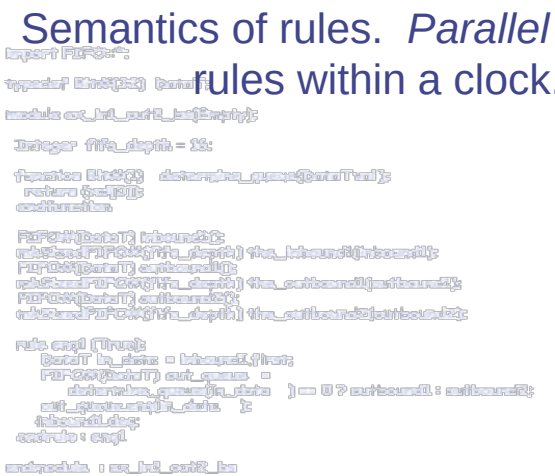
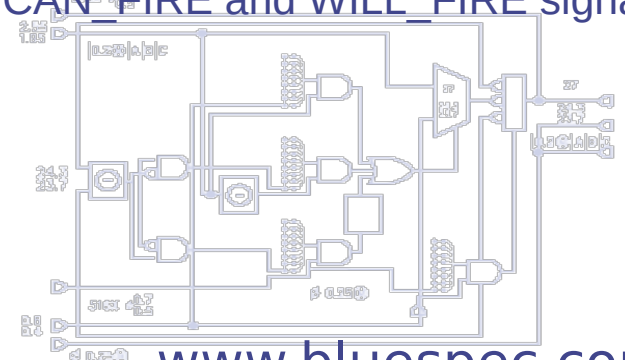
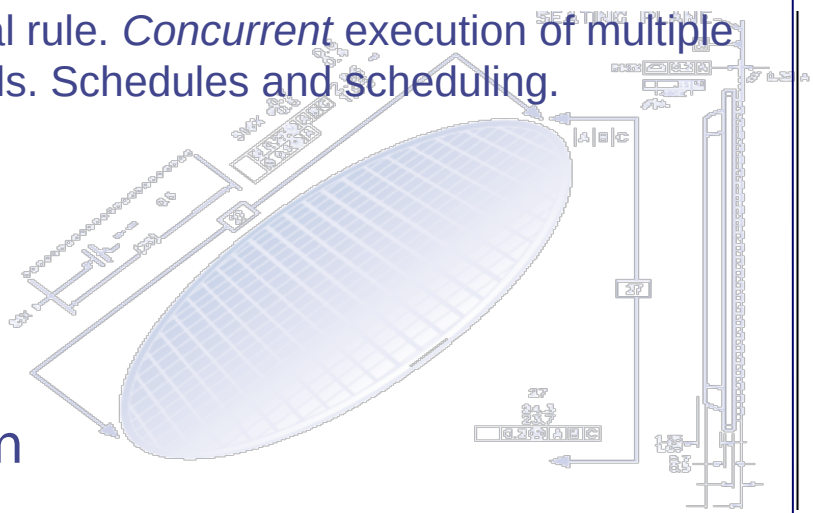
Bluespec Training

Lec_Rule_Semantics

Semantics of rules. *Parallel* execution of actions in an individual rule. *Concurrent* execution of multiple rules within a clock. CAN_FIRE and WILL_FIRE signals. Schedules and scheduling.





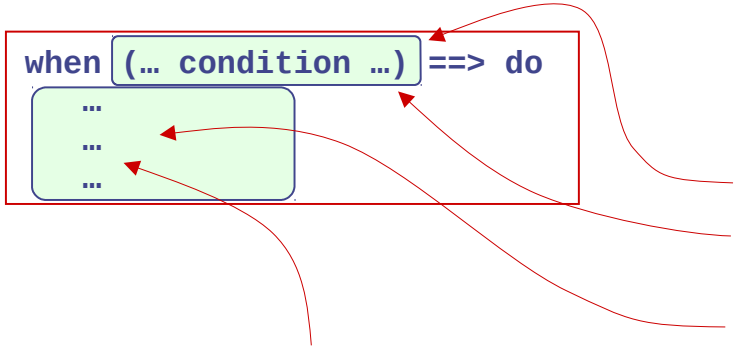


www.bluespec.com

Rule execution semantics, first approximation

(This is just a first approximation; more detail to follow)

Every rule has a rule name, and two semantically significant parts:



CAN_FIRE condition: a value of type **Bool**, representing the conjunction (&&) of:

- the explicit rule-condition
- the method-conditions of any method invoked in the rule-condition, and
- the method-conditions of all methods invoked in the rule-body.

Composite Action: the collection of all the Actions invoked in the rule-body.

To first approximation, a Bluespec program can be simulated by:

> whenever a rule's CAN_FIRE is True,
> do the rule-body actions

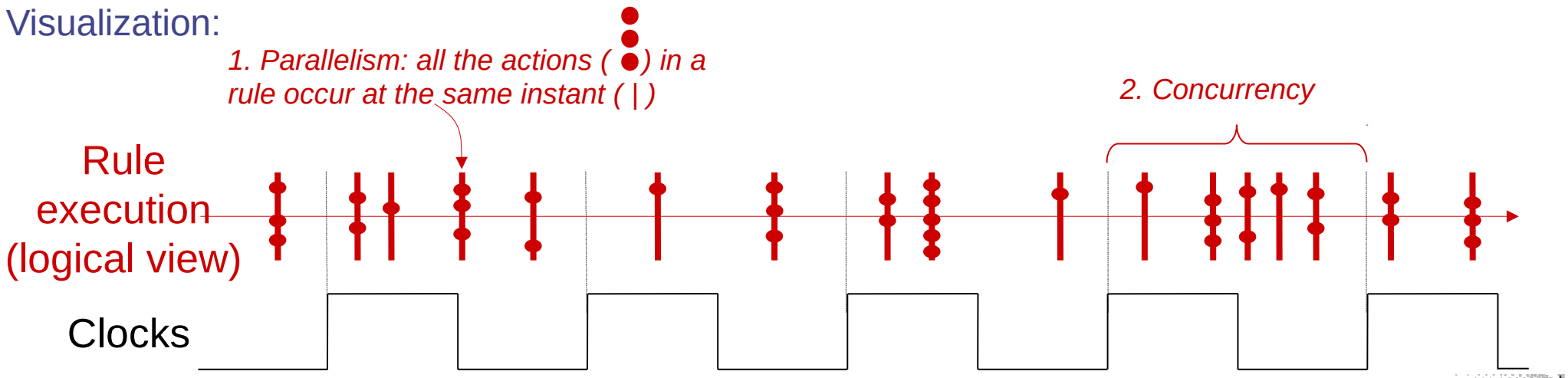
Rule semantics

Rule semantics are explained in two parts:

1. Individual rule: logically, a rule fires in a single instant. All the actions in the rule (and any methods it calls) happen at the same instant. We call this *parallelism of actions.*

2. Multiple rules within a clock: logically, they execute in sequence, so the overall state change in a clock can be understood as the simple sequential composition of the state changes of the individual rules. We call this the *concurrency* of rules.

