Bluespec-5: Scheduling & Rule Composition

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Executing Multiple Rules Per Cycle: *Conflict-free rules*

```
rule ra (z > 10);
  x <= x + 1;
endrule

rule rb (z > 20);
  y <= y + 2;
endrule
```

Parallel execution behaves like ra < rb = rb < ra

Rule$_a$ and Rule$_b$ are conflict-free if

$$\forall s \,.\, \pi_a(s) \wedge \pi_b(s) \Rightarrow \begin{array}{l} 1.\ \pi_a(\delta_b(s)) \wedge \pi_b(\delta_a(s)) \\ 2.\ \delta_a(\delta_b(s)) == \delta_b(\delta_a(s)) \end{array}$$

Parallel Execution can also be understood in terms of a composite rule

```
rule ra_rb((z>10)&&(z>20));
  x <= x+1; y <= y+2;
endrule
```

Executing Multiple Rules Per Cycle: *Sequentially Composable rules*

```
rule ra (z > 10);
  x <= y + 1;
endrule

rule rb (z > 20);
  y <= y + 2;
endrule
```

Parallel execution behaves like ra < rb

Rule$_a$ and Rule$_b$ are sequentially composable if

$$\forall s \,.\, \pi_a(s) \wedge \pi_b(s) \Rightarrow \pi_b(\delta_a(s))$$

Parallel Execution can also be understood in terms of a composite rule

```
rule ra_rb((z>10)&&(z>20));
  x <= y+1; y <= y+2;
endrule
```

Sequentially Composable rules ...

```
rule ra (z > 10);
  x <= 1;
endrule

rule rb (z > 20);
  x <= 2;
endrule
```

Parallel execution can behave either like ra < rb or rb < ra but the two behaviors are not the same

Composite rules

Behavior ra < rb

Behavior rb < ra

A property of rule-based systems

- Adding a new rule to a system can only introduce new behaviors
- If the new rule is a *derived* rule, then it does not add new behaviors

- Example of a derived rule:
  - Given rules:

    R_a: when $\pi_a(s)$ => s := $\delta_a(s)$;

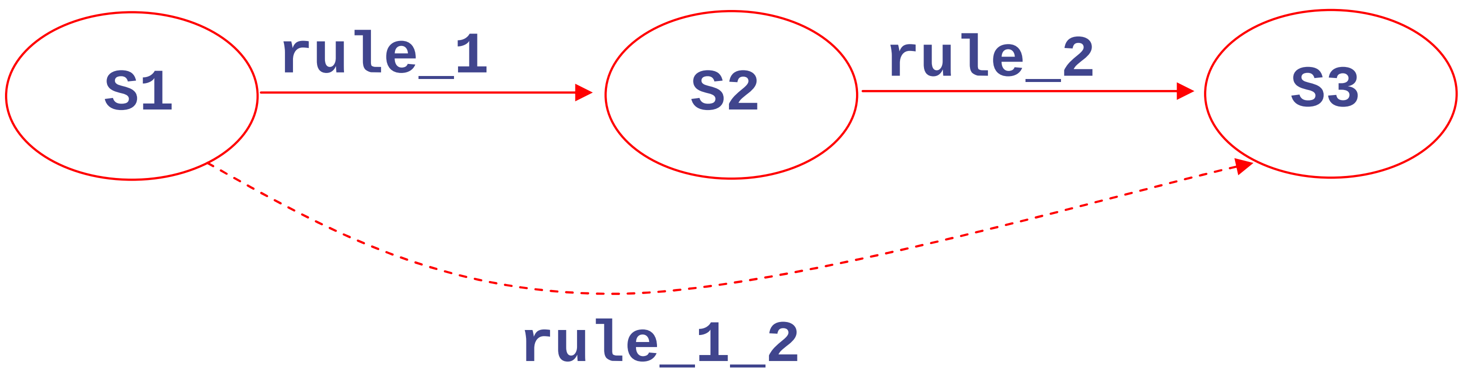
    R_b: when $\pi_b(s)$ => s := $\delta_b(s)$;
  - The following rule is a derived rule:

    $R_{a,b}$: when $\pi_a(s)$ & $\pi_b(\delta_a(s))$ => s := $\delta_b(\delta_a(s))$;

For CF rules $\pi_b(\delta_a(s)) = \pi_b(s)$ and s := $\delta_b(\delta_a(s)) = \delta_a(\delta_b(s))$;

For SC rules $\pi_b(\delta_a(s)) = \pi_b(s)$ and s := $\delta_b(\delta_a(s))$;

Rule composition




```
rule rule_1   (p1(s));           r <= f1(s); endrule
rule rule_2   (p2(s));           r <= f2(s); endrule
rule rule_1_2 (p1(s) && p2(s’); s <= f2(s’);endrule
                            where s’ = f1(s);
```

Semantics of rule based systems guarantee that rule_1_2 which takes s1 to s3 is *correct*

Such composed rules are called derived rules because they are mechanically derivable

