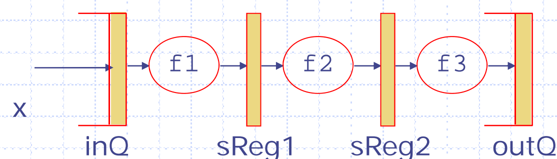
Elastic Pipelines and Basics of Multi-rule Systems

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

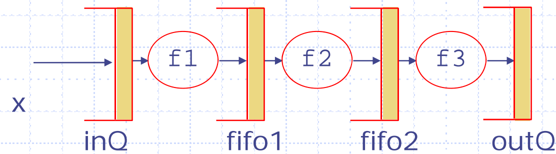
Inelastic Pipeline




```
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
```

Elastic 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?

Can tokens be left inside the pipeline?

No Maybe types?

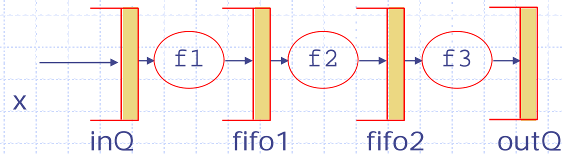
Easier to write?

Can all three rules fire concurrently?

Inelastic vs Elastic Pipelines

- In an Inelastic pipeline:
  - typically only one rule; the designer controls precisely which activities go on in parallel
  - *downside:* The rule can get too complicated -- easy to make a mistake; difficult to make changes
- In an Elastic pipeline:
  - several smaller rules, each easy to write, easier to make changes
  - *downside:* sometimes rules do not fire concurrently when they should

What behavior do we want?



- If inQ, fifo1 and fifo2 are not empty and fifo1, fifo2 and outQ are not full then we want all the three rules to fire
- If inQ is empty, fifo1 and fifo2 are not empty and fifo2 and outQ are not full then we want rules stage2 and stage3 to fire
- ...

Maximize concurrency - Fire maximum number of rules

The tension

- If multiple rules never fire in the same cycle then the machine can hardly be called a pipelined machine
- If all rules fire in parallel every cycle when they are enabled, then, in general, wrong results can be produced

Concurrency analysis and rule scheduling

Guarded Atomic Actions (GAA): Execution model

Repeatedly:

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

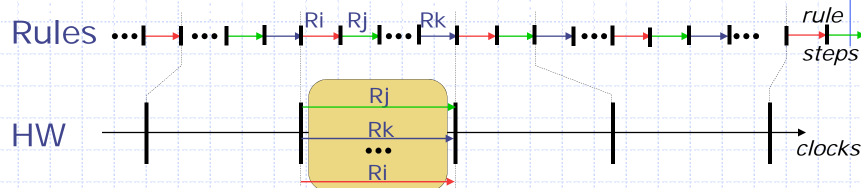
Highly non-deterministic

User annotations can help in rule selection

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

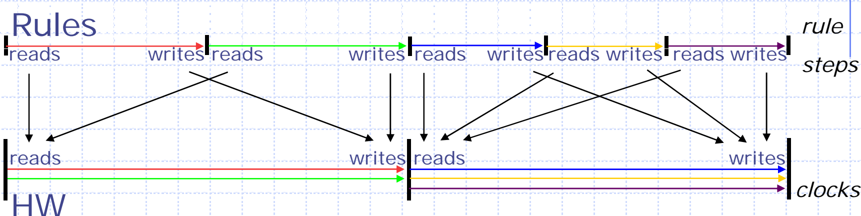
some insight into

Concurrent rule firing



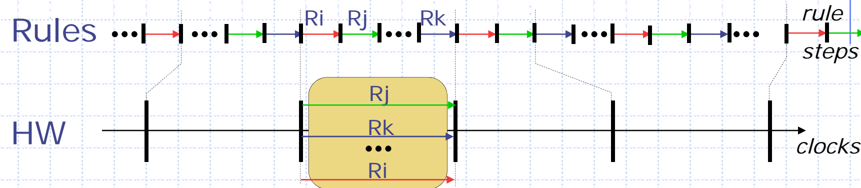
- There are more intermediate states in the rule semantics (a state after each rule step)
- In the HW, states change only at clock edges

