Constructive Computer Architecture:

Concurrency Analysis and Designing FIFOs

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

Contributors to the course material

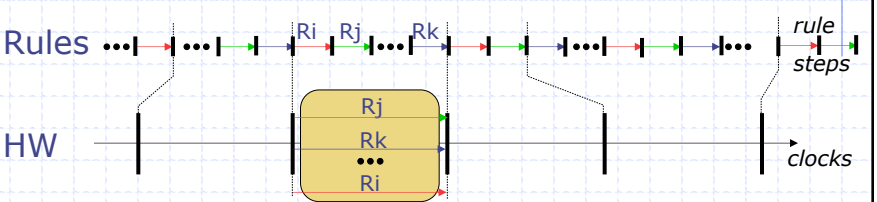
- Arvind, Rishiyur S. Nikhil, Joel Emer, Muralidaran Vijayaraghavan
- Staff and students in 6.375 (Spring 2013), 6.S195 (Fall 2012), 6.S078 (Spring 2012)
  - Asif Khan, Richard Ruhler, Sang Woo Jun, Abhinav Agarwal, Myron King, Kermin Fleming, Ming Liu, Li-Shiuan Peh
- External
  - Prof Amey Karkare & students at IIT Kanpur
  - Prof Jihong Kim & students at Seoul Nation University
  - Prof Derek Chiou, University of Texas at Austin
  - Prof Yoav Etsion & students at Technion

Contents

- Hardware Intuition
- Compiler analysis
- Limitations of Registers
- EHRs – Ephemeral History Registers
- FIFOs with concurrent enq and deq

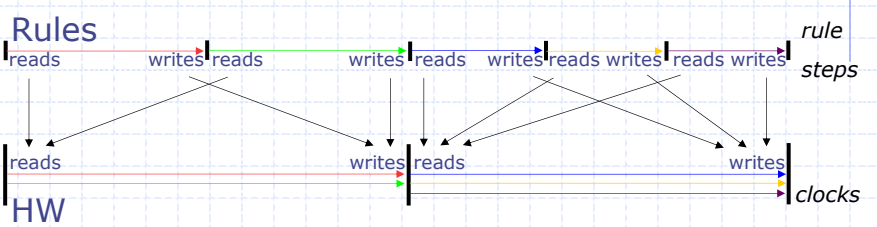
some insight into

Concurrent rule firing



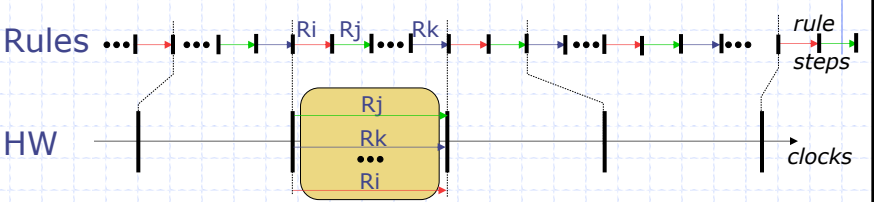
- There are more intermediate states in the rule semantics (a state after each rule step)
- In the HW, states change only at clock edges

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

One-rule-at-a-time semantics

- Rule execution ($\rightarrow$)

$$\frac{\text{Rule } r\ a \in P \qquad \langle S,\{\}\rangle \vdash a \Rightarrow U}{P \vdash S \rightarrow \text{update}(S,U)}$$

Where update(S,U)[x] = if (x,v) $\in$ U the v else S[x]