Implementation oriented view of concurrency

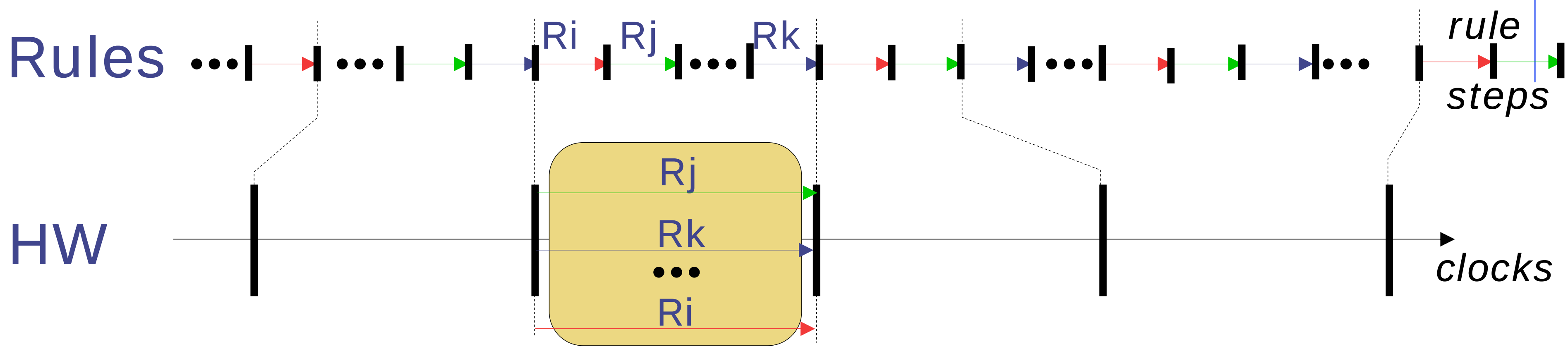
A. When executing a set of rules in a clock cycle, each rule reads state from the leading clock edge and sets state at the trailing clock edge

$\Rightarrow$ none of the rules in the set can see the effects of any of the other rules in the set

B. However, in one-rule-at-a-time semantics, each rule sees the effects of all previous rule executions

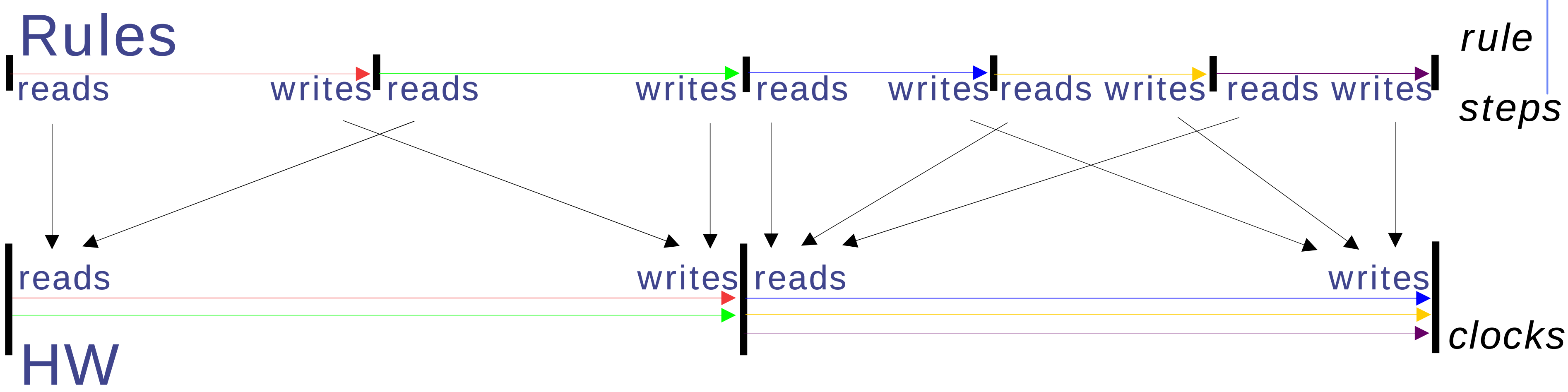
Thus, a set of rules can be *safely* executed together in a clock cycle only if A and B produce the same net state change

Pictorially



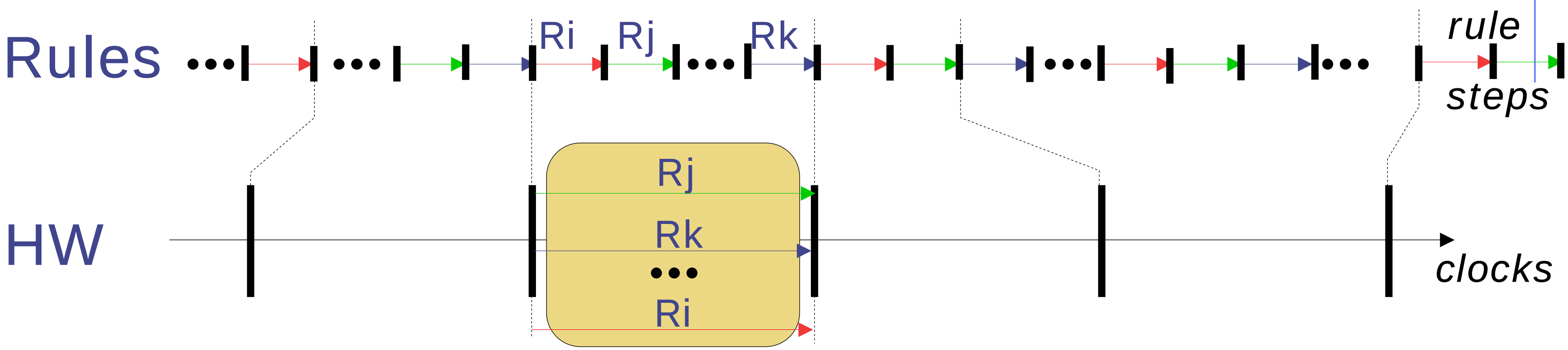
- There are more intermediate states in the rule semantics (a state after each rule step)
- In the HW, states change only at clock edges

