Constructive Computer Architecture:

Multirule systems and Concurrent Execution of Rules

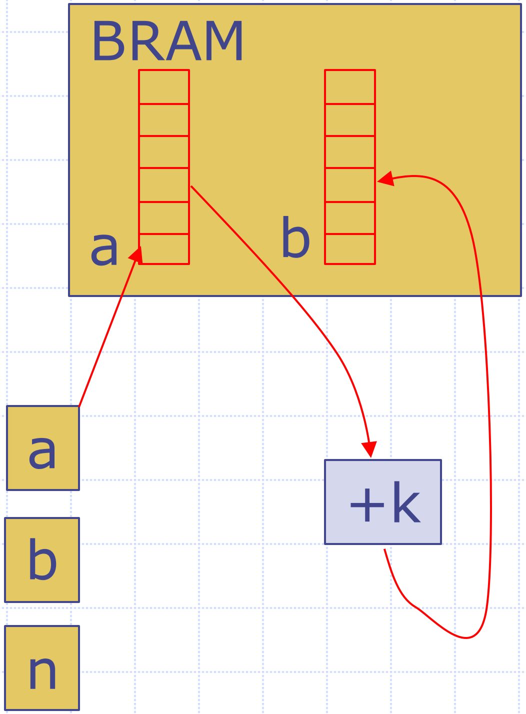
Yann Herklotz - EPFL

Slides adapted from 6.1920 with Thomas and Arvind
(Spring 23, MIT)

Systems with multiple rules

A one-instruction vector machine

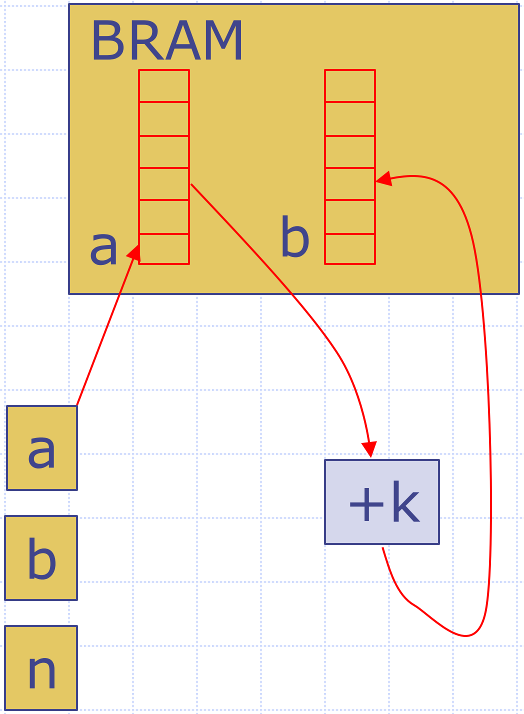
- Adds k to each element of an n-element vector a and store it in b
- Vectors are too large to be stored in registers and therefore stored in BRAM
- BRAM interface is single ported
- Let a and b be the current pointers to the vectors, and n be the number of elements in the vector that remain to be processed



Let us build such a machine

Steps in processing

- Assume **a, b, n** and **BRAM** have been initialized properly
- Initiate a BRAM request to read an element of **a**; increment **a**
- Wait for the result; add **k**;
- initiate a write of the result in b; increment **b**; decrement **n**
- Repeat these steps if **n>0**



Let a and b be the current pointers to the vectors, and n be the number of elements in the vector that remain to be processed

Rules for the vector machine

```
rule init_read_a if (n>0 && turn == 0);
  mem.portA.request.put(BRAMReq{write:False,writeRes:False,
                                address:a, datain:?});
  a <= a + 1;
  n <= n - 1;
  turn <= 1;
endrule

rule read_result if (turn == 1);
  let x <- mem.portA.response.get();
  mem.portA.request.put(BRAMReq{write:True, writeRes:False,
                                address:b, datain:x + k});
  b <= b + 1;
  turn <= 0;
endrule
```

Multi-rule Systems

Repeatedly:

- Select a rule to execute
- Compute the state updates
- Make the state updates

Non-deterministic choice; User annotations can be used in rule selection

One-rule-at-a-time-semantics: Any legal behavior of a Bluespec program can be explained by observing the state updates obtained by applying only one rule at a time

However, for performance we execute multiple rules concurrently whenever possible

Rules for the vector machine

```
rule init_read_a if (n>0~~ && turn == 0~~);
  mem.portA.request.put(BRAMReq{write:False,writeRes:False,
                                address:a, datain:?});
  a <= a + 1;
  n <= n - 1;
  ~~turn <= 1;~~
endrule

rule read_result ~~if (turn == 1)~~;
  let x <- mem.portA.response.get();
  mem.portA.request.put(BRAMReq{write:True, writeRes:False,
                                address:b, datain:x + k});
  b <= b + 1;
  ~~turn <= 0;~~
endrule
```

