Serializability of Concurrent Execution of Rules

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Linearizability

- *Rule atomicity* says that execution no other rule appears to be interleaved with the execution of a rule
- This is also known as *linearizability*, i.e., other rules appear to execute before or after a given rule
- The following example has no double write error but concurrent execution of ra and rb violates linearizability

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

initially x=0, y=0

$$\begin{tikzcd}
 & \text{Final value of } (x,y) \\
\text{ra ; rb} & (1,3) \arrow[dd, dash, bend left=40, "\neq"] \\
\text{rb ; ra} & (3,2) \arrow[d, dash, bend right=40, "\neq"'] \\
\text{Concurrent Execution} & (1,2)
\end{tikzcd}$$

we say rules ra and rb conflict and should not be executed concurrently

Serializability

- We could say that the concurrent execution of rules or methods is allowed as long as no double write error is possible and the execution is linearizable
- In fact, we impose the additional constraint of *serializability* on the concurrent execution of rules:

> *Serializability* means that a concurrent execution of rules must match some serial execution of rules, aka one-rule-at-a-time execution of rules

- The serializability constrain is imposed to make it easier to analyze the behavior of concurrent systems; it is a common and well established practice all distributed systems and databases

> In the hardware domain the idea of serializability is new at the design level but it has been used extensively in proving properties of designs

An example to illustrate Serializability

```
rule ra;
   x <= y+1;
endrule
rule rb;
   y <= z+2;
endrule
rule rc;
   z <= x+3;
endrule
```

initially
`x=0,y=0,z=0`

Any two rules can be executed concurrently but not all three

(x,y,z) after each cycle

Sequential Executions	
ra ; rb ; rc	(1,0,0) -> (1,2,0) -> (1,2,4)
ra ; rc ; rb	(1,0,0) -> (1,0,4) -> (1,6,4)
rb ; rc ; ra	(0,2,0) -> (0,2,3) -> (3,2,3)
rb ; ra ; rc	(0,2,0) -> (3,2,0) -> (3,2,6)
rc ; ra ; rb	(0,0,3) -> (1,0,3) -> (1,5,3)
rc ; rb ; ra	(0,0,3) -> (0,5,3) -> (6,5,3)

Parallel Executions		
ra | rb | rc	(1,2,3)	not allowed
(ra | rb) ; rc	(1,2,0) -> (1,2,4)	ra < rb
rc ; (ra | rb)	(0,0,3) -> (1,5,3)	
ra ; (rb | rc)	(1,0,0) -> (1,2,4)	rb < rc
(rb | rc) ; ra	(0,2,3) -> (3,2,3)	
rb ; (ra | rc)	(0,2,0) -> (3,2,3)	rc < ra
(ra | rc) ; rb	(1,0,3) -> (1,5,3)	

Notice ra<rb, rb<rc does not imply that ra<rc

Why is serializability important?

- As you have seen it is straight forward to build hardware so that ra and rb will execute concurrently. However, in general, it is difficult to derive the behavior of the resulting circuit
- Serializability, lets us apply one rule at a time in some order to derive the behavior of the composite system
- Without serializabilty, the atomicity of each rule has no meaning in a complex system
- Even though serializability imposes an additional constraint, and will make us reject some RTL implementations for a Bluespec design, in practice its advantages far outweigh its disadvantages in debugging and verification

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

initially (x,y) are (0,0)

Concurrent Execution	(1,2)
ra < rb	(1,3)
rb < ra	(3,2)

Conflict Matrix

- We can define a Conflict Matrix (CM), which specifies for a given pair of methods, or a pair of rules, or a method and rule, the effect of concurrent execution
  - ra < rb : ra and rb can be executed concurrently; the net effect is as if ra executed before rb
  - ra CF rb: ra and rb can be executed concurrently; the net effect is the same as (ra<rb) and (rb<ra)
  - ra C rb: ra and rb *Conflict*; either the concurrent execution will cause a *double-write* error or the resulting effect is neither (ra<rb) nor (rb<ra)
  - ra ME rb: the guards of ra and rb are mutually exclusive and thus, ra and rb can never be rdy together

The derivation of CM

- There is a natural ordering between the values of CM entries

$$\begin{tikzcd}
& \mathrm{CF} = \{<,>\} & \\
\{<\} \arrow[ur, dash] & & \{>\} \arrow[ul, dash] \\
& \mathrm{C} = \{\} \arrow[ul, dash] \arrow[ur, dash] &
\end{tikzcd}$$

  - This ordering permits us to take intersections of conflict information, e.g.,
    - $\{>\} \cap \{<,>\} = \{>\}$
    - $\{>\} \cap \{<\} = \{\}$
- We use the CM of primitive modules (register, EHR) to derive the CM for the interface methods of a module

Deriving the Conflict Matrix (CM) of a module interface

- Let g1 and g2 be the two methods defined by a module, such that

Methods called by g1

mcalls(g1)={g11,g12...g1n}

mcalls(g2)={g21,g22...g2m}

  - CM[g1,g2] = conflict(g11,g21) ∩ conflict(g11,g22) ∩...
    ∩ conflict(g12,g21) ∩ conflict(g12,g22) ∩...
    …
    ∩ conflict(g1n,g21) ∩ conflict(g1n,g22) ∩...
  - conflict(x,y) = if x and y are methods of the same module then CM[x,y] else CF