Visualization:



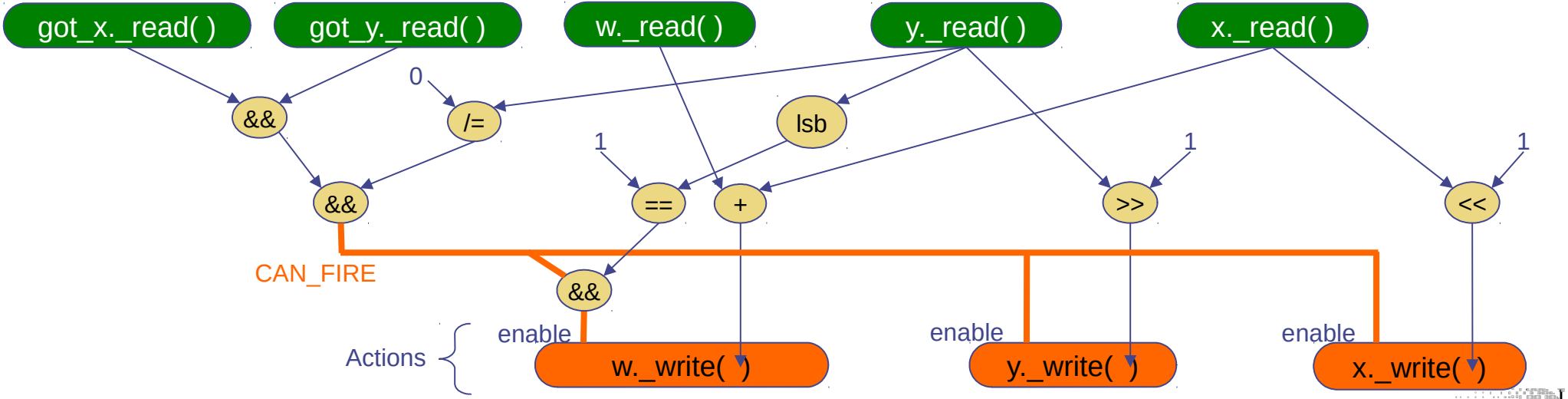
1. Parallelism: semantics of an individual rule, involving the simultaneous, instantaneous execution of all its “Actions”

Semantics of a single rule (parallelism of its actions)

Every rule can be regarded as a flow of data from constants and outputs of methods, through functions and operators, ending in inputs to Action methods (this flow is always a combinational circuit).

Example: below is a visualization of the flow for the rule shown at right

```
when ((y /= 0) && got_x && got_y) ==> do
   if1 (lsb(y) == 1) w := w + x
   x := x << 1
   y <= y >> 1
```



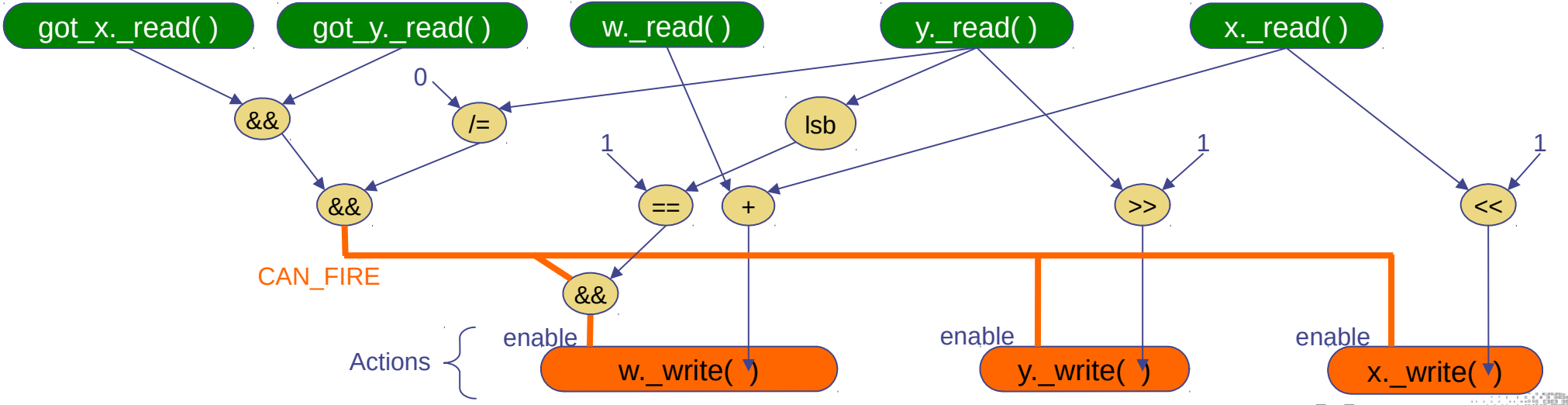
Semantics of a single rule (parallelism of its actions)

Every Action method has an “enable” input: when True, the action is performed, using the argument values on its remaining inputs (if any).

The semantics of an individual rule are simple:

- (Simultaneously) for each of the rule’s actions whose “enable” is True, do the action

Note: as illustrated below, if the CAN_FIRE condition of the rule is False, no Action will be enabled.
Further, for w.write() to be enabled, the condition **`(lsb(y) == 1)`** must also be true.

