Constructive Computer Architecture:

Well formed BSV programs

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

A semantic questions

- Is a rule well-formed? That is, is there a possibility of *double write error* or does it contain a combinational cycle?

This question can be answered by analyzing the properties of the methods called by the rules

Illegal Actions – Double Write

- x <= e1; x <= e2;
- x <= e1; if(p) x <= e2;
- if(p) x <= e1; else x <= e2;

Not an error

Parallel composition of two actions is illegal if it creates the possibility of a *double-write error,* that is, if its constituent (sub)actions invoke the same action method

Illegal actions: Dataflow violations

Assume that x and y are EHRs

Read "<" as happens before

- x[0] <= x[1]
  - Syntax mandated: read x[1] < write x[0]
  - EHR def: write x[0] < read x[1]

contradiction!

- if (x[1]) x[0] <= e;
  - Syntax mandated: read x[1] < write x[0]
  - EHR def: write x[0] < read x[1]

contradiction!

- x[0] <= y[1]; y[0] <= x[1]
  - Syntax mandated:

    read y[1] < write x[0], read x[1] < write y[0]
  - EHR def:

    write x[0] < read x[1], write y[0] < read y[1]

Contradiction! No total ordering of methods

What is the general rule for detecting violations?

Total ordering of methods

- It should be possible to put a total order on methods without violating the syntax or CM imposed restrictions
- x[0] <= y[1]; y[0] <= x[1]

read y[1] < write x[0] < read x[1] < write y[0]

from EHR CM

contradiction!

- x <= y ; y <= x

{read x, read y} < {write x, write y}

No contradiction!

“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 an 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

Some definitions

- mcalls(x) is the set of method called by x
- mcalls(x) $<_s$ mcalls(y) means that every pair of methods (a,b) such that a $\in$ mcalls(x) and b $\in$ mcalls(y), either (a < b) or (a CF b)

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}

- Derivation
  - CM[g1,g2] = conflict(g11,g21) ∩ conflict(g11,g22) ∩...
    ∩ conflict(g12,g21) ∩ conflict(g12,g22) ∩...
    …
    ∩ conflict(g1n,g21) ∩ conflict(g12,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

Deriving CM for One-Element Pipeline FIFO

```
module mkPipelineFifo(Fifo#(1, t)) provisos(Bits#(t, tSz));
  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) =
{d.w, v.w1}

mcalls(deq) =
{v.w0}

mcalls(first) =
{d.r}

CM for One-Element Pipeline FIFO

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

CM[enq,deq] = conflict[d.w,v.w0] ∩ conflict[v.w1,v.w0]
= {>}   This is what we expected!

	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)) provisos(Bits#(t, tSz));
  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) =
{d.w0, v.w0}

mcalls(deq) =
{v.w1}

mcalls(first) =
{d.r1}

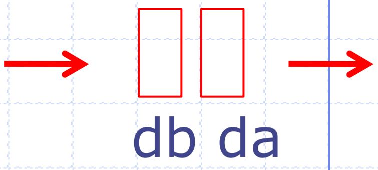
CM for One-Element Bypass FIFO

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

CM[enq,deq] = conflict[d.w0,v.w1] $\cap$ conflict[v.w0,v.w1]
= {<}   This is what we expected!

	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)) provisos(Bits#(t, tSz));
  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						

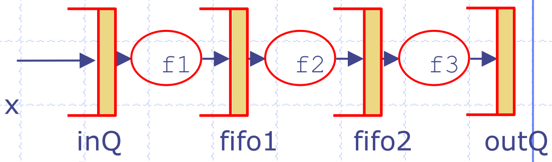
General rule for determining legal actions: syntax imposed restrictions

- a1 ; a2
  - either mcalls(a1) $<_s$ mcalls(a2) or mcalls(a2) $<_s$ mcalls(a1)
- if (e) a
  - mcalls(e) $<_s$ mcalls(a)
- m.g(e)
  - mcalls(e) $<_s$ {m.g}
- t = e ; a
  - mcalls(e) $<_s$ mcalls(a)

An action is *legal* if these syntax imposed constraints

1. are consistent with constraints defined by CM for each module
2. allow a total ordering of methods

Legal rule analysis



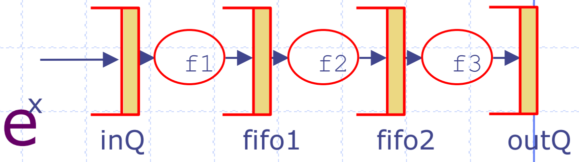
```
rule ArithPipe;
    if(inQ.notEmpty && fifo1.notFull)
       begin fifo1.enq(f1(inQ.first); inQ.deq end;
    if(fifo1.notEmpty && fifo2.notFull)
       begin fifo2.enq(f2(fifo1.first); fifo1.deq end;
    if(fifo2.notEmpty && outQ.notFull)
       begin outQ.enq(f3(fifo2.first); fifo2.deq end;
endrule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

- Syntactic constraints
  - `{inQ.notEmpty,fifo1.notFull}` $<_s$ `{fifo1.enq,inQ.first,inQ.deq}`
  - `{inQ.first}` $<_s$ `{fifo1.enq}`
  - `{fifo1.notEmpty,fifo2.notFull}` $<_s$ `{fifo2.enq,fifo1.first,fifo1.deq}`
  - `{fifo1.first}` $<_s$ `{fifo2.enq}`
  - `{fifo2.notEmpty,outQ.notFull}` $<_s$ `{outQ.enq,fifo2.first,fifo2.deq}`
  - `{fifo2.first}` $<_s$ `{outQ.enq}`