Compiler can derive the CM for a module by starting with the innermost modules in the module instantiation tree

CM for rules

- The conflict between two rules or a rule and a method can be derived in a similar manner by examining the CM properties of the constituent method calls

Example 1

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

	ra	rb
ra	C	CF
rb	CF	C

Example 2

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

	ra	rb
ra	C	C
rb	C	C

Example 3

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

	ra	rb
ra	C	<
rb	>	C

Example 1: Compiler Analysis

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

```
mcalls(ra) = {x.w, x.r}
mcalls(rb) = {y.w, y.r}

CM(ra, rb) =
   conflict(x.w, y.w) ∩ conflict(x.w, y.r)
 ∩ conflict(x.r, y.w) ∩ conflict(x.r, y.r)
            = CF ∩ CF ∩ CF ∩ CF
            = CF
```

Rules ra and rb can be scheduled together without violating the one-rule-at-a-time-semantics. We say rules ra and rb are CF

Example 2: Compiler Analysis

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

```
mcalls(ra) = {x.w, y.r}
mcalls(rb) = {y.w, x.r}

CM(ra, rb) =
   conflict(x.w, y.w) ∩ conflict(x.w, x.r)
 ∩ conflict(y.r, y.w) ∩ conflict(y.r, x.r)
       = CF ∩ {>} ∩ {<} ∩ CF
       = C
```

Rules ra and rb cannot be scheduled together without violating the one-rule-at-a-time semantics.
Rules ra and rb Conflict

Example 3: Compiler Analysis

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

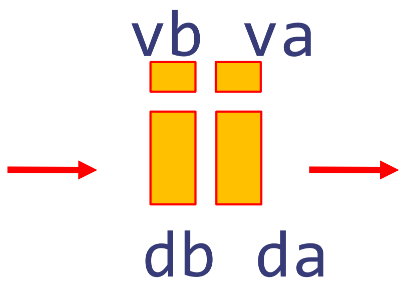
```
mcalls(ra) = {x.w, y.r}
mcalls(rb) = {y.w, y.r}

CM(ra, rb) =
  conflict(x.w, y.w) ∩ conflict(x.w, y.r)
∩ conflict(y.r, y.w) ∩ conflict(y.r, y.r)
        =  CF ∩ CF ∩ {<} ∩ CF
        = {<}
```

Rules ra and rb can be scheduled together without violating the one-rule-at-a-time-semantics.

Rule ra < rb

Two-Element FIFO
Deriving the CM

```
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
```

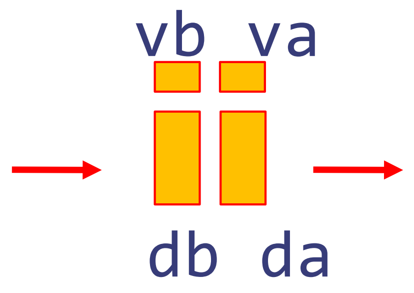


We can derive a conservative CM by ignoring the conditionals

```
  mcalls(enq) = {vb.r, va.r, db.w, vb.w, da.w, va.w}
  mcalls(deq) = {va.r, vb.r, da.w, db.r, vb.w, va.w}

  CM[enq,deq] =
  CM[vb.r,va.r]∩CM[vb.r,vb.r]∩CM[vb.r,da.w]∩CM[vb.r,db.r]∩CM[vb.r,vb.w]∩CM[vb.r,va.w]
∩CM[va.r,va.r]∩CM[va.r,vb.r]∩CM[va.r,da.w]∩CM[va.r,db.r]∩CM[va.r,vb.w]∩CM[va.r,va.w]
∩CM[db.w,va.r]∩CM[db.w,vb.r]∩CM[db.w,da.w]∩CM[db.w,db.r]∩CM[db.w,vb.w]∩CM[db.w,va.w]
∩CM[vb.w,va.r]∩CM[vb.w,vb.r]∩CM[vb.w,da.w]∩CM[vb.w,db.r]∩CM[vb.w,vb.w]∩CM[vb.w,va.w]
∩CM[da.w,va.r]∩CM[da.w,vb.r]∩CM[da.w,da.w]∩CM[da.w,db.r]∩CM[da.w,vb.w]∩CM[da.w,va.w]
∩CM[va.w,va.r]∩CM[va.w,vb.r]∩CM[va.w,da.w]∩CM[va.w,db.r]∩CM[va.w,vb.w]∩CM[va.w,va.w]
                                   = CF ∩ {<} ∩ CF ∩ {<} ∩ {>} ∩ {>} ∩ C ∩ C ∩ {>} ∩ C
                                   = C
```

Two-Element FIFO
More accurate 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
```



- For more accurate analysis we should consider the conditions under which both rules will be ready, i.e., va =True and vb = False

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

Thus, enq and deq do not conflict but the BSV complier is unable to deduce this

Using the CM to Enforce Serializability

- Suppose we are given a "rule ordering", that is, our preference about the behavior we would like to see given a set of rules
- We can keep scheduling the rules in that order, and if we find a conflict with an already scheduled rule we skip that rule and go to the next one
- Mathematically, given the list $\{r_1, r_2, \ldots r_n\}$

$$\begin{aligned}\text{will-fire}(r_i) &= \text{can-fire}(r_i)\ \&\& \\ &\quad \forall_{k<i}\ \{\text{if will-fire}(r_k)\text{ then }(CM[r_k, r_i] \cap \{<\}) = \{<\}\} \\ &= \text{can-fire}(r_i)\ \&\& \\ &\quad \forall_{k<i}\ \{!\text{will-fire}(r_k)\ ||\ (CM[r_k, r_i] \cap \{<\}) = \{<\}\}\end{aligned}$$

An example schedule

Ex 4

