An EHR based methodology for Concurrency management

Arvind (with Asif Khan)
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Guarded Atomic Actions (GAA): Execution model

Repeatedly:

- Select a rule to execute ← Highly non-deterministic
- Compute the state updates
- Make the state updates

User annotations can help in rule selection

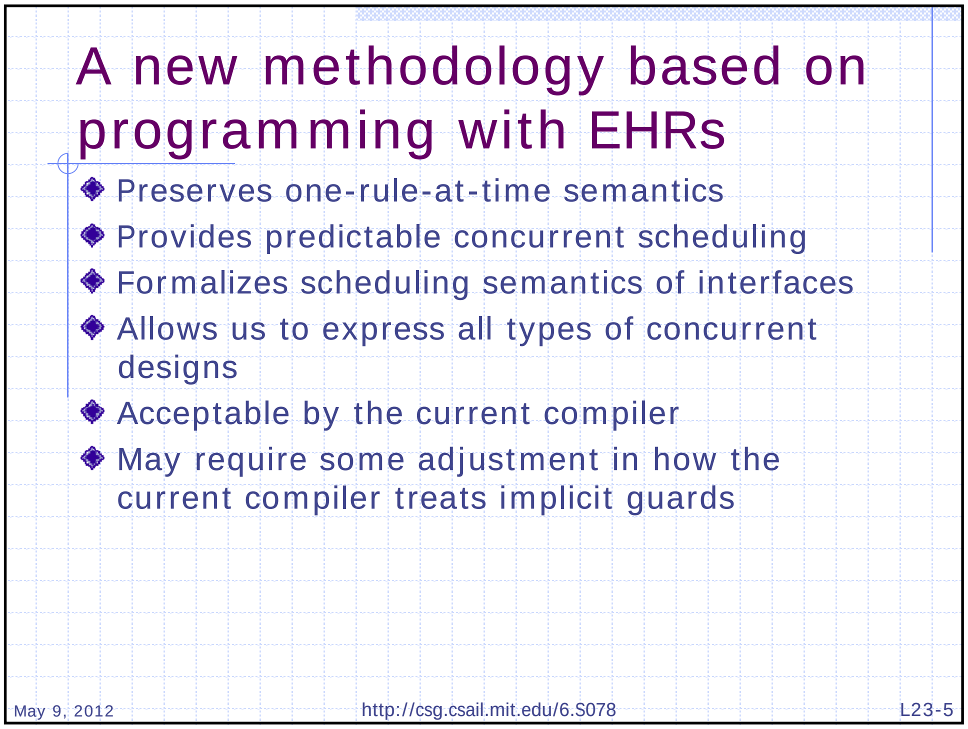
Implementation concern: Schedule multiple rules concurrently without violating one-rule-at-a-time semantics

Language issue

- BSV with registers only in not expressive enough to capture the desired degree of parallelism in designs
- BSV extended with *wires, reconfig regs,* etc. is used commonly and can express all types of parallelism but it
  - destroys one-rule-at-time semantics; it is essential to take scheduling as done by the current compiler into account to argue about functional correctness
  - is fragile – even textual reordering of rules can break programs
  - allows too many latent bugs which show up later when program fragments are used in slightly different contexts