Parallel execution reorders reads and writes



- In the rule semantics, each rule sees (reads) the effects (writes) of previous rules
- In the HW, rules only see the effects from previous clocks, and only affect subsequent clocks

Correctness



- Rules are allowed to fire in parallel only if the net state change is equivalent to sequential rule execution (i.e., CF or SC)
- Consequence: the HW can never reach a state unexpected in the rule semantics

Compiler determines if two rules can be executed in parallel

Rule$_a$ and Rule$_b$ are conflict-free if

$$\forall s\ .\ \pi_a(s) \wedge \pi_b(s) \Rightarrow$$

1. $\pi_a(\delta_b(s)) \wedge \pi_b(\delta_a(s))$
2. $\delta_a(\delta_b(s)) == \delta_b(\delta_a(s))$

Rule$_a$ and Rule$_b$ are sequentially composable if

$$\forall s\ .\ \pi_a(s) \wedge \pi_b(s) \Rightarrow \pi_b(\delta_a(s))$$

These properties can be determined by examining the domains and ranges of the rules in a pairwise manner.

Mutually Exclusive Rules

- $Rule_a$ and $Rule_b$ are mutually exclusive if they can never be enabled simultaneously

$$\forall s \,.\, \pi_a(s) \Rightarrow \sim \pi_b(s)$$

Mutually-exclusive rules are Conflict-free even if they write the same state

Mutual-exclusive analysis brings down the cost of conflict-free analysis

Conflict-Free Scheduler

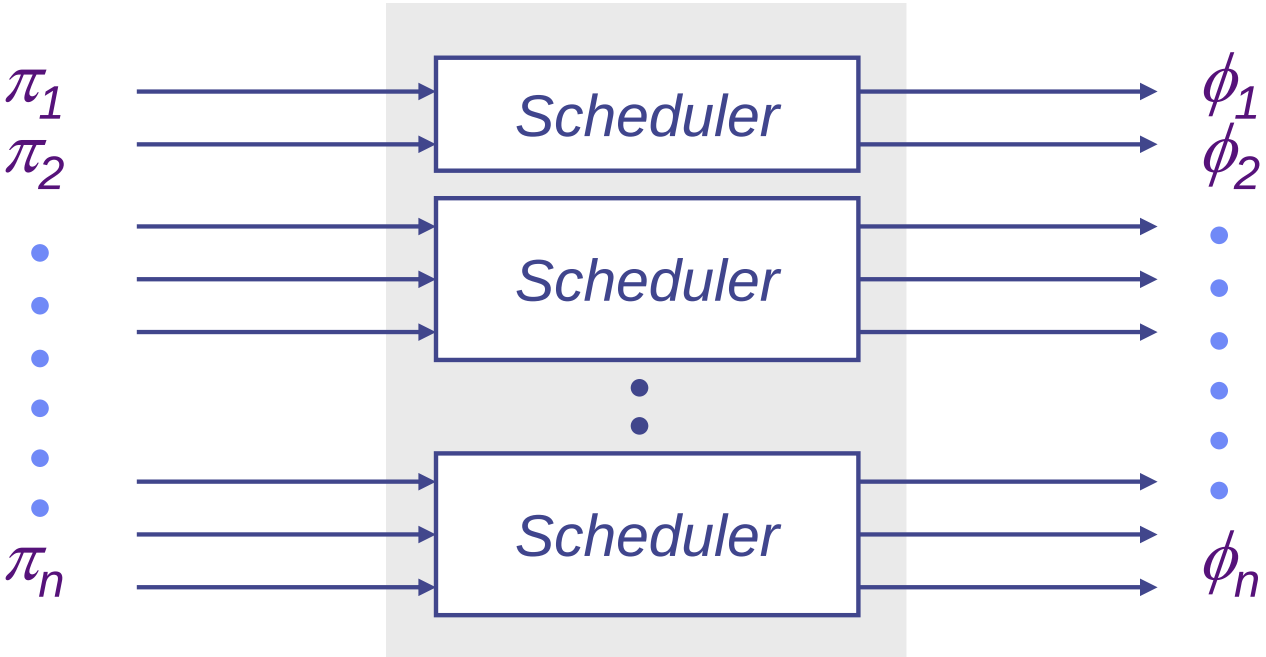
- Partition rules into maximum number of disjoint sets such that
  - a rule in one set may conflict with one or more rules in the same set
  - a rule in one set is conflict free with respect to all the rules in all other sets

(Best case: All sets are of size 1!!)

- Schedule each set independently
  - Priority Encoder, Round-Robin Priority Encoder
  - Enumerated Encoder

The state update logic depends upon whether the scheduler chooses "sequential composition" or not

Multiple-Rules-per-Cycle Scheduler



Divide the rules into smallest conflicting groups; provide a scheduler for each group

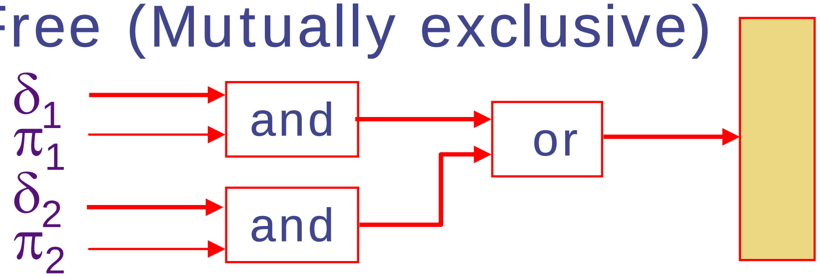
1. $\phi_i \Rightarrow \pi_i$
2. $\pi_1 \vee \pi_2 \vee .... \vee \pi_n \Rightarrow \phi_1 \vee \phi_2 \vee .... \vee \phi_n$
3. *Multiple operations such that*

