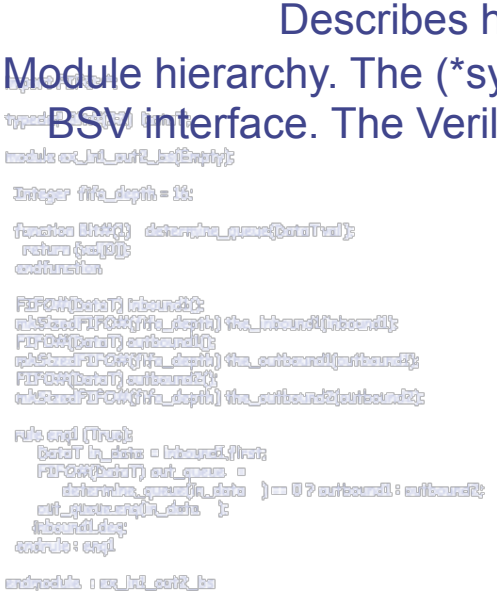
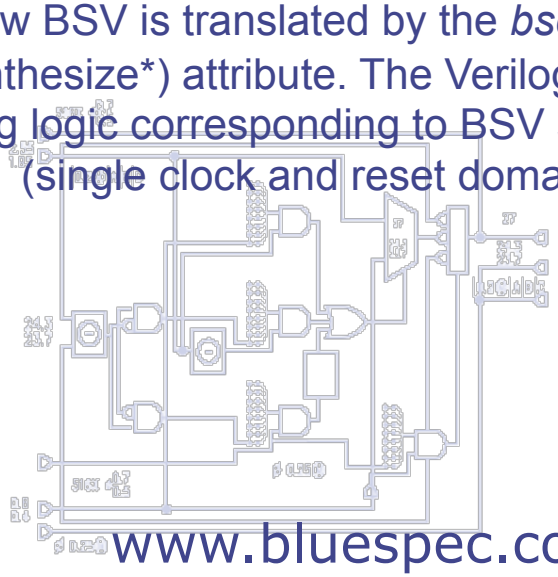
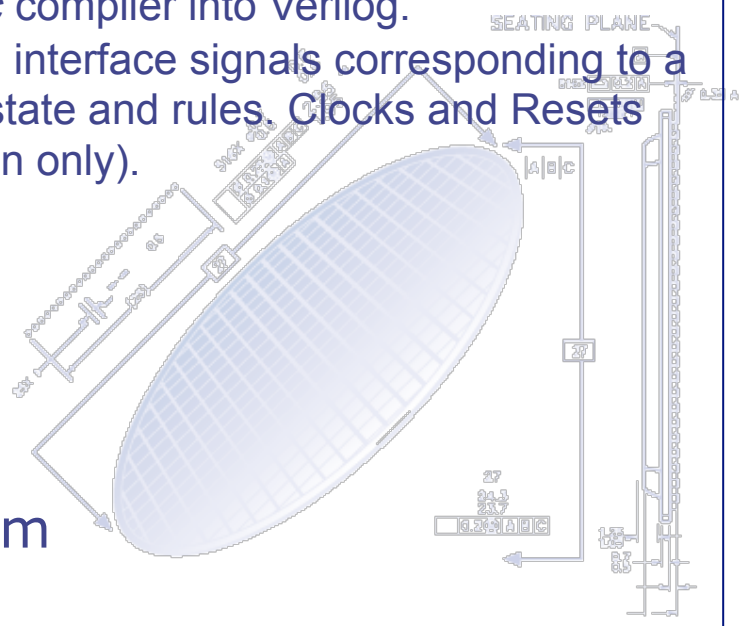
BSV Training

Lec_BSV_to_Verilog

Describes how BSV is translated by the *bsc* compiler into Verilog. Module hierarchy. The (*synthesize*) attribute. The Verilog interface signals corresponding to a BSV interface. The Verilog logic corresponding to BSV state and rules. Clocks and Resets (single clock and reset domain only).







www.bluespec.com

© Bluespec, Inc., 2013

Introduction

This lecture is optional, and is intended for those who have prior experience with Verilog and digital hardware, and are curious about the Verilog produced by the *bsc* tool from BSV source code. It is also useful for those who need to interface BSV-generated Verilog to other existing Verilog/VHDL.

For everybody else, this lecture can safely be skipped.

Analogy: One can learn a programming language (e.g., C, C++, Java) and use it productively without knowing anything about the machine code produced by compilers for the language. Similarly, BSV is self-contained and has a clear semantics independent of any particular implementation, and so can be used and debugged by just understanding the source semantics. Verilog is just the "assembly language" for BSV.

© Bluespec, Inc., 2013

Contrasting Verilog and BSV module structure

Verilog params are typically scalar numbers.
Verilog interfaces are signal port lists.

BSV params can be of arbitrary type
(incl. functions, interfaces, modules, …).
BSV interfaces are interface types (with methods)

```
module m #(params) (ports)

  input …
  output …
  wire …

  reg x;
  reg y;

  module m1 #(params) p (port connections);
  module m1 #(params) q (port connections);
  module m2 #(params) r (port connections);

  assign w = 10 + wire from instance q
  assign …

  always @(posedge clk) …

  always @(posedge clk) …

endmodule
```

wire decls
The only type is 'bits'

'reg' is not a module.
'reg' may not even be a register.
'reg' just holds bits.

Module instantiation

"Behavior"

```
module m #(params) (interface type);

  Reg #(t1) x <- mkReg (0);
  Reg #(t2) y <- mkReg (12);

  Ifc_m1 p <- mkM1a (params);
  Ifc_m1 q <- mkM1b (params);
  Ifc_m2 r <- mkM2 (params);

  int w = 10 + q.method();

endmodule
```

Registers are instantiated just like any other module, and are strongly typed.

