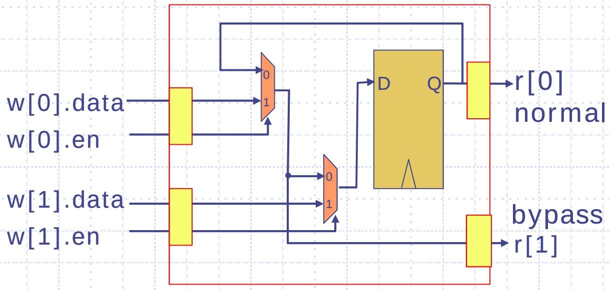
EHR: Ephemeral History Register

A new primitive element to design modules with concurrent methods

---

Ephemeral History Register (EHR) Dan Rosenband [MEMOCODE'04]



r[1] returns:

- the current state if w[0] *is not enabled*
- the value being written (w[0].data) if w[0] *is enabled*

w[i+1] takes precedence over w[i]

r[0] < w[0]

r[1] < w[1]

w[0] < w[1] < ….

“Happens before” (<) relation

- “happens before” relation between the methods of a module governs how the methods behave when called by a rule, action, method or exp
  - f < g : f happens before g (g cannot affect f within an action)
  - f > g : g happens before f
  - C : f and g conflict and cannot be called together
  - CF : f and g are conflict free and do not affect each other
- This relation is defined as a conflict matrix (CM) for the methods of primitive modules like registers and EHRs and derived for the methods of all other modules

---

Conflict Matrix of Primitive modules: Registers and EHRs

Register

	reg.r	reg.w
reg.r	CF	<
reg.w	>	C

EHR

	EHR.r0	EHR.w0	EHR.r1	EHR.w1
EHR.r0	CF	<	CF	<
EHR.w0	>	C	<	<
EHR.r1	CF	>	CF	<
EHR.w1	>	>	>	C

Designing FIFOs using EHRs

- *Conflict-Free FIFO:* Both enq and deq are permitted concurrently as long as the FIFO is not-full and not-empty
  - The effect of enq is not visible to deq, and vise versa
- *Pipeline FIFO:* An enq into a full FIFO is permitted provided a deq from the FIFO is done simultaneously
- *Bypass FIFO:* A deq from an empty FIFO is permitted provided an enq into the FIFO is done simultaneously

---

One-Element *Pipelined FIFO*

```
module mkPipelineFifo(Fifo#(1, t));
  Reg#(t) d <- mkRegU;
  Ehr#(2, Bool) v <- mkEhr(False);

  method Bool notFull = !v[1];
  method Bool notEmpty = v[0];
  method Action enq(t x);
    d <= x;
    v[1] <= True;
  endmethod

  method Action deq;
    v[0] <= False;
  endmethod

  method t first;
    return d;
  endmethod
endmodule
```

Desired behavior
deq < enq
first < deq
first < enq

No double write error

In any given cycle:
- If the FIFO is not empty then simultaneous enq and deq are permitted;
- Otherwise, only enq is permitted

One-Element Bypass FIFO

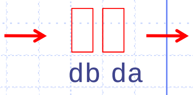
```
module mkBypassFifo(Fifo#(1, t));
  Ehr#(2, t) d <- mkEhr(?);
  Ehr#(2, Bool) v <- mkEhr(False);

  method Bool notFull = !v[0];
  method Bool notEmpty = v[1];
  method Action enq(t x);
    d[0] <= x;
    v[0] <= True;
  endmethod

  method Action deq;
    v[1] <= False;
  endmethod

  method t first;
    return d[1];
  endmethod
endmodule
```

Desired behavior
enq < deq
first < deq
enq < first

No double write error

In any given cycle:
- If the FIFO is not full then simultaneous enq and deq are permitted;
- Otherwise, only deq is permitted

