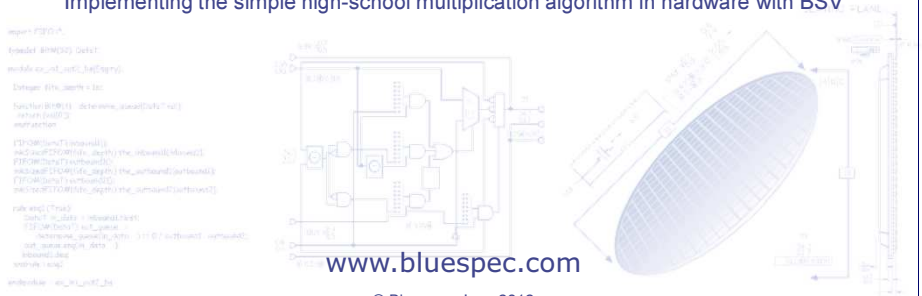
bluespec

BSV Training

Section: A small example

Implementing the simple high-school multiplication algorithm in hardware with BSV



www.bluespec.com

High-school multiplication algorithm, in binary

```
  1001      // d = 4'd9
x 0101      // r = 4'd5
------
  1001      // d << 0      (since r[0] == 1)
 0000       // 0 << 1      (since r[1] == 0)
1001        // d << 2      (since r[2] == 1)
0000        // 0 << 3      (since r[3] == 0)
-------
0101101     // product = 45
```

These tests can be performed by repeatedly shifting r right by 1 place and testing its LSB

bluespec

BSV code for simple binary multiplier and a test driver

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

   rule rl_go;
      m.put_x (9);
      m.put_y (5);
   endrule

   rule rl_finish;
      let z = m.get_z ();
      $display ("Product = %d", z);
      $finish ();
   endrule
endmodule: mkTest
```

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

```
(* synthesize *)
module mkMult (Mult_ifc);
   Reg#(int)  product <- mkReg (0);
   Reg#(int)  d       <- mkReg (0);
   Reg#(int)  r       <- mkReg (0);
   Reg#(Bool) got_x   <- mkReg (False);
   Reg#(Bool) got_y   <- mkReg (False);

   rule rl_compute ((r != 0) && got_x && got_y) ;
      if (pack(r)[0] == 1) product <= product + d;
      d <= d << 1;
      r <= r >> 1;
   endrule

   method Action put_x (int x) if (! got_x);
      d <= x; product <= 0; got_x <= True;
   endmethod

   method Action put_y (int y) if (! got_y);
      r <= y; got_y <= True;
   endmethod

   method ActionValue #(int) get_z ()
               if ((r == 0) && got_x && got_y);
      got_x <= False; got_y <= False;
      return product;
   endmethod
endmodule: mkMult
```

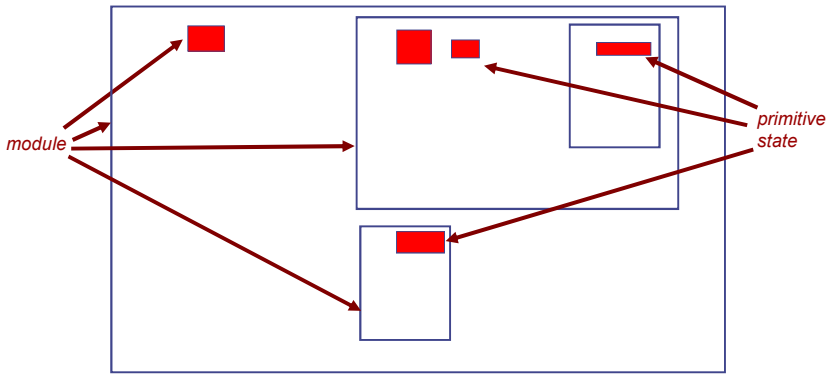

bluespec

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.

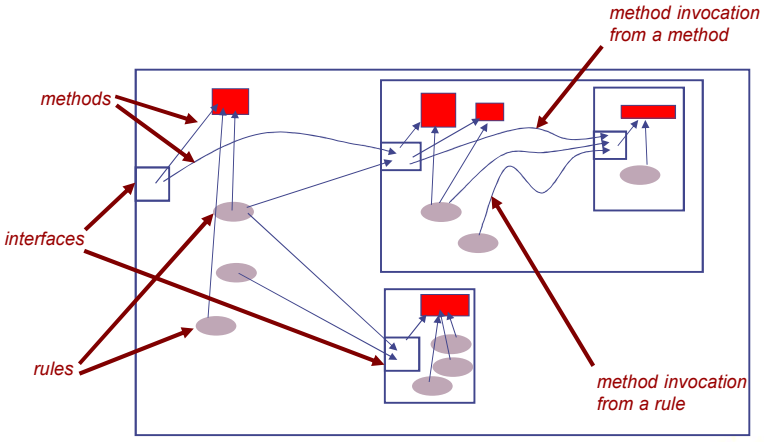
bluespec

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 (fully object-oriented)

A method can itself use methods of other modules.



bluespec

Even registers are modules …

Registers are just modules with the following interface:

```
interface Reg #(t);
   method Action _write (t x);
   method t      _read  ();
endinterface: Mult_ifc
```

Where "t" is "int" or "Bool" or some other type

Following standard BSV syntax, a register update would look like this:

```
d._write (d._read () << 1);
```

But, for convenience, the BSV compiler allows you to omit "._read()" from register reads, and will insert it for you. So, our update becomes:

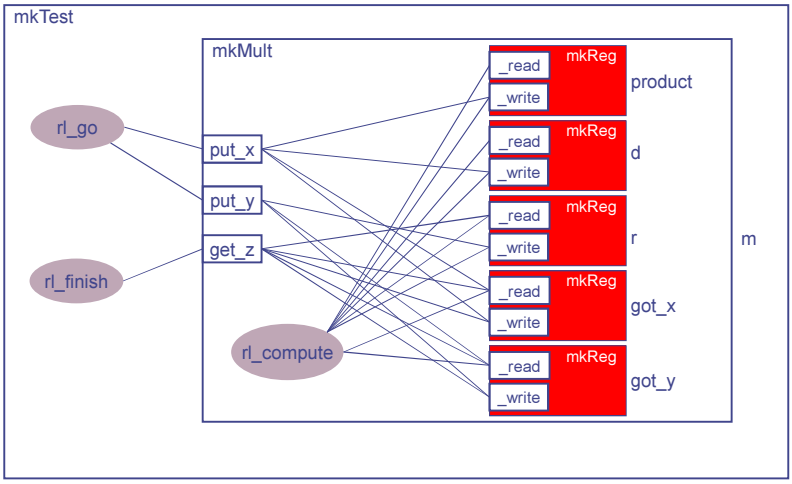
```
d._write (d << 1);
```

And, for convenience, BSV provides special syntax using the conventional "<=" for register assignment for "._write()". So, our update becomes:

```
d <= d << 1;
```


bluespec

Specifically, in our multiplier example …



bluespec

Module instantiation

Module instantiation has the following syntax:

```
interface_type   instance_name  <-  module_name ( module_parameters ) ;
```

"(*module_parameters*)" can be omitted if a module has no parameters.

Examples:

```
Mult_ifc m <- mkMult;

Reg#(int)  product <- mkReg (0);
Reg#(Bool) got_x   <- mkReg (False);
```


bluespec

Basic syntax elements, and identifiers

Basic syntax elements (identifiers, comments, whitespace, strings, integer constants, infix operators, etc.) all follow standard SystemVerilog syntax.
Standard static scoping rules for identifier visibility and shadowing.

Identifiers are case sensitive.
The case of the first letter in an identifier is significant:
- Constants begin with an uppercase letter: Int, UInt, Bool, True, False, ...
- Variables begin with a lowercase letter (type1, x, y, product, mkMyModule, ...)

Exceptions: 'int' and 'bit'
- *These are familiar type constants from Verilog*
- *You can use the standard BSV syntax Int#(32) and Bit#(1) if you prefer*

Module naming convention: in all our examples you will notice we use names like "mkTest" and "mkProd"

- The "mk" prefix is pronounced "make" and reinforces the idea that, as in Verilog, a module declaration is a generator, i.e., the module can be *instantiated* multiple times, each time providing a fresh instance.

- This is just a convention, for style and readability (not enforced by the compiler)

bluespec

Method and rule conditions

Rules and methods can have a boolean condition indicating when they are "enabled" (see ( ) at right). Default if omitted: True

A method is ultimately invoked from some rule (perhaps via other methods of intermediate modules).

The method's condition additionally affects rule enablement.

Thus, a rule's "overall condition", which we also call the rule's CAN_FIRE condition, is the AND of:
- any condition explicitly attached to the rule
- all conditions attached to methods invoked by the rule