Typed var decls

Rules

Methods

© Bluespec, Inc., 2013

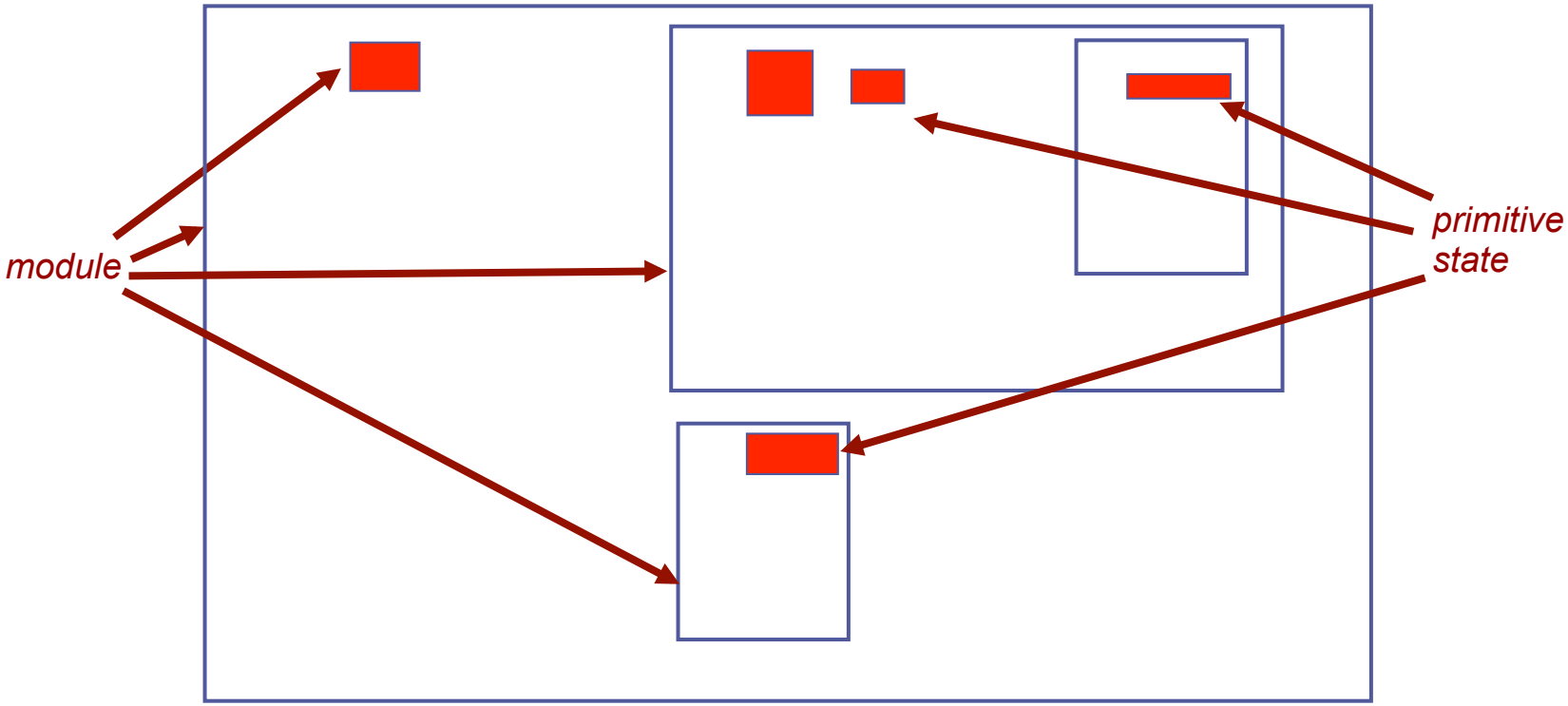
bluespec

Module hierarchy and state

A BSV design consists of a *module hierarchy* (just like in Verilog, SystemVerilog and SystemC)

The leaves of the hierarchy are "primitive" state elements, including registers, FIFOs, etc.

Even registers are (semantically) modules (unlike in Verilog, SystemVerilog, ...).



All "primitives" in BSV are in fact implemented in Verilog and "imported" using BSV's standard import mechanism. Hence, you can easily create new primitives or import existing Verilog IP.

