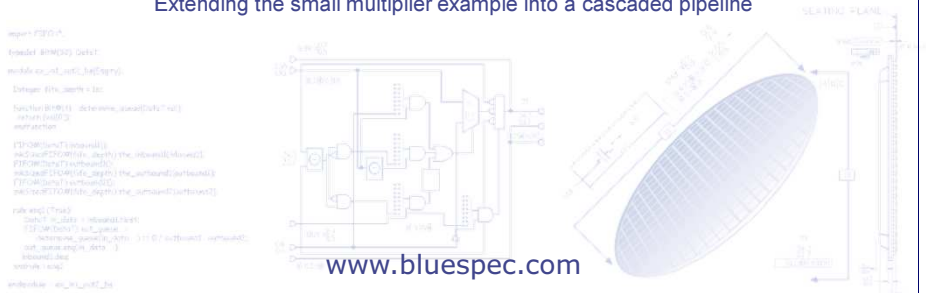
bluespec

BSV Training

Section: Concurrency

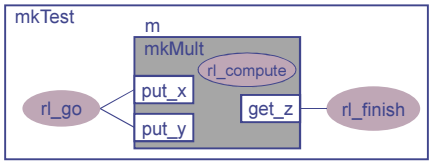
Extending the small multiplier example into a cascaded pipeline



www.bluespec.com

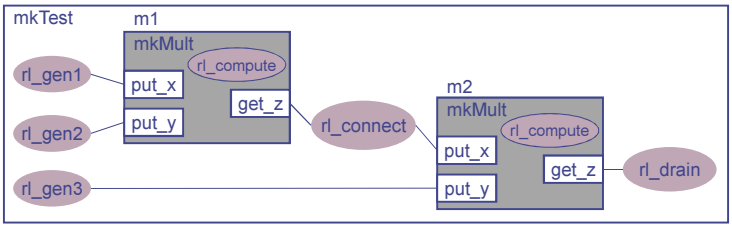
Extending example 1 to example 2, with concurrency

Example 1



Note: we use the same mkMult (twice), and only change mkTest

Example 2

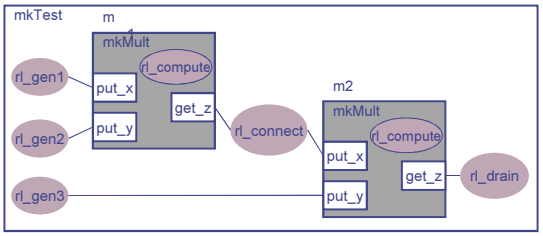


bluespec

Example 2: BSV code for module mkTest

```
(* synthesize *)
module mkTest (Empty);
   Mult_ifc m1 <- mkMult;
   Mult_ifc m2 <- mkMult;

   Reg #(int) rg_arg1 <- mkReg (1);
   Reg #(int) rg_arg2 <- mkReg (1);
   Reg #(int) rg_arg3 <- mkReg (1);

   rule rl_gen1;
      m1.put_x (rg_arg1);
      rg_arg1 <= rg_arg1 + 1;
   endrule

   rule rl_gen2;
      m1.put_y (rg_arg2);
      rg_arg2 <= rg_arg2 + 2;
   endrule

   rule rl_gen3;
      m2.put_y (rg_arg3);
      rg_arg3 <= rg_arg3 + 3;
   endrule
```