   $\phi_i \wedge \phi_j \Rightarrow R_i$ *and* R_j *are conflict-free or sequentially composable*

Muxing structure

- Muxing logic requires determining for each register (action method) the rules that update it and under what conditions

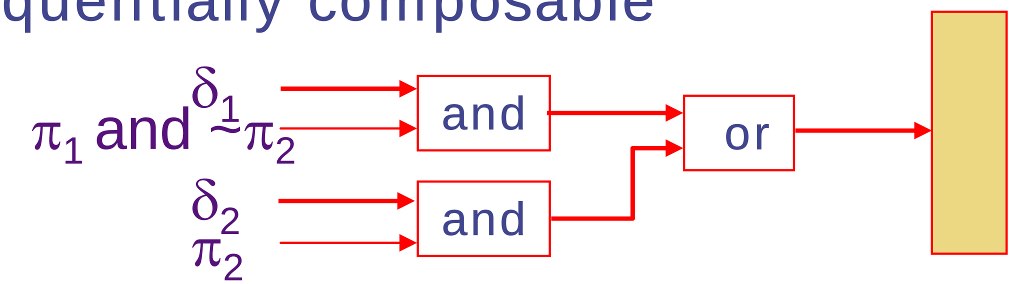
Conflict Free (Mutually exclusive)



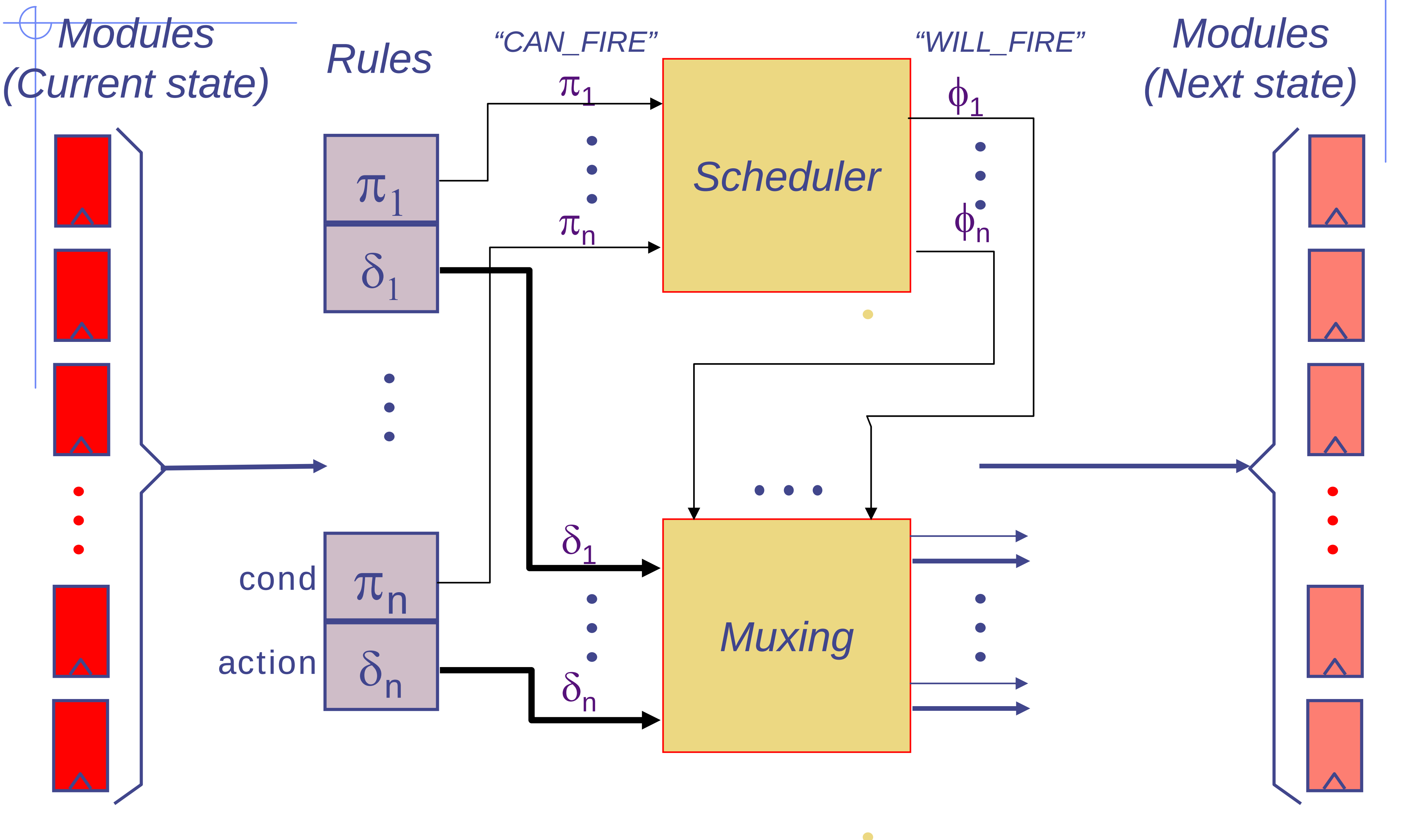
CF rules either do not update the same element or are ME