© Bluespec, Inc., 2013

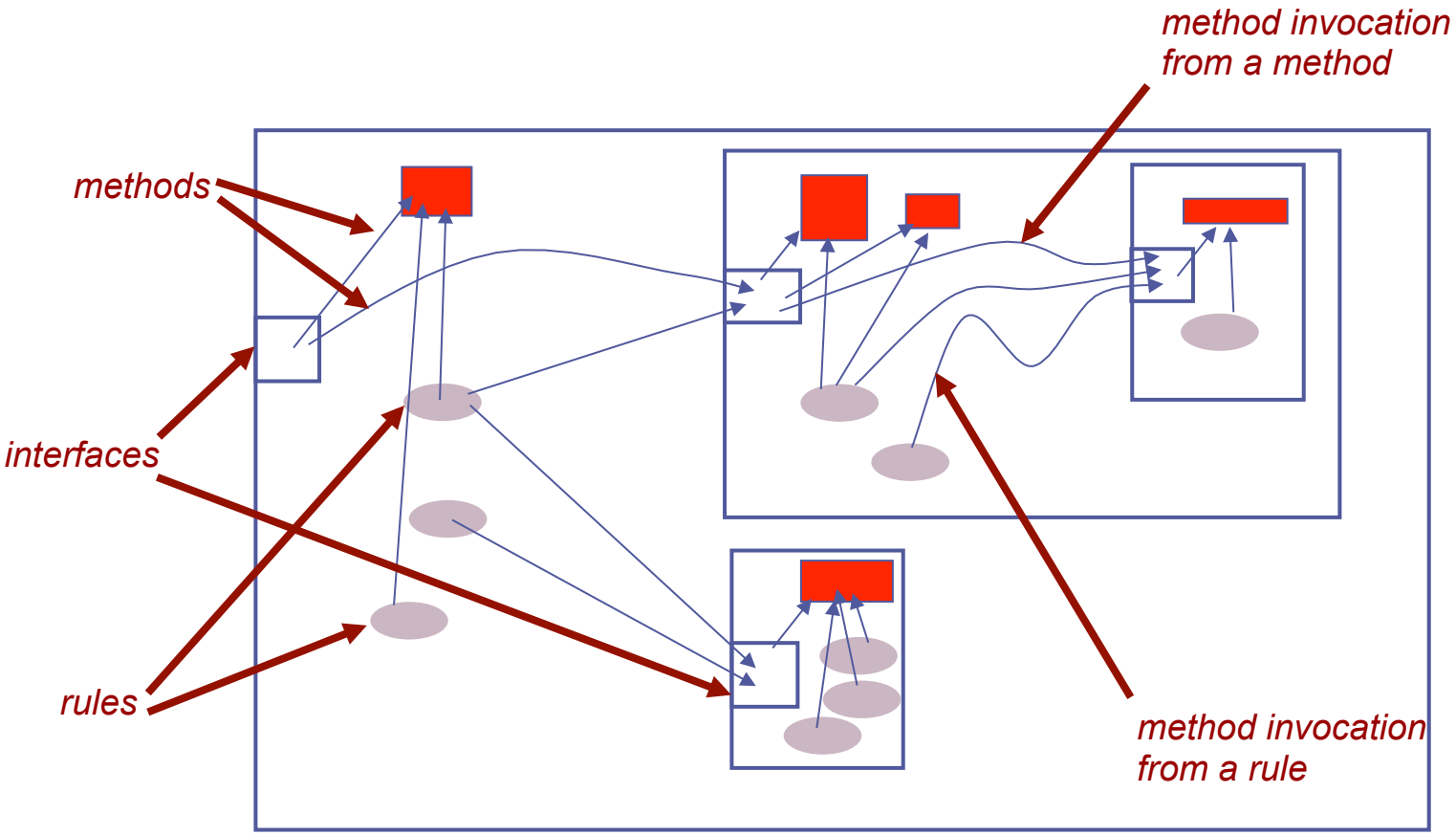
bluespec

Rules and interface methods

Modules provide interfaces, which contain *interface methods.*

Modules contain rules, which use methods in other modules.
All inter-module communication is via methods (object-oriented)

A method can itself use methods of other modules.



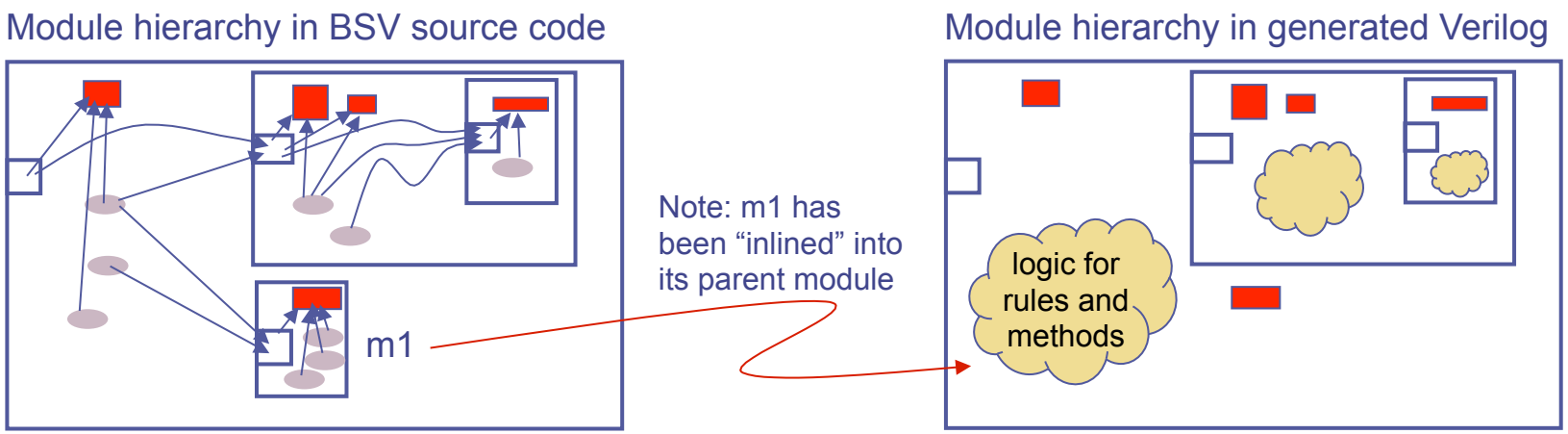
© Bluespec, Inc., 2013

From BSV module hierarchy to Verilog module hierarchy

Broadly speaking, the BSV module hierarchy can be preserved in the generated Verilog:

- Each BSV module mkM becomes a corresponding Verilog module mkM (in file mkM.v)
- If a BSV module mkM1 instantiates a BSV module mkM2,
  the corresponding Verilog mkM1 instantiates the corresponding Verilog mkM2

However, a BSV module may be “inlined” wherever it is instantiated.
In this case, there will be no corresponding separate Verilog module.



This inlining is controlled by the optional (* synthesize *) attribute (see next slide)

© Bluespec, Inc., 2013

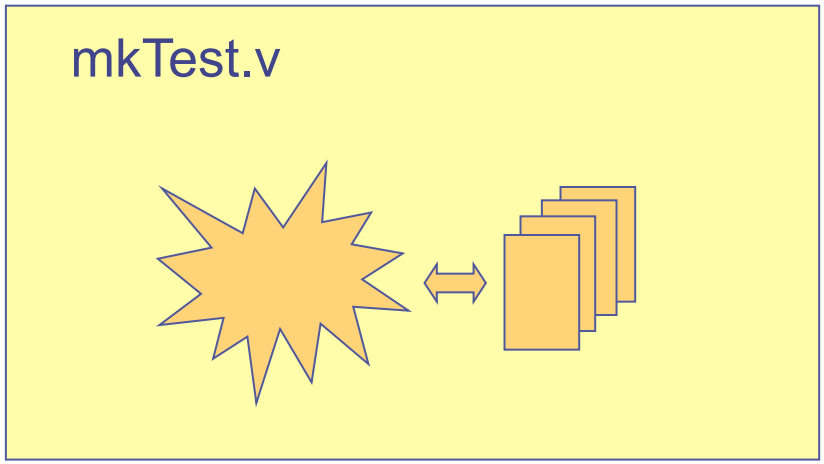
Example of the optional (* synthesize *) attribute

BSV source code

```
(* synthesize *)
module mkTest (Empty);
   Mult_ifc m <- mkMult;
   …
endmodule: mkTest
```

