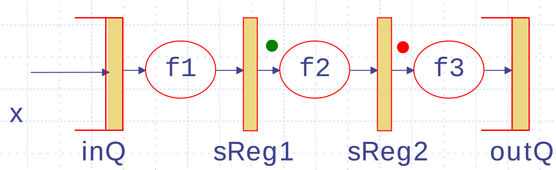
Basics of Multi-rule Systems

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

September 17, 2009

Synchronous Pipeline




```
rule sync-pipeline (True);
  inQ.deq();
  sReg1 <= f1(inQ.first());
  sReg2 <= f2(sReg1);
  outQ.enq(f3(sReg2));
endrule
```

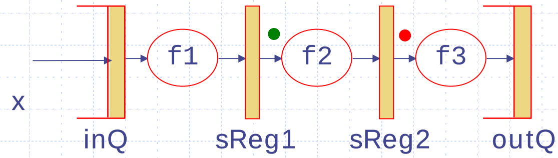
Red and Green tokens must move even if there is nothing in the inQ!

Also if there is no token in sReg2 then nothing should be enqueued in the outQ

Modify the rule to deal with these conditions

Valid bits or the Maybe type

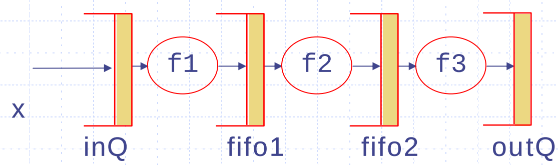
Synchronous Pipeline using the Maybe type data




```
rule sync-pipeline (True);
 if (inQ.notEmpty())
  begin sReg1 <= Valid f1(inQ.first()); inQ.deq(); end
  else  sReg1 <= Invalid;
 case (sReg1) matches
  tagged Valid .sx1: sReg2 <= Valid f2(sx1);
  tagged Invalid:    sReg2 <= Invalid;
 case (sReg2) matches
  tagged Valid .sx2: outQ.enq(f3(sx2));
endrule
```


Asynchronous pipeline

Use FIFOs instead of pipeline registers




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

Firing conditions?

Asynchronous pipeline: Some Issues

- Easier to write but will not behave like a pipeline unless all rules can execute simultaneously
- It must be possible to enqueue and dequeue in a FIFO simultaneously

Rule scheduling and the synthesis of a scheduler

Guarded Atomic Actions (GAA): Execution model

Repeatedly:

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

Implementation concern: Schedule multiple rules concurrently without violating one-rule-at-a-time semantics

Rule: As a State Transformer

A rule may be decomposed into two parts $\pi(s)$ and $\delta(s)$ such that

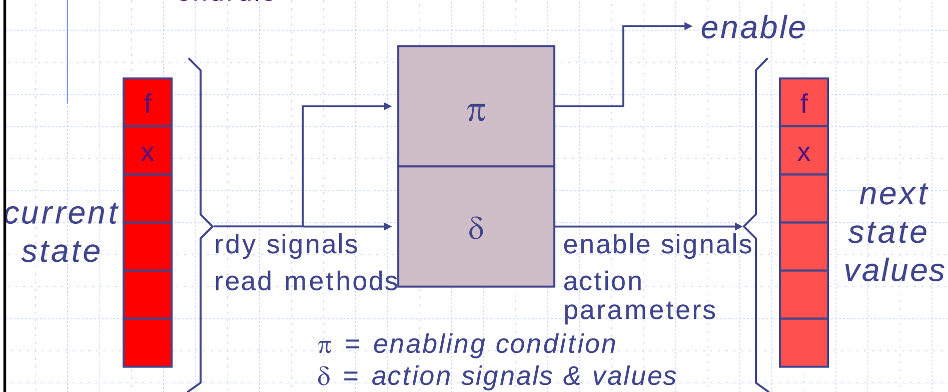
$$s_{next} = \textit{if } \pi(s) \textit{ then } \delta(s) \textit{ else } s$$

$\pi(s)$ is the condition (predicate) of the rule, a.k.a. the "CAN_FIRE" signal of the rule. π is a conjunction of explicit and implicit conditions

$\delta(s)$ is the "state transformation" function, i.e., computes the next-state values from the current state values

