  - Describe how the interface method arguments, results, enables, clocks, resets, etc. connect to the Verilog module ports
  - Describe the BSV *scheduling constraints* on the methods

## Example: importing a MAC (multiply-accumulate) module

The Verilog module we wish to import:

```
module mymac(EN, a, b, clear_value, clear, out, clk, rst_b);  
  
input a, b, EN, clear, clear_value, clk, rst_b;  
output out;  
  
reg [15:0] out;  
wire [15:0] a, b, clear_value;  
  
always@(posedge clk or negedge rst_b)  
if (!rst_b)  
    out <= 0;  
else  
    out <= clear ? clear_value : (EN ? out+a*b: out);  
endmodule
```

- When clear == 1, out <= clear\_value
- When EN == 1, out <= out + a \* b
- The value of the “out” register is continuously available as an output

# Define the desired BSV interface

```
interface Mac_IFC ;  
    method Action acc (Int#(16) aa, Int#(16) bb);  
    method Action reset_acc (Int#(16) value);  
    method Int#(16) read_y;  
endinterface
```

- When `clear == 1`, `out <= clear_value`
  - *The related Verilog signals “clear” and “clear\_value” will be part of the BSV “reset\_acc” method*
- When `EN == 1`, `out <= out + a * b`
  - *The related Verilog signals “EN”, “a” and “b” will be part of the BSV “acc” method*
- The value of the “out” register is continuously available as an output
  - *This will be done with the “read\_y” method*

# Define the module wrapper

*Verilog module name*

*BSV module name*

```
import "BVI" mymac =  
  module mkMac (Mac_IFC);  
  
    default_clock dclk (clk);  
    default_reset dreset (rst_b);  
  
    method acc(a, b)          enable(EN);  
    method reset_acc(clear_value) enable(clear);  
    method out read_y();  
  
    schedule (read_y) SB (reset_acc, acc);  
    schedule (acc) C (reset_acc);  
  
  endmodule
```

*Clock and reset*

*Methods*

*Scheduling constraints*

- The “default\_clock” line indicates that the BSV default clock will be connected to Verilog “clk”, and that it will be known within mkMac as “dclk” (although in this example it is not used elsewhere; typically it’s used in a “clocked\_by” clause).
- In the “method” lines we can see how method signals (arguments, results, ENABLEs) connect to various Verilog ports (which are shown in green). Since the optional “ready()” clauses are missing, these methods will be always ready.



# Define the module wrapper

*Verilog module name*

*BSV module name*

```
import "BVI" mymac =  
  module mkMac (Mac_IFC);  
  
    default_clock dclk (clk);  
    default_reset dreset (rst_b);  
  
    method acc(a, b)          enable(EN);  
    method reset_acc(clear_value) enable(clear);  
    method out read_y();  
  
    schedule (read_y) SB (reset_acc, acc);  
    schedule (acc) C (reset_acc);  
  
  endmodule
```

*Clock and reset*

*Methods*

*Scheduling constraints*

The “schedule” lines inform the *bsc* compiler about scheduling constraints on the methods.

- The first line indicates that `read_y < reset_acc` and `read_y < acc` (“SB” = “sequenced before” = “<”). This captures the Verilog semantics that “out” carries the current value in the register, and that any change due to `reset_acc` or `acc` will only be visible on the next clock.
- The second line indicates that `acc` and `reset_acc` conflict (“C”), so they can never fire concurrently (in the same clock). This design choice is stricter than the Verilog, which *does* allow both to happen in the same clock, giving priority to `reset_acc`. To match those semantics we could have instead specified: `read_y SB acc` and `acc SB reset_acc`

# Notes on the import Verilog mechanism

The previous slides showed only a simple example. The Reference Guide Sec. 15 goes into a lot more detail:

- Connecting clocks and resets
- Connecting method ENABLE and RDY signals
- The method-to-RTL connections need not be 1-to-1, and can involve other logic
- The available scheduling annotations
- Caveat: the schedule is just an assertion by you taken at face value by the compiler when using this module in your BSV program; the compiler makes no attempt to ‘verify’ that the scheduling assertions are sensible