```
rule ra;
   x <= y+1;
endrule
rule rb;
   y <= z+2;
endrule
rule rc;
   z <= x+3;
endrule
```

- CM = {(ra<rb), (rb<rc), (rc<ra)}
- Scheduling priority = {ra,rb,rc}

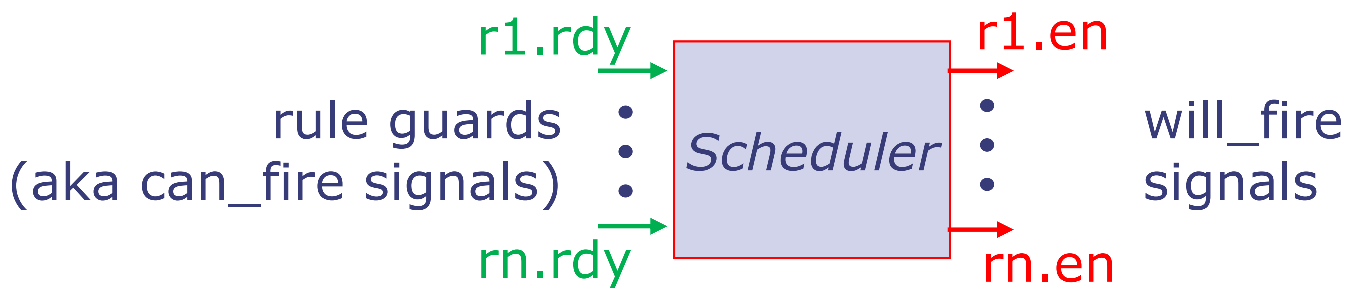
will-fire(r_i) = can-fire(r_i) &&
$\forall_{k<i}$ {!will-fire(r_k) || (CM[r_k, r_i]$\cap${<}) = {<}}

- can-fire(r_k) is true for all the rules every cycle

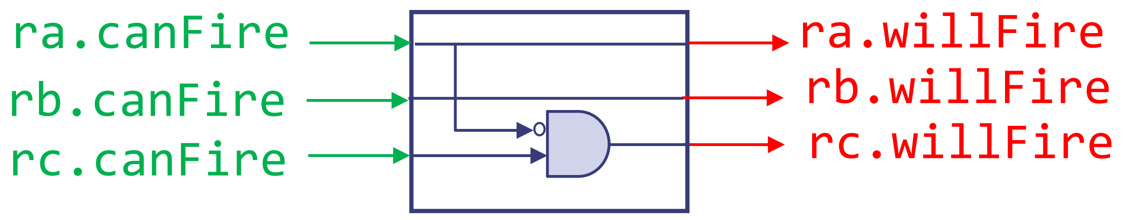
- will-fire(ra) = can-fire(ra)
- will-fire(rb) = can-fire(rb) &&
  (!can-fire(ra) || ((CM[ra, rb]$\cap${<}) = {<})
  = can-fire(rb) &&((! can-fire(ra)) || True)
  = can-fire(rb)
- will-fire(rc) = can-fire(rc) &&
  {(!(can-fire(ra)) || (CM[ra, rc]$\cap${<}) = {<})
  && (!can-fire(rb) || (CM[rb, rc]$\cap${<}) = {<}))}
  = can-fire(rc) &&
  {(!(can-fire(ra))||False)) && (!can-fire(rb)||True)}
  = can-fire(rc)&&!can-fire(ra)

Synthesis of the scheduler

- CM = {(ra<rb), (rb<rc), (rc<ra)}
- Scheduling priority = {ra,rb,rc}



- will-fire(ra) = can-fire(ra)
- will-fire(rb) = can-fire(rb)
- will-fire(rc) = can-fire(rc)&&!can-fire(ra)



Since in Ex 4, all `canFire` signals are true, the scheduler gets simplified to (True,True,False)

Slightly modified example