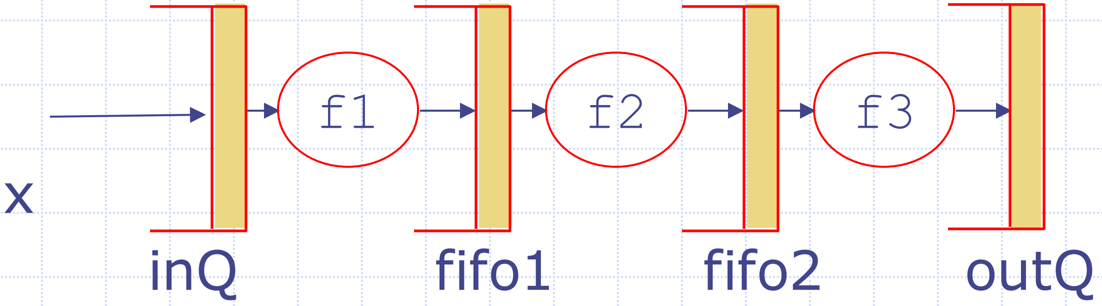
The design still works without taking turn! Or does it?

Homework

- To get familiarized with this programming/execution model:
  - Lab1-b: debug a simple multi-rule machine that fail to compute dot products
  - Lab2: write your own matrix/matrix multiply module

Multi-rule systems and concurrency

Elastic pipeline



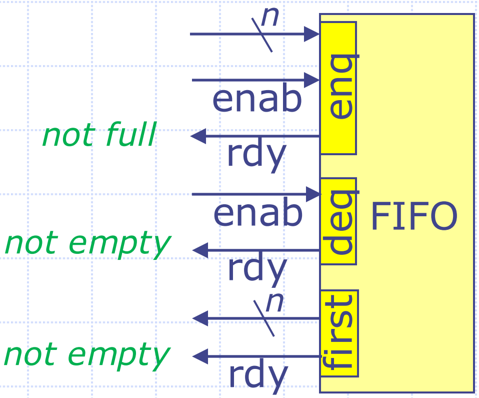
```
rule stage1;
   fifo1.enq(f1(inQ.first));
   inQ.deq();          endrule
rule stage2;
   fifo2.enq(f2(fifo1.first));
   fifo1.deq;          endrule
rule stage3;
   outQ.enq(f3(fifo2.first));
   fifo2.deq;          endrule
```

- Can these rules fire concurrently?

Yes, but it must be possible to do enq and deq on a fifo simultaneously

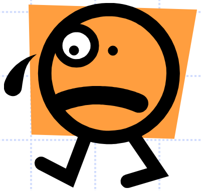
One-Element FIFO Implementation

```
module mkFifo (Fifo#(1, t));
   Reg#(t)     d   <- mkRegU;
   Reg#(Bool)  v   <- mkReg(False);
   method Action enq(t x) if (!v);
      v <= True; d <= x;
   endmethod
   method Action deq if (v);
      v <= False;
   endmethod
   method t first if (v);
      return d;
   endmethod
endmodule
```

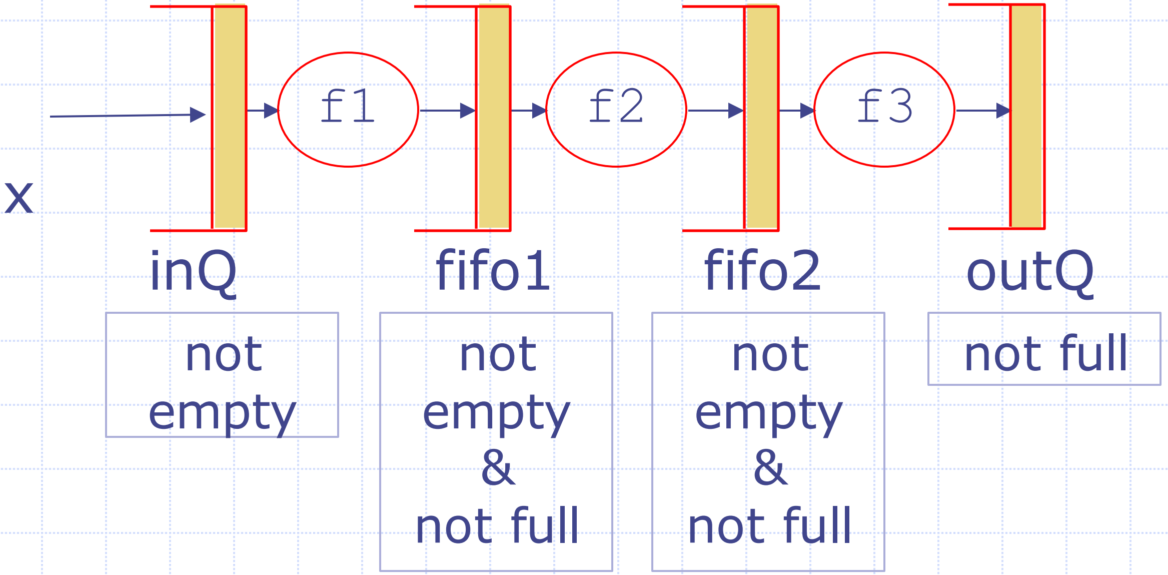


Can enq and deq methods be ready at the same time?