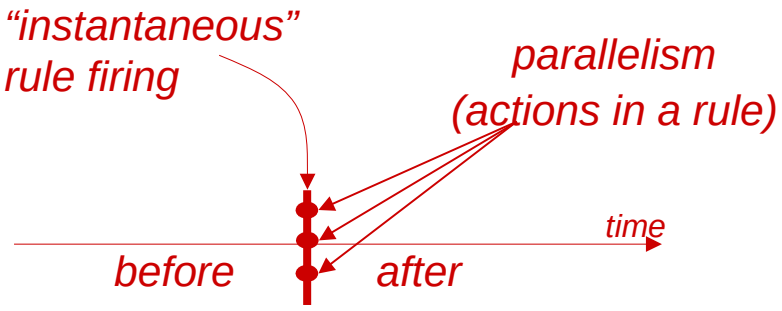


Semantics of a single rule (parallelism of its actions)

Some important points to observe and remember:

- There is no ordering among the actions (their textual order in the rule is not relevant). We truly think of the actions as "simultaneous"
- Because of conditionals inside a rule, not all its actions may be performed (e.g., w._write)

- Any values "read" by the rule are from "before the firing instant" (from previous rules)
- Any values "written" by the rule are only visible "after the firing instant" (for later rules)



All actions in a rule are simultaneous

To emphasize that all actions in a rule are simultaneous, note:

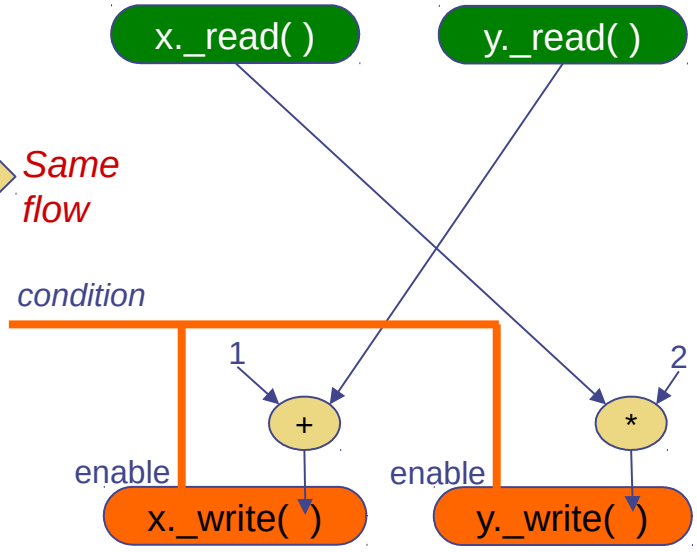
- The textual ordering of actions in a rule is not important
- You can even exchange values in two registers

```
"rule2a": when (condition) ==> do
   x := y + 1
   y := x * 2
```

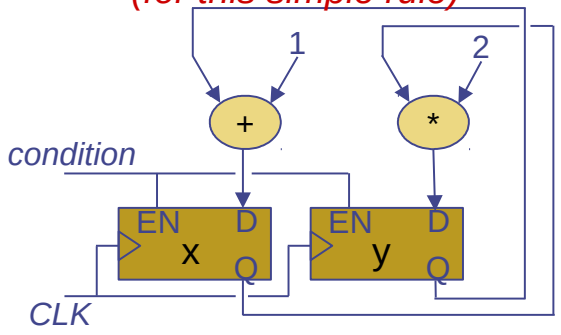
Textual order of actions is irrelevant

```
"rule2b": when (condition) ==> do
   y := x * 2
   x := y + 1
```

Same flow



Some HW intuition (for this simple rule)



Not all actions can be combined within a rule

```
when (…) ==> do
  valuea := expr1
  valuea := expr2
  …
```

Cannot instantaneously update a register with two values

```
when (…) ==> do
  fifo.enq  23
  fifo.enq  34
  …
```

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

```
when (…) ==> do
  let x = regFile.read 5
      y = regFile.read 7
  …
```

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 different register, FIFO or regFile modules that have *multiple* ports that can be accessed simultaneously, in which case those simultaneous accesses can be used in a single rule.

2. Concurrency: semantics of multiple rules within a clock (while preserving a logical sequential order)

Overview of rule concurrency

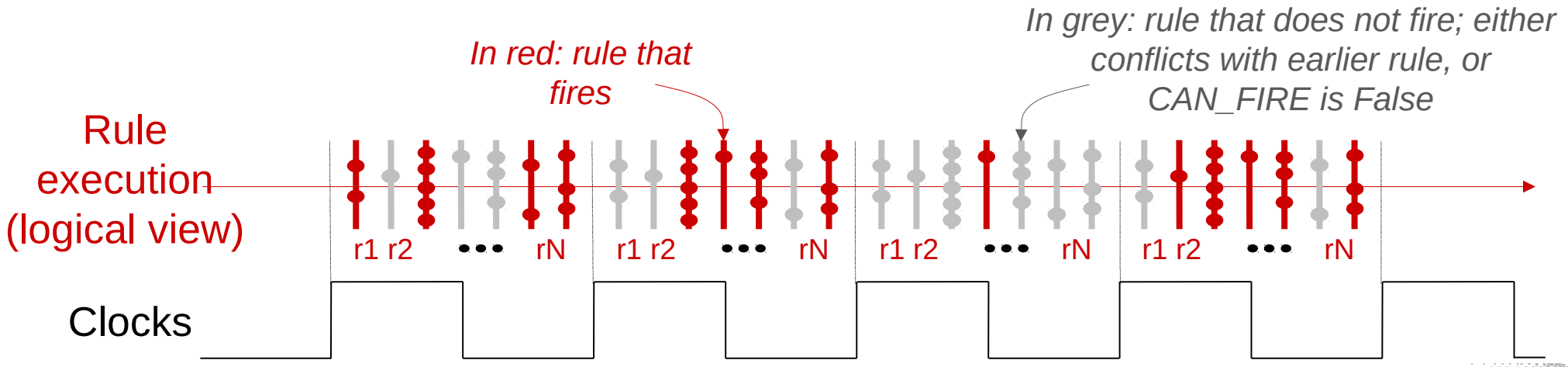
Define a *schedule* as some linear ordering of all rules in a program: r1 r2 ••• rN

Then, the semantics of multiple rules in a clock (logical concurrency) is simple:

For each clock:

> Consider each rule *in order*
>
> If rJ does not *conflict* with earlier rules (r1..rJ-1)
>
> Execute rJ according to the per-rule semantics
>
> (i.e., if its CAN_FIRE is true, do its actions)

(we'll discuss conflicts shortly)



Method Ordering Constraints (inducing *conflicts*)

Consider two rules rule1 and rule2, in that order.

Consider two method calls x.mA in rule1, and x.mB in rule2, on a common module x. These methods can be either in the condition or body or rule1 and rule2.

For every module (x), the compiler knows certain ordering constraints on its methods:

Constraint	Meaning
mA *conflict_free* mB	Rules invoking mA and mB can fire concurrently (either order)
mA < mB	Rules invoking mA and mB can fire concurrently, provided the rule invoking mA is earlier than the rule invoking mB
mB < mA	Rules invoking mA and mB can fire concurrently, provided the rule invoking mB is earlier than the rule invoking mA
mA *conflict* mB	Rules invoking mA and mB cannot fire concurrently (neither order)

For primitive modules the ordering constraints on its interface methods are built-in (“given”).

For a user-defined module, the ordering constraints on its interface methods are inferred by *bsc*, the compiler, when the module is compiled.