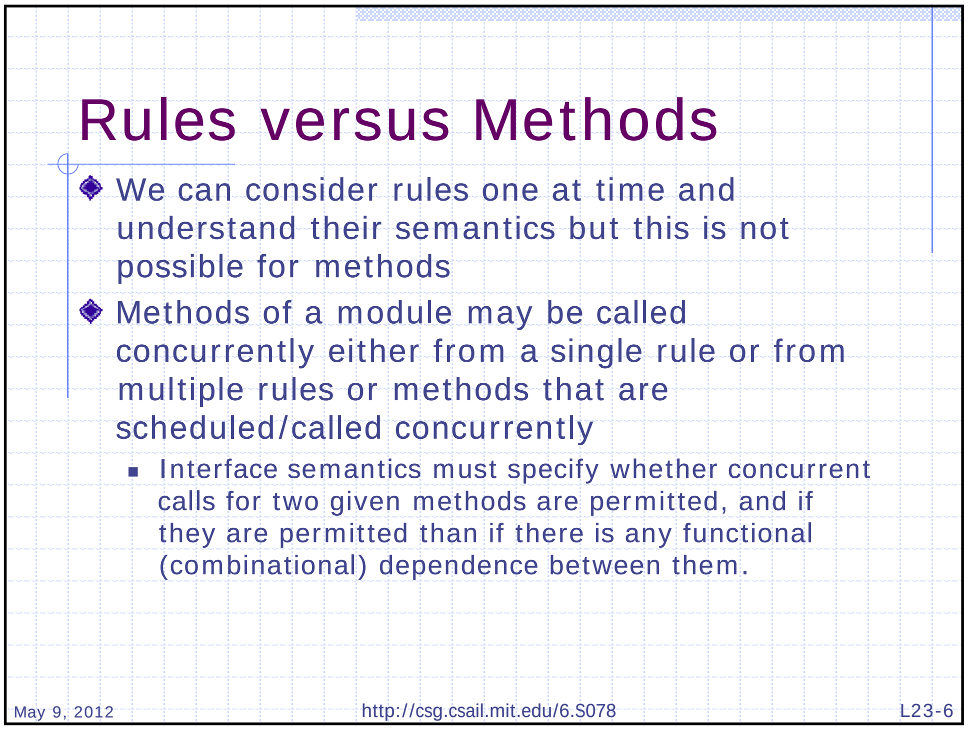
Other solutions

- *performance guarantees* [Rosenband 2005]: Compiled using EHRs. Clean semantics but not so good for modularity and synthesis boundaries; difficult to implement manually
- *sequential connective* [Dave 2011]: compiling is well understood. Clean semantics but not so good for modularity and synthesis boundaries. Not easy to retrofit in the current BSV compiler
- *single rule* [Khan, Muralidharan]: Express the whole design as a single rule. Tight control over scheduling, minimal use of wires but modularity is destroyed



A new methodology based on programming with EHRs

- Preserves one-rule-at-time semantics
- Provides predictable concurrent scheduling
- Formalizes scheduling semantics of interfaces
- Allows us to express all types of concurrent designs
- Acceptable by the current compiler
- May require some adjustment in how the current compiler treats implicit guards



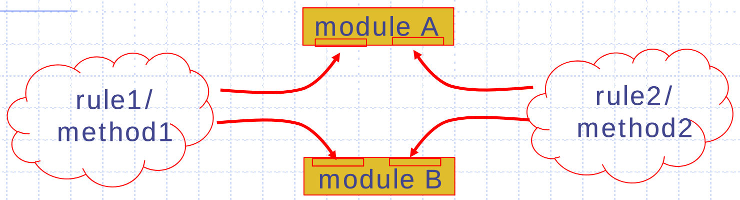
Rules versus Methods

- We can consider rules one at time and understand their semantics but this is not possible for methods
- Methods of a module may be called concurrently either from a single rule or from multiple rules or methods that are scheduled/called concurrently
  - Interface semantics must specify whether concurrent calls for two given methods are permitted, and if they are permitted than if there is any functional (combinational) dependence between them.

Interface properties

- Conflicting (m1,m2): m1 and m2 cannot be called together, i.e.,
  - if they are called from the same method or rule than that method or rule is invalid
  - if they are called from two different rules than those rules cannot be scheduled concurrently
- CF (m1,m2): m1 and m2 are conflict free and can be called together, however, no effect of m1 can be seen by m2 in the current cycle or vise versa
- m1 < m2: m1 and m2 can be called together, but m1 may affect m2 in the current cycle (similarly for m2 < m1)

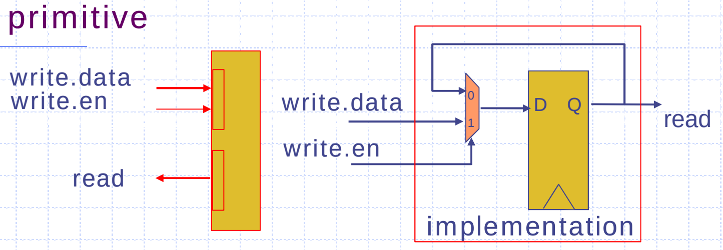
Scheduling constraints on rules



- Scheduling constraints on the methods of modules A and B induce scheduling constraints on rules and methods calling them
- Such scheduling constraints can be determined and enforced by the compiler bottoms' up