```
(* synthesize *)
module mkMult (Mult_ifc);
   Reg#(int)  product <- mkReg (0);
   Reg#(int)  d       <- mkReg (0);
   Reg#(int)  r       <- mkReg (0);
   Reg#(Bool) got_x   <- mkReg (False);
   Reg#(Bool) got_y   <- mkReg (False);

   rule rl_compute ((r != 0) && got_x && got_y) ;
      if (pack(r)[0] == 1) product <= product + d;
      d <= d << 1;
      r <= r >> 1;
   endrule

   method Action put_x (int x) if (! got_x);
      d <= x; product <= 0; got_x <= True;
   endmethod

   method Action put_y (int y) if (! got_y);
      r <= y; got_y <= True;
   endmethod

   method ActionValue #(int) get_z ()
               if ((r == 0) && got_x && got_y);
      got_x <= False; got_y <= False;
      return product;
   endmethod
endmodule: mkMult
```


bluespec

Method and rule conditions

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

   rule rl_go;
      m.put_x (9);
      m.put_y (5);
   endrule

   rule rl_finish;
      let z = m.get_z ();
      $display ("Product = %d", z);
      $finish ();
   endrule
endmodule: mkTest
```

CAN_FIRE condition for rl_go:
(! got_x) && (! got_y)

CAN_FIRE condition for rl_finish:
((r == 0) && got_x && got_y)

CAN_FIRE condition for rl_compute:
((r != 0) && got_x && got_y)

```
(* synthesize *)
module mkMult (Mult_ifc);
   Reg#(int)  product <- mkReg (0);
   Reg#(int)  d       <- mkReg (0);
   Reg#(int)  r       <- mkReg (0);
   Reg#(Bool) got_x   <- mkReg (False);
   Reg#(Bool) got_y   <- mkReg (False);

   rule rl_compute ((r != 0) && got_x && got_y) ;
      if (pack(r)[0] == 1) product <= product + d;
      d <= d << 1;
      r <= r >> 1;
   endrule

   method Action put_x (int x) if (! got_x);
      d <= x; product <= 0; got_x <= True;
   endmethod

   method Action put_y (int y) if (! got_y);
      r <= y; got_y <= True;
   endmethod

   method ActionValue #(int) get_z ()
               if ((r == 0) && got_x && got_y);
      got_x <= False; got_y <= False;
      return product;
   endmethod
endmodule: mkMult
```


bluespec

Rule logical execution semantics

Rule logical execution semantics are surprisingly simple:

```
while (at least one rule's condition is ready)
      choose any such rule
      perform its body action(s)
```

- Note: this is sequential (exec one rule before doing the next)
  - ... and, as a result, trivially *atomic*
  - In reasoning about correctness, we can think about one rule at a time (do not have to worry about interleavings of multiple rules reading/writing shared state)
    - Unwanted interleavings are the root cause of race conditions and bugs in parallel software and in HW designs
  - Executing a rule is not the same a clock cycle! Later we'll see how rule executions are mapped into clocks.
- Note: this is non-deterministic ("choose any such rule")
  - Don't worry, by the time we map it into real hardware, it will be fully deterministic
  - (Note: non-determinism is often preferred by people who do formal verification, because it allows a *family* of equivalent behaviors without over-constraining the schedule by which they are reached)