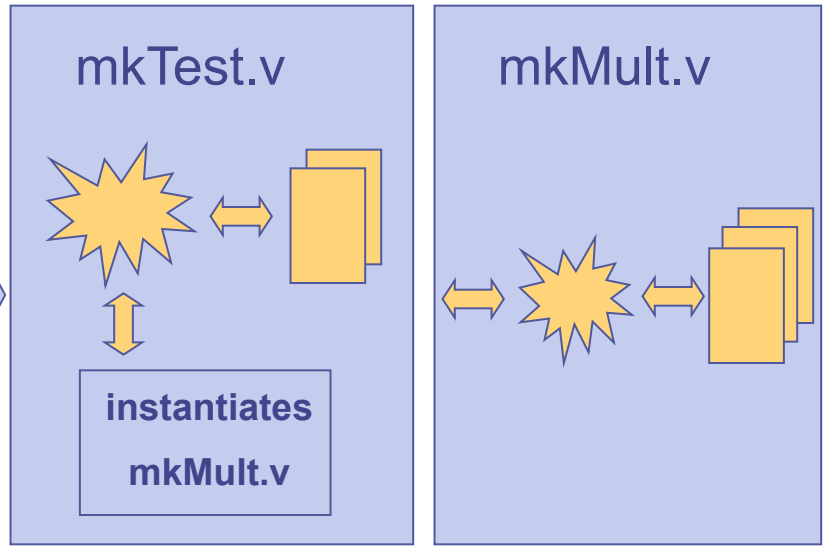
```
module mkMult (Mult_ifc);
  …
endmodule: mkMult
```

Generated Verilog




```
(* synthesize *)
module mkTest (Empty);
   Mult_ifc m <- mkMult;
   …
endmodule: mkTest
```

```
(* synthesize *)
module mkMult (Mult_ifc);
  …
endmodule: mkMult
```



Note: saying “-g mkTest” on the *bsc* command line is equivalent to adding the (*synthesize*) attribute

© Bluespec, Inc., 2013

bluespec

Limitation on the "(*synthesize*)" attribute

The "(* synthesize *)" is written just before a "module mkFoo (…);" header.

It can only be written before certain modules (*bsc* will complain if it is disallowed).
This is because Verilog is less expressive than BSV:

- In BSV, module parameters and interfaces can contain arbitrary types, including functions, other interfaces, modules, etc.
- In Verilog, module parameters and ports can only carry bits, scalars and bit-vectors

Thus, the "(* synthesize *)" attribute can only be placed on those BSV modules whose parameters and interfaces can be mapped to Verilog parameters and ports.

Important note: when a BSV module cannot have a "(*synthesize*)" attribute, that does not mean that it cannot be used in synthesizable designs; it simply means that it cannot be *separately* synthesized. It can still be instantiated in other BSV modules which, in turn, can be synthesized. In this sense, *all* of BSV can be used in synthesizable code; there is no limitation of a "synthesizable subset" which is common in other High-Level Synthesis tools based on C and C++.

© Bluespec, Inc., 2013

Mapping BSV interfaces to Verilog ports

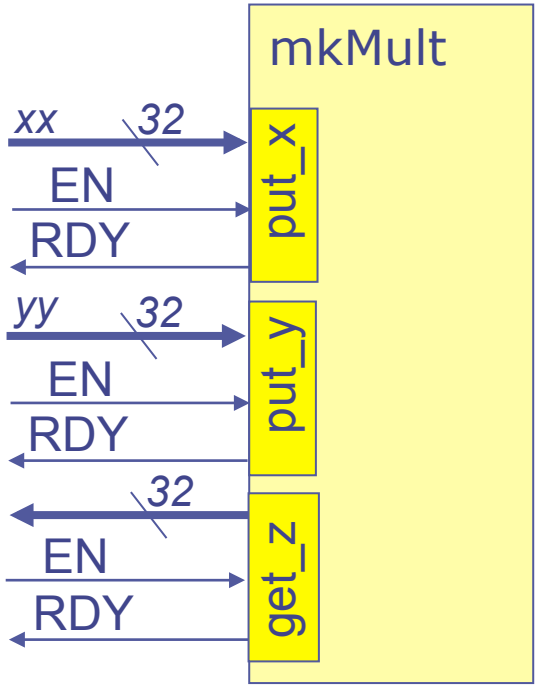
Interface methods are mapped directly to Verilog ports, based on their types:

- BSV method arguments become Verilog module input ports
- BSV method results become Verilog module output ports
- A BSV method condition becomes a Verilog “ready” (RDY) output signal
- BSV Action and ActionValue methods also have a Verilog “enable” (EN) input signal

© Bluespec, Inc., 2013

Mapping interfaces to HW: example

```
interface Mult_ifc;
   method Action put_x (int xx);
   method Action put_y (int yy);
   method ActionValue #(int) get_z ();
endinterface: Mult_ifc
```



- RDY = the method condition
- EN = signal asserted by external rule when it invokes an Action or ActionValue method (causes the internal Actions to happen)

In a separate lecture on interfacing to RTL we'll see that it is possible:

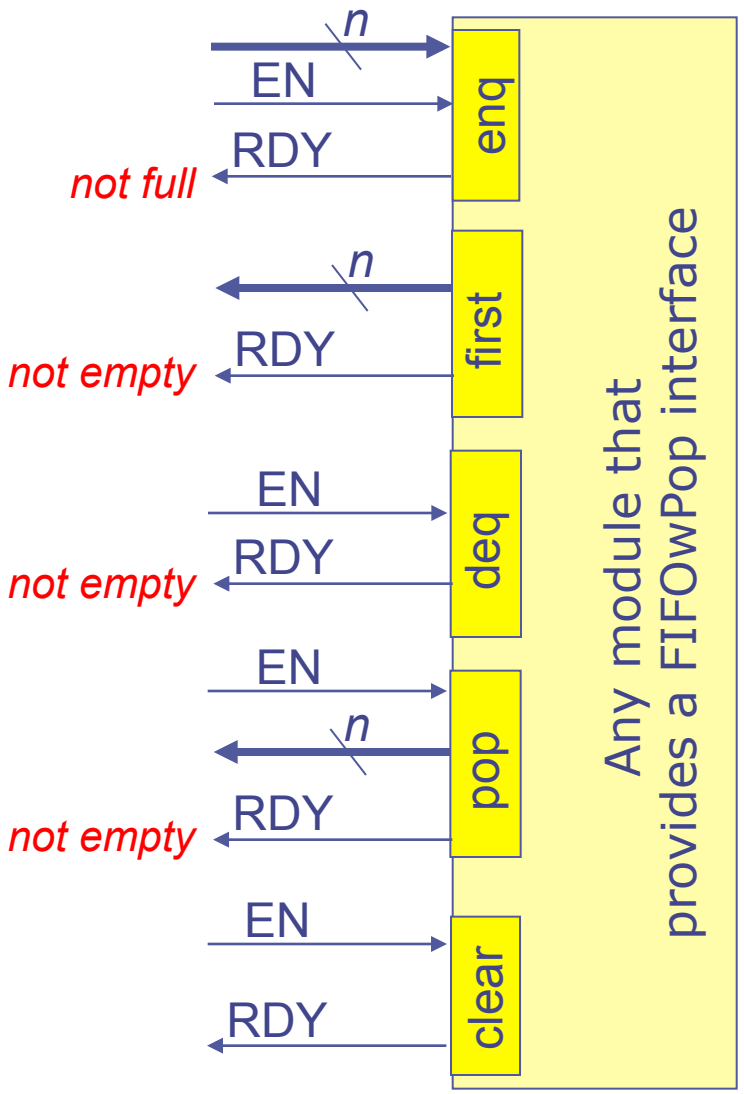
- to eliminate the RDY signal when the method is always ready, and
- to eliminate the EN signal when external rule invokes the method on every clock.