- Legal states: S is a legal state if and only if given an initial state S_0 , there exists a sequence of rules $r_{j1},\ldots,r_{jn}$ such that $S= r_{jn}(\ldots(r_{j1}(S_0))\ldots)$

$$\frac{P \vdash S_0 \rightarrow^* S}{S \in \text{LegalState}(P,S_0)}$$

where $\rightarrow^*$ is the transitive reflexive closure of $\rightarrow$

Concurrent scheduling of rules

- rule r_1 a_1 and rule r_2 a_2 can be scheduled concurrently, preserving one-rule-at-a-time semantics, if and only if
  - for all S. $(a_1|a_2)(S)$ = either $a_2(a_1(S))$ or $a_1(a_2(S))$
- rule r_1 a_1 to rule r_n a_n can be scheduled concurrently, preserving one-rule-at-a-time semantics, if and only if there exists a permutation $(p_1,\ldots,p_n)$ of $(1,\ldots,n)$ such that
  - for all S. $(a_1|\ldots|a_n)(S) = a_{pn}(\ldots(a_{p1}(S))$

Compiler test for concurrent scheduling

Let RS(r) be the set of registers rule r may read
Let WS(r) be the set of registers rule r may write

- Rules ra and rb are *conflict free* (CF) if
  $(RS(ra) \cap WS(rb) = \varphi) \wedge (RS(rb) \cap WS(ra) = \varphi) \wedge (WS(ra) \cap WS(rb) = \varphi)$
- Rules ra and rb are *sequentially composable* (SC) (ra<rb) if
  $(RS(rb) \cap WS(ra) = \varphi) \wedge (WS(ra) \cap WS(rb) = \varphi)$
- Rules ra and rb *conflict* if they are not CF or SC

Theorem: If ra < rb then for all S. (a|b) (S) = b(a(S))

Non-conflicting rules can be executed concurrently without violating the one-rule-at-a-time-semantics

Example 1: Compiler Analysis

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

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

RS(ra) = {z, x}
WS(ra) = {x}
RS(rb) = {z, y}
WS(rb) = {y}

$RS(ra) \cap WS(rb) = \varphi$
$RS(rb) \cap WS(ra) = \varphi$
$WS(ra) \cap WS(rb) = \varphi$
ra and rb are Conflict free

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

- $\{x0,y0,30\} \Rightarrow_{ra} \{x0+1,y0,30\} \Rightarrow_{rb} \{x0+1,y0+2,30\}$
  $\{x0,y0,30\} \Rightarrow_{rb} \{x0,y0+2,30\} \Rightarrow_{ra} \{x0+1,y0+2,30\}$
  $\{x0,y0,30\} \Rightarrow_{rb|ra} \{x0+1,y0+2,30\}$
- $\{x0,y0,15\} \Rightarrow_{ra} \{x0+1,y0,15\} \Rightarrow_{rb} \{x0+1,y0,15\}$
  $\{x0,y0,15\} \Rightarrow_{rb} \{x0,y0,15\} \Rightarrow_{ra} \{x0+1,y0,15\}$
  $\{x0,y0,15\} \Rightarrow_{rb|ra} \{x0+1,y0,15\}$

Example 2: Compiler Analysis

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

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

RS(ra) = {z, y}
WS(ra) = {x}
RS(rb) = {z, x}
WS(rb) = {y}

RS(ra)∩WS(rb) = y ra and rb are
RS(rb)∩WS(ra) = x neither CF or
WS(ra)∩WS(rb) = φ SC

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

- {x0,y0,30} $\Rightarrow_{ra}$ {y0+1,y0,30} $\Rightarrow_{rb}$ {y0+1,y0+1+2,30}
- {x0,y0,30} $\Rightarrow_{rb}$ {x0,x0+2,30} $\Rightarrow_{ra}$ {x0+2+1,x0+2,30}
- {x0,y0,30} $\Rightarrow_{rb|ra}$ {y0+1,x0+2,30}

Example 3: Compiler Analysis

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

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

RS(ra) = {z, y}
WS(ra) = {x}
RS(rb) = {z, y}
WS(rb) = {y}

RS(ra)∩WS(rb) = y
RS(rb)∩WS(ra) = φ ra and rb are
WS(ra)∩WS(rb) = φ SC (ra<rb)