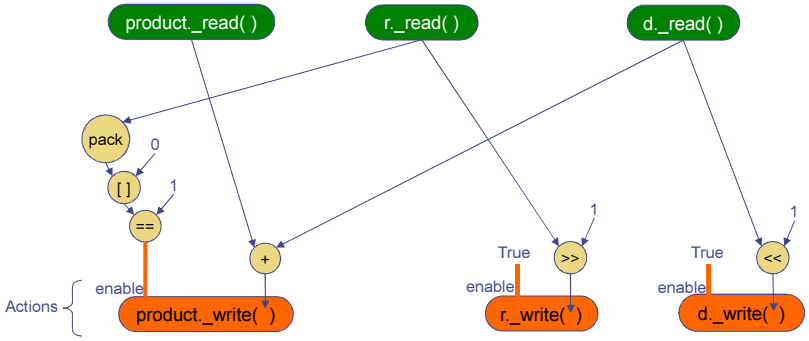
bluespec

Structure of a rule's body

The body of a rule contains one or more "Action" methods, such as _write methods of registers (although, due to conditionals, only some may be enabled).

Everything else is a classical, pure expression built from constants, operators, and invocations of "value" methods (such as _read methods of registers). "Pure" means that there are no side-effects, and they can visualized as an expression graph/ dataflow graph/ and even a hardware combinational circuit.

For example: here is the structure of the body of rule "rl_compute" from our multiplier:



bluespec

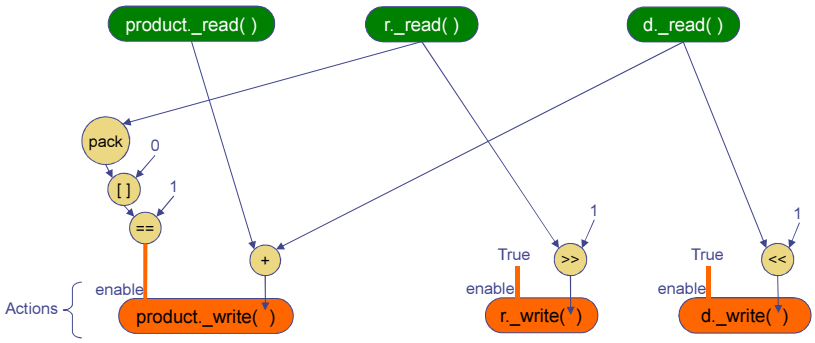
Executing a rule's body

Executing a rule's body means:

- Evaluate all the pure expressions fully (everything except the orange boxes in the diagram)
- Perform all the enabled Actions (all orange boxes with enable == True)

Executing a rule is considered to be "instantaneous".
All actions executed in a rule are considered to happen "simultaneously".

- This is a semantic abstraction, like vertical clock edges; of course, in real life, signals take time to propagate



bluespec

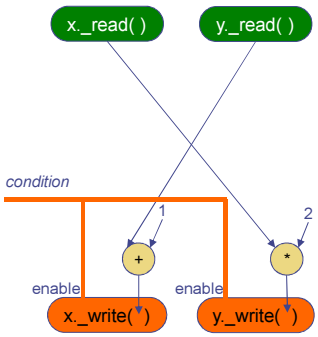
All actions in a rule are simultaneous

To emphasize that all actions in a rule are simultaneous, note:
- The textual ordering of actions in a rule is not important
- You can even exchange values in two or more registers

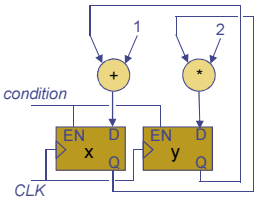
```
rule rule2a (condition);
   x <= y + 1;
   y <= x * 2;
endrule : rule2a
```

Textual order of actions is irrelevant

```
rule rule2b (condition);
   y <= x * 2;
   x <= y + 1;
endrule : rule2b
```



Some HW intuition (for this simple rule)



bluespec

A rule's actions must be capable of being simultaneous

```
rule rule3 (…);
  valuea <= expr1;
  valuea <= expr2;
  …
endrule : rule3
```