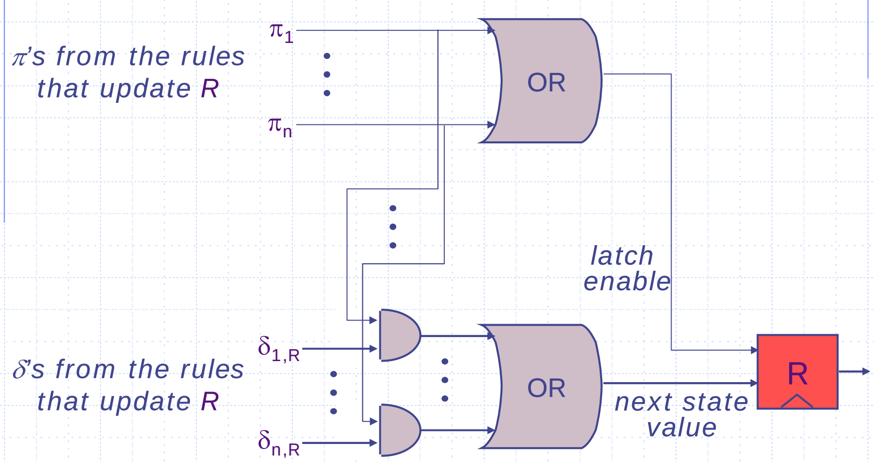
Compiling a Rule

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

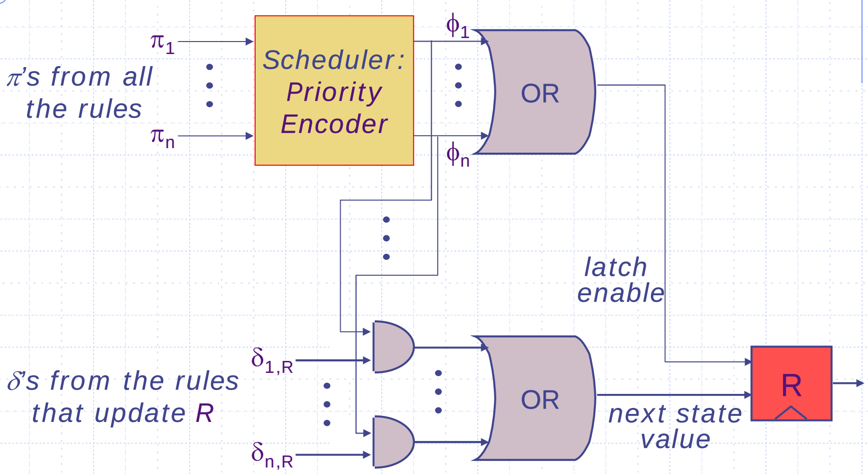


Combining State Updates:

strawman

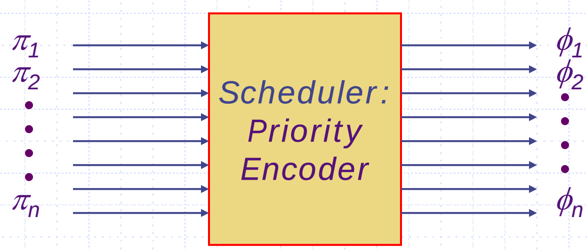


Combining State Updates



Scheduler ensures that at most one ϕ_i is true

One-rule-at-a-time Scheduler



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. One rewrite at a time
   i.e. at most one ϕ_i is true

A compiler can determine if two rules can be executed in parallel without violating the one-rule-at-a-time semantics

James Hoe, Ph.D., 2000

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 or equivalently rb < ra

$Rule_a$ 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. $\delta_a(\delta_b(s)) == \delta_b(\delta_a(s))$

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 by definition

Executing Multiple Rules Per Cycle: *Sequentially Composable rules*

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

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

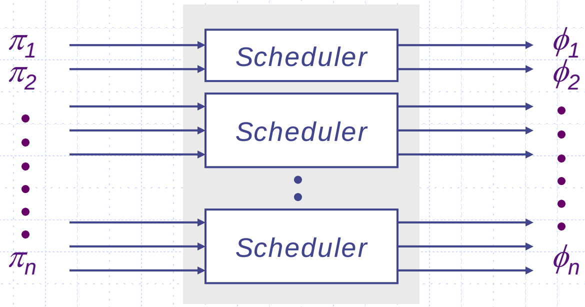
Parallel execution behaves like ra < rb

- R(rb) is the range of rule rb
- Prj_{st} is the projection selecting st from the total state