No! Therefore, they cannot execute concurrently!

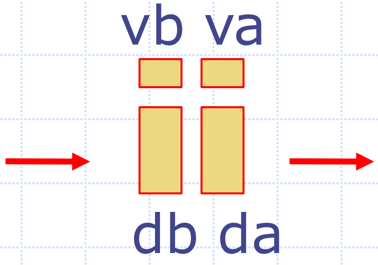


Concurrency when the FIFOs do not permit concurrent enq and deq



At best alternate stages in the pipeline will be able to fire concurrently

Two-Element FIFO

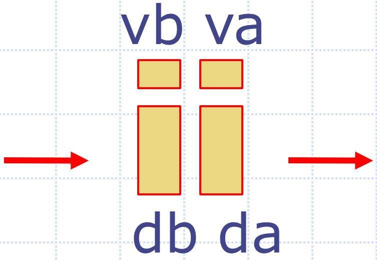


Assume, if there is only one element in the FIFO it resides in da

- Initially, both va and vb are false
- First enq will store the data in da and mark va true
- An enq can be done as long as vb is false; a deq can be done as long as va is true

Two-Element FIFO

BSV code



Assume, if there is only one element in the FIFO it resides in da

```
module mkFifo (Fifo#(2, t));
  Reg#(t)     da  <- mkRegU();
  Reg#(Bool)  va  <- mkReg(False);
  Reg#(t)     db  <- mkRegU();
  Reg#(Bool)  vb  <- mkReg(False);
  method Action enq(t x) if !vb;
    if (va) begin db <= x; vb <= True; end
       else begin da <= x; va <= True; end
  endmethod
  method Action deq if va;
    if (vb) begin da <= db; vb <= False; end
       else begin va <= False; end
  endmethod
  method t first if va; return da;
  endmethod
endmodule
```

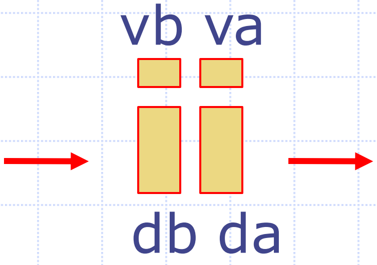
Can both enq and deq be ready at the same time?

yes

Two-Element FIFO

Sequential behavior analysis

```
method Action enq(t x) if !vb;
 if (va) begin db <= x; vb <= True; end
    else begin da <= x; va <= True; end
endmethod
method Action deq if va;
 if (vb) begin da <= db; vb <= False; end
    else begin va <= False; end
endmethod
```

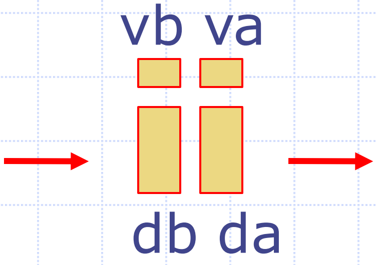


- Suppose, initially `vb=false` and `va=true (there is an element)`
- Suppose `enq` executes before `deq`
  - `enq` executes: `db <= x; vb <= True;`
  - `deq` executes: `da <= x; vb <= False;`
  - Final values: `da == x; db == x; va == True; vb == False;`
- Suppose `deq` executes before `enq`
  - `deq` executes: `va <= False;`
  - `enq` executes: `da <= x; va <= True;`
  - Final values: `da == x; db == ?; va == True; vb == False;`

Two-Element FIFO

concurrency analysis

```
method Action enq(t x) if !vb;
 if (va) begin db <= x; vb <= True; end
    else begin da <= x; va <= True; end
endmethod
method Action deq if va;
 if (vb) begin da <= db; vb <= False; end
    else begin va <= False; end
endmethod
```



we can't get into this state if enq and deq are performed in some order

- Will concurrent execution of `enq` and `deq` cause a double write error?
  - Initially `vb=False` and `va=True`
  - `enq` will execute: `db <= x; vb <= True;`
  - `deq` will execute `va <= False;`

no double-write error

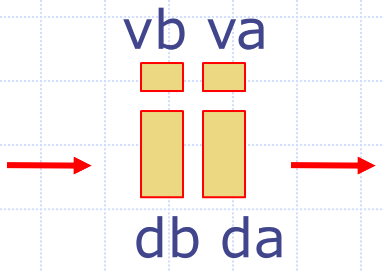
- The final state will be `va = False` and `vb = True;` with the old data in `da` and new data in `db`

oops!

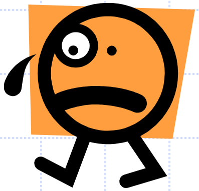
Two-Element FIFO

concurrency analysis - continued

```
method Action enq(t x) if !vb;
 if (va) begin db <= x; vb <= True; end
    else begin da <= x; va <= True; end
endmethod
method Action deq if va;
 if (vb) begin da <= db; vb <= False; end
    else begin va <= False; end
endmethod
```



- In this implementation, `enq` and `deq` should not be called concurrently
  - later we will present a systematic procedure to decide which methods of a module can be called concurrently
- First, we will study when two rules that only use registers can be executed concurrently



Concurrent execution of rules

- Two rules can execute concurrently, if concurrent execution would not cause a double-write error, *and*
- The final state can be obtained by executing rules one-at-a-time in some sequential order

Can these rules execute concurrently?