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
- Consequence: the HW can never reach a state unexpected in the rule semantics

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

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

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

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

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

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

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

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

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

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

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 conditions are sufficient but not necessary

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

Parallel execution of CF and SC rules does not increase the critical path delay

Conflicting rules

```
rule ra (True);
   x <= y + 1;
endrule

rule rb (True);
   y <= x + 2;
endrule
```

Assume `x` and `y` are initially zero

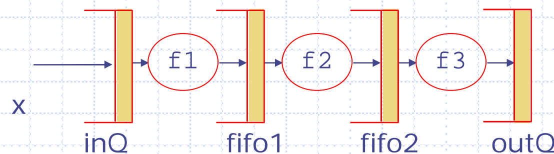
- Concurrent execution of these can produce x=1 and y=2 but these values cannot be produced by any sequential execution

The compiler issue

- Can the compiler detect all the conflicting conditions?
  - Important for correctness
- Does the compiler detect conflicts that do not exist in reality?
  - False positives lower the performance
  - The main reason is that sometimes the compiler cannot detect under what conditions the two rules are mutually exclusive or conflict free
- What can the user specify easily?
  - Rule priorities to resolve nondeterministic choice

In many situations the correctness of the design is not enough; the design is not done unless the performance goals are met

Concurrency in Elastic pipeline




```
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?

Consider rules stage1 and stage2:

No conflict around inQ or fifo2.

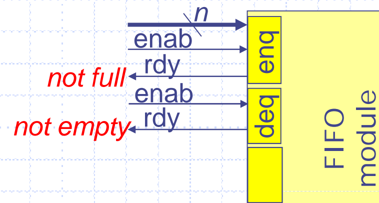
What can we assume about enq, deq and first methods of fifo1?

we want the FIFO to behave as if first < deq < enq

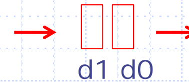
Concurrency in FIFOs

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
```



Two-Element FIFO

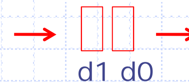



```
module mkFIFO (FIFO#(t));
  Reg#(t)     d0  <- mkRegU();
  Reg#(Bool)  v0  <- mkReg(False);
  Reg#(t)     d1  <- mkRegU();
  Reg#(Bool)  v1  <- mkReg(False);
  method Action enq(t x) if (!v1);
    if v0 then begin d1 <= x; v1 <= True; end
           else begin d0 <= x; v0 <= True; end endmethod
  method Action deq() if (v0);
    if v1 then begin d0 <= d1; v1 <= False; end
           else begin v0 <= False; end endmethod
  method t first() if (v0);
    return d0; endmethod
  method Action clear();
    v0<= False; v1 <= False; endmethod
endmodule
```

Assume, if there is only one element in the FIFO it resides in d0

Two-Element FIFO

another version




```
module mkFIFO (FIFO#(t));
  Reg#(t)     d0  <- mkRegU();
  Reg#(Bool)  v0  <- mkReg(False);
  Reg#(t)     d1  <- mkRegU();
  Reg#(Bool)  v1  <- mkReg(False);
  method Action enq(t x) if (!v1);
    v0 <= True; v1 <= v0;
    if v0 then d1 <= x; else d0 <= x; endmethod
  method Action deq() if (v0);
    v1 <= False; v0 <= v1; d0 <= d1; endmethod
  method t first() if (v0);
    return d0; endmethod
  method Action clear();
    v0<= False; v1 <= False; endmethod
endmodule
```

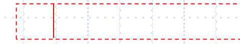
Assume, if there is only one element in the FIFO it resides in d0

enq and deq can be enabled together but apparently conflict

Compiler has no chance to be able to deduce the concurrency of enq and deq

RWire to rescue

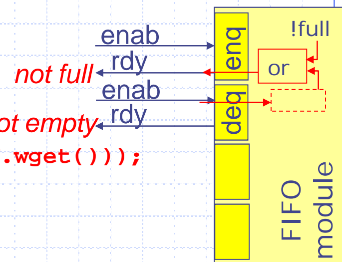
```
interface RWire#(type t);
      method Action wset(t x);
      method Maybe#(t) wget();
endinterface
```