$Rule_a$ and $Rule_b$ are sequentially composable if

$$\forall s \,.\, \pi_a(s) \wedge \pi_b(s) \Rightarrow 1.\ \pi_b(\delta_a(s))$$
$$2.\ Prj_{R(rb)}(\delta_b(s)) == Prj_{R(rb)}(\delta_b(\delta_a(s)))$$

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. $\pi_1 \vee \pi_2 \vee \ldots. \vee \pi_n \Rightarrow \phi_1 \vee \phi_2 \vee \ldots. \vee \phi_n$
3. Multiple operations such that

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

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. $\delta_a(\delta_b(s)) == \delta_b(\delta_a(s))$

$D(Ra) \cap R(Rb) = \phi$
$D(Rb) \cap R(Ra) = \phi$
$R(Ra) \cap R(Rb) = \phi$

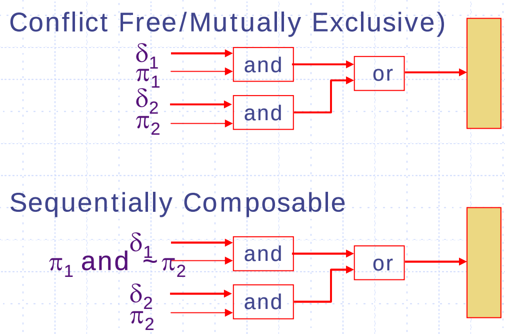
$Rule_a$ and $Rule_b$ are sequentially composable if
$\forall s \,.\, \pi_a(s) \wedge \pi_b(s) \Rightarrow$
1. $\pi_b(\delta_a(s))$
2. $Prj_{R(Rb)}(\delta_b(s)) == Prj_{R(Rb)}(\delta_b(\delta_a(s)))$

$D(Rb) \cap R(Ra) = \phi$

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

Muxing structure

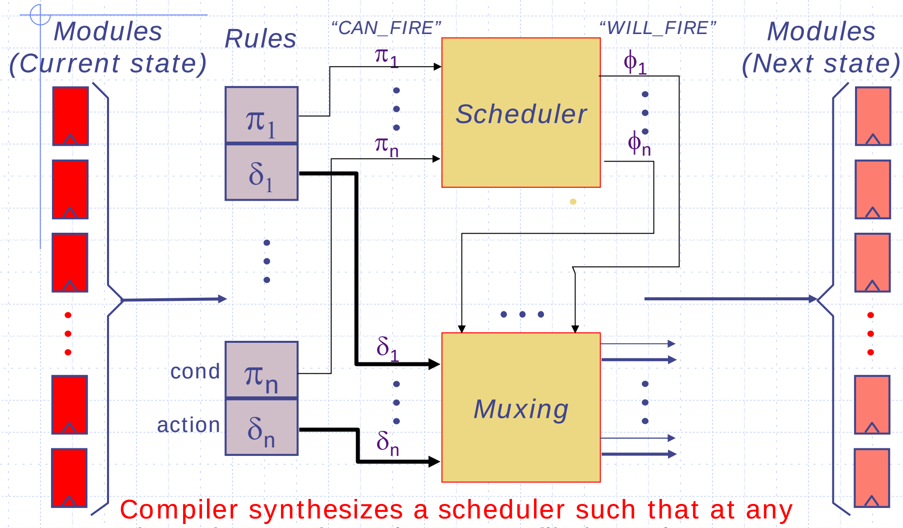
- Muxing logic requires determining for each register (action method) the rules that update it and under what conditions



If two CF rules update the same element then they must be *mutually exclusive*

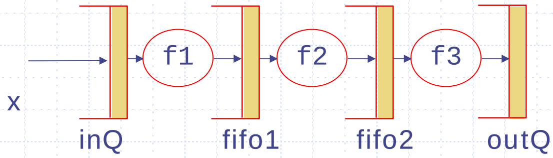
$(\pi_1 \rightarrow \sim\pi_2)$

Scheduling and control logic



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

Does our pipeline behave properly?




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

Can all three rules fire concurrently?

next time