Register interface

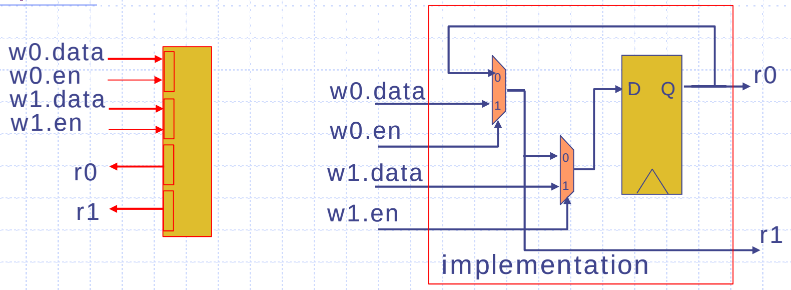
BSV primitive



- methods can be called concurrently but read does not see the effect of write
  - read < write
- these methods have no guard, i.e., they are always “ready”
- the write method is an action method and therefore has an “enable” signal

Ephemeral History Register (EHR)

BSV primitive Dan Rosenband [MEMOCODE’04]



- methods can be called concurrently
  - r0 < w0 < r1 < w1
  - r1 can see w0 and w1 takes precedence over w0
- methods have no guards
- Primitive register is a special case of EHR

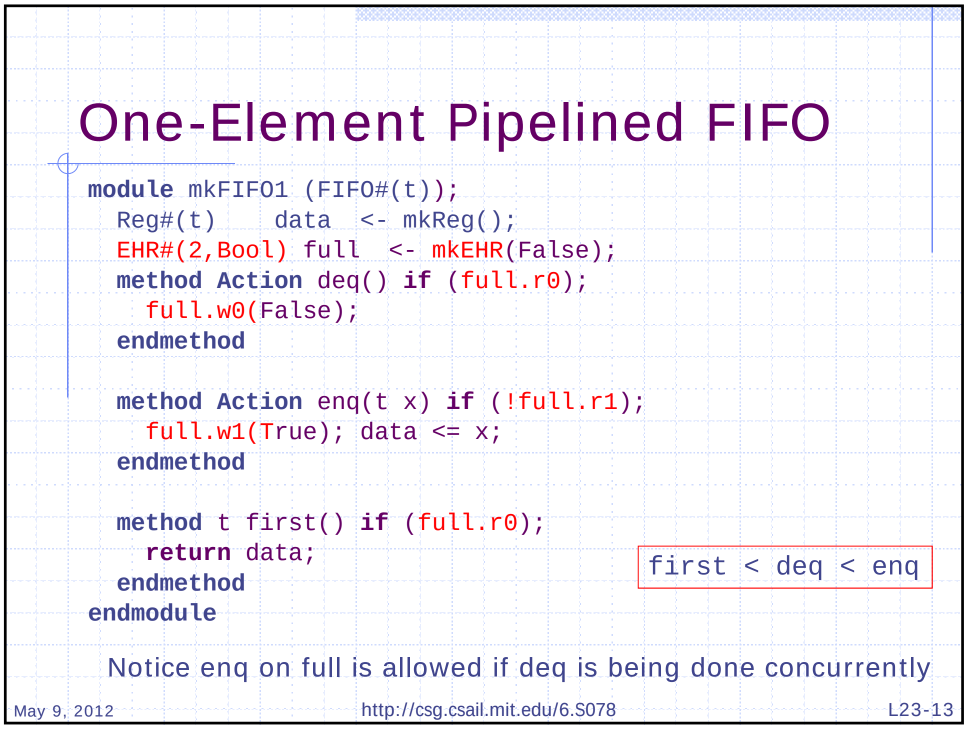
Using EHRs to express the desired concurrency