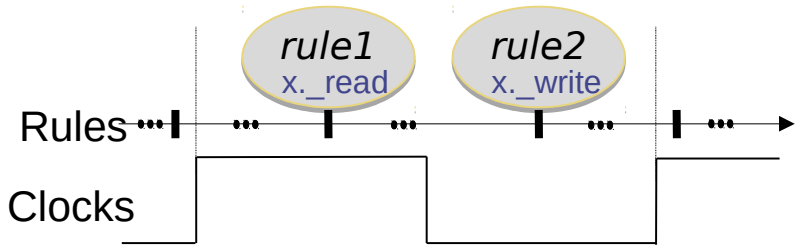
Example: method ordering constraints on registers

For the mkReg register primitive, a method ordering constraint is: _read < _write

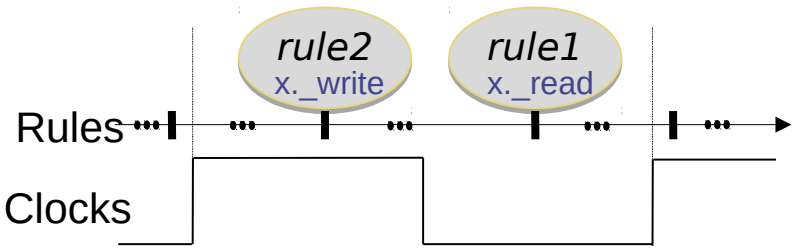
(One can see that this maps easily into hardware: during a clock, we can only read the old value (from the last clock edge) and when we write, it is only visible after the next clock edge.)

These constraints may result in a conflict for some rule schedules.

No conflict: rule order is consistent with method ordering constraint (and maps naturally to HW semantics)



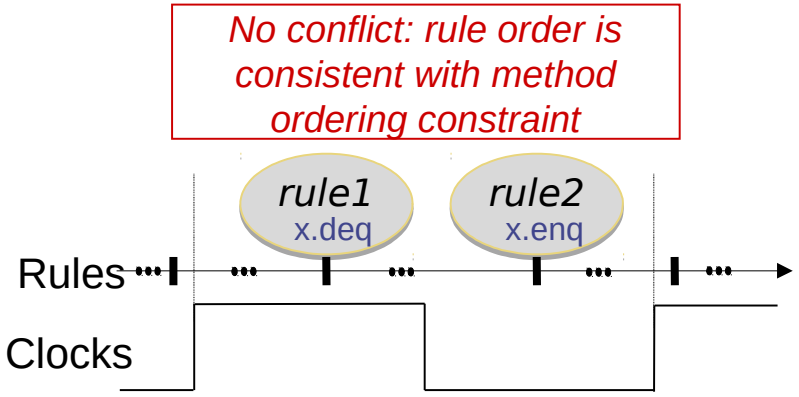
Conflict: rule order is inconsistent with method ordering constraint

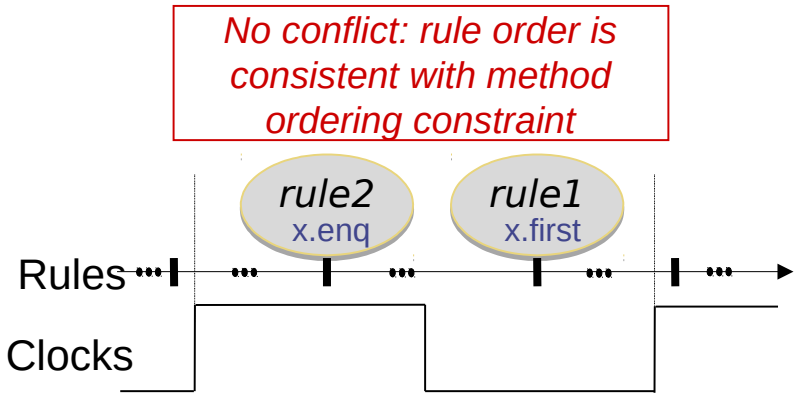


Example: method ordering constraints for FIFOs

The mkFIFO primitive has this method ordering constraint:

{deq, first} *conflict_free* enq



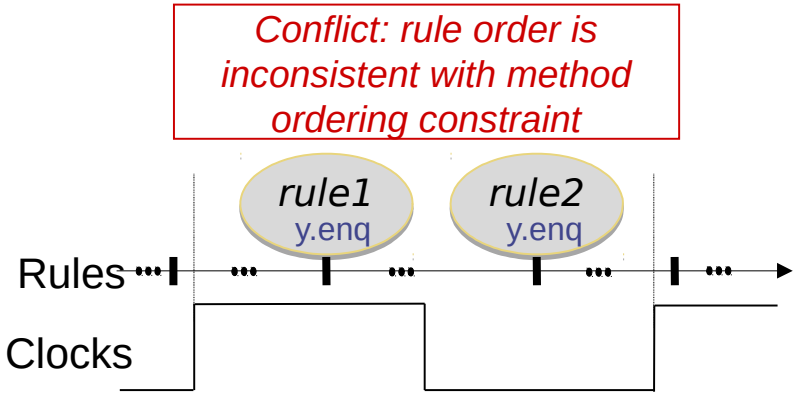


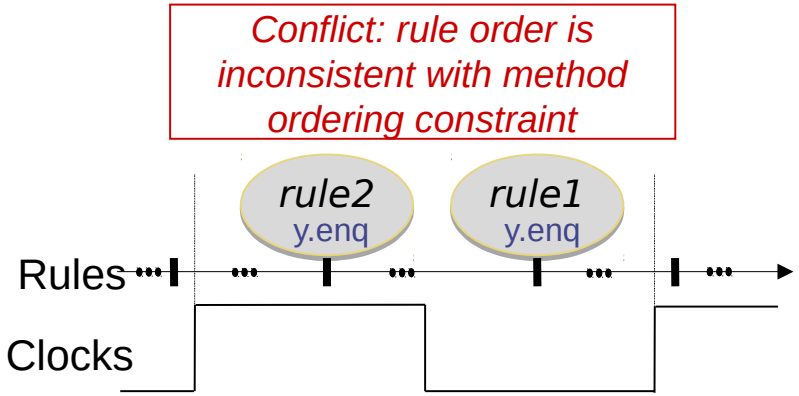
Example: method ordering constraints for FIFOs

The mkFIFO primitive also has this method ordering constraint:

enq *conflict* enq

(Again, one can imagine the hardware considerations underlying this constraint: in a single-port FIFO, at most one item can be enqueued in each clock.)