Thus, Verilog ports are just a special case of BSV methods.

© Bluespec, Inc., 2013

Mapping interfaces to HW: another example

```
interface FIFOwPop #(type t);
  method Action            enq (t x);
  method t                 first;
  method Action            deq;
  method ActionValue#(t)   pop;
  method Action            clear;
endinterface
```

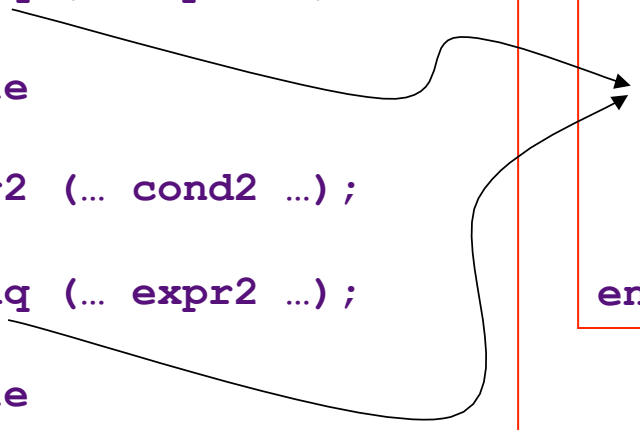


© Bluespec, Inc., 2013

Sharing: two rules invoking a common method

```
module mkTest (…);
  …
  FIFO#(int) f <- mkFIFO;
  …
  rule r1 (… cond1 …);
    …
    f.enq (… expr1 …);
    …
  endrule

  rule r2 (… cond2 …);
    …
    f.enq (… expr2 …);
    …
  endrule
endmodule: mkTest
```

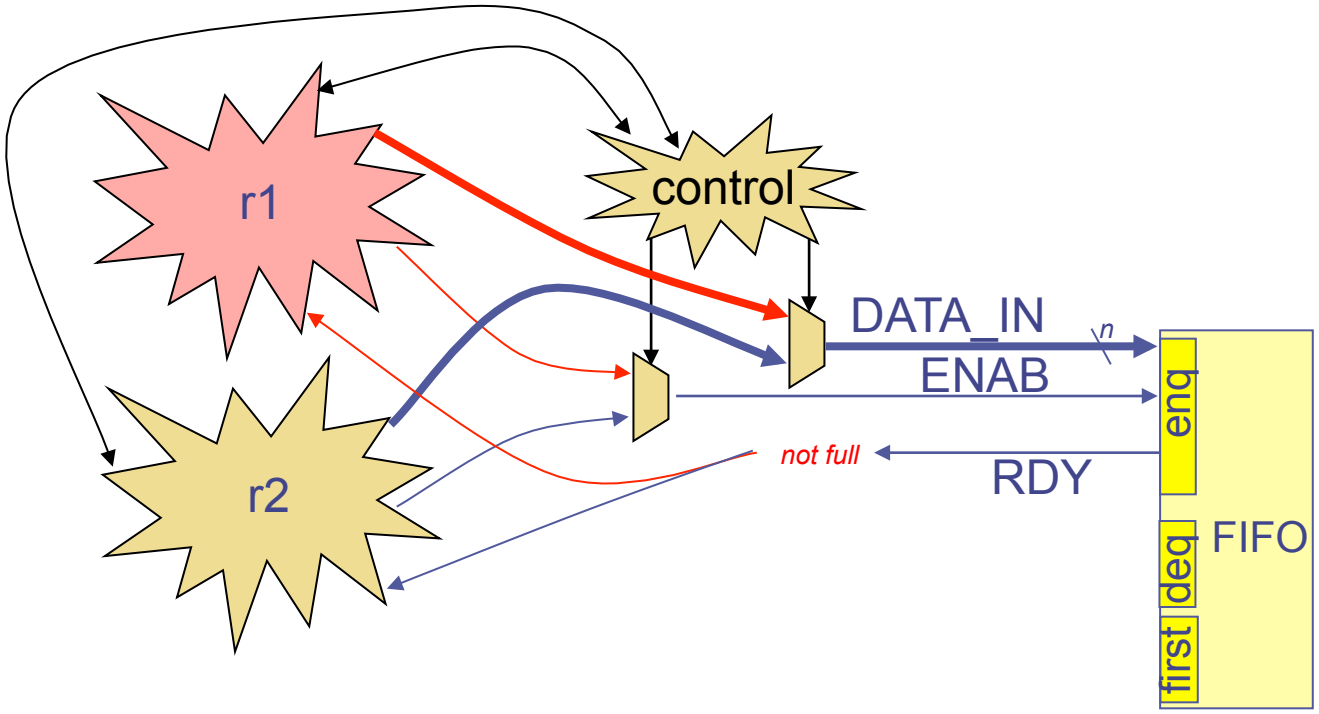
```
interface FIFO#(type t);
  Action enq (t n);
  …
endinterface

module mkFIFO (…);
  …
  method enq(x) if (…notFull…);
    …
  endmethod
  …
endmodule: mkFIFO
```