The actual Verilog code is not touched by the *bsc* compiler. The *bsc* compiler simply generates a Verilog instantiation of the module, and your RTL simulator finds and instantiates the Verilog module

These import mechanisms can also be used for importing VHDL and SystemVerilog, and RTL-style SystemC

- Most RTL simulators allow free intermixing of Verilog, VHDL, SystemVerilog and SystemC

# What about Bluesim, when you import RTL?

Bluesim does not currently support “co-simulation” of Bluesim with a Verilog simulator

- If you just import a Verilog module, your only simulation option is Verilog simulation, i.e.,
  - Use *bsc* to generate Verilog from the BSV code
  - Use a Verilog simulator to simulate all the Verilog code (BSV-generated, and imported)

How do the primitives in the BSV library work in Bluesim?

- Every primitive in the BSV library is implemented both in Verilog and in C
  - The C code is imported using the “import BDPI” mechanism (described later), into a module with exactly the same BSV interface
- For every primitive, we have a wrapper module that instantiates one or the other, like this:

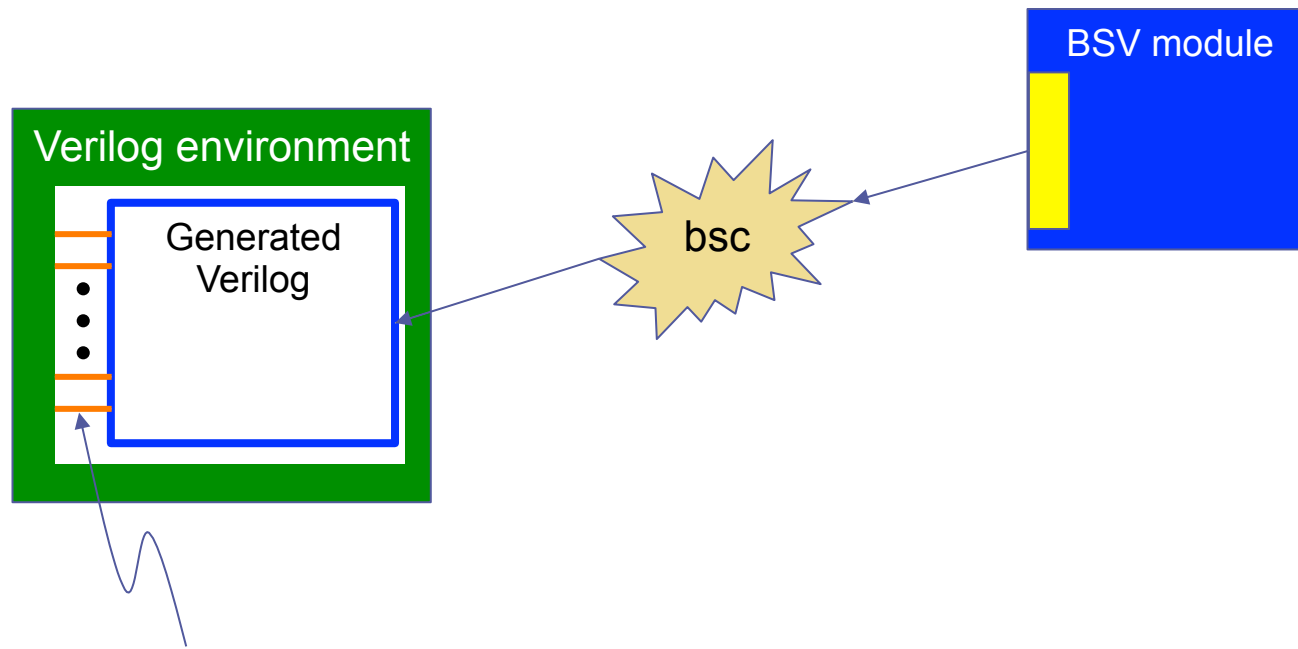
```
IfcType m <- if (genVerilog) mkVerilogWrapper;  
             else mkCWrapper;
```

- “genVerilog” is a built-in boolean which is true when compiling for Verilog and False when compiling for Bluesim (Reference Guide Sec. B.6). Thus, at compile time, *bsc* chooses the appropriate instantiation (i.e., this is a “static elaboration” choice).