Two-Element Conflict-free FIFO



```
module mkCFFifo(Fifo#(2, t));
  Ehr#(2, t) da <- mkEhr(?);
  Ehr#(2, Bool) va <- mkEhr(False);
  Ehr#(2, t) db <- mkEhr(?);
  Ehr#(2, Bool) vb <- mkEhr(False);


  method Bool notFull = !vb[0];
  method Bool notEmpty = va[0];
  method Action enq(t x);
    db[0] <= x; vb[0] <= True; endmethod
  method Action deq;
    va[0] <= False; endmethod
  method t first;
    return da[0]; endmethod
endmodule
```

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

Desired behavior
enq CF deq

In any given cycle:
- Simultaneous enq and deq are permitted only if the FIFO is not full and not empty

Deriving the Conflict Matrix (CM) of a module

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

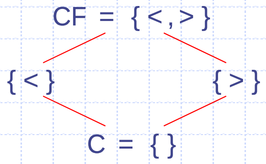
  mcalls(g1)={g11,g12...g1n}
  mcalls(g2)={g21,g22...g2m}

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

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

---

Conflict ordering

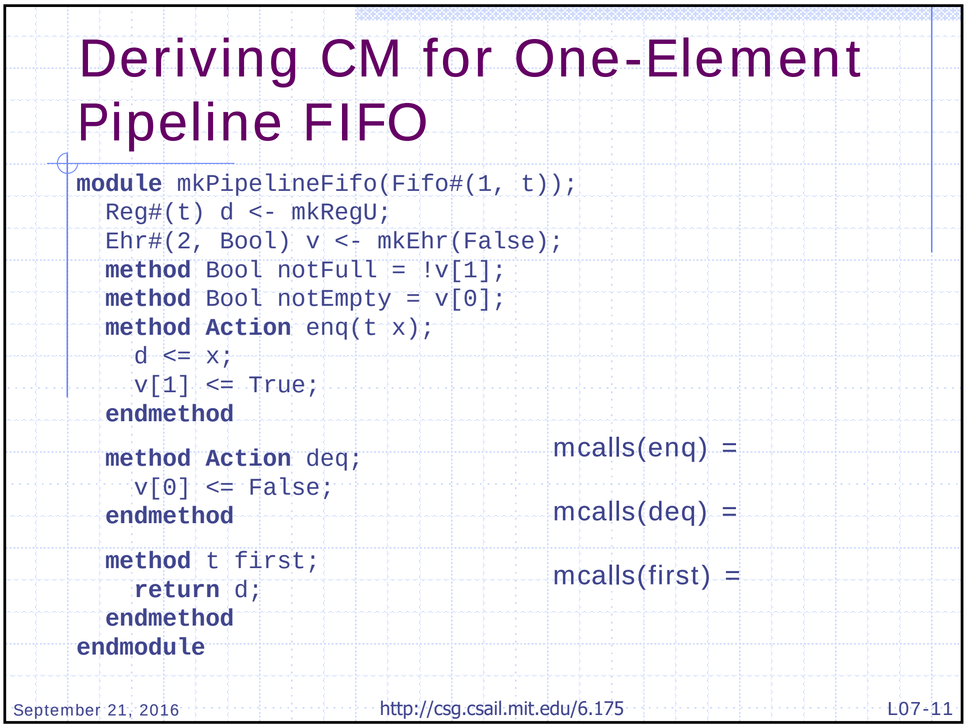


CF = {<,>}

{<} {>}

C = {}

- This permits us to take intersections of conflict information, e.g.,
  - {>}∩{<,>} = {>}
  - {>}∩{<} = {}