© Bluespec, Inc., 2013

bluespec

HW for a shared method



Note:

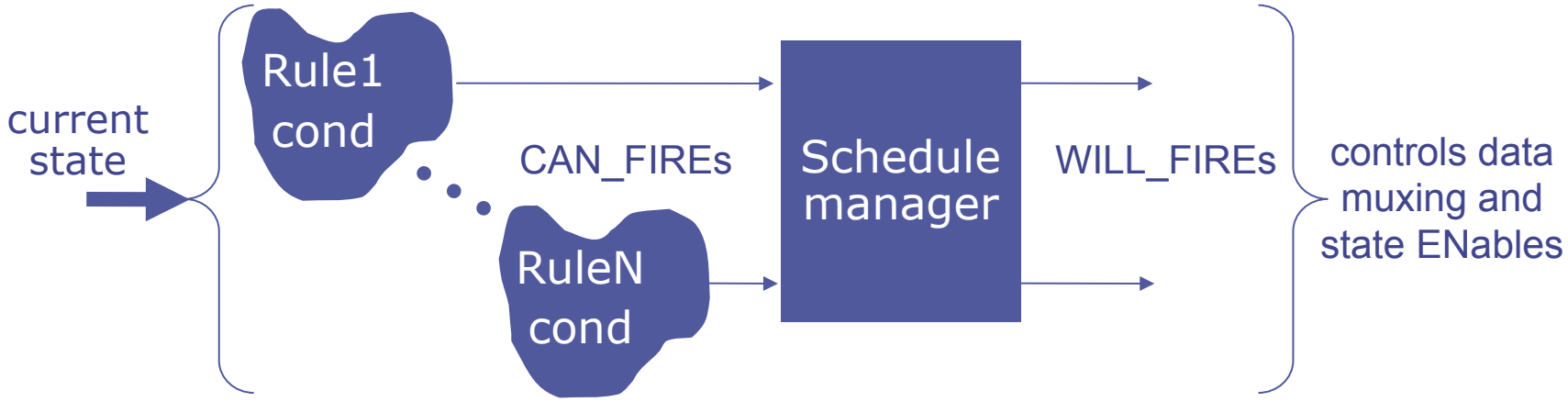
- *Any* input wire is potentially a rule resource conflict
  - (only one rule can drive it in each clock)
- Examples:
  - Argument of any method (whether value, Action or ActionValue)
  - EN signal of any Action or ActionValue method

Corollary:

- Only 0-argument value methods don't have resource conflicts (no input wires)

© Bluespec, Inc., 2013

CAN_FIRE and WILL_FIRE signals in synthesized HW

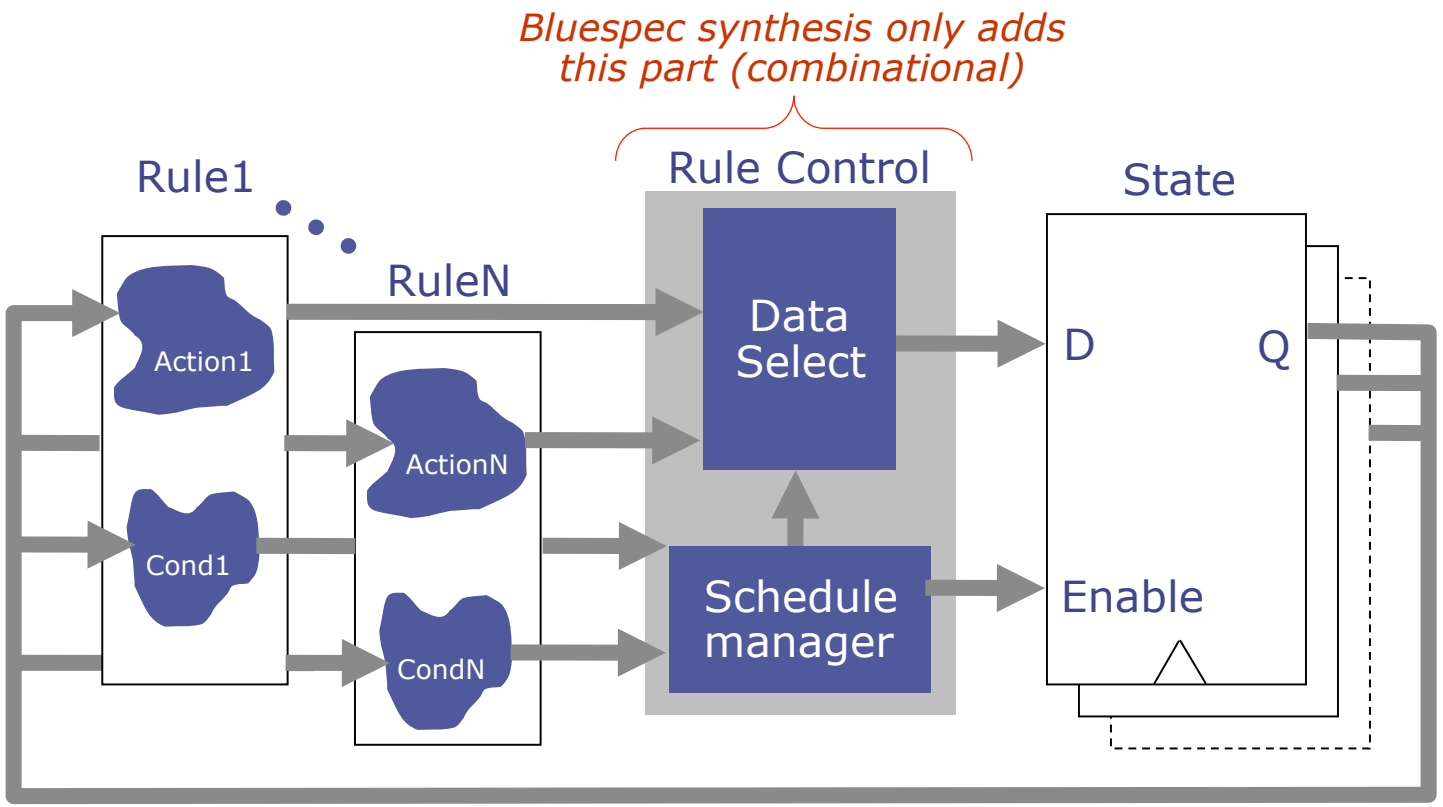


- The compiler performs conflict analysis of all rules in a BSV program, and generates a corresponding HW schedule manager (purely combinational)
- The CAN_FIRE signal of a rule = its rule condition along with the conditions of methods that it invokes, directly or indirectly
- The WILL_FIRE signal of a rule =
  CAN_FIRE && (! WILL_FIRE of any earlier rule that conflicts with this rule)