# Plugging BSV into RTL

# Plugging BSV into RTL: key issues

When plugging BSV into RTL (more accurately: BSV-generated RTL into RTL), the key issues are:



We need:

- Exactly the right set of input and output signal ports, with exactly the right port names, as expected by the RTL environment
- Exactly the right signaling protocols on these ports, as expected by the RTL environment (may be different from standard BSV method protocol of RDY and EN signal)

Example: the “Verilog environment” may be an AMBA AXI bus port

# Exposing method conditions

- A method's condition can always be exposed as an explicit Bool method
- The examples below are from the BSV library
  - Note: in the lib FIFO, enq, first and deq are still guarded by their implicit conditions; the notFull and notEmpty methods are just additional methods

```
interface FIFO#(type t);  
  // enq has "notFull" condition  
  method Action      enq(t x);  
  // first/deq have "notEmpty" condition  
  method t           first;  
  method Action      deq;  
  method Action      clear;  
endinterface: FIFO
```

```
interface FIFO#(type t);  
  // enq has "notFull" condition  
  method Action      enq(t x);  
  // first/deq have "notEmpty" condition  
  method t           first;  
  method Action      deq;  
  method Action      clear;  
  method Bool        notEmpty;  
  method Bool        notFull;  
endinterface: FIFO
```

# Exposing method conditions

- Implementing exposed conditions is trivial: just replicate the method condition
  - E.g., below, `enq()`'s condition `canEnque` is returned explicitly as a Boolean in `notFull()`
  - Don't worry: the generated code will share a single instance of the condition logic
    - Or, share it explicitly by writing `let x = canEnque` and using `x` twice

```
module mkFIFO (FIFO#(type t));  
  ...  
  method Action enq(t x)      if (canEnque); ... endmethod  
  method t      first        if (canDequeue); ... endmethod  
  method Action deq          if (canDequeue); ... endmethod  
  
  method Bool  notFull;      return canEnque; endmethod  
  method Bool  notEmpty;     return canDequeue; endmethod  
endmodule: mkFIFO
```

# Removing RDY signals

- The “always\_ready” attribute can be applied to a method to indicate
  - that it is always ready (the compiler will check this!)
  - that the RDY signal must be removed from the generated Verilog

```
(* always_ready = “add, result” *)
```

```
module mkAdder (Adder);  
  Reg#(int) acc <- mkRegA(0);
```

```
  method Action add(a, b);  
    acc <= a + b;  
  endmethod
```

```
  method int result;  
    return acc;  
  endmethod  
endmodule
```

```
interface Adder;  
  method Action add(int a, int b);  
  method int result;  
endinterface
```

*The compiler will check that add and result are always ready to be invoked, i.e. their implicit and explicit conditions are always True*

*The compiler will not generate any RDY signal for add and result*

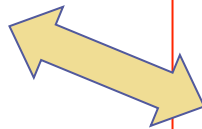


# Removing ENABLE signals

- The “always\_enabled” attribute can be attached to a method (Action or ActionValue) to indicate
  - that it is assumed True, i.e., the data input signals (args) are driven by the environment on *every* cycle
  - Of course, this implies “always ready”, i.e., the method’s condition must be always True (otherwise it would be an error to drive the EN signal)
  - that the EN signal must be removed from the generated Verilog

```
(* always_enabled = “add” *)  
module mkAdder (Adder);  
  Reg#(int) acc <- mkRegA(0);  
  
  method Action add(a, b);  
    acc <= a + b;  
  endmethod  
  
  method int result;  
    return acc;  
  endmethod  
endmodule
```

```
interface Adder;  
  method Action add(int a, int b);  
  method int result;  
endinterface
```



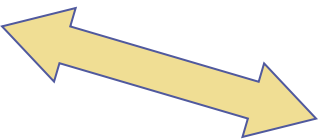
*The compiler will check that add can be always invoked, i.e. its method condition is always True*