Example 2

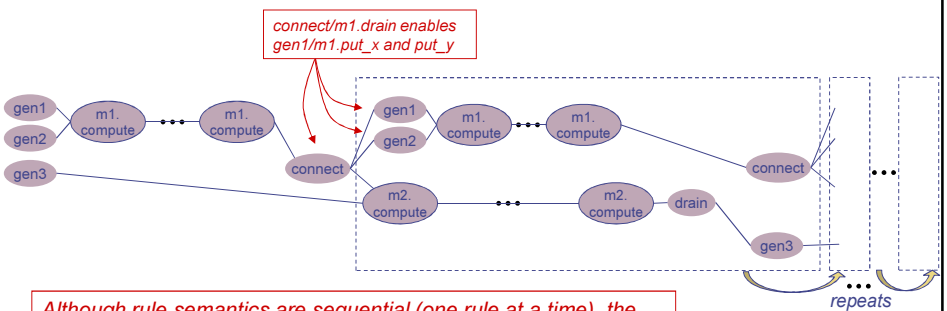



```
   rule rl_connect_m1_to_m2;
      let z1 <- m1.get_z ();
      m2.put_x (z1);
   endrule

   Reg #(int) rg_j    <- mkReg (0);

   rule rl_drain;
      let z2 <- m2.get_z ();
      $display ("Product [%0d]: %0d x %0d x %0d =
%0d",
                rg_j, rg_j+1, rg_j*2+1, rg_j*3+1, z2);
      if (rg_j == 10) $finish ();
      rg_j <= rg_j + 1;
   endrule
endmodule: mkTest
```


bluespec

Possible rule firing orders

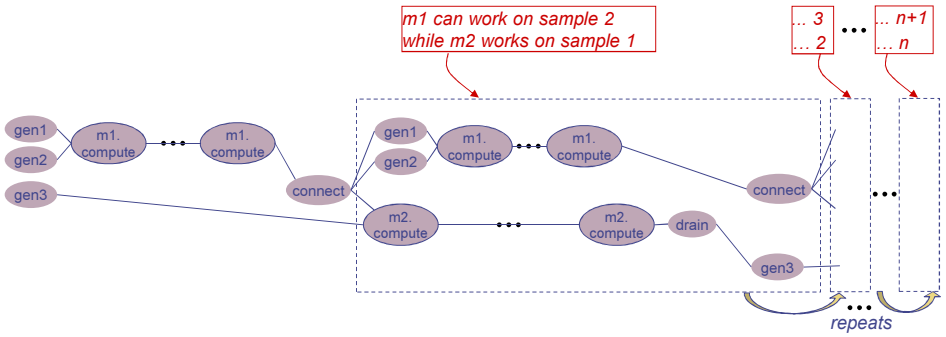
- In the diagram below, a line between two rules represent a precedence, i.e., the rule to the left enables some condition of the rule to the right
- This is a *partial order*:
  - A rule cannot fire until all rules connected to its left have fired
  - Two rules unconnected by lines (directly or indirectly) may fire in any order
- According to the rule semantics, any sequential order that is consistent with the partial order (i.e., any topological sort of this graph) is a possible rule firing order



Although rule semantics are sequential (one rule at a time), the partial order shown hints at how we can compile them into hardware that will execute multiple rules in each clock.
(it is analogous to packing multiple sequential instructions into a wide VLIW instruction whenever they are independent—i.e., whenever the dependence partial order allows)

bluespec

Observation: example2 could form a pipeline



This particular "pipeline" is very dynamic and elastic, since the number of steps taken by m1.compute and by m2.compute is highly data-dependent.

Method and rule conditions provide a simple and natural basis for flow-control (no danger of dropping or overwriting values, nor of race conditions).

(This not your classical pipeline that may be "balanced" in terms of latency of each stage, or that may move its data in synchronous lock-step; we'll see examples of those later.)

