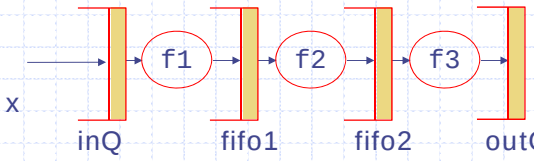
EHRs: Designing modules with concurrent methods

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

Elastic pipeline




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

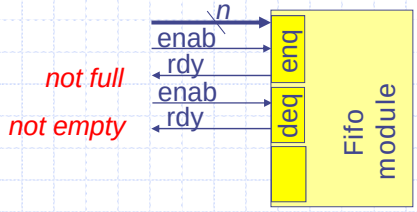
Whether these rules can fire concurrently depends crucially on the properties of fifo methods

Designing FIFOs

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



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?

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

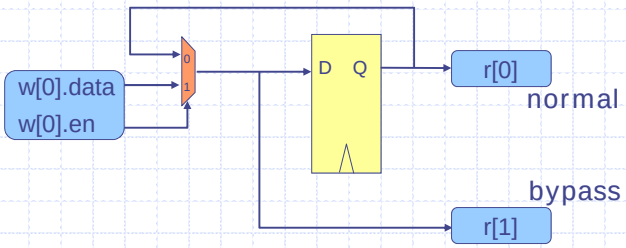
BSV model needs to be extended to express Fifos with concurrent methods

EHR: Ephemeral History Registers

EHR: Register with a bypass Interface

r[0] < w[0]

w[0] < r[1]



r[1] – if write *is not enabled* it returns the current state and if write *is enabled* it returns the value being written

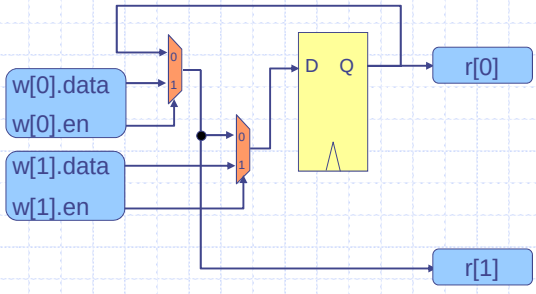
Ephemeral History Register (EHR)

Dan Rosenband [MEMOCODE'04]

r[0] < w[0]

r[1] < w[1]

w[0] < w[1] < ….



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

Designing FIFOs using EHRs

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
```

One can enq into a full Fifo provided someone is trying to deq from it simultaneously.

```
first < enq
deq   < enq
first < deq
```

One-Element Bypass FIFO

using EHRs

```
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[1];
  endmethod
endmodule
```

One can deq from an empty Fifo provided someone is trying to enq into it simultaneously.

```
enq   < first
enq   < deq
first < deq
```

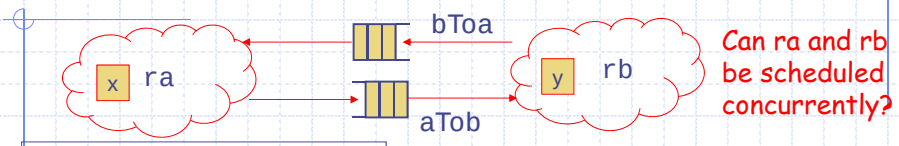

Two-Element Conflict-free FIFO


```
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

```
first CF enq
deq   CF enq
first < deq
```

Scheduling constraints due to multiple modules




```
rule ra;
  aTob.enq(fa(x));
  if (bToa.nonEmpty)
   begin
    x <= ga(bToa.first);
    bToa.deq; end
endrule
rule rb;
   y <= gb(aTob.first);
   aTob.deq;
   bToa.enq(fb(y));
endrule
```

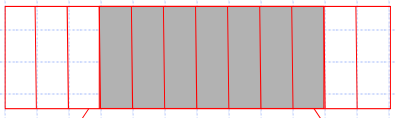
fifos are initially empty

aTob fifo	bToa fifo	Concurrent scheduling?
CF	CF	√
CF	pipeline	
CF	bypass	
pipeline	CF	
pipeline	pipeline	
pipeline	bypass	
bypass	CF	
bypass	pipeline	
bypass	bypass	

N-element Conflict-free FIFO

locks for preventing accesses until canonicalization

enqEn | deqEn



temp storage to hold values until canonicalization

oldEnqP

newData

- an enq updates enqP and puts the old value of enqP and enq data into oldEnqP and newData, respectively. It also sets enqEn to false to prevent further enqueues
- a deq updates deqP and sets deqEn to false to prevent further dequeues
- Canonicalize rule calculates the new count and puts the new data into the array and sets the enqEn and deqEn bits appropriately