(without violating the one-rule-at-a-time-semantics)

Example 1

```
rule ra;
   x <= x+1;
endrule
rule rb;
   y <= y+2;
endrule
```

Example 2

```
rule ra;
   x <= y+1;
endrule
rule rb;
   y <= x+2;
endrule
```

Example 3

```
rule ra;
   x <= y+1;
endrule
rule rb;
   y <= y+2;
endrule
```

Final value of (x,y) (initial values (0,0))

	Exam 1	Exam2	Exam3
Concurrent Execution	(1,2)	(1,2)	(1,2)
ra<rb	(1,2)	(1,3)	(1,2)
rb<ra	(1,2)	(3,2)	(3,2)
	No Conflict	Conflict	ra<rb

Rule scheduling

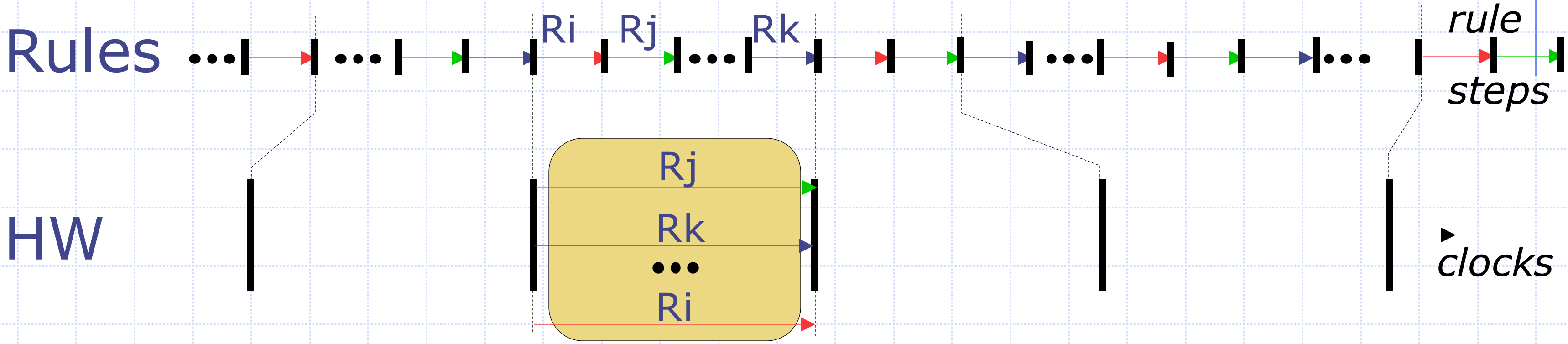
- The BSV compiler schedules as many rules as possible for concurrent execution among the rules that are enabled (i.e., whose guards are true), provided it can ensure that the chosen rules don't *conflict* with each other
- Conflict:
  - Double write
  - If the effect of rule execution does not appear to be as if one rule executed after the other