bluespec

High-Level Synthesis: mapping rules to clocked HW

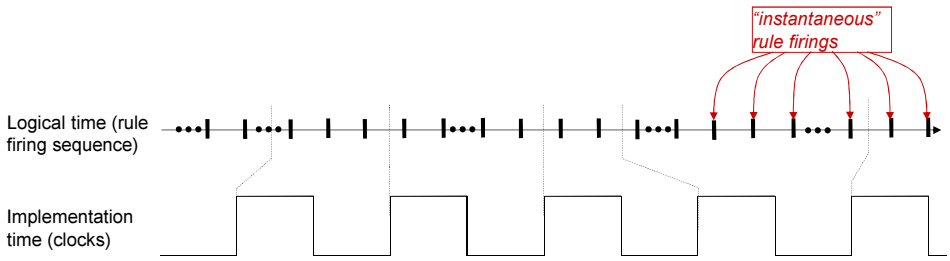
- *bsc* (the BSV compiler/synthesis tool) transforms BSV source code into traditional, clocked Verilog RTL
  - The RTL can be simulated in a Verilog simulator
  - The RTL is also further synthesizable into gates using existing tools like DC (ASIC)and XST (FPGA)
  - (*bsc* can also produce a directly compiled simulation, called Bluesim. Actually we mostly debug using Bluesim, and often skip Verilog sim completely.)
- Although we may execute Verilog, we always think of execution (and debug) in BSV source terms (sequence of rule firings, BSV data types)
  - A tool (Bluespec Development Workstation, BDW) provides a source-level view, even though you may be executing Verilog or Bluesim
- Analogy:
  - We compile C code to assembly/machine code
  - We actually execute machine code
  - But we think (and debug) in terms of C source-level statements, variables and types
  - Tools (like gdb) preserve this illusion

It is very important to keep this distinction between "logical" semantics (BSV rules) and "implementation semantics" (Verilog RTL).

bluespec

Mapping rules to clocks

bsc-compiled hardware logically executes multiple rules per clock:



The number of rules that fire will vary from clock to clock

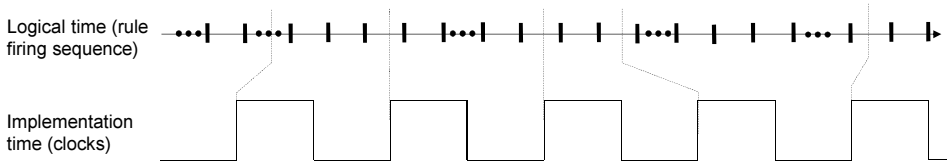
- because the set of eligible rules (rule conditions true) will vary
- because of *conflicts* between eligible rules

Terminology:

- Actions within a rule happen *simultaneously* (at a logical instant)
- Multiple rules in a clock cycle execute *concurrently*
- (conversely, we never use *concurrently* to refer to multiple actions in a rule, and we never use *simultaneously* to refer to multiple actions in a rule

bluespec

Mapping rules to clocks



For maximum performance we may wish to execute as many rules as possible within a clock. However, hardware properties will always limit what we can reasonably do within a clock cycle. Key questions:

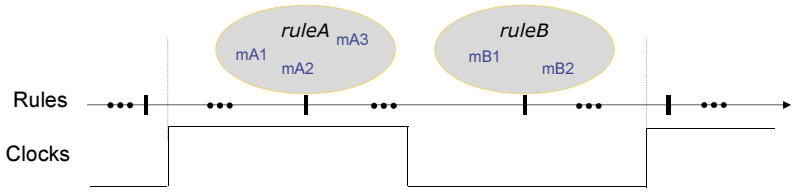
- What rules can execute concurrently (within a clock)?
- With what *schedule* (in what logical rule firing order)?

Note: there is no new semantic question—the meaning of a program execution is still explained in terms of a logical sequence of rule firings.

bluespec

Scheduling constraints

In BSV, we formalize hardware-inspired constraints into a simple semantic model of *scheduling constraints on pairs of methods*.



ruleA (invoking methods mAi) and ruleB (invoking methods mBj) may fire in the same clock, and in the order shown, if:

- (of course) both their conditions (CAN_FIRE) are true
- All *constraints* between each method mAi and method mBj are satisfied

Every primitive module comes with the constraints on each pair of its methods:

Constraint	Meaning
mA *conflict_free* mB	ruleA and ruleB can be concurrent, in either order
mA < mB	ruleA and ruleB can be concurrent, with ruleA before ruleB
mB < mA	ruleA and ruleB can be concurrent, with ruleB before ruleA
mA *conflict* mB	ruleA and ruleB can never be concurrent

bluespec

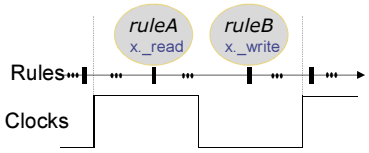
Method schedule constraints

Where do these scheduling constraints on methods of primitive modules come from? From practical HW considerations (ease of mapping into HW).