Cannot instantaneously update a register with two values

```
rule rule4 (…);
  fifo.enq (23);
  fifo.enq (34);
  …
endrule : rule4
```

Cannot instantaneously enq two values into one enq port of a FIFO

```
rule rule5 (…);
  let x = regFile.read (5);
  let y = regFile.read (7);
  …
endrule : rule5
```

Cannot instantaneously read two registers from one read port of a register file

- The *bsc* compiler will flag such errors
  - "Cannot compose certain actions in parallel"
- Note: it is of course possible to have alternate register, FIFO or regFile implementations that have *multiple* ports that can be accessed simultaneously, in which case those simultaneous accesses can be used in a single rule

bluespec

Rule execution sequence for simple binary multiplier

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

   rule rl_go;
      m.put_x (9);
      m.put_y (5);
   endrule

   rule rl_finish;
      let z = m.get_z ();
      $display ("Product = %d", z);
      $finish ();
   endrule
endmodule: mkTest
```

If we execute this program according to these semantics (paying attention to rule CAN_FIREs), we get the following rule firing sequence:

rl_go
rl_compute
rl_compute
...
rl_compute
rl_finish

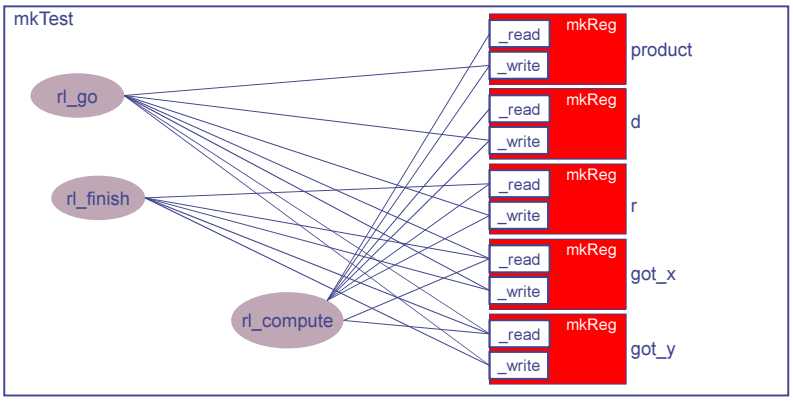
```
(* synthesize *)
module mkMult (Mult_ifc);
   Reg#(int)  product <- mkReg (0);
   Reg#(int)  d       <- mkReg (0);
   Reg#(int)  r       <- mkReg (0);
   Reg#(Bool) got_x   <- mkReg (False);
   Reg#(Bool) got_y   <- mkReg (False);

   rule rl_compute ((r != 0) && got_x && got_y) ;
      if (pack(r)[0] == 1) product <= product + d;
      d <= d << 1;
      r <= r >> 1;
   endrule

   method Action put_x (int x) if (! got_x);
      d <= x; product <= 0; got_x <= True;
   endmethod

   method Action put_y (int y) if (! got_y);
      r <= y; got_y <= True;
   endmethod