Note:
- Conflict analysis requires sophisticated analysis of rule conditions and resources, potentially across module boundaries
- In practice, the "schedule manager" is not a monolithic circuit; it is distributed across modules and is built incrementally

© Bluespec, Inc., 2013
bluespec

Overall schematic of synthesized HW



- The schedule manager ensures consistency with Rule logical semantics
  - Represents control logic that is normally hand-written in RTL in ad hoc ways, and usually the most error-prone part of RTL
  - Here, correct-by-construction, because of rule semantics

- Bluespec patented technology

© Bluespec, Inc., 2013

A small example to build HW intuitions about rules

Can you guess what these rules compute? (Hint: "Euclid")

```
rule decr ( x <= y && y != 0 );
    y <= y - x;
endrule : decr

rule swap (x > y && y != 0);
    x <= y; y <= x;
endrule: swap
```

Answer:

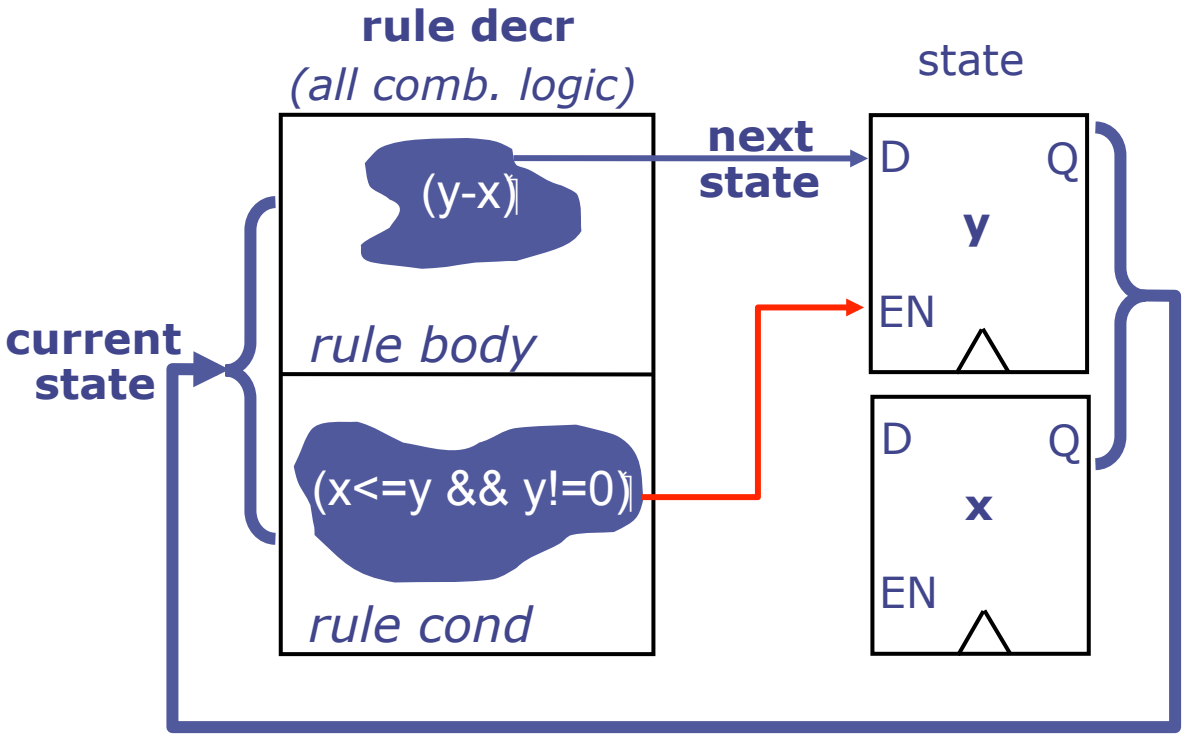
Euclid's algorithm for computing GCD (Greatest Common Divisor)
of initial values in x and y registers; result is in x

© Bluespec, Inc., 2013

bluespec

HW for one of the rules

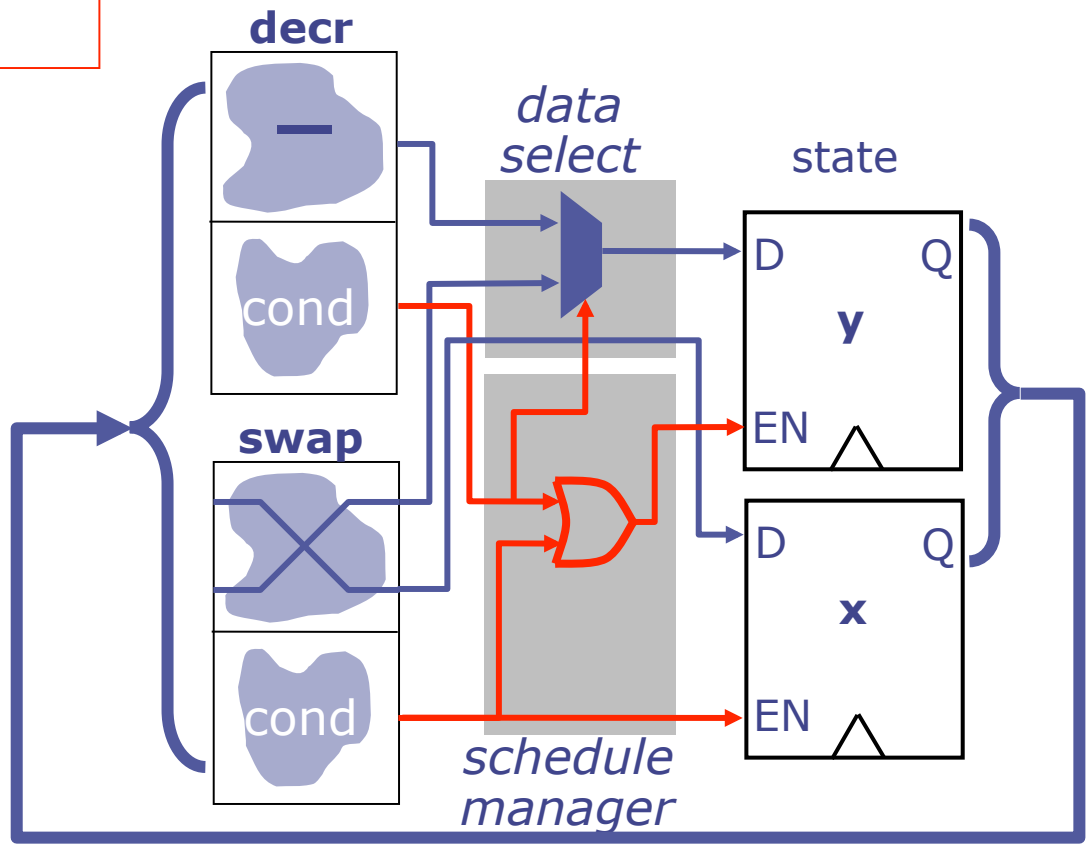
```
rule decr ( x <= y && y != 0 );
    y <= y - x;
endrule : decr
```