```
rule ra if (y==1);
   x <= y+1;
endrule
rule r;
   y <= z+2;
endrule
rule rc;
   z <= x+3;
endrule          Ex 5
```

- CM = {(ra<rb), (rb<rc), (rc<ra)}
- Scheduling priority = {ra,rb,rc}

$$\text{will-fire}(r_i) = \text{can-fire}(r_i)\ \&\&\ \forall_{k<i}\ \{!\text{will-fire}(r_k)\ \|\ (CM[r_k, r_i] \cap \{<\}) = \{<\}\}$$

- can-fire(r_k) is true for all the rules every cycle

- will-fire(ra) = can-fire(ra)
- will-fire(rb) = can-fire(rb)
- will-fire(rc) = can-fire(rc)&&!can-fire(ra)

Simplification for Ex 5 will lead to

- will-fire(ra) = (y==1)
- will-fire(rb) = True
- will-fire(rc) = !(y==1)

Preserving atomicity while preventing a rule from firing

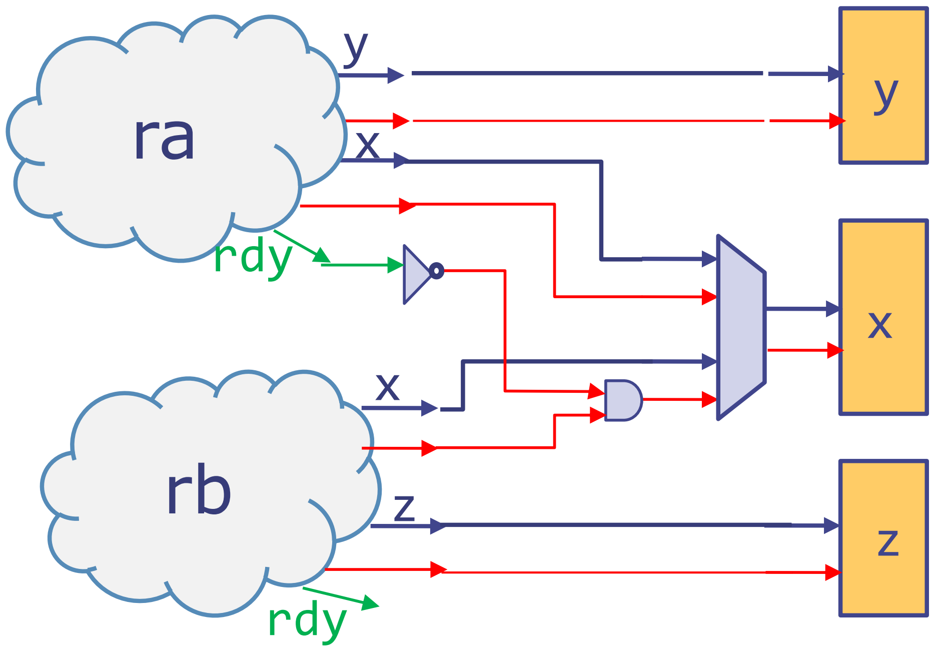
```
rule ra;
   x <= e1; y <= e2;
endmethod
rule rb;
   x <= e3; z <= e4;
endmethod
```

- ra and rb conflict because of a double write in x
- Suppose we want to prevent rb from firing

What is wrong with this circuit?

The atomicity of rule rb is violated: y may be updated without x being updated!

fix?

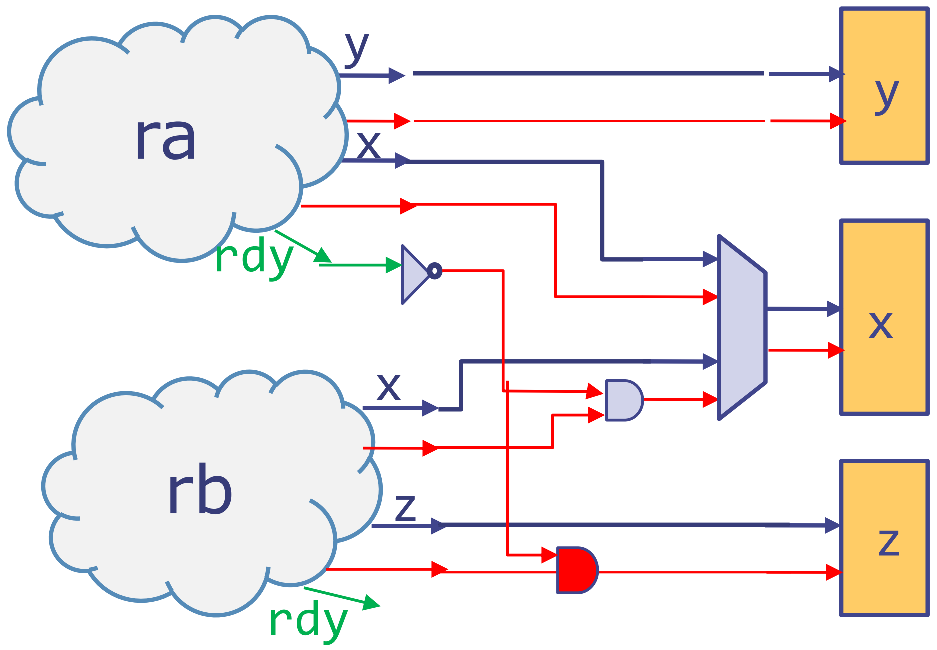


Preserving atomicity while preventing a rule from firing