$\pi_1 \rightarrow \sim\pi_2$

Sequentially composable



Scheduling and control logic



Synthesis Summary

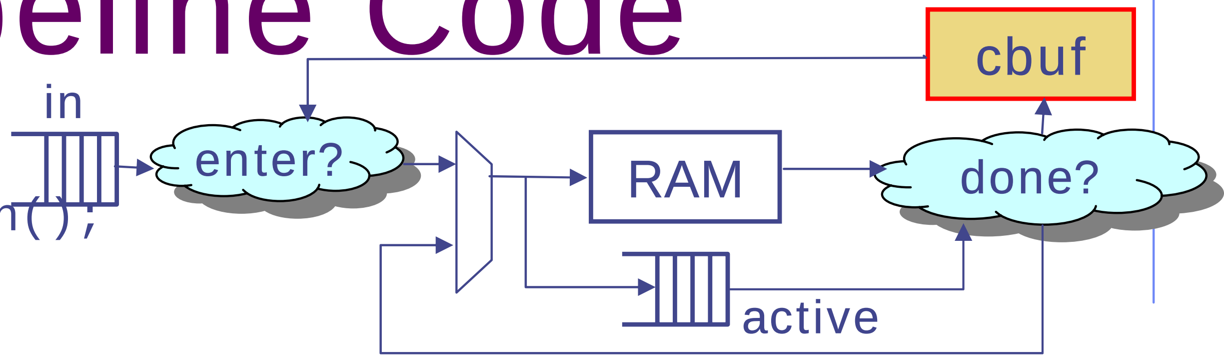
- Bluespec generates a *combinational hardware scheduler* allowing multiple enabled rules to execute in the same clock cycle
  - The hardware makes a rule-execution decision on every clock (i.e., it is not a static schedule)
  - Among those rules that CAN_FIRE, only a subset WILL_FIRE that is consistent with a Rule order
- Since multiple rules can write to a common piece of state, the compiler introduces appropriate muxing logic

Scheduling conflicting rules

- When two rules conflict on a shared resource, they cannot both execute in the same clock
- The compiler produces logic that ensures that, when both rules are applicable, only one will fire
  - Which one?

source annotations

Circular Pipeline Code




```
rule enter (True);
   Token t <- cbuf.getToken();
   IP ip = in.first();
   ram.req(ip[31:16]);
   active.enq(tuple2(ip[15:0], t)); in.deq();
endrule
rule done (True);
   TableEntry p <- ram.resp();
   match {.rip, .t} = active.first();
   if (isLeaf(p)) cbuf.done(t, p);
   else begin
       active.enq(rip << 8, t);
       ram.req(p + signExtend(rip[15:7]));
     end
   active.deq();
endrule
```

Can rules enter and done be applicable simultaneously?

Which one should go?

Concurrency Expectations

- Register

	read2	write2
read1		
write1		

- FIFO

	enq2	first2	deq2	clear2
enq1				
first1				
deq1				
clear1				

One Element FIFO

```
module mkFIFO1 (FIFO#(t));
  Reg#(t)    data  <- mkRegU();
  Reg#(Bool) full  <- mkReg(False);
  method Action enq(t x) if (!full);
    full <= True;     data <= x;
  endmethod
  method Action deq() if (full);
    full <= False;
  endmethod
  method t first() if (full);
    return (data);
  endmethod
  method Action clear();
    full <= False;
  endmethod
endmodule
```

Concurrency?

enq and deq ?

Two-Element FIFO

```
module mkFIFO2#(FIFO#(t));
  Reg#(t) data0 <-mkRegU; Reg#(Bool) full0 <- mkReg(False);
  Reg#(t) data1 <-mkRegU; Reg#(Bool) full1 <- mkReg(False);

  method Action enq(t x) if (!(full0 && full1));
    data1 <= x; full1 <= True;
    if (full1) then begin data0 <= data1; full0 <= True; end
  endmethod
  method Action deq() if (full0 || full1);
    if (full0) full0 <= False; else full1 <= False;
  endmethod
  method t first() if (full0 || full1);
    return ((full0)?data0:data1);
  endmethod
  method Action clear();
    full0 <= False; full1 <= False;
  endmethod
endmodule
```

Shift register implementation

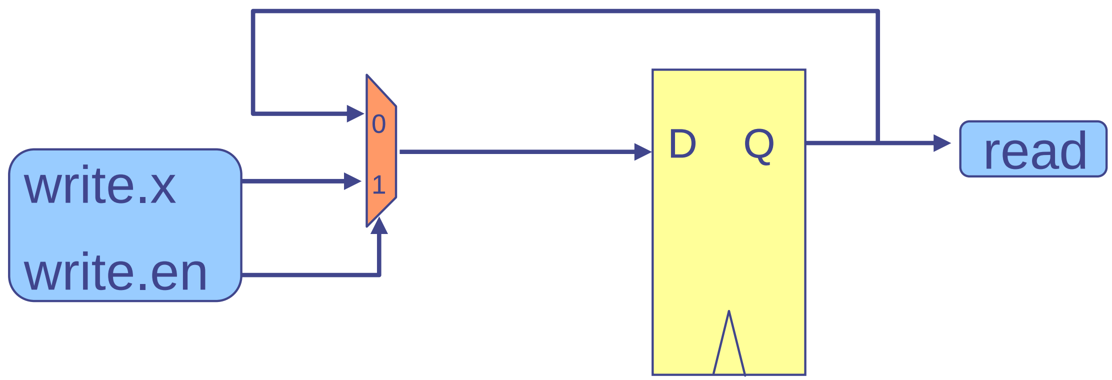
The good news ...