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

- {x0,y0,30} $\Rightarrow_{ra}$ {y0+1,y0,30} $\Rightarrow_{rb}$ {y0+1,y0+2,30}
- {x0,y0,30} $\Rightarrow_{rb}$ {x0,y0+2,30} $\Rightarrow_{ra}$ {y0+2+1,y0+2,30}
- {x0,y0,30} $\Rightarrow_{ra|rb}$ {y0+1,y0+2,30}

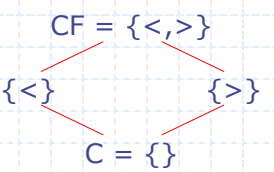
Analysis of method calls for concurrent scheduling

- Conflict analysis has to be performed in terms of the properties of the ports of a module rather than the module (e.g. register) it self
- Register conflicts:

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

- Let mcalls(x) represent the (multi-)set of methods called by x where x may be a method definition or a rule

Conflict ordering



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

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 {<,>}

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

Shorthand notation for Conflict relation

- h1<h2 ⇔ conflict(h1, h2) = {<}
- h1>h2 ⇔ conflict(h1, h2) = {>}
- h1 CF h2 ⇔ conflict(h1, h2) = {<,>}
- h1 C h2 ⇔ conflict(h1, h2) = {}

One-Element FIFO

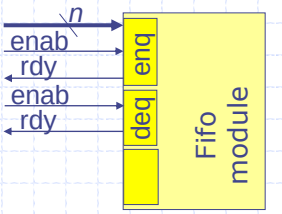
```
module mkCFFifo (Fifo#(1, t));
  Reg#(t)    data  <- mkRegU;
  Reg#(Bool) full  <- mkReg(False);
  method Action enq(t x) if (!full);
    full <= True;     data <= x;
  endmethod
  method Action deq if (full);
    full <= False;
  endmethod
  method t first if (full);
    return (data);
  endmethod
endmodule
```

Can **enq** and **deq** execute concurrently

enq and deq cannot even be enabled together much less fire concurrently!

not full

not empty



Two-Element FIFO


```
module mkCFFifo (Fifo#(2, t));
  Reg#(t)    da  <- mkRegU();
  Reg#(Bool) va  <- mkReg(False);
  Reg#(t)    db  <- mkRegU();
  Reg#(Bool) vb  <- mkReg(False);
  method Action enq(t x) if (!vb);
    if va then begin db <= x; vb <= True; end
          else begin da <= x; va <= True; end
  endmethod
  method Action deq if (va);
    if vb then begin da <= db; vb <= False; end
          else begin va <= False; end
  endmethod
  method t first if (va);
    return da;
  endmethod
endmodule
```

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

Can **enq** and **deq** be ready concurrently?

yes

Do **enq** and **deq** conflict?

yes, both read/write the same elements

Limitations of registers

- Limitations of a language with only the register primitive
  - No communication between rules or between methods or between rules and methods in the same atomic action i.e. clock cycle
  - Can't express a FIFO with concurrent enq and deq

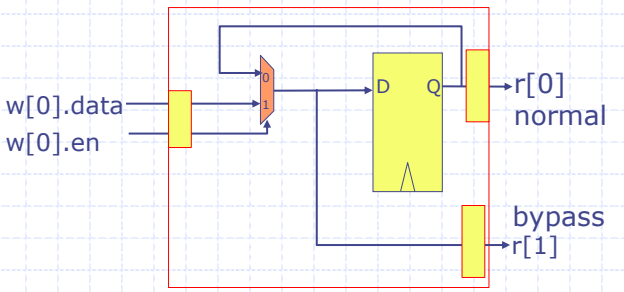
EHR: Ephemeral History Register

A new primitive element to design modules with concurrent methods

EHR: Register with a bypass Interface

r[0] < w[0]

w[0] < r[1]