Scheduling, systematically

First register only, and with arbitrary modules

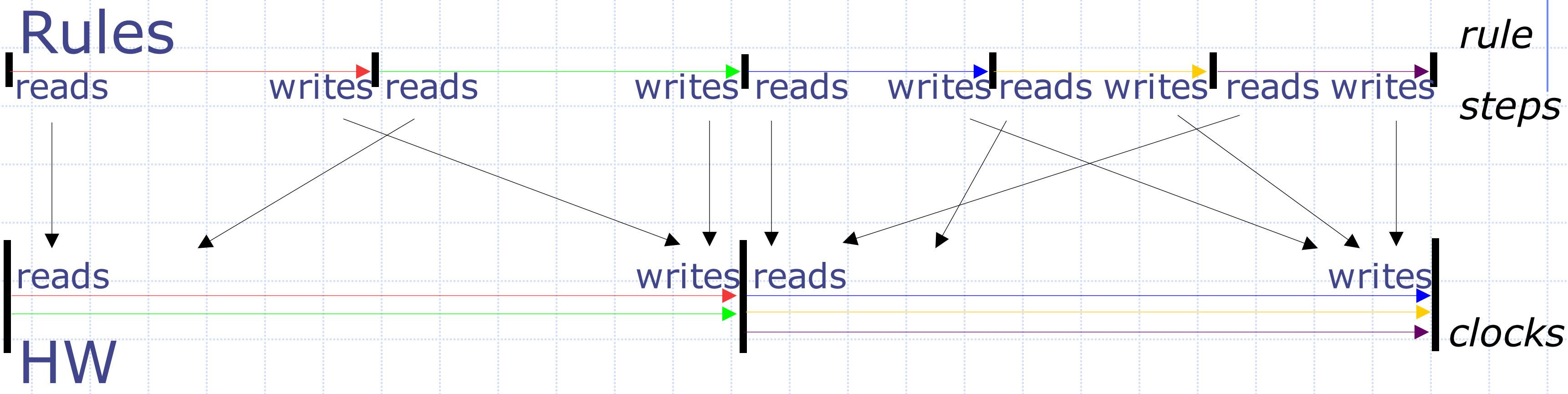
some insight into

Concurrent rule execution



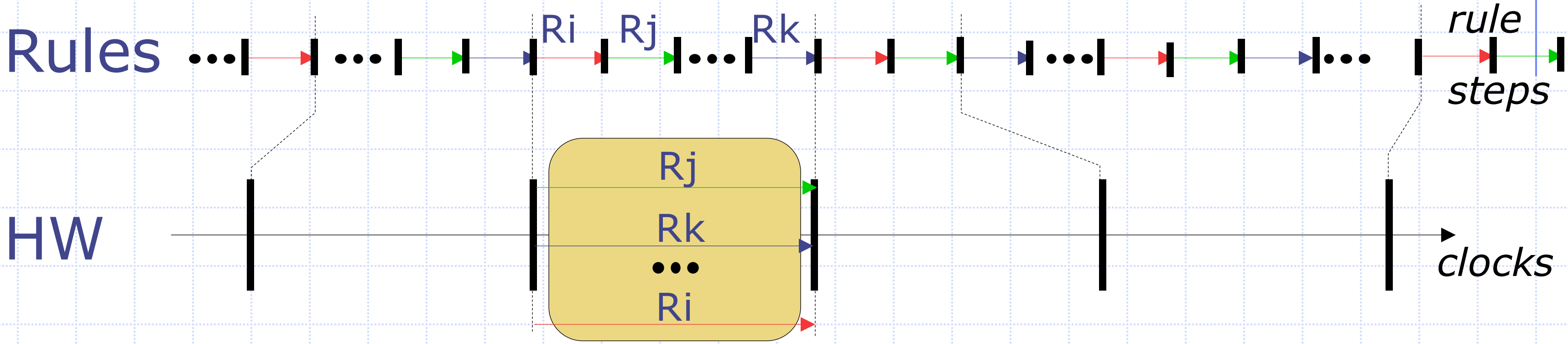
- There are more intermediate states in the rule semantics (a state after each rule step)
- In the HW, states change only at clock edges

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



- The compiler will schedule rules concurrently only if the net state change is equivalent to a sequential rule execution

Compiler test for concurrent rule execution James Hoe, Ph.D., 2000

- Let RS(r) be the set of registers rule r may read
- Let WS(r) be the set of registers rule r may write

- Rules ra and rb are *conflict free* (CF) if

  $(RS(ra) \cap WS(rb) = \varphi) \wedge (RS(rb) \cap WS(ra) = \varphi) \wedge (WS(ra) \cap WS(rb) = \varphi)$

- Rules ra and rb are *sequentially composable* (SC) (ra<rb) if

  $(RS(rb) \cap WS(ra) = \varphi) \wedge (WS(ra) \cap WS(rb) = \varphi)$

- If Rules ra and rb *conflict* if they are not CF or SC

Compiler analysis

Example 1

```
rule ra;
   x <= x+1;
endrule
rule rb;
   y <= y+2;
endrule
```

Example 2

```
rule ra;
   x <= y+1;
endrule
rule rb;
   y <= x+2;
endrule
```

Example 3

```
rule ra;
   x <= y+1;
endrule
rule rb;
   y <= y+2;
endrule
```

	Exam 1	Exam2	Exam3
RS(ra)	{x}	{y}	{y}
WS(ra)	{x}	{x}	{x}
RS(rb)	{y}	{x}	{y}
WS(rb)	{y}	{y}	{y}
RS(ra)∩WS(rb)	φ	{y}	{y}
RS(rb)∩WS(ra)	φ	{x}	φ
WS(ra)∩WS(rb)	φ	φ	φ
Conflict?	CF	C	SC

Concurrent scheduling

- The BSV compiler determines which rules among the rules whose guards are ready can be executed concurrently
- It builds a simple **greedy** list-based scheduler:
  - Picks the first enabled rule in the list
  - Schedules the next enabled rule if it does not conflict with any of the rules scheduled so far
  - Repeats the process until no more rules can be scheduled

The list is built using textual ordering of rules but can be changed by user annotations