Like a register in that you can read and write it but unlike a register
- read happens after write
- data disappears in the next cycle

One-Element Pipeline FIFO

```
module mkLFIFO1 (FIFO#(t));
  Reg#(t)    data  <- mkRegU();
  Reg#(Bool) full  <- mkReg(False);
  RWire#(void) deqEN <- mkRWire();
  Bool         deqp = isValid (deqEN.wget()));
  method Action enq(t x) if
                    (!full || deqp);
    full <= True;    data <= x;
  endmethod
  method Action deq() if (full);
    full <= False; deqEN.wset(?);
  endmethod
  method t first() if (full);
    return (data);
  endmethod
  method Action clear();
    full <= False;
  endmethod endmodule
```



FIFOs

- Ordinary one element FIFO
  - deq & enq conflict – won't do
- Pipeline FIFO
  - first < deq < enq < clear
- Bypass FIFO
  - enq < first < deq < clear

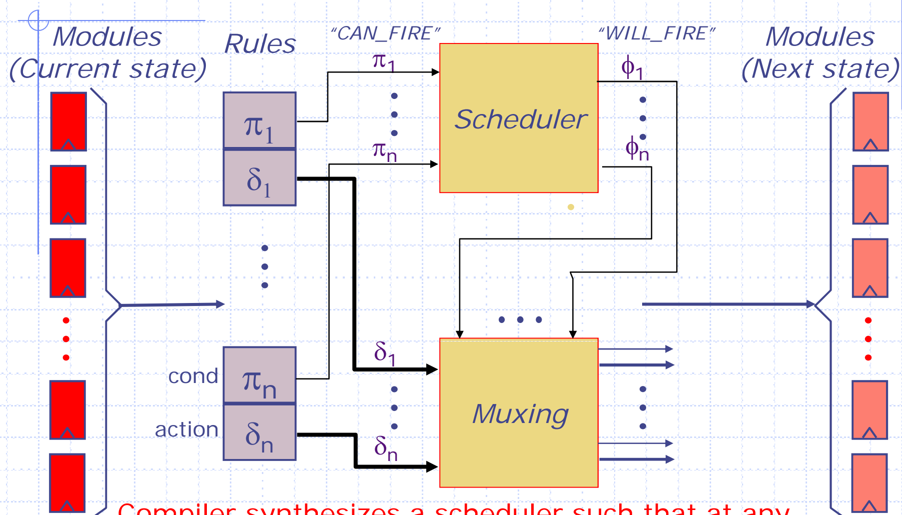
Takeaway

- FIFOs with concurrent operations are quite difficult to design, though the amount of hardware involved is small
  - FIFOs with appropriate properties are in the BSV library
- Various FIFOs affect performance but not correctness
- For performance, concentrate on high-level design and then search for modules with appropriate properties

Extras

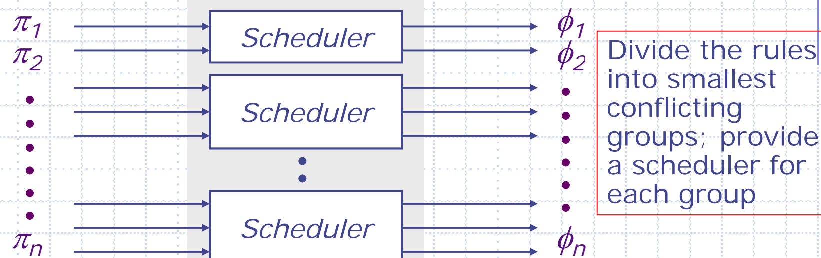
Scheduler synthesis

Scheduling and control logic



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

Multiple-Rules-per-Cycle Scheduler



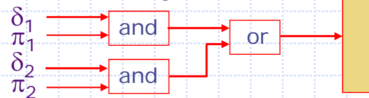
Divide the rules into smallest conflicting groups; provide a scheduler for each group

1. $\phi_i \Rightarrow \pi_i$
2. $\pi_1 \vee \pi_2 \vee \dots \vee \pi_n \Rightarrow \phi_1 \vee \phi_2 \vee \dots \vee \phi_n$
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)



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

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

Sequentially Composable