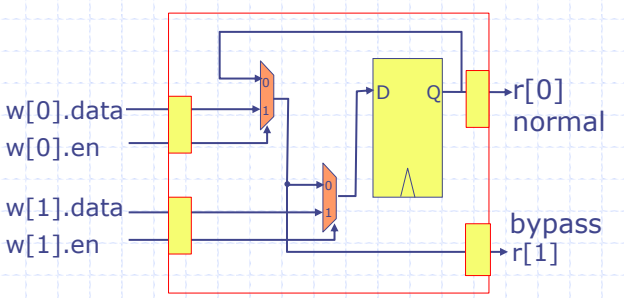
r[1] returns:
- the current state if write *is not enabled*
- the value being written if write *is enabled*

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

r[0] < w[0]

r[1] < w[1]

w[0] < w[1] < ….



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

Conflict Matrix of Primitive modules: Registers and EHRs

Register

	reg.r0	reg.w0
reg.r0	CF	<
reg.w0	>	C

EHR

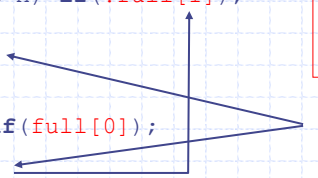
	EHR.r0	EHR.w0	EHR.r1	EHR.w1
EHR.r0	CF	<	<	<
EHR.w0	>	C	<	<
EHR.r1	>	>	CF	<
EHR.w1	>	>	>	C

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)) provisos(Bits#(t, tSz));
  Reg#(t) data <- mkRegU;
  Ehr#(2, Bool) full <- mkEhr(False);

  method Action enq(t x) if(!full[1]);
    data <= x;
    full[1] <= True;
  endmethod

  method Action deq if(full[0]);
    full[0] <= False;
  endmethod

  method t first if(full[0]);
    return data;
  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

Deriving CM for One-Element Pipelined FIFO

```
module mkPipelineFifo(Fifo#(1, t)) provisos(Bits#(t, tSz));
  Reg#(t) data <- mkRegU;
  Ehr#(2, Bool) full <- mkEhr(False);

  method Action enq(t x) if(!full[1]);
    data <= x;
    full[1] <= True;
  endmethod

  method Action deq if(full[0]);
    full[0] <= False;
  endmethod

  method t first if(full[0]);
    return data;
  endmethod
endmodule
```

mcalls(enq) =
{full.r1, data.w, full.w1}
mcalls(deq) =
{full.r0, full.w0}
mcalls(first) =
{full.r0, data.r}

CM for One-Element Pipelined FIFO

mcalls(enq) = {full.r1, data.w, full.w1}
mcalls(deq) = {full.r0, full.w0}
mcalls(first) = {full.r0, data.r}

CM[enq,deq] = conflict[full.r1,full.r0] ∩ conflict[full.r1,full.w0]
∩ conflict[data.w,full.r0]∩conflict[data.w,full.w0]
∩ conflict[full.w1,full.r0]∩conflict[full.w1,full.w0]

= {>} ∩ {>}
∩ {<,>} ∩ {<,>}
∩ {>} ∩ {>}
= {>}

This is what we expected!

One-Element Bypass FIFO

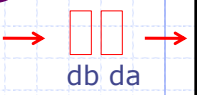
```
module mkBypassFifo(Fifo#(1, t)) provisos(Bits#(t, tSz));
  Ehr#(2, t) data <- mkEhr(?);
  Ehr#(2, Bool) full <- mkEhr(False);

  method Action enq(t x) if(!full[0]);
    data[0] <= x;
    full[0] <= True;
  endmethod

  method Action deq if(full[1]);
    full[1] <= False;
  endmethod

  method t first if(full[1]);
    return data;
  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)) 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 Action enq(t x) if(!vb[0]);
    db[0] <= x; vb[0] <= True; endmethod
  method Action deq if (va[0]);
    va[0] <= False; endmethod
  method t first if(va[0]);
    return da[0]; endmethod
endmodule
```

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

Desired behavior

enq CF deq

first < deq

first CF enq

In any given cycle:

- Simultaneous enq and deq are permitted only if the FIFO is not full and not empty