- It is always possible to transform your design to meet desired concurrency and functionality

Register Interfaces

read < write

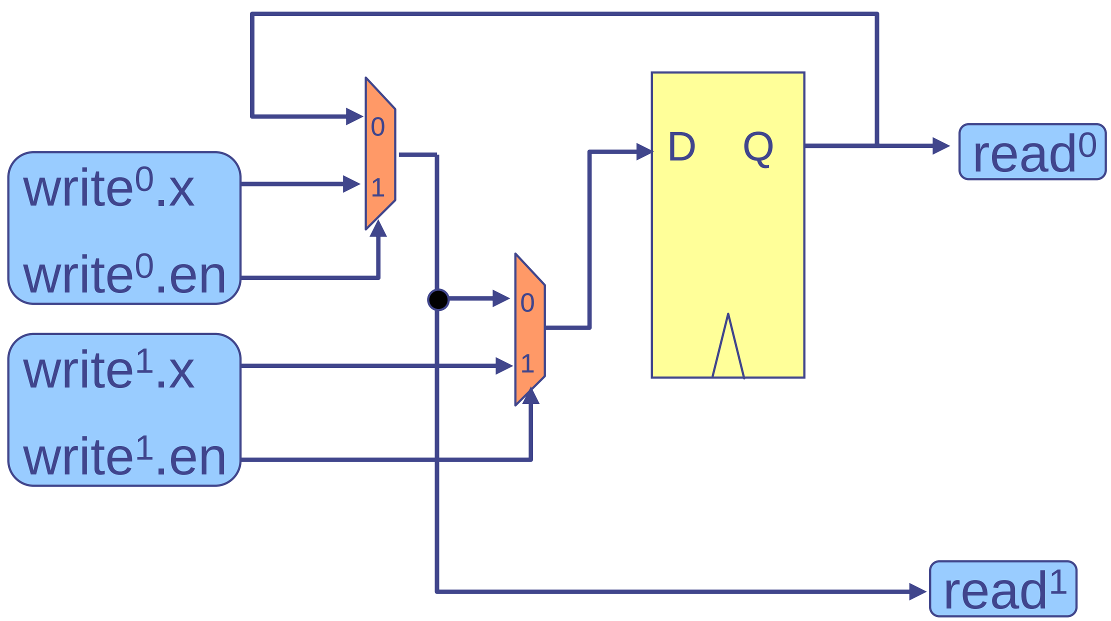
write < read ?



Ephemeral History Register (EHR)

[MEMOCODE'04]

read⁰ < write⁰ < read¹ < write¹ < ….