```
rule ra;
   x <= e1; y <= e2;
endmethod
rule rb;
   x <= e3; z <= e4;
endmethod
```

- ra and rb conflict because of a double write in x
- Suppose we want to prevent rb from firing

What is wrong with this circuit?

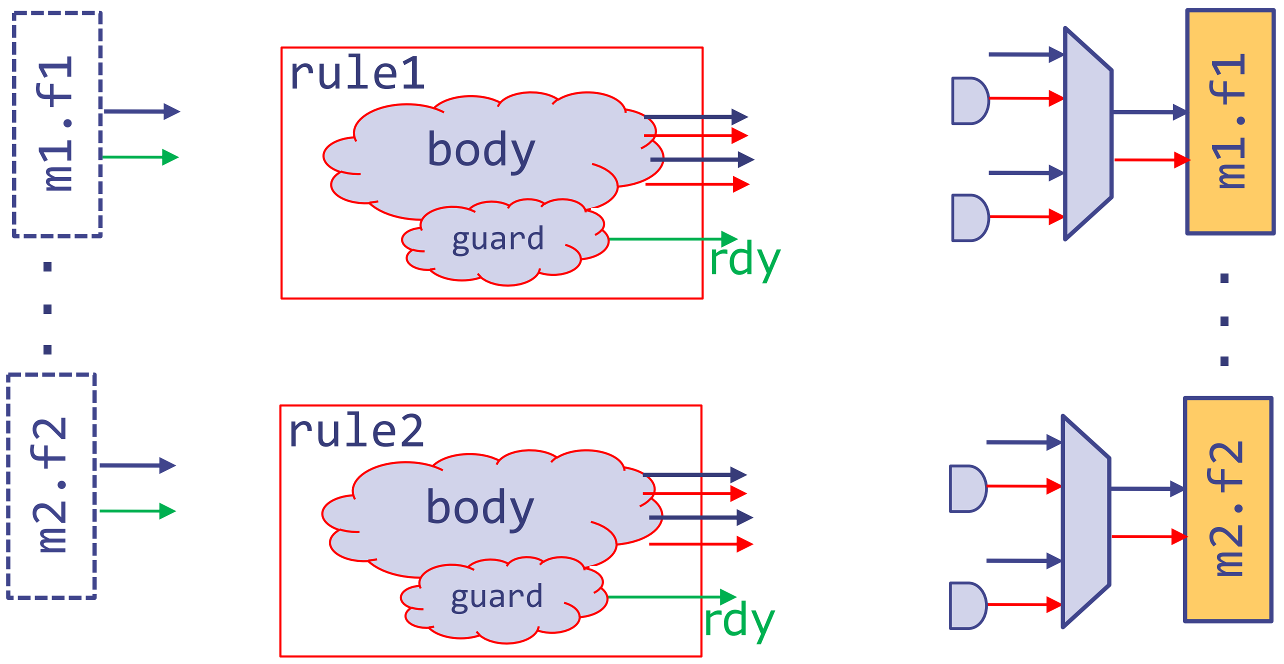


The atomicity of rule rb is violated: y may be updated without x being updated!

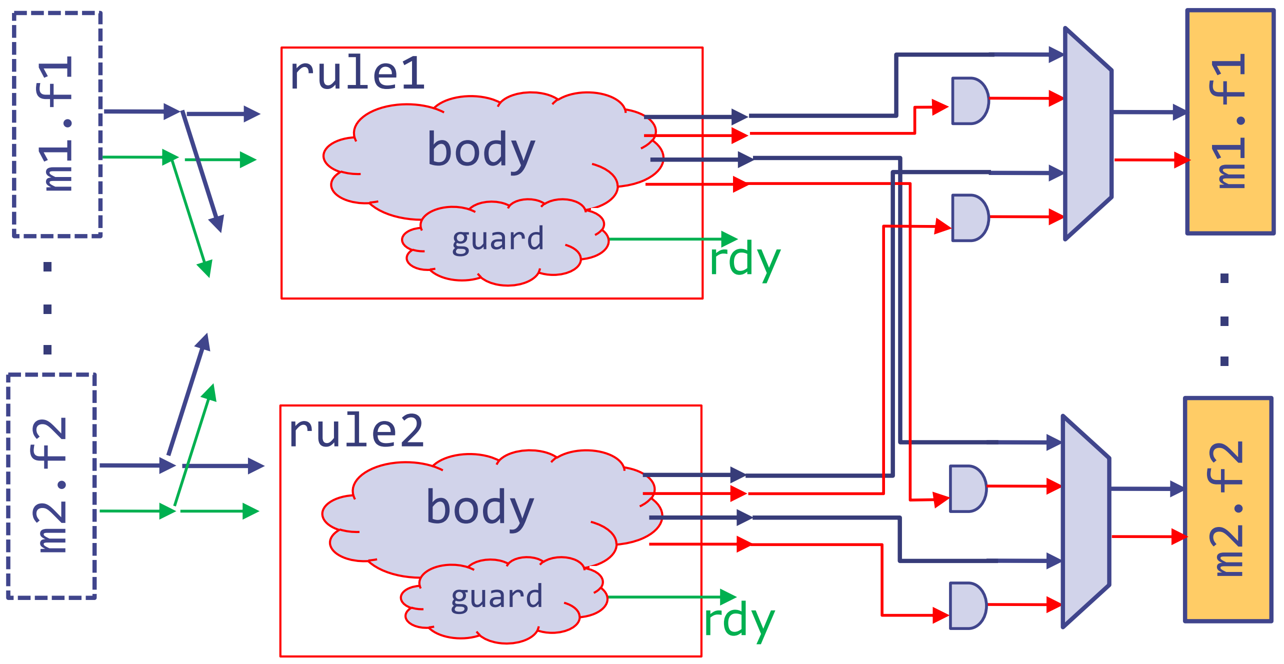
fix?

When we do not want a rule to fire all its state updates must be stopped

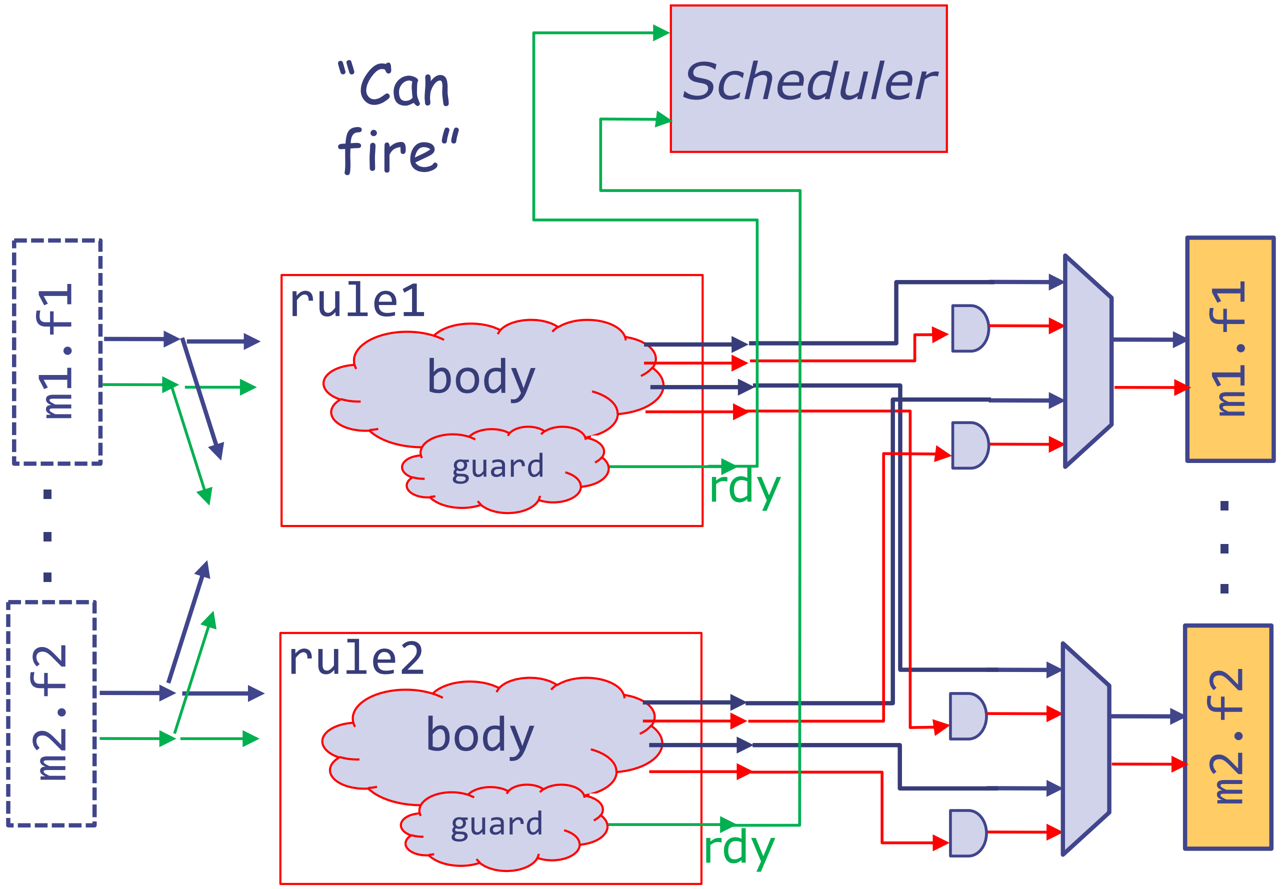
A general method for inhibiting rule execution



A general method for inhibiting rule execution

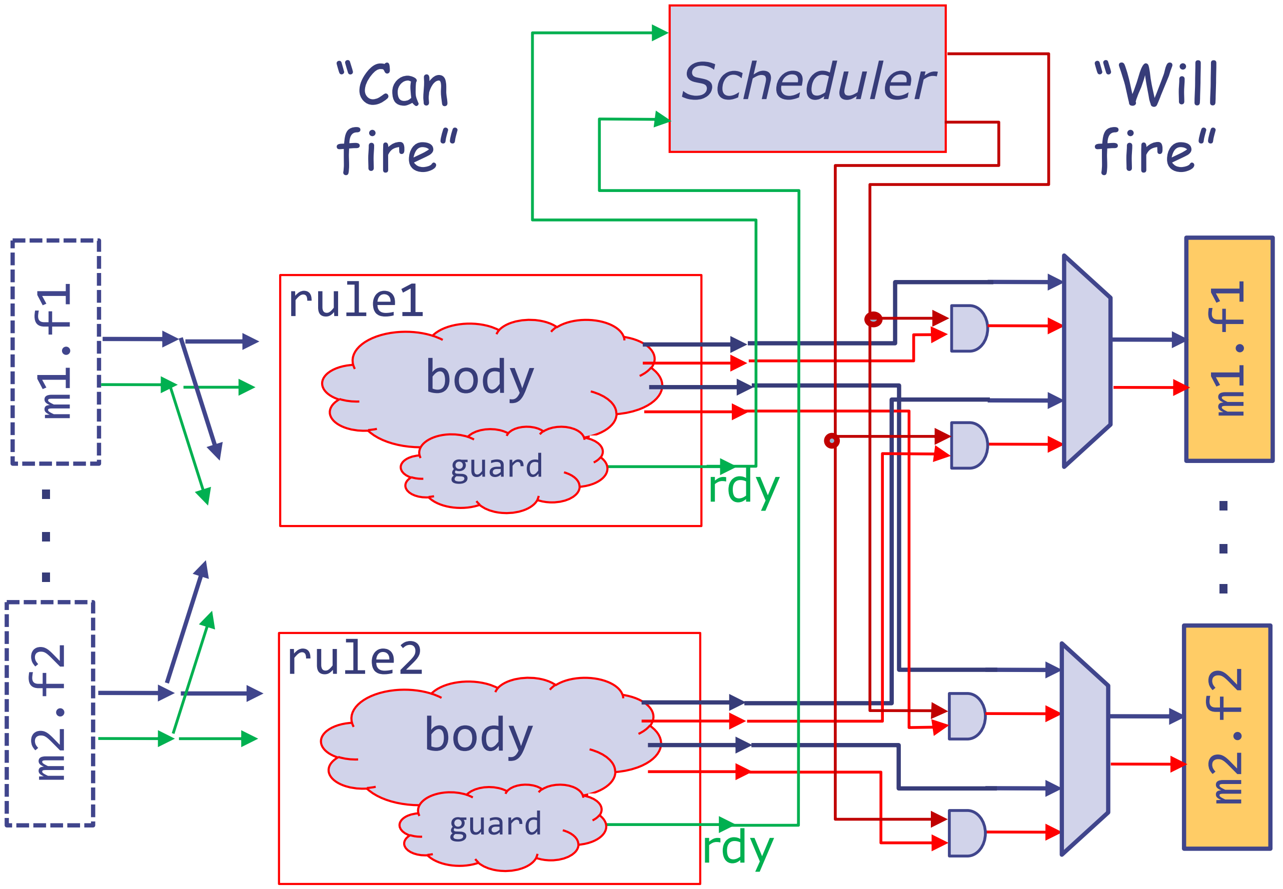


A general method for inhibiting rule execution