Pointer comparison

- enqP and deqP can contain indices for upto twice the size of the FIFO, to distinguish between full and empty conditions
- Full: enqP == deqP + FIFO_size
- Empty: enqP == deqP

N-element Conflict-free FIFO

```
module mkCFFifo(Fifo#(n, t))
  provisos(Bits#(t, tSz), Add#(n, 1, n1), Log#(n1, sz),
Add#(sz, 1, sz1));
  Integer ni = valueOf(n); Bit#(sz1) nb = fromInteger(ni);
  Bit#(sz1) n2 = 2*nb;
  Vector#(n, Reg#(t)) data <- replicateM(mkRegU);
  Ehr#(2, Bit#(sz1)) enqP <- mkEhr(0);
  Ehr#(2, Bit#(sz1)) deqP <- mkEhr(0);
  Ehr#(2, Bool) enqEn <- mkEhr(True);
  Ehr#(2, Bool) deqEn <- mkEhr(False);
  Ehr#(2, t) newData <- mkEhr(?);
  Ehr#(2, Maybe#(Bit#(sz1))) oldEnqP <- mkEhr(Invalid);
```

N-element Conflict-free FIFO continued-1

```
rule canonicalize;
  Bit#(sz1) cnt = enqP[1] >= deqP[1]?
                    enqP[1] - deqP[1]:
                    (enqP[1]%nb + nb) - deqP[1]%nb;
  if(!enqEn[1] && cnt != nb) enqEn[1] <= True;
  if(!deqEn[1] && cnt != 0) deqEn[1] <= True;
  if(isValid(oldEnqP[1])) begin
    data[validValue(oldEnqP[1])] <= newData[1];
    oldEnqP[1] <= Invalid;
  end
endrule
```


N-element Conflict-free FIFO continued-2

```
method Action enq(t x) if(enqEn[0]);
  newData[0] <= x;
  oldEnqP[0] <= Valid (enqP[0]%nb);
  enqP[0] <= (enqP[0] + 1)%n2;
  enqEn[0] <= False;
endmethod

method Action deq if(deqEn[0]);
  deqP[0] <= (deqP[0] + 1)%n2; deqEn[0] <= False;
endmethod

method t first if(deqEn[0]);
  return data[deqP[0]%nb];
endmethod
endmodule
```

Register File:
normal and bypass

- Normal rf: {rd1, rd2} < wr; the effect of a register update can only be seen a cycle later, consequently, reads and writes are conflict-free
- Bypass rf: wr < {rd1, rd2}; in case of concurrent reads and write, check if rd1==wr or rd2==wr then pass the new value as the result and update the register file, otherwise the old value in the rf is read

Normal Register File

```
module mkRFile(RFile);
  Vector#(32,Reg#(Data)) rfile <- replicateM(mkReg(0));

  method Action wr(Rindx rindx, Data data);
    if(rindx!=0) rfile[rindx] <= data;
  endmethod
  method Data rd1(Rindx rindx) = rfile[rindx];
  method Data rd2(Rindx rindx) = rfile[rindx];
endmodule
```

{rd1, rd2} < wr

Bypass Register File using EHR

```
module mkBypassRFile(RFile);
  Vector#(32,EHR#(2, Data)) rfile <-
                              replicateM(mkEHR(0));

  method Action wr(Rindx rindx, Data data);
    if(rindex!==0) rfile[rindex][0] <= data;
  endmethod
  method Data rd1(Rindx rindx) = rfile[rindx][1];
  method Data rd2(Rindx rindx) = rfile[rindx][1];
endmodule
```

wr < {rd1, rd2}

Bypass Register File

with external bypassing

```
module mkBypassRFile(BypassRFile);
  RFile            rf <- mkRFile;
  Fifo#(1, Tuple2#(RIndx, Data))
                bypass <- mkBypassSFifo;
  rule move;
    begin rf.wr(bypass.first); bypass.deq end;
  endrule
  method Action wr(RIndx rindx, Data data);
    if(rindex!==0) bypass.enq(tuple2(rindx, data));
  endmethod
  method Data rd1(RIndx rindx) =
      return (!bypass.search1(rindx)) ? rf.rd1(rindx)
             : bypass.read1(rindx);
  method Data rd2(RIndx rindx) =
      return (!bypass.search2(rindx)) ? rf.rd2(rindx)
             : bypass.read2(rindx);
endmodule
```