© Bluespec, Inc., 2013

HW for two mutually exclusive rules

```
rule decr ( x <= y && y != 0 );
    y <= y - x;
endrule : decr

rule swap (x > y && y != 0);
    x <= y; y <= x;
endrule: swap
```



© Bluespec, Inc., 2013

bluespec

A brief word about clocks and resets

For BSV designs with a single clock and reset domain:

- There is no mention of clocks or resets in the BSV source code
- In the generated Verilog, every module has an additional CLK and RST input port, and these are connected to all sub-modules and state elements (including registers and memories)

[There is a separate lecture going into more detail on clocks and resets, including multiple clock and reset domains, clock domain-crossing synchronizers, positive and negative resets, synchronous and asynchronous resets, etc.]

© Bluespec, Inc., 2013

Hands-on

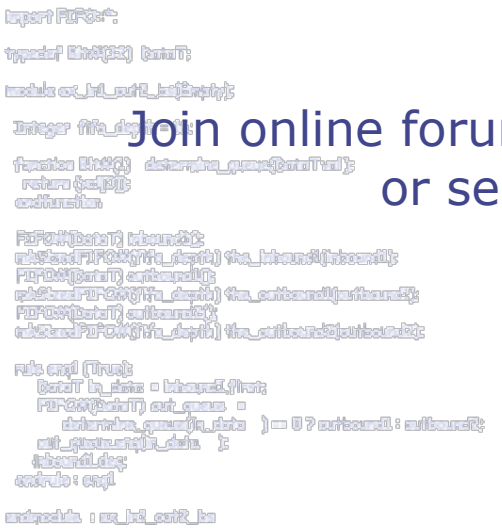
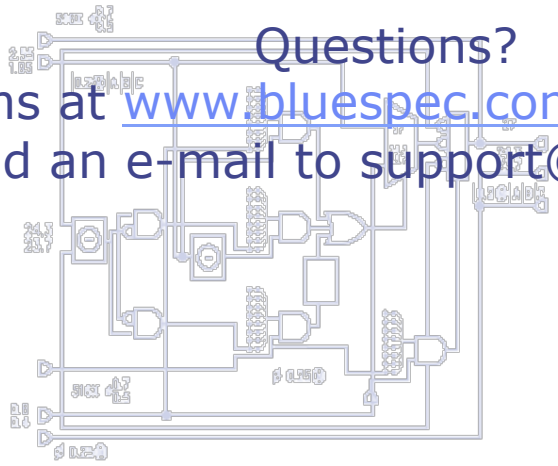
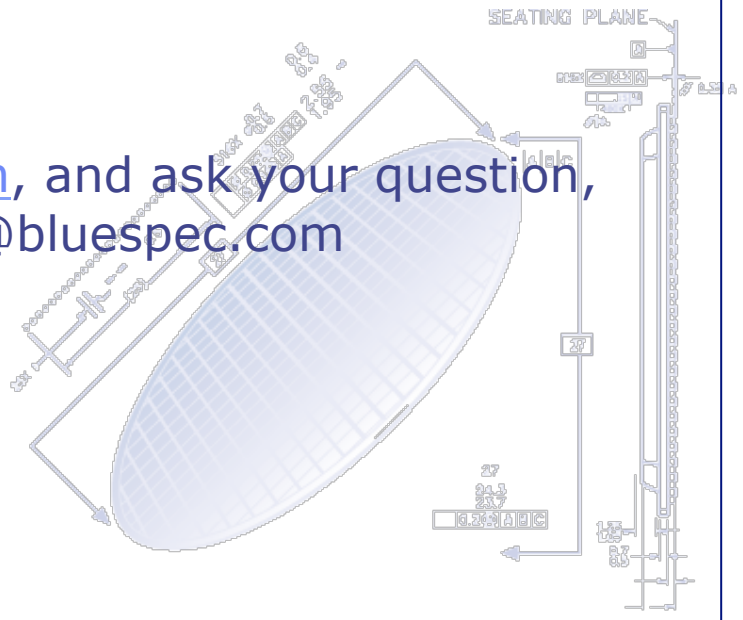
For any of the example codes in this training course, or in the BSV-by-Example book,

- Use *bsc* to compile it to Verilog using the "-verilog" flag (instead of the "-sim" flag which is used to create a Bluesim executable)
- Examine the generated Verilog files in light of the descriptions in this lecture
- Run the Verilog code in a Verilog simulator (3rd party, not supplied by Bluespec) and use the simulator's facilities to examine the Verilog code, single-step it, generate and view waveforms, etc.
  - You can use commercial simulators like ModelSim (Mentor Graphics), VCS (Synopsys), NCSim/Incisive (Cadence), Riviera Pro (Aldec), ISim (Xilinx), etc.
  - You can use the free tools available for Linux: Verilog simulator iverilog and waveform viewer gtkwave

© Bluespec, Inc., 2013

bluespec

End

Questions?

Join online forums at www.bluespec.com, and ask your question,
or send an e-mail to support@bluespec.com