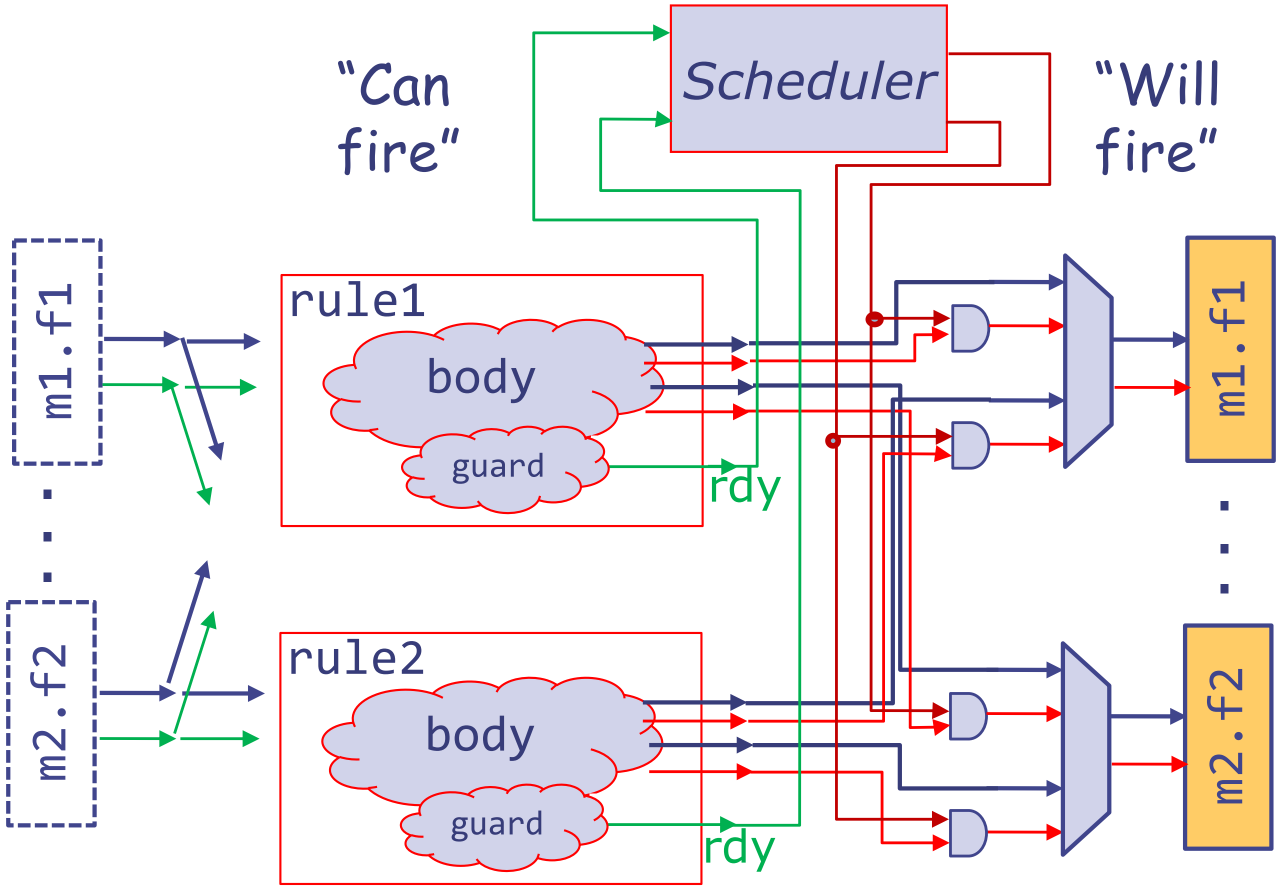
- We introduce a scheduler to control which rules among the ready rules should execute
  - We feed it the rdy signals of all the rules

A general method for inhibiting rule execution



- We introduce a scheduler to control which rules among the ready rules should execute
  - We feed it the rdy signals of all the rules
- The scheduler lets only *non-conflicting* rules proceed
  - It turns off some of the "can fire" signals

A general method for inhibiting rule execution

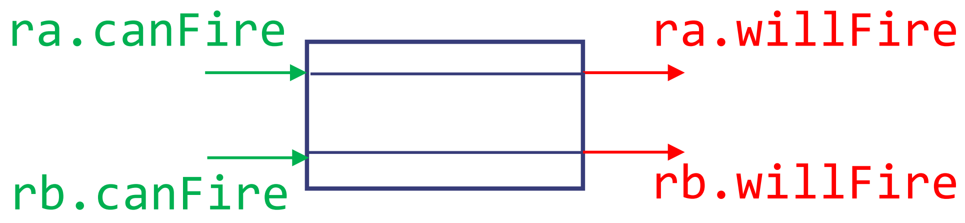


Scheduler is a pure combinational circuit with a small number of gates