{rf.rd1, rf.rd2} < rf.wr

wr < {rd1, rd2}

Extras

may be good for a lab exercise

N-element searchable FIFO

search CF {deq, enq}

Need a oldDeqP also to avoid scheduling constraints between deq and Search

```
module mkCFSFifo#(function Bool isFound(t v, st k))(SFifo#(n,
t, st))
  provisos(Bits#(t, tSz), Add#(n, 1, n1), Log#(n1, sz),
Add#(sz, 1, sz1));
  Integer ni = valueOf(n); Bit#(sz1) nb = fromInteger(ni);
  Bit#(sz1) n2 = 2*nb;
  Vector#(n, Reg#(t)) data <- replicateM(mkRegU);
  Ehr#(2, Bit#(sz1)) enqP <- mkEhr(0);
  Ehr#(2, Bit#(sz1)) deqP <- mkEhr(0);
  Ehr#(2, Bool) enqEn <- mkEhr(True);
  Ehr#(2, Bool) deqEn <- mkEhr(False);
  Ehr#(2, t) newData <- mkEhr(?);
  Ehr#(2, Maybe#(Bit#(sz1))) oldEnqP <- mkEhr(Invalid);
  Ehr#(2, Maybe#(Bit#(sz1))) oldDeqP <- mkEhr(Invalid);
```

N-element searchable FIFO
search CF {deq, enq} continued-1

```
Bit#(sz1) cnt0 = enqP[0] >= deqP[0]?  enqP[0] - deqP[0]:
                 (enqP[0]%nb + nb) - deqP[0]%nb;
Bit#(sz1) cnt1 = enqP[1] >= deqP[1]?  enqP[1] - deqP[1]:
                 (enqP[1]%nb + nb) - deqP[1]%nb;
rule canonicalize;
  if(!enqEn[1] && cnt2 != nb) enqEn[1] <= True;
  if(!deqEn[1] && cnt2 != 0) deqEn[1] <= True;
  if(isValid(oldEnqP[1])) begin
    data[validValue(oldEnqP[1])] <= newData[1];
    oldEnqP[1] <= Invalid;
  end
  if(isValid(oldDeqP[1])) begin
    deqP[0] <= validValue(oldDeqP[1]); oldDeqP[1]<=Invalid;
  end
endrule
```


N-element searchable FIFO
search CF {deq, enq} continued-2

```
  method Action enq(t x) if(enqEn[0]);
    newData[0] <= x; oldEnqP[0] <= Valid (enqP[0]%nb);
    enqP[0] <= (enqP[0] + 1)%n2; enqEn[0] <= False;
  endmethod

  method Action deq if(deqEn[0]);
    oldDeqP[0] <= Valid ((deqP[0] + 1)%n2);
    deqEn[0] <= False;
  endmethod

  method t first if(deqEn[0]);
    return data[deqP[0]%nb];
  endmethod
```

N-element searchable FIFO

search CF {deq, enq} continued-3

```
  method Bool search(st s);
    Bool ret = False;
    for(Bit#(sz1) i = 0; i < nb; i = i + 1)
    begin
      let ptr = (deqP[0] + i)%nb;
      if(isFound(data[ptr], s) && i < cnt0)
        ret = True;
    end
    return ret;
  endmethod
endmodule
```


N-element searchable FIFO

deq < search, deq < enq

This will make a good lab assignment

```
module mkPipelineSFifo#(function Bool isFound(t v, st
k))(SFifo#(n, t, st))
  provisos(Bits#(t, tSz), Add#(n, 1, n1), Log#(n1, sz),
Add#(sz, 1, sz1), Bits#(st, stz));
  Integer ni = valueOf(n);
  Bit#(sz1) nb = fromInteger(ni);
  Bit#(sz1) n2 = 2*nb;
  Vector#(n, Reg#(t)) data <- replicateM(mkRegU);
  Ehr#(2, Bit#(sz1)) enqP <- mkEhr(0);
  Ehr#(2, Bit#(sz1)) deqP <- mkEhr(0);
```

N-element searchable FIFO

deq < search, deq < enq

```
Bit#(sz1) cnt0 = enqP[0] >= deqP[0]? enqP[0] - deqP[0]:
                 (enqP[0]%nb + nb) - deqP[0]%nb;
Bit#(sz1) cnt1 = enqP[0] >= deqP[1]? enqP[0] - deqP[1]:
                 (enqP[0]%nb + nb) - deqP[1]%nb;

method Action enq(t x) if(cnt1 < nb);
  enqP[0] <= (enqP[0] + 1)%n2; data[enqP[0]%nb] <= x;
endmethod
method Action deq if(cnt0 != 0);
  deqP[0] <= (deqP[0] + 1)%n2;
endmethod
method t first if(cnt0 != 0);
  return data[deqP[0]%nb];
endmethod
```


N-element searchable FIFO

deq < search, deq < enq

```
  method Bool search(st s);
    Bool ret = False;
    for(Bit#(sz1) i = 0; i < nb; i = i + 1)
    begin
      let ptr = (deqP[1] + i)%nb;
      if(isFound(data[ptr], s) && i < cnt1)
        ret = True;
    end
    return ret;
  endmethod
endmodule
```