1. Are these constraints consistent with the CM for various FIFOs?
2. Can these method calls be put in a total order?

Legal rule analysis

syntactic constraints for each module^{x}



- Syntactic constraints of the rule
  - {inQ.notEmpty,fifo1.notFull} $<_s$ {fifo1.enq,inQ.first,inQ.deq}
  - {inQ.first} $<_s$ {fifo1.enq}
  - {fifo1.notEmpty,fifo2.notFull} $<_s$ {fifo2.enq,fifo1.first,fifo1.deq}
  - {fifo1.first} $<_s$ {fifo2.enq}
  - {fifo2.notEmpty,outQ.notFull} $<_s$ {outQ.enq,fifo2.first,fifo2.deq}
  - {fifo2.first} $<_s$ {outQ.enq}

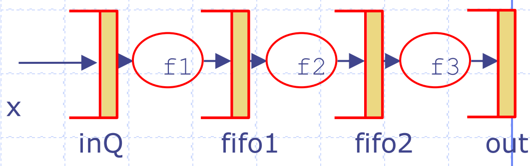
- Syntactic constraints for each FIFO
  - {inQ.notEmpty} $<_s$ {inQ.first,inQ.deq}
  - {fifo1.notFull} $<_s$ {fifo1.enq}
  - {fifo1.notEmpty} $<_s$ {fifo1.first,fifo1.deq}
  - {fifo2.notFull} $<_s$ {fifo2.enq}
  - {fifo2.notEmpty} $<_s$ {fifo2.first,fifo2.deq}
  - {outQ.notFull} $<_s$ {outQ.enq}

True for all types of FIFOs!

2. Can these method calls be put in a total order?

Legal rule analysis

assume all FIFOs are pipeline FIFOs

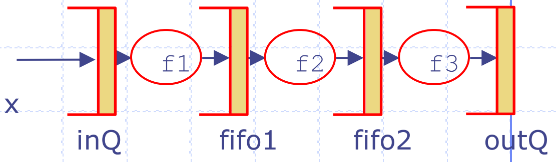


- Additional constraints because of pipeline FIFOs fifo1
  - `fifo1.notFull < fifo1.enq`
  - `fifo1.notEmpty CF fifo1.first`
  - `fifo1.notEmpty < fifo1.deq`
  - `fifo1.deq < fifo1.enq`
  - `fifo1.first < fifo1.enq`
  - `fifo1.notEmpty < fif01.enq`
  - `fifo1.notFull > fifo1.deq`
- Effect of additional constraints
  - {inQ.notEmpty,fifo1.notFull} $<_s$ fifo1.enq,inQ.first,inQ.deq}
  - {inQ.first} $<_s$ {fifo1.enq}
  - {fifo1.notEmpty,fifo2.notFull} $<_s$ {fifo2.enq,fifo1.first,fifo1.deq}
  - …

Can the method calls be put in a total order?

Legal rule analysis

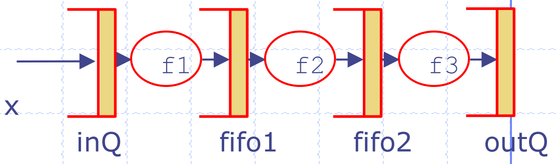
all FIFOs are pipeline FIFOs



- Syntactic constraints of the rule
  - {inQ.notEmpty,fifo1.notFull} $<_s$ fifo1.enq,inQ.first,inQ.deq}
  - {inQ.first} $<_s$ {fifo1.enq}
  - {fifo1.notEmpty,fifo2.notFull} $<_s$ {fifo2.enq,fifo1.first,fifo1.deq}
  - {fifo1.first} $<_s$ {fifo2.enq}
  - {fifo2.notEmpty,outQ.notFull} $<_s$ {outQ.enq,fifo2.first,fifo2.deq}
  - {fifo2.first} $<_s$ {outQ.enq}
- A total order

  {fifo2.notEmpty,outQ.notFull} $<_s$ {fifo2.first}
  $<_s$ {outQ.enq}
  $<_s$ {fifo2.deq}
  $<_s$ {fifo1.notEmpty,fifo2.notFull} $<_s$ {fifo1.first}
  $<_s$ {fifo2.enq}
  $<_s$ {fifo1.deq}
  $<_s$ {inQ.notEmpty,fifo1.notFull} $<_s$ {inQ.first}
  $<_s$ {fifo1.enq}
  $<_s$ {inQ.deq}

Legal rule analysis



- Syntactic constraints of the rule
  - {inQ.notEmpty,fifo1.notFull} $<_s$ fifo1.enq,inQ.first,inQ.deq}
  - {inQ.first} $<_s$ {fifo1.enq}
  - {fifo1.notEmpty,fifo2.notFull} $<_s$ {fifo2.enq,fifo1.first,fifo1.deq}
  - {fifo1.first} $<_s$ {fifo2.enq}
  - {fifo2.notEmpty,outQ.notFull} $<_s$ {outQ.enq,fifo2.first,fifo2.deq}
  - {fifo2.first} $<_s$ {outQ.enq}

- Can we find a total order on methods, assuming
  - All FIFOs are Pipeline FIFOs
  - All FIFOs are Bypass FIFOs
  - All FIFOs are CF
  - The design mixes different types of FIFOs

Real legal-rule analysis is more complicated: Predicated calls

- The analysis we presented would reject the following rule because of method conflicts

```
if (p) m.g(e1) ; if (!p) m.g(e2)
```

- We need to keep track of the predicates associated with each method call

m.g is called with predicates p and !p

which are disjoint – therefore no conflict