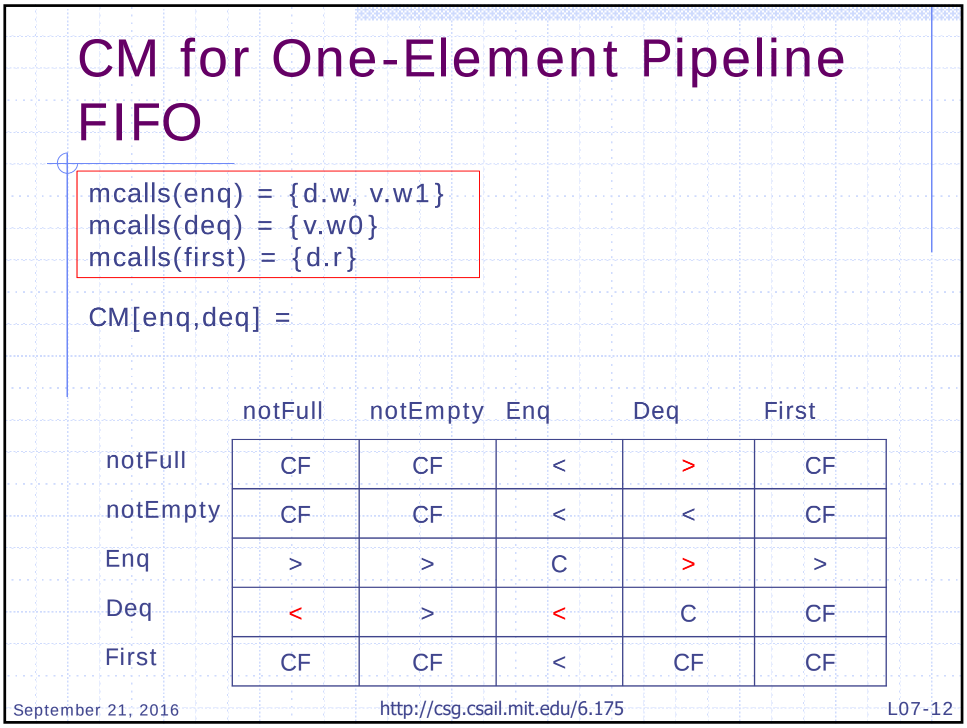
Deriving CM for One-Element Pipeline FIFO

```
module mkPipelineFifo(Fifo#(1, t));
  Reg#(t) d <- mkRegU;
  Ehr#(2, Bool) v <- mkEhr(False);
  method Bool notFull = !v[1];
  method Bool notEmpty = v[0];
  method Action enq(t x);
    d <= x;
    v[1] <= True;
  endmethod

  method Action deq;
    v[0] <= False;
  endmethod

  method t first;
    return d;
  endmethod
endmodule
```

mcalls(enq) =

mcalls(deq) =

mcalls(first) =



CM for One-Element Pipeline FIFO

mcalls(enq) = {d.w, v.w1}
mcalls(deq) = {v.w0}
mcalls(first) = {d.r}

CM[enq,deq] =

	notFull	notEmpty	Enq	Deq	First
notFull	CF	CF	<	>	CF
notEmpty	CF	CF	<	<	CF
Enq	>	>	C	>	>
Deq	<	>	<	C	CF
First	CF	CF	<	CF	CF

Deriving CM for One-Element Bypass FIFO

```
module mkBypassFifo(Fifo#(1, t));
  Ehr#(2, t) d <- mkEhr(?);
  Ehr#(2, Bool) v <- mkEhr(False);

  method Bool notFull = !v[0];
  method Bool notEmpty = v[1];
  method Action enq(t x);
    d[0] <= x;
    v[0] <= True;
  endmethod

  method Action deq;
    v[1] <= False;
  endmethod

  method t first;
    return d[1];
  endmethod
endmodule
```

mcalls(enq) =

mcalls(deq) =

mcalls(first) =

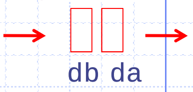
CM for One-Element Bypass FIFO

mcalls(enq) = {d.w0, v.w0}
mcalls(deq) = {v.w1}
mcalls(first) = {d.r1}

CM[enq,deq] =

	notFull	notEmpty	Enq	Deq	First
notFull	CF	CF	<	<	CF
notEmpty	CF	CF	>	<	CF
Enq	>	<	C	<	<
Deq	>	>	>	C	CF
First	CF	CF	>	CF	CF