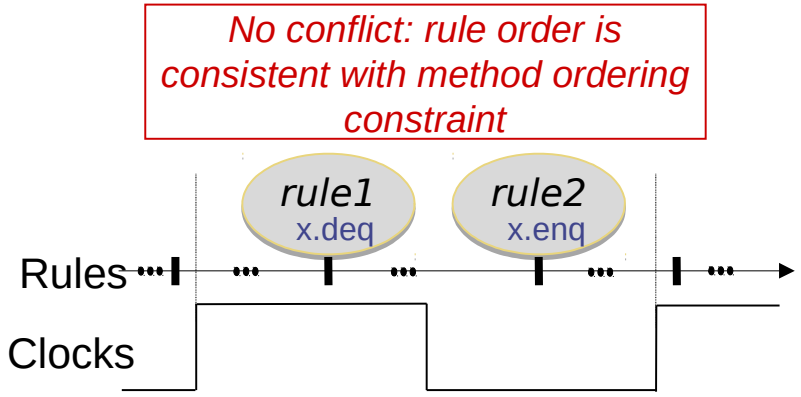


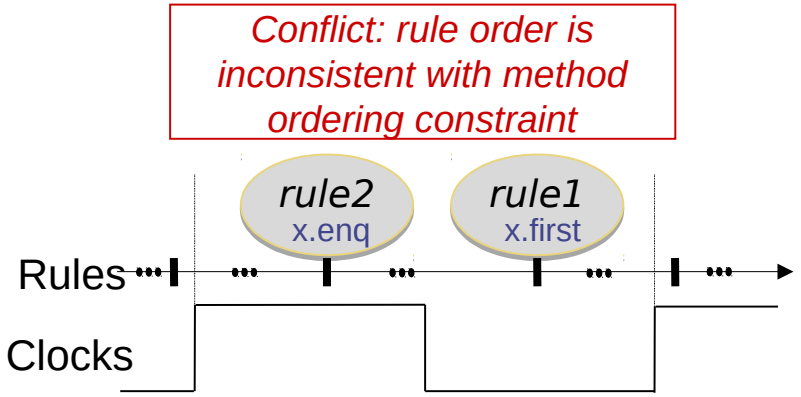


Example: method ordering constraints for FIFOs

The mkPipelineFIFO primitive (which provides the same interface as mkFIFO, but has a different implementation) has this method ordering constraint:

{deq, first} < enq



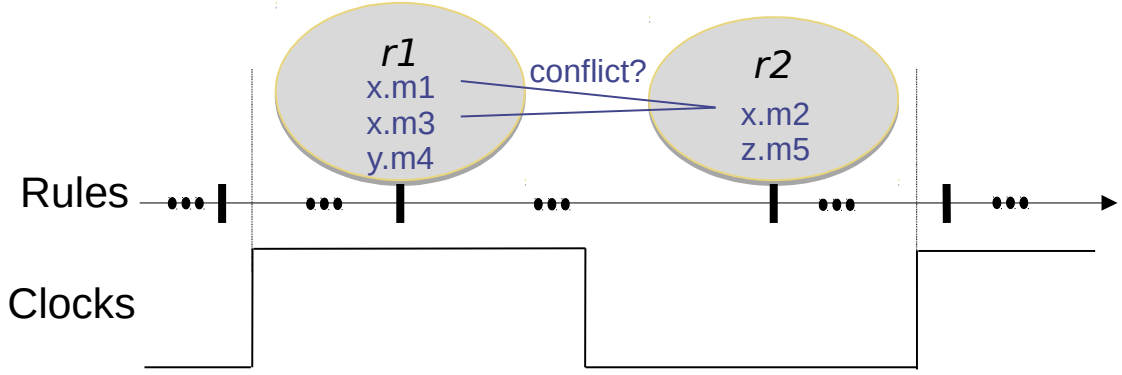


From method ordering to rule ordering

In the past few slides, we have focused on some particular pair of methods in two rules that are arranged in some particular order.

Of course, rules typically invoke several methods, often on several sub-modules.

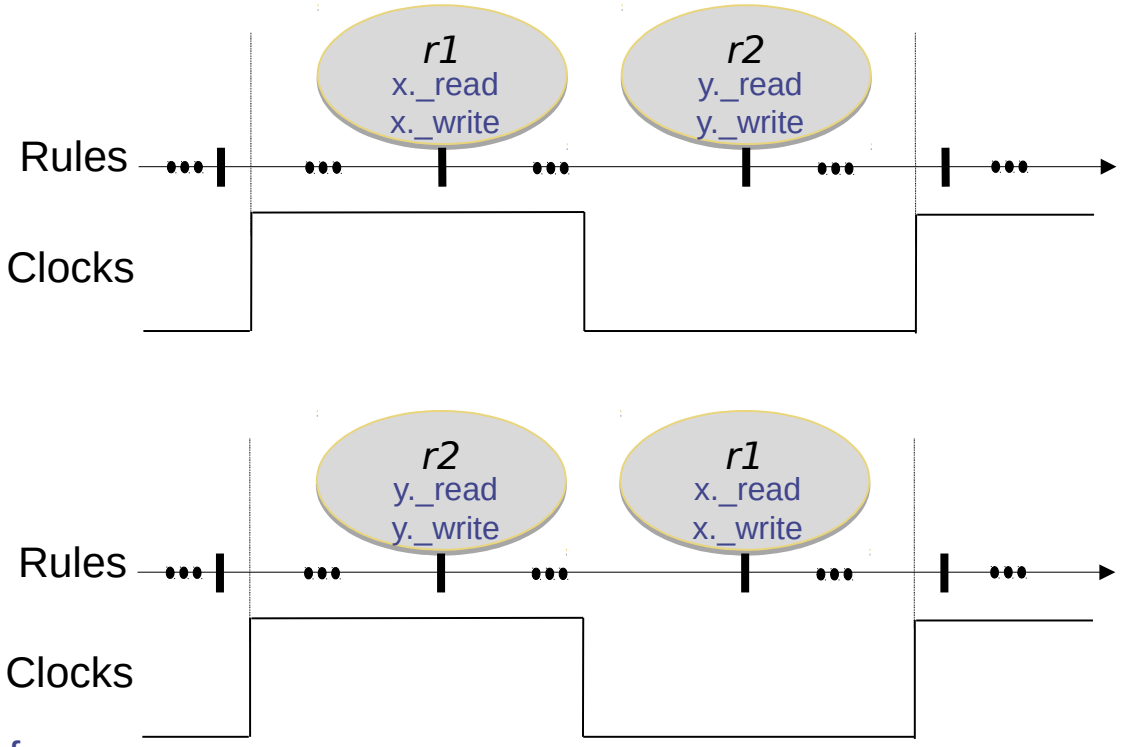
We say that there is a conflict between an ordered pair of rules r1 and r2
if there is a conflict between *any* pair of methods x.m1 in r1 and x.m2 in r2.