- We introduce a scheduler to control which rules among the ready rules should execute
  - We feed it the rdy signals of all the rules
- The scheduler lets only *non-conflicting* rules proceed
  - It turns off some of the "can fire" signals

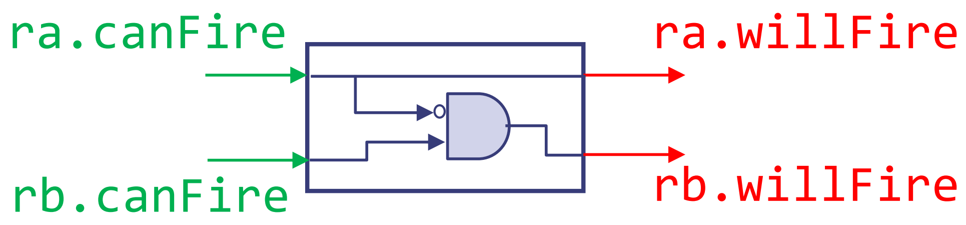
What is inside the scheduler

- Suppose rules ra and rb can be executed concurrently – no double write
    - Scheduler

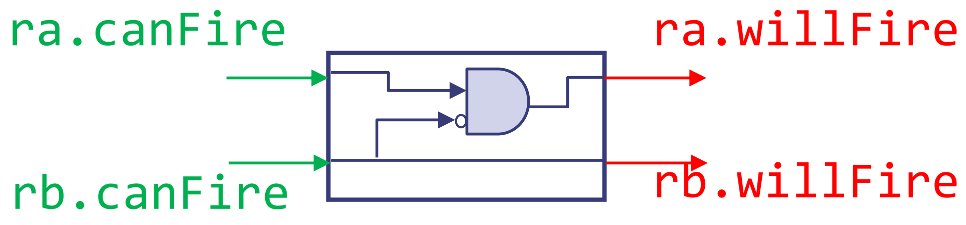


What is inside the scheduler

- Suppose rules ra and rb should not be executed concurrently
  - Schedule 1: rule ra has priority, i.e., if ra can fire rb will not fire



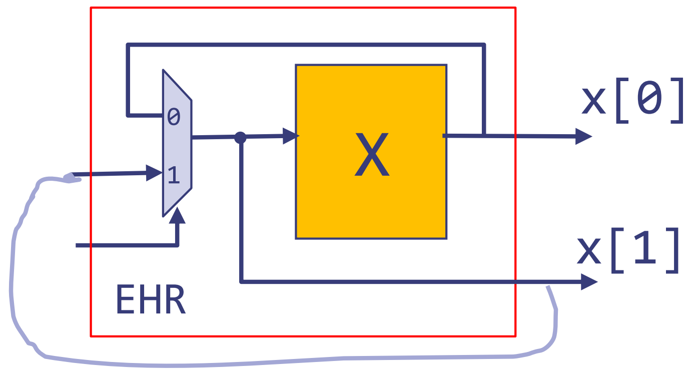
  - Schedule 2: rule rb has priority, i.e., if rb can fire ra will not fire



The choice is specified by scheduling annotations in the BSV program

Combinational cycles

- Rules containing following types of actions will be rejected by the BSV compiler because they are meaningless and will generate combinational cycles
  - x[0] <= x[1]
  - x[0] <= y[1]; y[0] <= x[1]
  - if (x[1]) x[0] <= e;



Takeaway

- One-rule-at-a-time semantics are important to understand the legal behaviors of a system
- 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
  - It takes user advice in scheduling conflicting rules
- For high-performance designs we have to worry about the CM characteristics of our modules