One-Element FIFO

No concurrent enq / deq

```
module mkFIFO1 (FIFO#(t));
  Reg#(t)    data  <- mkReg();
  Reg#(Bool) full  <- mkReg(False);

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

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

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

deq and enq cannot be enabled together

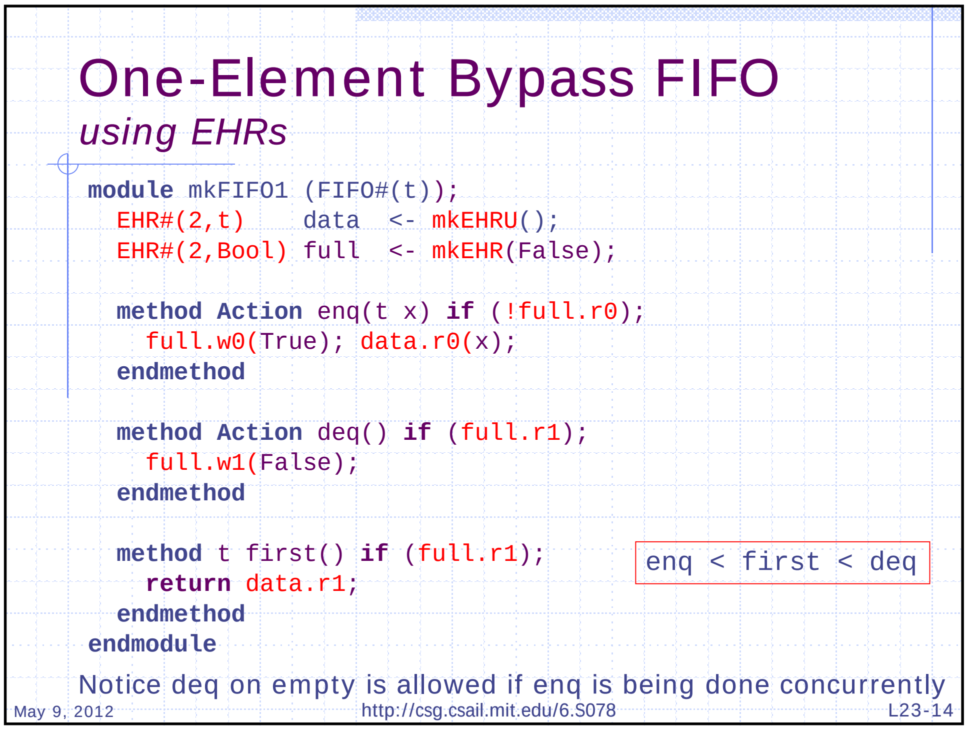


One-Element Pipelined FIFO

```
module mkFIFO1 (FIFO#(t));
  Reg#(t)     data  <- mkReg();
  EHR#(2,Bool) full  <- mkEHR(False);
  method Action deq() if (full.r0);
    full.w0(False);
  endmethod

  method Action enq(t x) if (!full.r1);
    full.w1(True); data <= x;
  endmethod

  method t first() if (full.r0);
    return data;
  endmethod
endmodule
```

first < deq < enq

Notice enq on full is allowed if deq is being done concurrently



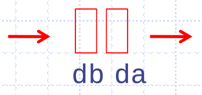
One-Element Bypass FIFO
using EHRs

```
module mkFIFO1 (FIFO#(t));
  EHR#(2,t)    data  <- mkEHRU();
  EHR#(2,Bool) full  <- mkEHR(False);

  method Action enq(t x) if (!full.r0);
    full.w0(True); data.r0(x);
  endmethod

  method Action deq() if (full.r1);
    full.w1(False);
  endmethod

  method t first() if (full.r1);
    return data.r1;
  endmethod
endmodule
```

enq < first < deq

Notice deq on empty is allowed if enq is being done concurrently

Two-Element FIFO



```
module mkFIFO (FIFO#(t));
  EHR#(t)    da   <- mkEHRU();
  EHR#(Bool) va   <- mkEHR(False);
  EHR#(t)    db   <- mkEHRU();
  EHR#(Bool) vb   <- mkEHR(False);

  rule canonicalize (vb.r1 & !va.r1);
          da.w1(db.r1); va.w1(True); vb.w1(False);
  endrule