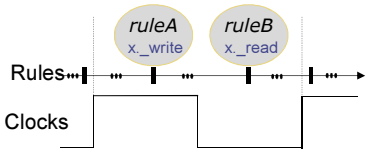
E.g., for the register primitive, a scheduling constraint is: _read < _write

One can see that this maps easily into hardware: during a clock, we can only read the old value (from previous clock edge) and when we write, it is only visible at the next clock edge.

Allowed by method constraints, and maps naturally to HW semantics



Disallowed by method constraints. Making the effect of the _write visible to the _read, within a clock, would involve more complex HW (bypassing)



bluespec

Method schedule constraints

Where do these scheduling constraints on methods of primitive modules come from? From practical HW considerations (ease of mapping into HW).

Another reason why a primitive may have a scheduling constraint: *resource limitation*

E.g., a register file with one read port:

```
interface RegFile #(type index_t, type data_t);
   method Action upd(index_t addr, data_t d);
   method data_t sub(index_t addr);
endinterface: RegFile
```

The primitive has a constraint:

sub *conflict* sub

In HW, the module has one set of input wires to carry the "addr" argument of "sub". These wires can only carry one set of values during a clock.

The "conflict" constraint prevents two rules using "sub" on the same regfile from being scheduled into the same clock.

bluespec

Method schedule constraints

To summarize about method scheduling constraints on primitive modules:

- Different HW primitive modules have different considerations about whether more than one of its interfaces can be operated within a clock, and what is the meaning of such same-clock invocation (orderings, resource constraints, etc.)
- BSV abstracts and formalizes these diverse considerations into a simple, uniform model, method constraints, that directly translate into the simple concept of rule orderings.

bluespec

From method constraints to rule orderings

bsc (the BSV compiler/synthesis tool) converts BSV source code into Verilog RTL.

Conceptually, the synthesized circuits do the following. On each clock,

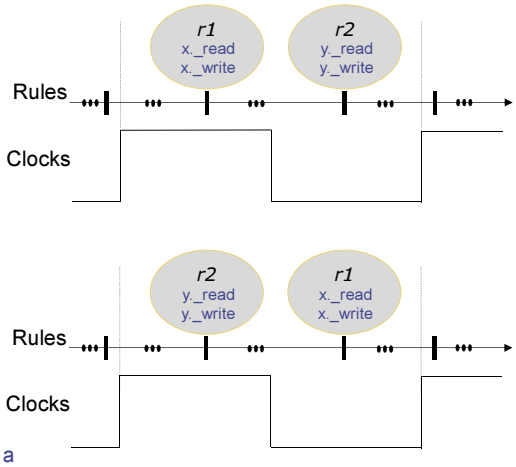
- all rule conditions are evaluated to see which ones are eligible (CAN_FIRE)
- among the eligible rules, a subset is allowed to fire (equivalently, the complementary subset is disallowed to fire) by suitable gating of their action "enables". Always, the subset that fires will form a legal ordering with respect to the scheduling constraints.