CM for Two-Element Conflict-free FIFO



```
module mkCFFifo(Fifo#(2, t));
  Ehr#(2, t) da <- mkEhr(?);
  Ehr#(2, Bool) va <- mkEhr(False);
  Ehr#(2, t) db <- mkEhr(?);
  Ehr#(2, Bool) vb <- mkEhr(False);

  rule canonicalize;
    if(vb[1] && !va[1])
    (da[1] <= db[1]|
     va[1] <= True| vb[1] <= False) endrule

  method Bool notFull = !vb[0];
  method Bool notEmpty = va[0];
  method Action enq(t x);
    db[0] <= x; vb[0] <= True; endmethod
  method Action deq;
    va[0] <= False; endmethod
  method t first;
    return da[0]; endmethod
endmodule
```

Derive the CM

CM for Two-Element Conflict-free FIFO

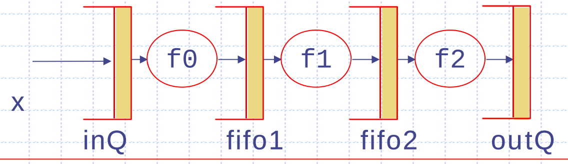
mcalls(enq) = { }
mcalls(deq) = { }
mcalls(first) = { }

Fill the CM

CM[enq,deq] =

	notFull	notEmpty	Enq	Deq	First	Canon
notFull	CF	CF			CF	
notEmpty	CF	CF			CF	
Enq			C			
Deq				C		
First	CF	CF			CF	
Canon						

Rewriting Elastic pipeline as a multirule system



```
rule stage1;
  if(inQ.notEmpty && fifo1.notFull)
    begin fifo1.enq(f0(inQ.first)); inQ.deq; end endrule
rule stage2;
  if(fifo1.notEmpty && fifo2.notFull)
    begin fifo2.enq(f1(fifo1.first)); fifo1.deq; end endrule
rule stage3;
  if(fifo2.notEmpty && outQ.notFull)
    begin outQ.enq(f2(fifo2.first)); fifo2.deq; end endrule
```

- How does such a system function?

Bluespec Execution Model

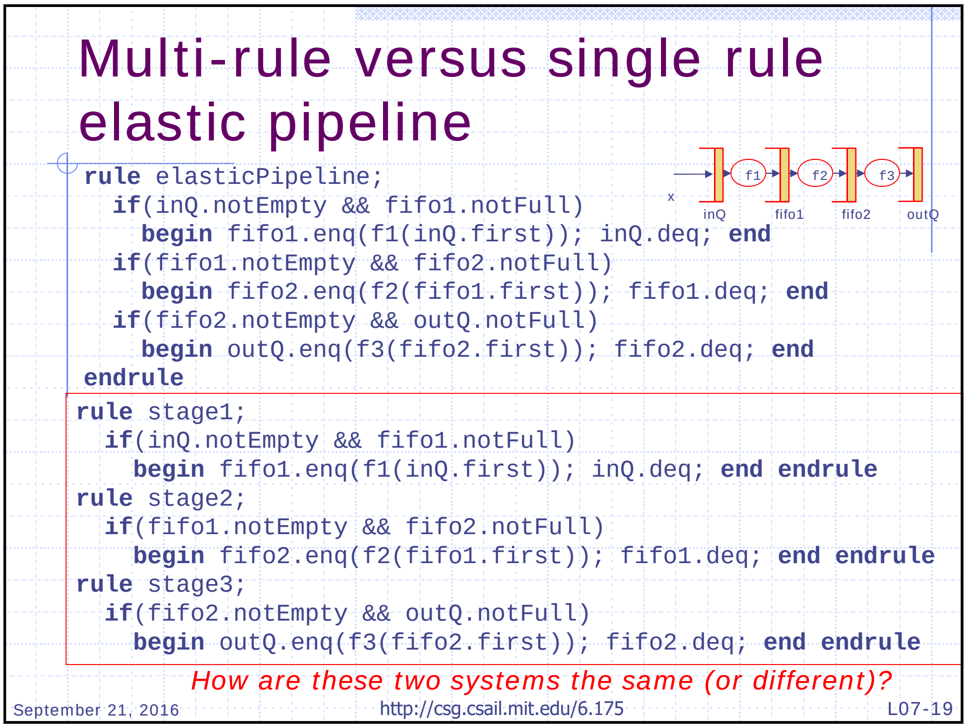
Repeatedly:

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

Highly non-deterministic; 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 need to execute multiple rules concurrently if possible

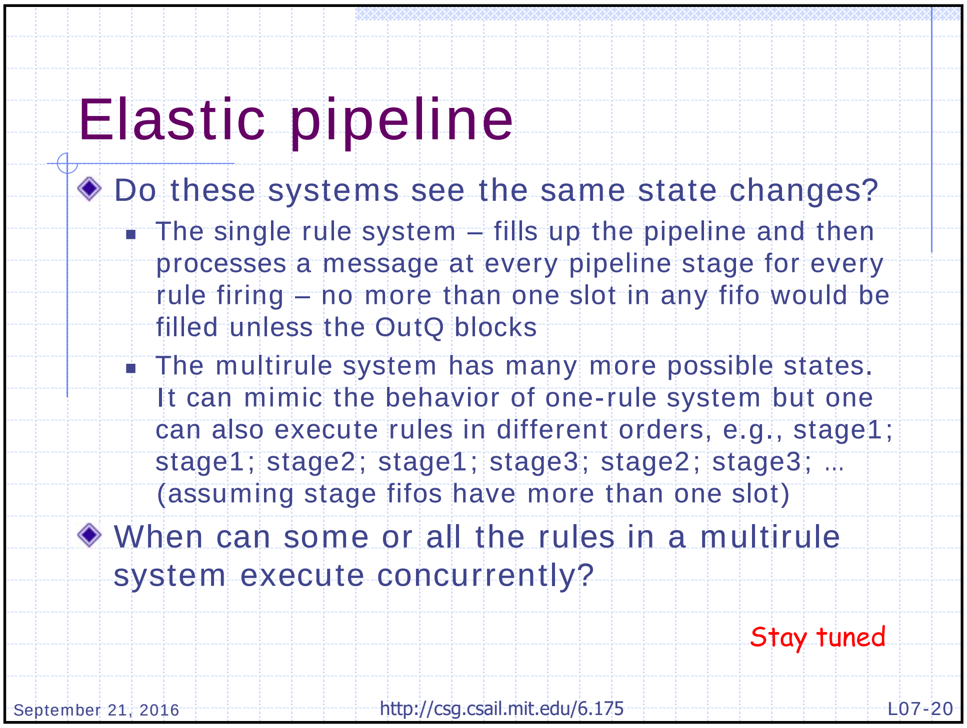


Multi-rule versus single rule elastic pipeline

```
rule elasticPipeline;
  if(inQ.notEmpty && fifo1.notFull)
    begin fifo1.enq(f1(inQ.first)); inQ.deq; end
  if(fifo1.notEmpty && fifo2.notFull)
    begin fifo2.enq(f2(fifo1.first)); fifo1.deq; end
  if(fifo2.notEmpty && outQ.notFull)
    begin outQ.enq(f3(fifo2.first)); fifo2.deq; end
endrule
```

```
rule stage1;
  if(inQ.notEmpty && fifo1.notFull)
    begin fifo1.enq(f1(inQ.first)); inQ.deq; end endrule
rule stage2;
  if(fifo1.notEmpty && fifo2.notFull)
    begin fifo2.enq(f2(fifo1.first)); fifo1.deq; end endrule
rule stage3;
  if(fifo2.notEmpty && outQ.notFull)
    begin outQ.enq(f3(fifo2.first)); fifo2.deq; end endrule
```

How are these two systems the same (or different)?



Elastic pipeline

- Do these systems see the same state changes?
  - The single rule system – fills up the pipeline and then processes a message at every pipeline stage for every rule firing – no more than one slot in any fifo would be filled unless the OutQ blocks
  - The multirule system has many more possible states. It can mimic the behavior of one-rule system but one can also execute rules in different orders, e.g., stage1; stage1; stage2; stage1; stage3; stage2; stage3; … (assuming stage fifos have more than one slot)
- When can some or all the rules in a multirule system execute concurrently?

Stay tuned