   method ActionValue #(int) get_z ()
               if ((r == 0) && got_x && got_y);
      got_x <= False; got_y <= False;
      return product;
   endmethod
endmodule: mkMult
```


bluespec

The optional (* synthesize *) attribute

The optional (* synthesize *) attribute on a module declaration requests the BSV compiler to retain this bit of hierarchy in the generated Verilog, i.e., to generate a Verilog module for this BSV module, and instantiate it wherever it is used.

If we omit the attribute on mkMult, the module instance gets in-lined into mkTest at the instantiation location (the methods get in-lined into the invoking rules, and mkMult's sub-hierarchy—its registers—get added to mkMult's sub-hierarchy) like so:



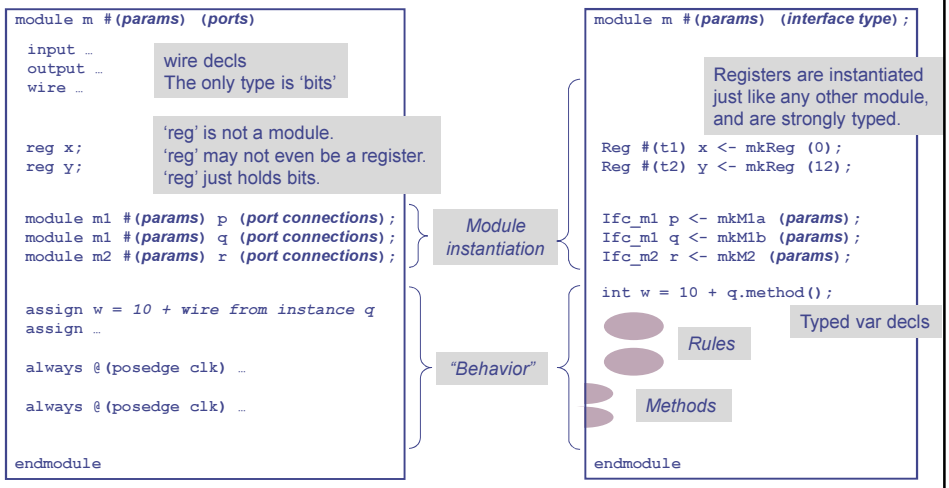
This further reinforces the idea that a method is just a piece of a rule, localized to a particular module. We effectively build rules from modular pieces.

bluespec

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)



bluespec

HW intuitions for interface methods

Although BSV code can be understood purely in source-code terms (because its semantics are defined in source-code terms), it is also illuminating to see how interface methods are compiled into hardware by *bsc*.

Interface methods can be directly mapped to HW, just by looking at the types:

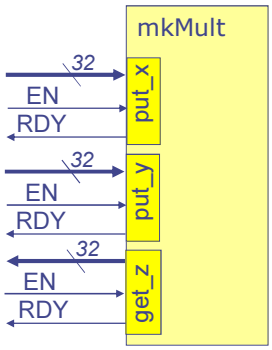
- Arguments become input signals (an input port) to the module
- Results become output signals (an output port) to the module
- The method condition becomes a "ready" (RDY) output signal
- Action and ActionValue methods have an "enable" (EN) input signal

bluespec

Mapping interfaces to HW: example

```
interface Mult_ifc;
   method Action put_x (int x);
   method Action put_y (int y);
   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)

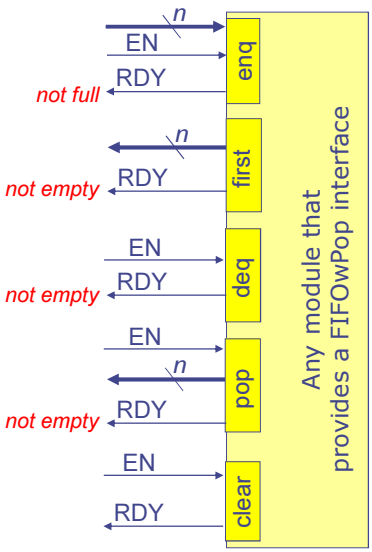


In later lectures we'll see that it is possible to eliminate the RDY signal when the method is always ready, and the EN signal when external rule invokes the method on every clock. Thus, Verilog ports are just a special case of BSV methods.

bluespec

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

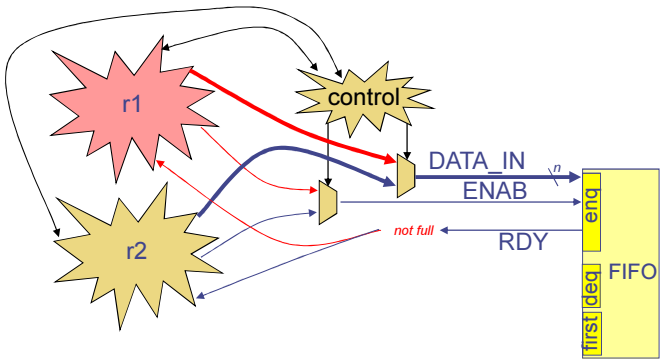
Sharing: two rules invoking a common method

```
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

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

bluespec

End

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