- Thus, the net state transition for each clock can *always* be understood in terms of the logical linear schedule of rules that fired in the clock
  - i.e., we can think (and debug) in BSV source code terms

bluespec

Example: no conflict

```
rule r1;
   x <= x + 1;
endrule

rule r2;
   y <= y + 2;
endrule
```

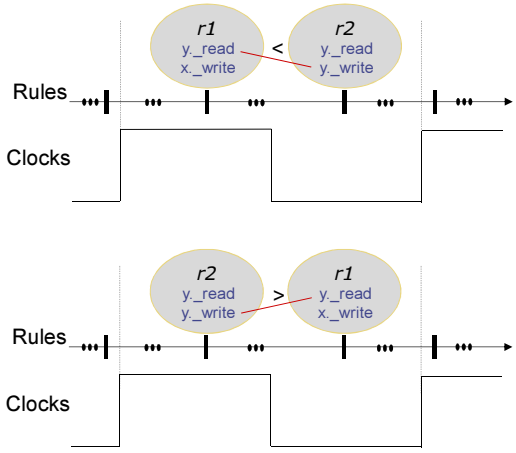


There are no constraints between a method of x and method of y, so both rule orderings are legal. bsc will synthesize hardware that allows both rules to execute in the same clock.

bluespec

Example: no conflict

```
rule r1;
   x <= y + 1;
endrule

rule r2;
   y <= y + 2;
endrule
```

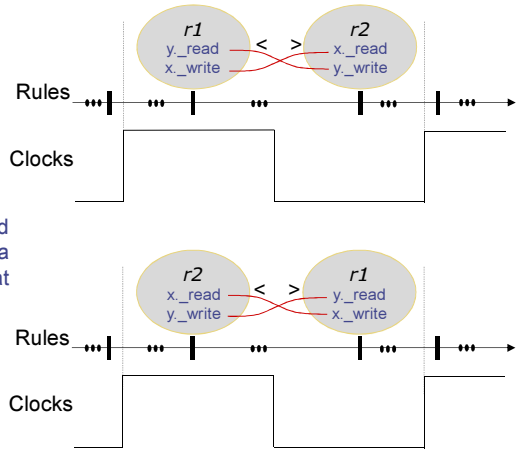


The only constraint is: y._read < y._write
(there are no constraints between methods of different registers x and y).
The upper ordering (or schedule) is consistent with this, the lower schedule is not. *bsc* will synthesize HW that allows concurrent execution of r1 and r2, with the ordering r1 < r2.

bluespec

Example: conflict

```
rule r1;
   x <= y + 1;
endrule

rule r2;
   y <= x * 2;
endrule
```



In both possible orderings, a method constraint is violated. This is also called a conflict, and *bsc* will generate HW that prevents concurrency for these rules.

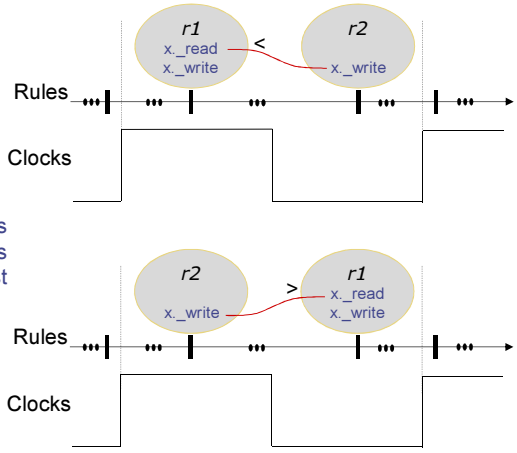
Compare: Lecture 2, slide 15 had the same two actions in a single rule, which was ok!
There, the actions were simultaneous (truly parallel).
Here, they are concurrent (logically sequential).

bluespec

Example: no conflict

```
rule r1;
   x <= x + 1;
endrule

rule r2;
   x <= 23;
endrule
```

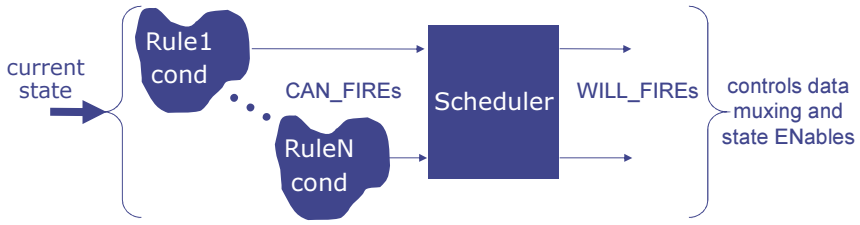
The first ordering is legal, the second is not. *bsc* will generate HW that allows concurrency for these rules, with the first ordering.