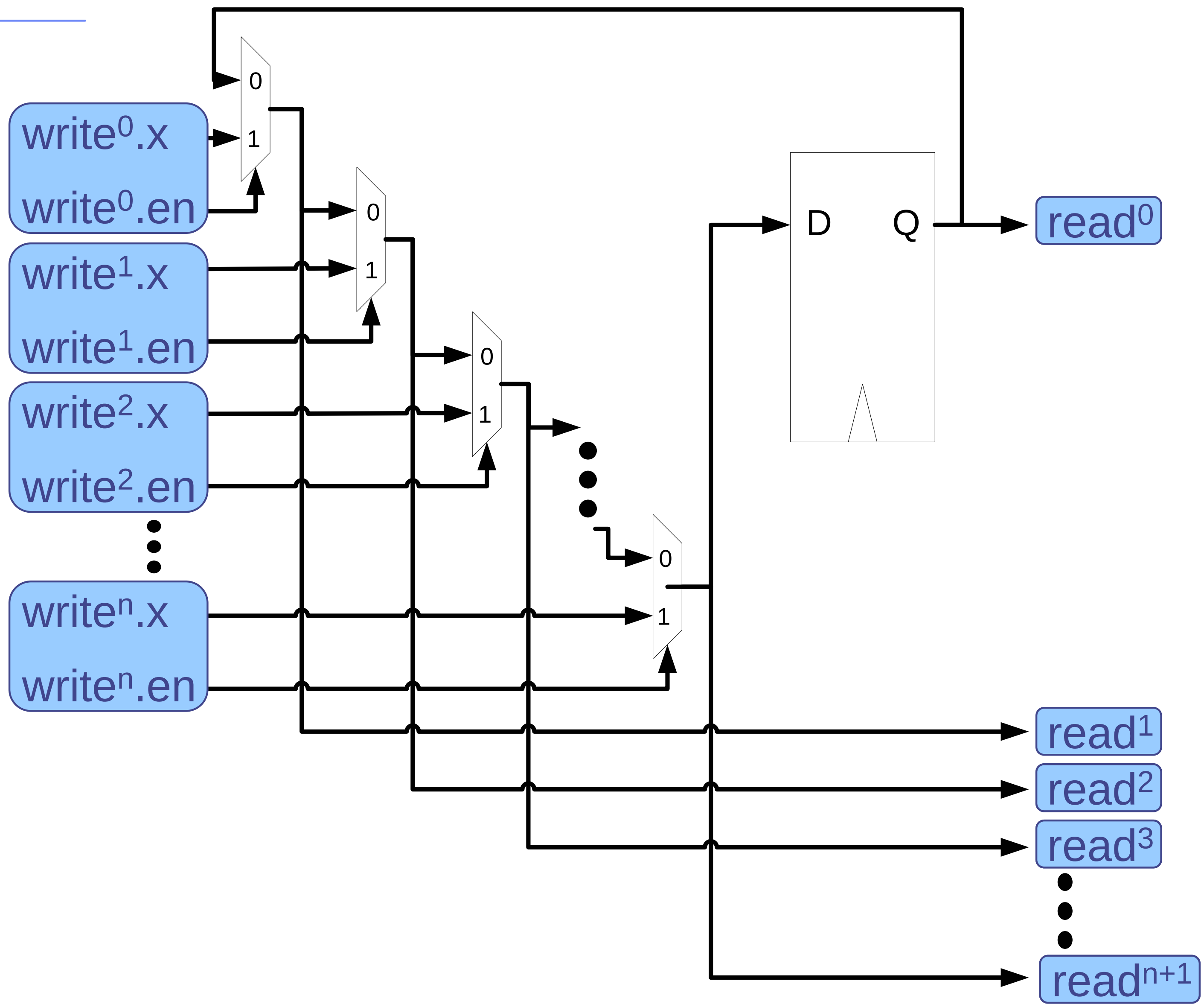
writeⁱ⁺¹ takes precedence over writeⁱ

One Element FIFO *using EHRs*

```
first0 < deq0 < enq1
```

```
module mkFIFO1 (FIFO#(t));
  EHReg2#(t)    data  <- mkEHReg2U();
  EHReg2#(Bool) full  <- mkEHReg2(False);
  method Action enq0(t x) if (!full.read0);
    full.write0 <= True;  data.write0 <= x;
  endmethod
  method Action deq0() if (full.read0);
    full.write0 <= False;
  endmethod
  method t first0() if (full.read0);
    return (data.read0);
  endmethod
  method Action clear0();
    full.write0 <= False;
  endmethod
endmodule
```

EHR as the base case?



The bad news ...

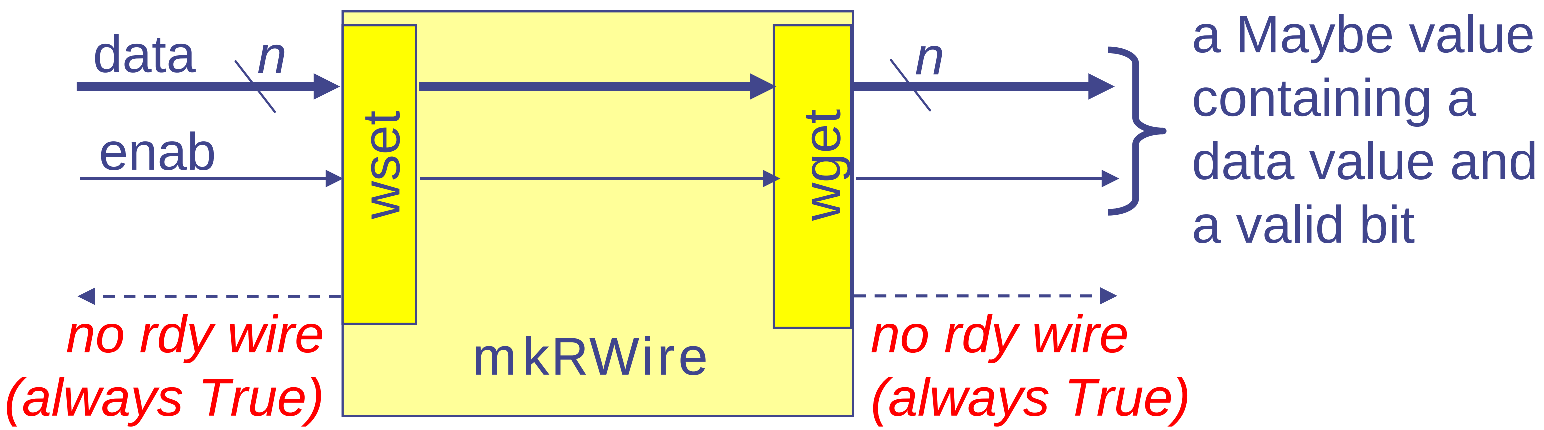
- EHR cannot be written in Bluespec as defined so far
- Even though this transformation to meet the performance "specification" is mechanical, the Bluespec compiler currently does not do this transformation. Choices:
  - do it manually and use a library of EHRs
  - rely on a low level (dangerous) programming mechanism.

Wires

RWires

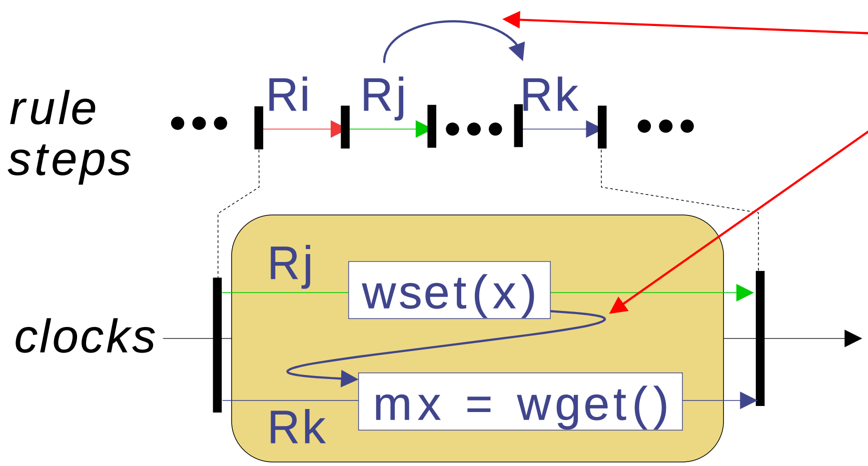
```
interface RWire #(type  t);
    method Action wset (t  data);
    method Maybe#(t)  wget ();
endinterface

module mkRWire (RWire#(t));
```