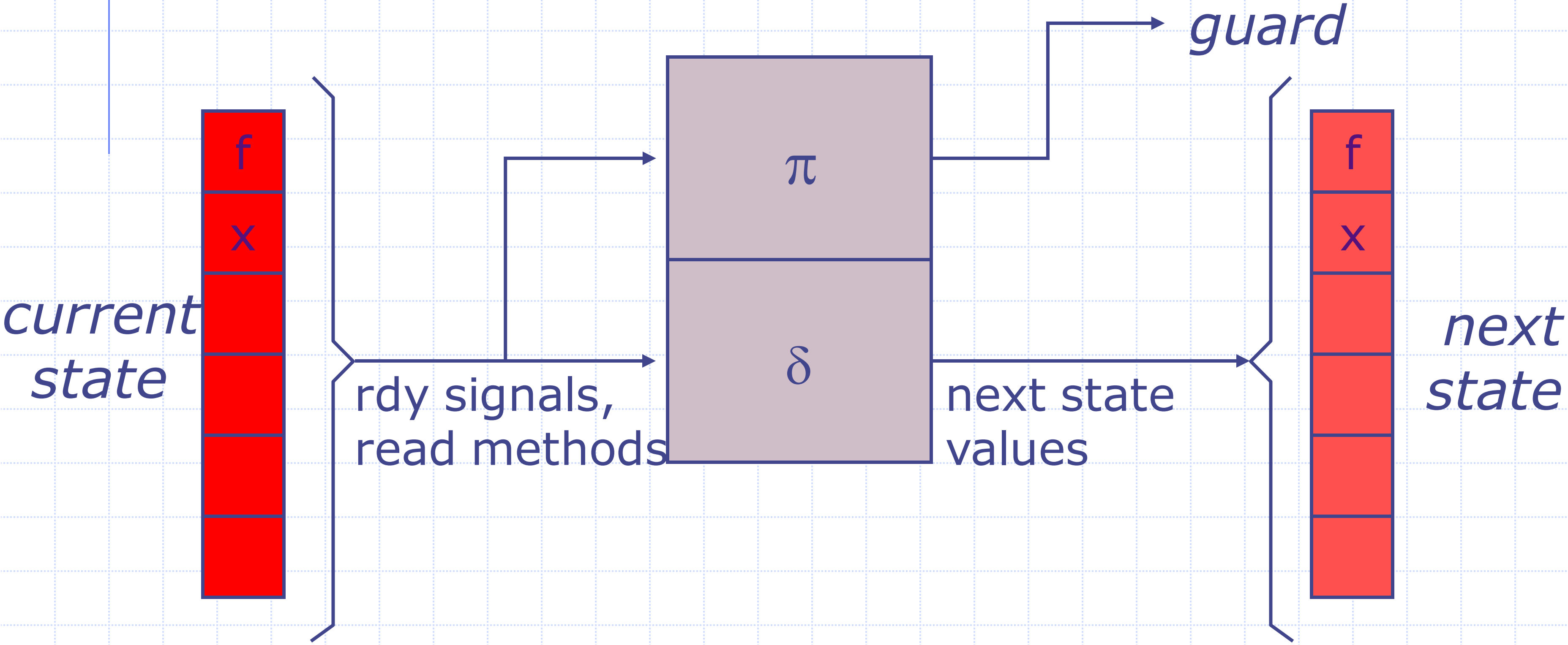
Such a scheduler can be built as a pure combinational circuit, *but* it is not fair

In practice it does fine, and one can get around it programmatically

Scheduling and Control Logic

Compiling a Rule

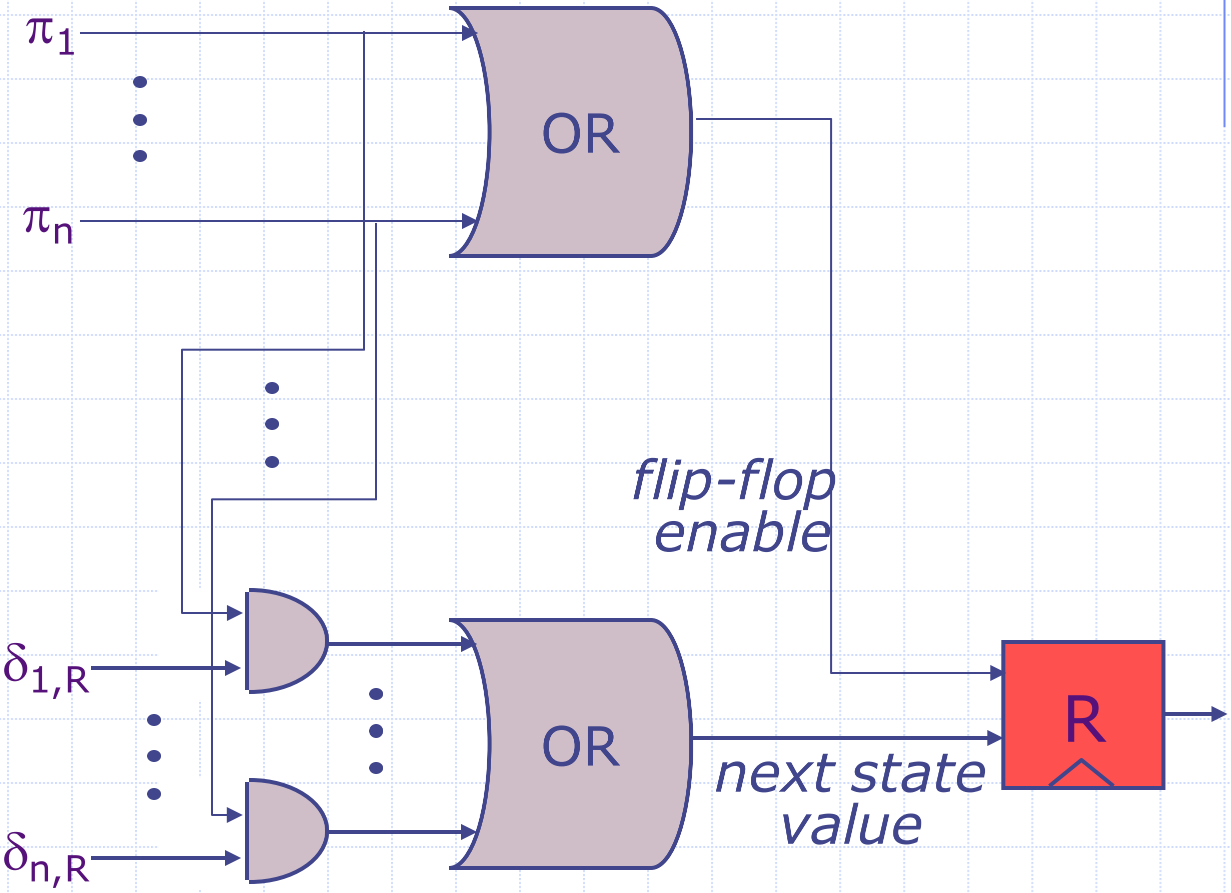
```
rule r (f.first() > 0) ;
          x <= x + 1 ;    f.deq ();
endrule
```