This example sometimes surprises HW designers since it appears we're doing two assignments to x in the same clock. But remember that that is in the logical semantics. bsc can synthesize hardware that does just one physical assignment, but whose net effect is the same as the two assignments in the logical semantics (by ignoring the first one).

bluespec

CAN_FIRE and WILL_FIRE signals in synthesized HW

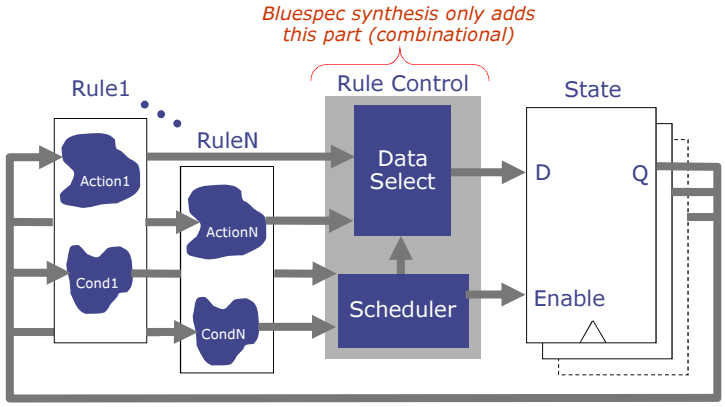


- The compiler performs conflict analysis of all rules in a BSV program, and generates a corresponding HW scheduler (purely combinational)
- The CAN_FIRE signal of a rule = its rule condition along with the conditions of methods that it invokes, directly or indirectly
- The WILL_FIRE signal of a rule = 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 "scheduler" is not a monolithic circuit; it is distributed across modules and is built incrementally

bluespec

Overall schematic of synthesized HW



- The scheduler 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

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
```

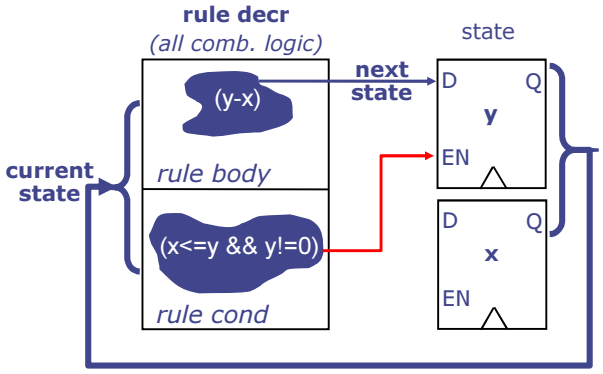
Answer:

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

bluespec

HW for one of the rules

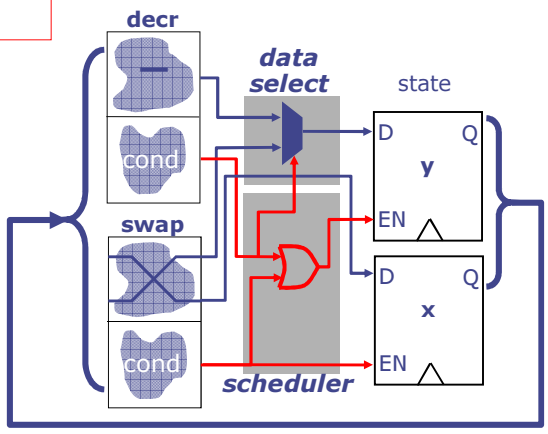
```
rule decr ( x <= y && y != 0 );
    y <= y - x;
endrule : decr
```



bluespec

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

bluespec

End

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