*The compiler will not generate any EN signal for add*

# Unguarded methods

- Suppose a module must read datum on EVERY CYCLE

```
(* always_enabled = "accept" *)  
module mkFoo (IfcFoo);  
  FIFO#(int) fifo <- mkFIFO;  
  ...  
  method Action accept (int);  
    fifo.enq(bus);  
  endmethod  
endmodule
```



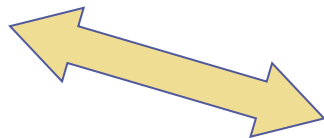
*The bsc compiler will signal an error, since the enq() method may not be ready (the fifo may be full).*

*It cannot verify that accept() is always ready, so it cannot allow the always\_enabled spec*

# Unguarded methods

- Answer: use “unguarded” methods, i.e., methods with no conditions, relying on you to guard them explicitly
  - The BSV library provides FIFOs with unguarded methods
  - Warning: these are dangerous! Use with care, only in these “impedance matching” situations

```
(* always_enabled = “accept” *)  
module mkFoo (IfcFoo);  
  FIFO#(int) fifo <- mkUGFIFO;  
  ...  
  method Action accept (int);  
    fifo.enq(bus);  
  endmethod  
endmodule
```



*This will work.*

*Warning: you have to use some other means  
to avoid buffer overflow!*

## Summary: plugging BSV into RTL

- A BSV Action method with one argument that is always ready and always enabled becomes, exactly, a Verilog input port. E.g.,

```
(* always_ready, always_enabled *)  
method Action accept (int);
```

- A BSV value method that is always ready becomes, exactly, a Verilog output port. E.g.,

```
(* always_ready *)  
method int yield_data;
```

- Sec. 13.2.1 shows attributes by which you can control the exact names of these generated Verilog ports

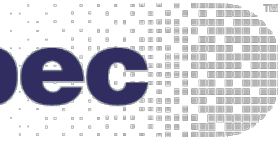
*With these capabilities, and unguarded methods, you can produce a Verilog interface with any desired ports, with any desired names, and with any desired behavior (protocol)*

- *The BSV AXI library has examples of interfaces to the AMBA AXI bus (masters, slaves, etc.)*

# Hands-on

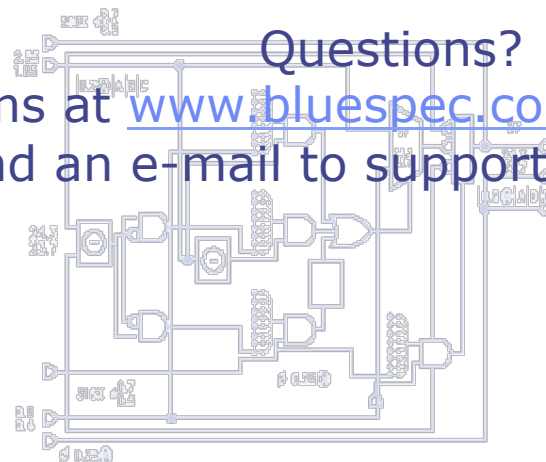
- BSV-by-Example book: Examples in Chapter 15

# bluespec



## End

```
import PCell*;
typedef Bit[24] StateT;
module ex_hdl_csrR_top(empty);
  Integer nfa_depth;
  function Bit[24] determine_pump(StateT s);
    return (s[0]);
  endfunction
  PCellT(bata1) inboud0;
  addSeedPCell(nfa_depth) the_inboud0(inboud0);
  PCellT(bata1) outboud0;
  addSeedPCell(nfa_depth) the_outboud0(outboud0);
  PCellT(bata1) outboud1;
  addSeedPCell(nfa_depth) the_outboud1(outboud1);
  rule exp1 (True);
    bata1 in_data = inboud0.first;
    PCellT(bata1) out_data =
      determine_pump(in_data) == 0 ? outboud0 : outboud1;
    out_data <- outboud1;
    inboud0 <- exp1;
  endrule; exp1
endmodule; ex_hdl_csrR_top
```



Questions?

Join online forums at [www.bluespec.com](http://www.bluespec.com), and ask your question,  
or send an e-mail to [support@bluespec.com](mailto:support@bluespec.com)