Combining State Updates: *strawman*

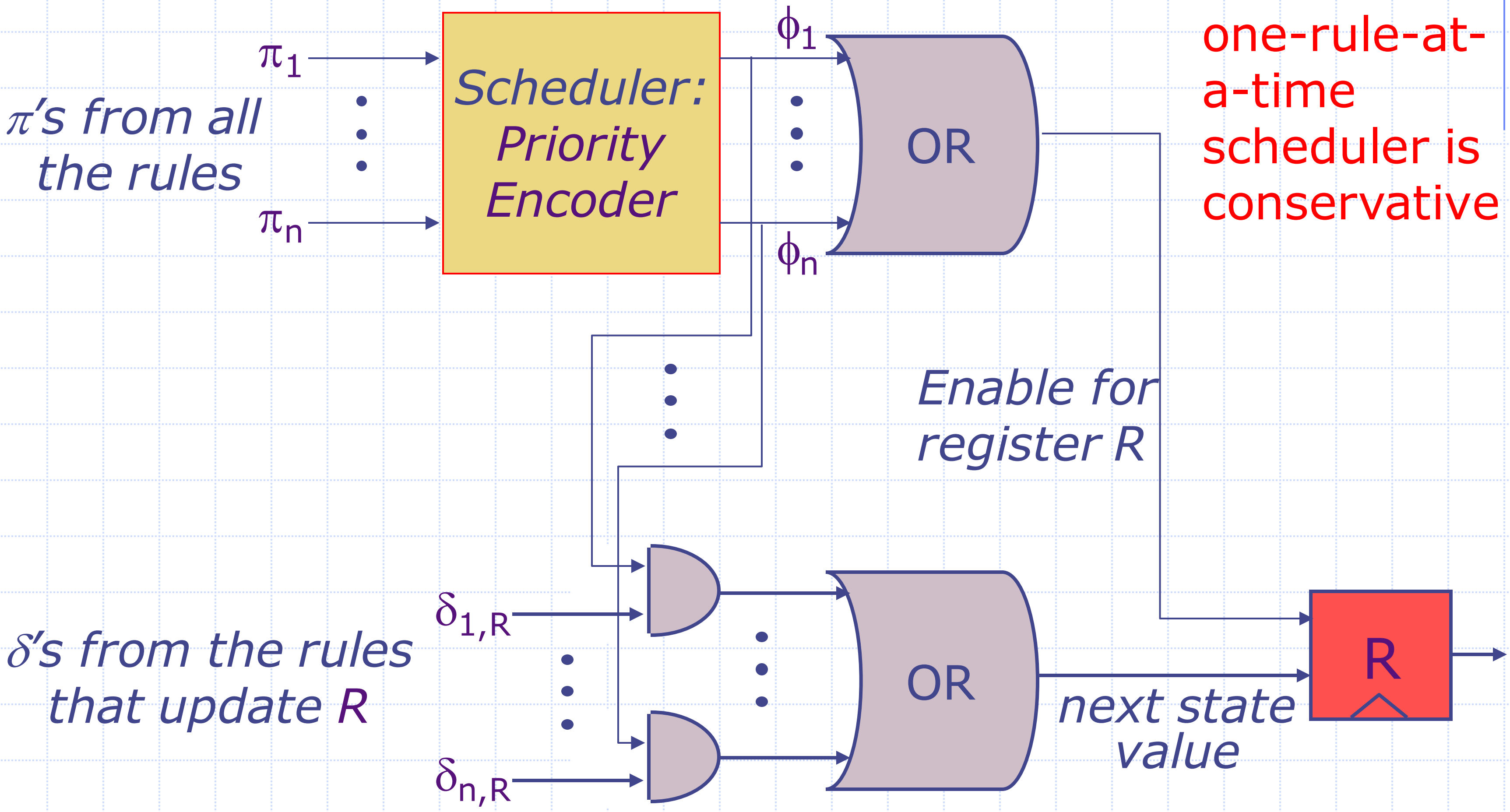
π's from the rules that update R

δ's from the rules that update R



What if more than one rule is enabled?