Note: conflicts and orderings only arise between methods of the *same* module (like x.m1 and x.m2), not between methods of different modules (like y.m4 and x.m2)

Example: no conflict

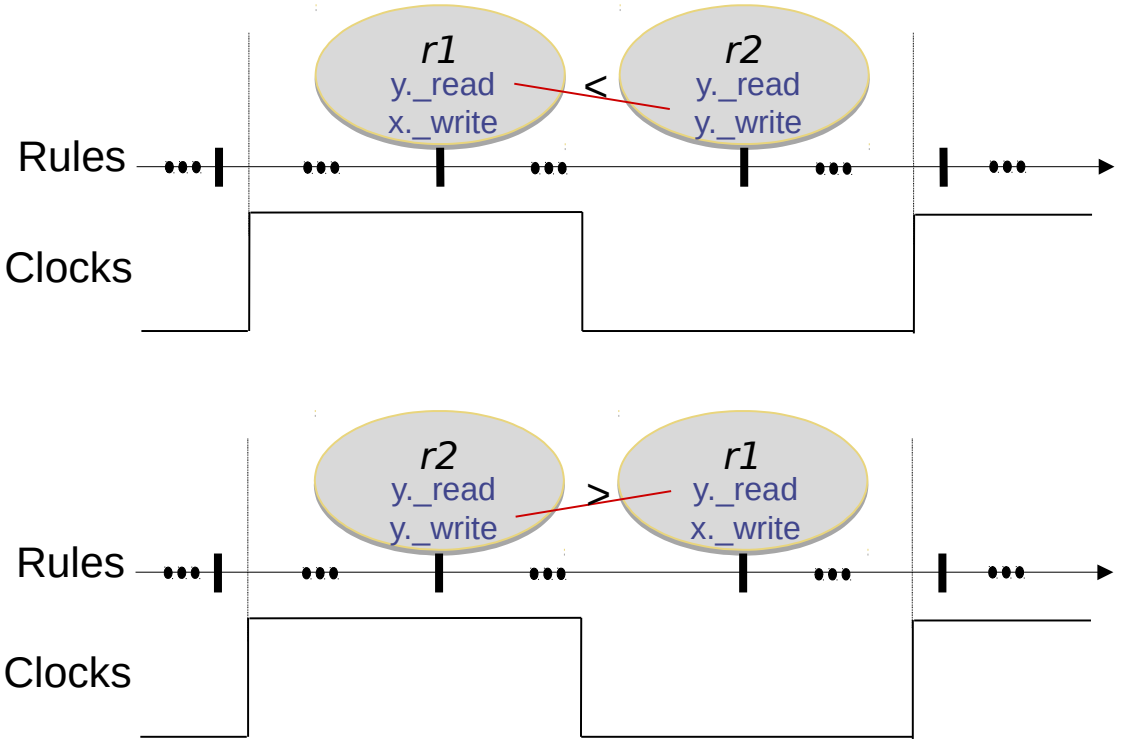
```
"r1": when True ==> do
   x := x + 1

"r2": when True ==> do
   y := y + 2
```



There are no constraints between a method of x and method of y, so both rule orderings are legal (no conflict).

Example: potential conflict

```
"r1": when True ==> do
   x := y + 1

"r2": when True ==> do
   y := y + 2
```

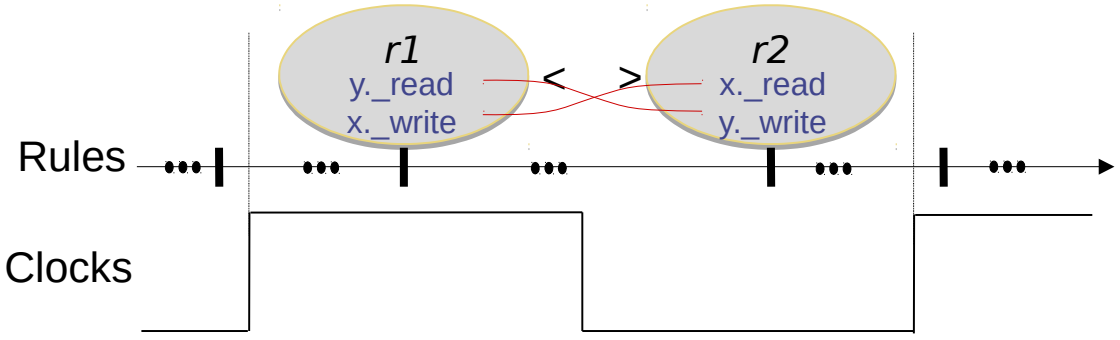


The only relevant constraint is: y._read < y._write
(there are no constraints between methods of different registers x and y).

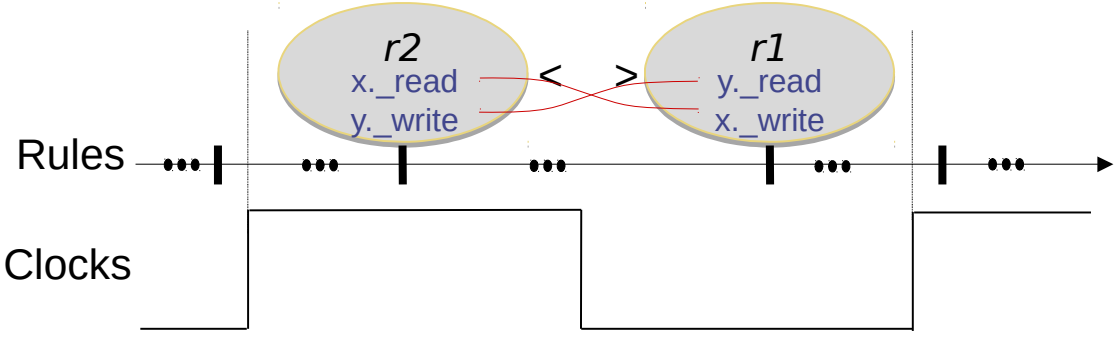
The upper ordering (or schedule) is consistent with this (no conflict).
The lower ordering (or schedule) is inconsistent with this (has a conflict).

Example: conflict

```
"r1": when True ==> do
    x <= y + 1

"r2": when True ==> do
    y <= x * 2
```



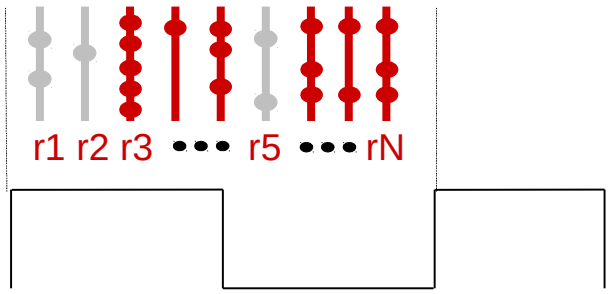
In both possible orderings, a method constraint is violated. This is a conflict.