  method Action enq(t x) if (!vb.r0);
          db.w0(x); vb.w0(True); endmethod
  method Action deq() if (va.r0);
          va.w0(False); endmethod
  method t first() if (va.r0);
    return da.r0; endmethod
endmodule
```

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

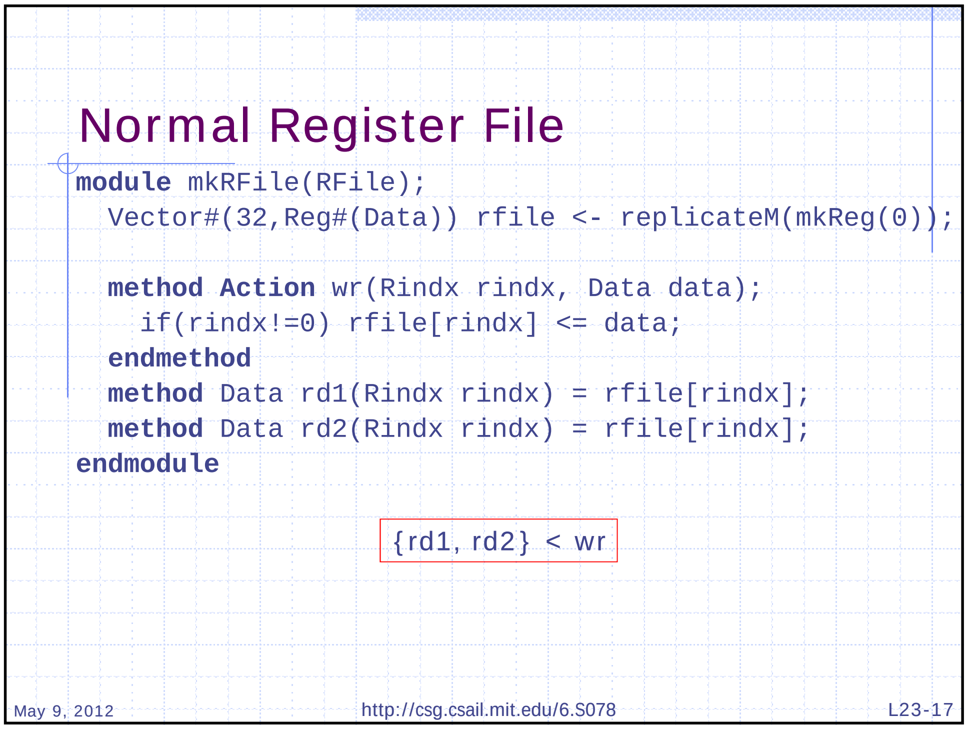
enq CF (first < deq)

All methods and rule canonicalize can be done concurrently

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

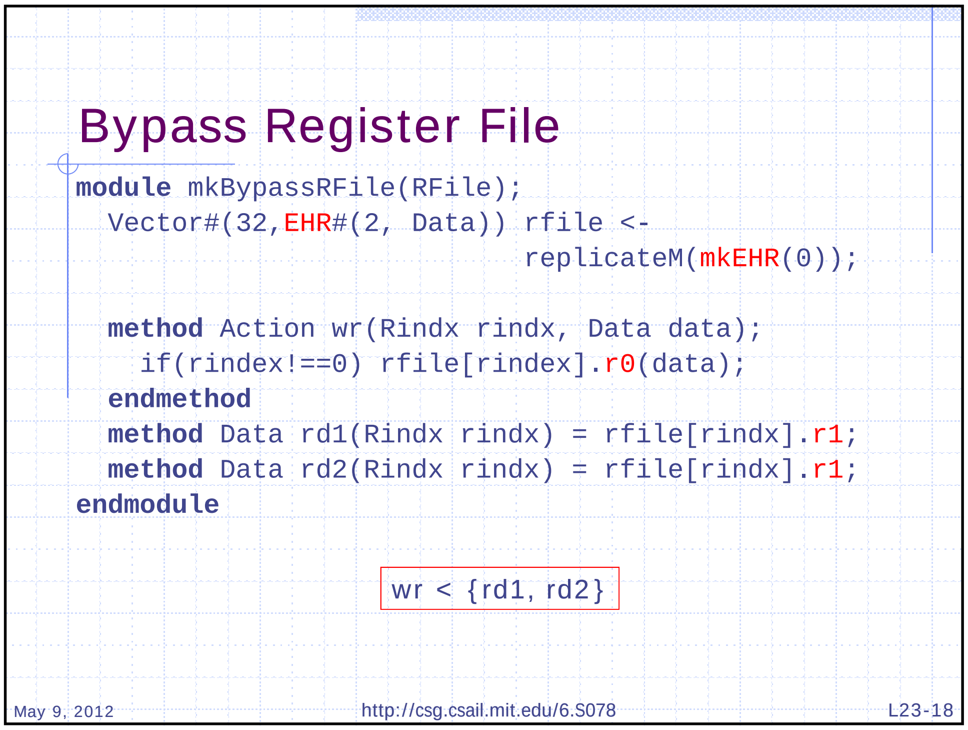


Normal Register File