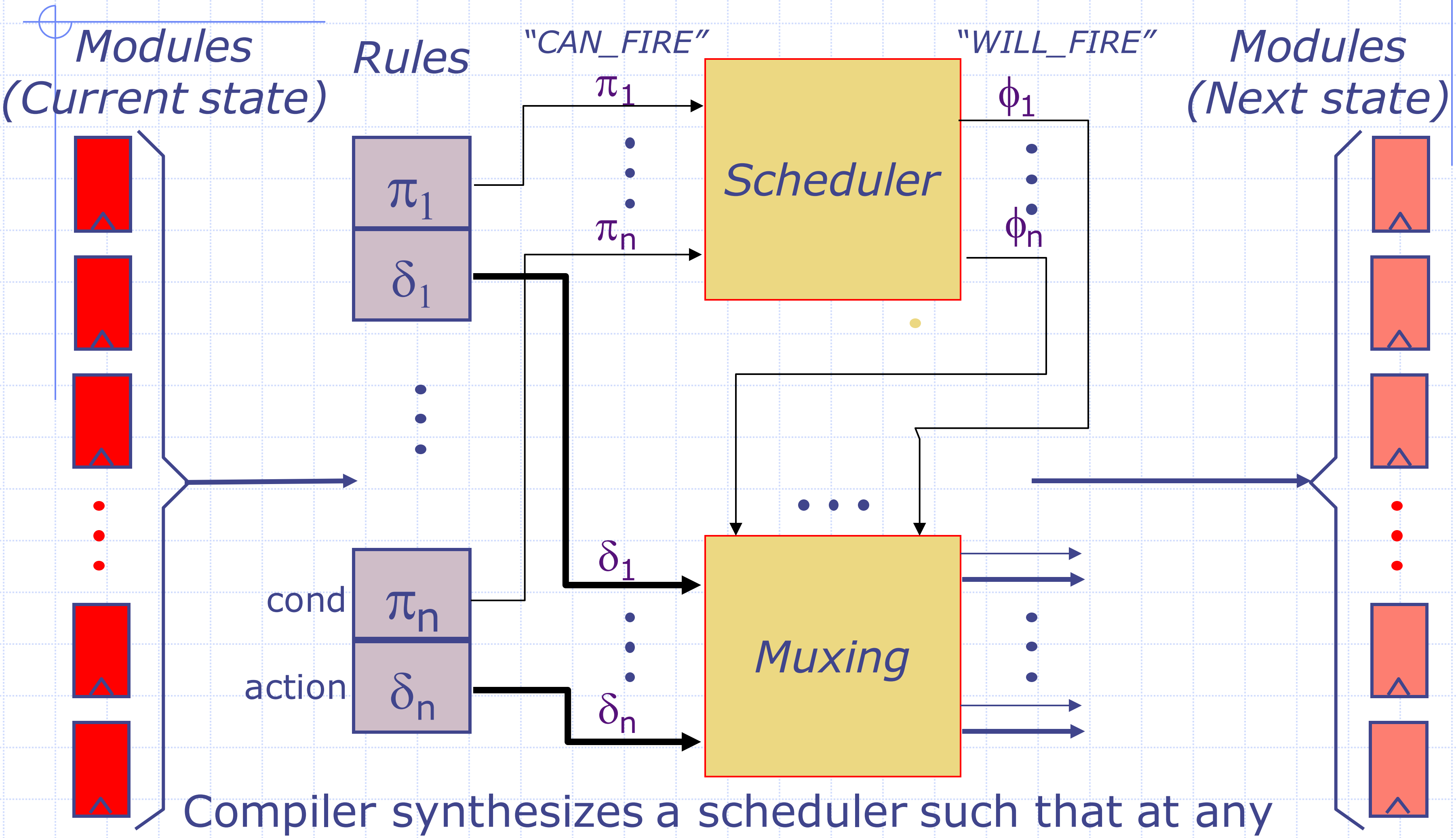
Combining State Updates



one-rule-at-a-time scheduler is conservative

Scheduler ensures that at most one ϕ_i is true

Scheduling and control logic



Compiler synthesizes a scheduler such that at any given time ϕ's for only non-conflicting rules are true

Takeaway

- One-rule-at-a-time semantics are very important to understand what behaviors a system can show
- Efficient hardware for multi-rule system requires that many rules execute in parallel without violating the one-rule-at-time semantics
- BSV compiler builds a scheduler circuit to execute as many rules as possible concurrently