Compare slide 8: we had the same two actions in a single rule, and it was ok!
This illustrates that some Actions may be simultaneous (in a single rule) but not concurrent (logically sequential in two different rules).

For conflicts, the earlier rule disables the later rule

A schedule (i.e., a particular linear ordering of rules) can contain conflicts.
A conflict between two rules in a schedule just means that the latter rule must be suppressed (not allowed to fire) if the earlier one fires.



Suppose the compiler has picked this schedule of rules.
In this schedule, suppose there is a conflict between r3 and r5.

Then it generates logic like this: WILL_FIRE_r5 = (! WILL_FIRE_r3) && CAN_FIRE_r5

i.e., on those cycles that r3 fires, r5 is disabled.
Rule r5 can fire only on those cycles when r3 does not fire.

Choice of schedule

Recall:

Define a *schedule* as some linear ordering of all rules in a program: r1 r2 rN

There are N! (factorial N) possible linear orderings of all the rules: which one does *bsc* pick?

bsc chooses a schedule trying to maximize rule concurrency (minimize conflicts).

This is based on a sophisticated analysis of the rules, the methods they invoke, the ordering constraints on the methods, and the boolean expressions representing rule conditions, method conditions, conditional statements and conditional expressions.
Occasionally you will see a compile-time informational message from bsc when It has to make an “arbitrary” ordering choice between two rules.

Logical semantics and implementations

We have described Rule Semantics purely in Bluespec source-code terms,
i.e., independent of any particular implementation, whether in Bluesim, Verilog simulation, or actual silicon.

This is how we always think about Bluespec programs, and this is how we debug them.

This separation of *logical* view and *implementation* view is present throughout Computer Science and Engineering. Example:

- *Logical* view: an assembly language (x86, ARM, … your favorite processor). Semantics are defined as a sequential execution, 1 instruction at a time. Also, this is how we debug, for example using gdb.
- *Implementation* view: what happens in a real CPU: pipeline parallelism, out-of-order execution, superscalar execution, branch prediction and speculation, … Further, these techniques will vary from one implementation to the next.

Similarly, rule orderings may not be evident in the Verilog generated by *bsc* from a Bluespec program. However, the implementation behavior will be identical to the logical semantics.

Rule semantics summary

All rules in a program are considered in a linear order called “the schedule”

On each clock, a rule fires if its CAN_FIRE condition is true, and if it does not conflict* with any earlier rule.

When a rule fires, it is logically at a single instant. All actions within the rule take place at that same instant.

(*) A conflict between two rules exists if the rule order violates any ordering constraint between methods called in the two rules.

Controlling rule scheduling

Controlling rule scheduling

Recall:

There are N! (factorial N) possible linear orderings of all the rules: which one does *bsc* pick?

bsc chooses an ordering trying to maximize rule concurrency (minimize conflicts).

The user can influence *bsc*'s choices with various *attributes* to control scheduling

- BSV attribute syntax (same as SystemVerilog) :

(* *attribute* = "*rule and method names*" *)

- These are written in a module, typically just before the rules mentioned in the attribute.
- Since methods are just rule fragments, these attributes can mention methods as well

Controlling scheduling: rule urgency

BSV notation

```
(* descending_urgency = “r1, r2” *)

rule r1 (c1);
   fifo.enq (e1);  // one enq per cycle
endrule

rule r2 (c2);
   fifo.enq (e2);  // one enq per cycle
endrule
```

Bluespec Classic notation

```
let r1 = rules
           when (c1) ==> do
             fifo.enq  e1

    r2 = rules
           when (c2) ==> do
             fifo.enq e2

addRules (rJoinDescendingUrgency r1 r2)
```

- Urgency is the order/priority in which WILL_FIRE conditions are computed
- In this example, r1 and r2 conflict because of ‘fifo.enq()’
  If r1 WILL_FIRE, we will suppress r2 (even if it CAN_FIRE)
- If the user does not specify urgency between conflicting rules, the compiler will pick an urgency order and notify the user

Controlling scheduling: rule preempts

BSV notation

```
(* preempts = “r1, r2" *)

rule r1 (upA);
   x <= x + 3;
endrule

rule r2;
   y <= y + 1;
endrule
```

Bluespec Classic notation

```
let r1 = rules
           when (upA) ==> do
             x := x + 3

    r2 = rules
           when (True) ==> do
             y := y + 1

addRules (rJoinPreempts r1 r2)
```

- This forces one rule’s firing to suppress another rule’s firing even if there is no conflict between the rules
  - This is equivalent to *forcing* a conflict between two rules
- In this example, whenever r1 fires the scheduler will suppress r2 (even if it CAN_FIRE)
  - For example, here y effectively counts “idle cycles” of r1

Controlling scheduling: rules mutually exclusive

BSV notation

```
(* mutually_exclusive = "updateBit0, updateBit1" *)

rule updateBit0 (oneHotNumber[0] == 1);
  x[0] <= 1;
endrule

rule updateBit1 (oneHotNumber[1] == 1);
  x[1] <= 1;
endrule
```

Bluespec Classic notation

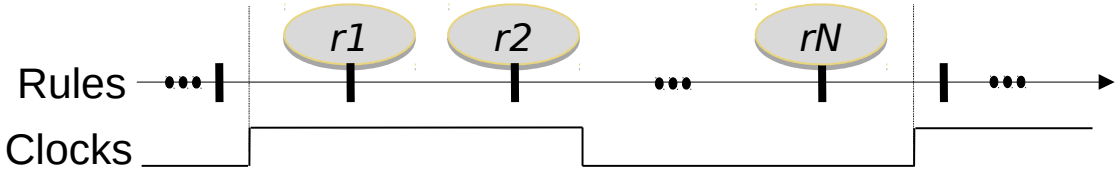
```
let r1 = rules
           when (oneHotNumber [0] == 1) ==> do
               (x !! 0) := 1

   r2 = rules
           when (oneHotNumber [1] == 1) ==> do
               (x !! 1) := 1

addRules (rJoinMutuallyExclusive  r1  r2)
```

- Asserts to the compiler that two rules' conditions are mutually exclusive (will never simultaneously be true in any clock)
- When rules are mutually exclusive, the compiler can generate better HW
  - E.g., simple muxes instead of priority muxes
- The compiler does sophisticated Boolean analysis to try to prove that two rule conditions are mutually exclusive
  - However, this question is undecidable in general. E.g.,
    - The conditions depend on external inputs
    - Mutual exclusivity depends on application-specific semantic knowledge (such as "one-hotness" of a bit-vector)
  - This assertion helps the compiler, when it is unable detect mutual exclusivity
  - For simulation, the compiler also generates code to verify mutual exclusivity

The last topic in this lecture (subtle distinction between “urgency” and “earliness”) can be skipped on first reading. It is a subtlety that matters only rarely.