- The mkRWire module contains no state and no logic: it's just wires!
- By testing the valid bit of wget() we know whether some rule containing wset() is executing concurrently (enab is True)



Intra-clock communication

- Suppose Rj uses rw.wset() on an RWire
- Suppose Rk uses rw.wget() on the same RWire
- If Rj and Rk execute in the same cycle then Rj always precedes Rk in the rule-step semantics
- Testing isValid(rw.wget()) allows Rk to test whether Rj is executing in the same cycle)
- wset/wget allows Rj to communicate a value to Rk



Intra-clock rule-to-rule communication, *provided* both rules actually execute concurrently (same cycle)

Forward communication only (in the rule-step ordering)

One Element FIFO w/ RWires

Pipeline FIFO

```
module mkFIFO1#(type t);
  Reg#(t)    data  <- mkRegU();
  Reg#(Bool) full  <- mkReg(False);
  PulseWire  deqW  <- mkPulseWire();
  method Action enq(t x) if (deqW || !full);
    full <= True;     data <= x;
  endmethod
  method Action deq() if (full);
    full <= False; deqW.send();
  endmethod
  method t first() if (full);
    return (data);
  endmethod
  method Action clear();
    full <= False;
  endmethod
endmodule
```

```
first < deq < enq
```

One Element FIFO w/ RWires

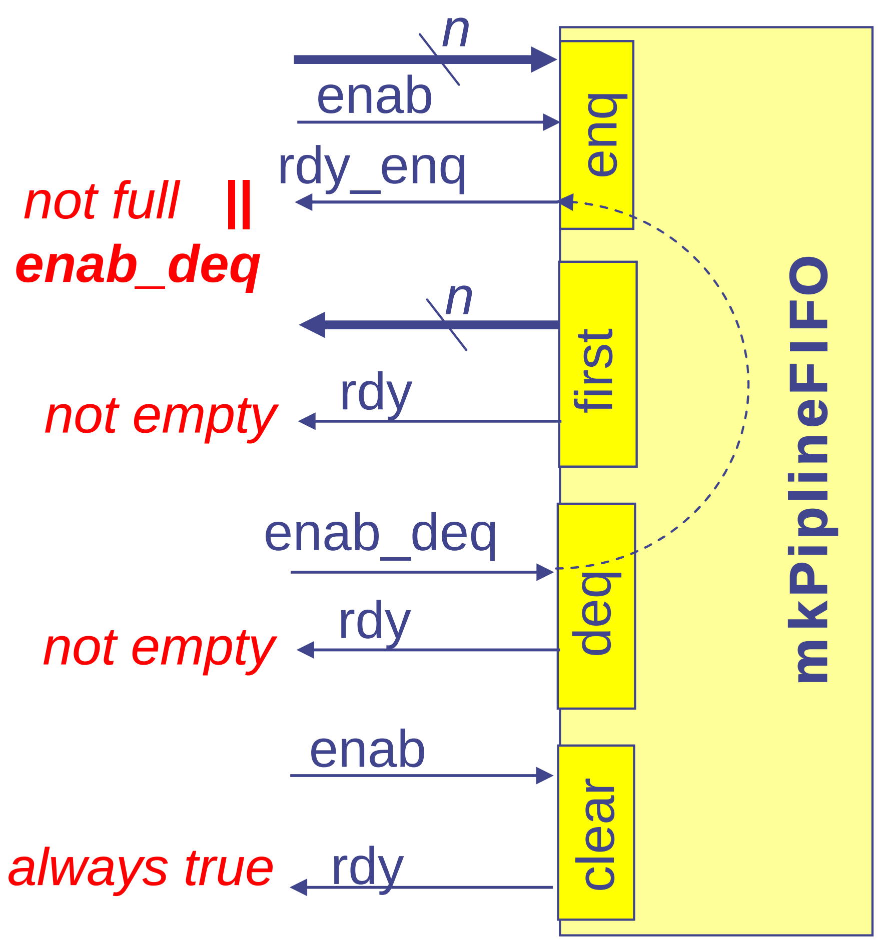
Bypass FIFO

```
module mkFIFO1#(type t);
  Reg#(t)    data  <- mkRegU();
  Reg#(Bool) full  <- mkReg(False);
  RWire#(t)  enqW  <- mkRWire();
  PulseWire  deqW  <- mkPulseWire();
  rule finishMethods(isJust(enqW.wget) || deqW);
    full  <= !deqW;
  endrule
  method Action enq(t x) if (!full);
    enqW.wset(x);  data <= x;
  endmethod
  method Action deq() if (full || isJust(enqW.wget()));
     deqW.send();
  endmethod
  method t first() if (full || isJust(enqW.wget()));
    return (full ? data : unJust(enqW.wget));
  endmethod
  method Action clear();
    full <= False;
  endmethod
endmodule
```

enq < first < deq

A HW implication of mkPipelineFIFO

- There is now a combinational path from enab_deq to rdy_enq (a consequence of the RWire)
- This is how a rule using enq() “knows” that it can go even if the FIFO is full, i.e., enab_deq is a signal that a rule using deq() is executing concurrently



Viewing the schedule

- The command-line flag **`-show-schedule`** can be used to dump the schedule
- Three groups of information:
  - method scheduling information
  - rule scheduling information
  - the static execution order of rules and methods