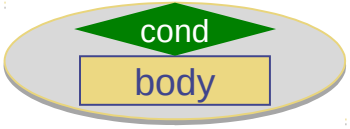
A refinement on rule ordering: urgency and earliness

Consider a set of concurrent rules (rules enabled because their rule conditions are true, and that fire in a clock in a logical order):

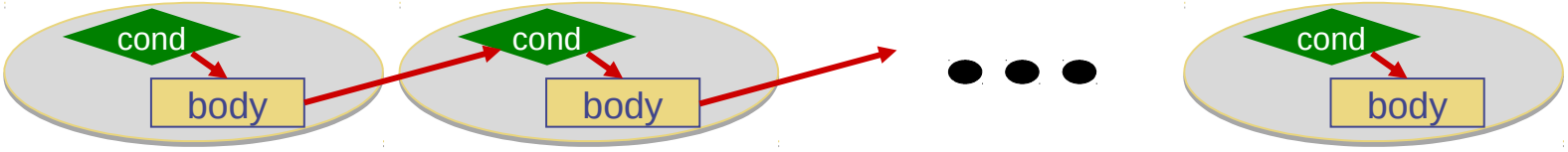


Executing each rule involves two separate activities:

- evaluate its condition
- evaluate its body



Thus, when executing a rule sequence, we're conceptually alternating these activities:



But note:

- Rule conditions are pure boolean expressions. They have no side effects. Thus, just evaluating the rule condition of rule rA can never affect another rule rB (neither evaluating rB's condition, nor what rB's body does).
- Many rule bodies do not affect other rule conditions

Thus, we can reorder condition and body evaluations (provided we preserve required orderings)

Thus, rule orderings can be refined into two orderings:

- *Urgency order*: order in which we evaluate rule conditions
- *Execution order*: order in which we evaluate rule bodies (technically this is original rule ordering), also called *Earliness*

Controlling scheduling: rule execution order

BSV notation

```
(* execution_order = “r1, r2" * )

rule r1;
   x <= 5;
endrule

rule r2;
   y <= 6;
endrule
```

Bluespec Classic notation

```
let r1 = rules
             when (True) ==> do
                 x := 5

    r2 = rules
             when (True) ==> do
                 y := 6

addRules (rJoinExecutionOrder  r1  r2)
```

- Forces an ordering in the logical semantics
  - Execution order is also called “earliness”

- In this example, forces “r1 < r2”

Urgency and Execution orders may be different

BSV notation

```
(* descending_urgency="enq_item, enq_bubble" *)
rule enq_item;
  outfifo.enq(infifo.first); infifo.deq;
  bubbles <= 0;
endrule

rule inc_bubbles;
  bubbles <= bubbles + 1;
endrule

rule enq_bubble;
  outfifo.enq(bubble_value);
  max_bubbles <= max (max_bubbles, bubbles);
endrule
```

Bluespec Classic notation

```
let eq_item = rules
                when True ==> do
                    outfifo.enq  infifo.first;  infifo.deq
                    bubbles := 0

    inc_bubbles = rules
                    when True ==> bubbles := bubbles + 1

    enq_bubble = rules
                   when True ==> do
                       outfifo.enq  bubble_value
                       max_bubbles := max (max_bubbles, bubbles)

addRules (rJoin (rJoinDescendingUrgency enq_item enq_bubble) inc_bubbles)
```

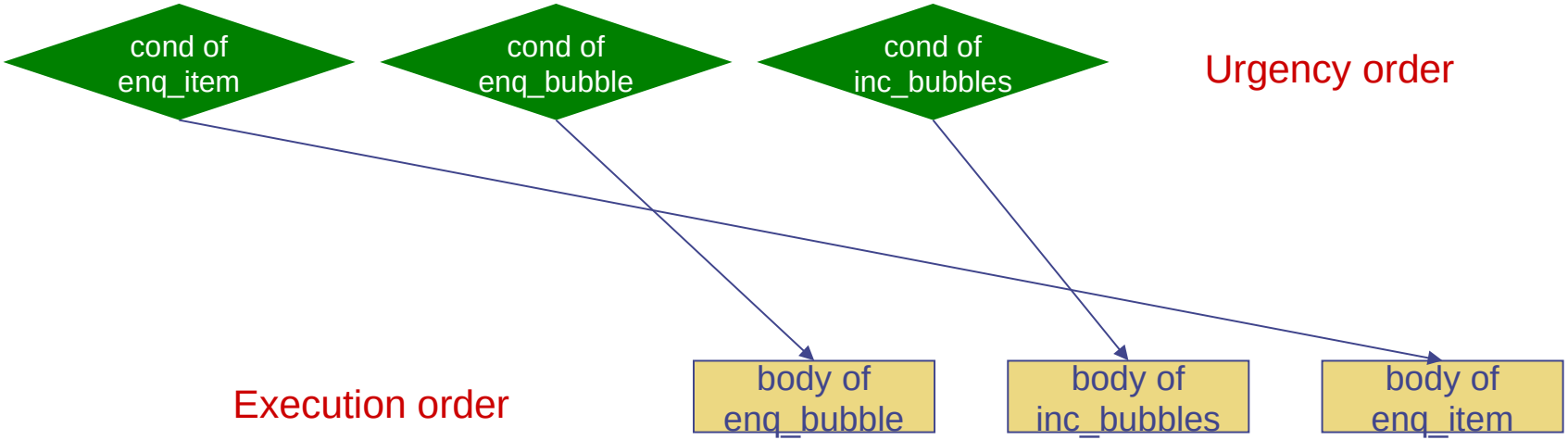
- This code enqueues items from the infifo into the outfifo, if available; otherwise, it enqueues a ‘bubble_value’
- It also computes the maximum stretch of bubbles

- The execution order is: enq_bubble < inc_bubbles < enq_item
  because reads of ‘bubbles’ must precede writes of ‘bubbles’

- However, we have forced the urgency to be: enq_item < enq_bubble

Urgency and Execution orders may be different

Pictorially:



Hands-on

- BSV-by-Example book: Examples in Chapter 7

bluespec

End

Questions?

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