```
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

May 9, 2012 http://csg.csail.mit.edu/6.S078 L23-17



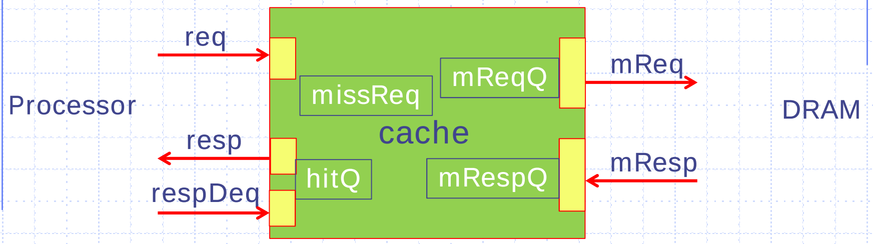
Bypass Register File

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

  method Action wr(Rindx rindx, Data data);
    if(rindex!==0) rfile[rindex].r0(data);
  endmethod
  method Data rd1(Rindx rindx) = rfile[rindx].r1;
  method Data rd2(Rindx rindx) = rfile[rindx].r1;
endmodule
```

wr < {rd1, rd2}

May 9, 2012 http://csg.csail.mit.edu/6.S078 L23-18

Blocking Cache Interface



```
interface Cache;
     method Action req(MemReq r);
     method MemResp resp;
     method Action respDeq;

     method ActionValue#(MemReq) mReq;
     method Action mResp(MemResp r);
endinterface
```

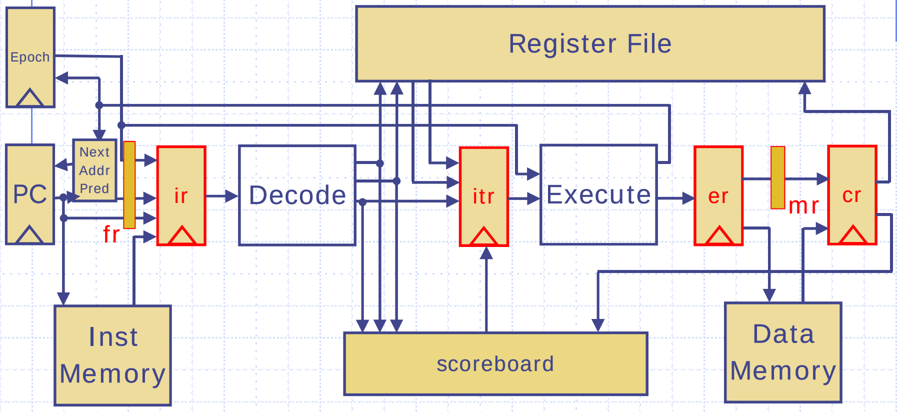
Blocking I-Cache

processor-side methods

```
method Action req(MemReq r) if (status==Rdy);
  Index idx = truncate(r.addr>>2);
  Tag tag = truncateLSB(r.addr);
  Bool valid = vArray[idx];
  Bool tagMatch = tagArray[idx]==tag;
  if(valid && tagMatch)
    hitQ.enq(r);
  else begin
    missReq <= r; status <= FillReq;  end
endmethod
method MemResp resp;
  let r = hitQ.first;
  Index idx = truncate(r.addr>>2);
  return dataArray[idx];
endmethod
method respDeq;
  hitQ.deq;
endmethod
```

hitQ is a bypass FIFO

Multi-Stage SMIPS

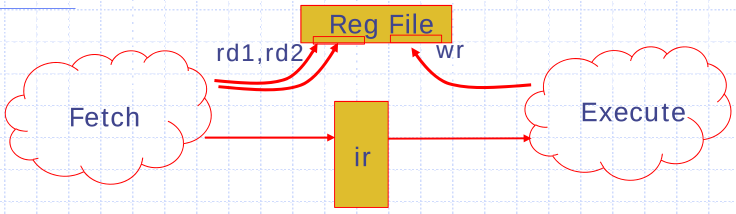


▮ insert bypass FIFO's to deal with (0,n) cycle memory response

Fetch rules

```
rule doFetch1 (fr.notFull);
  iCache.req(TypeMemReq{op:Ld, addr:pc.r1, data:?});
  let ppc = bpred.prediction(pc.r1);
  fr.enq(TypeFecth2Fetch{pc:pc.r1, ppc:ppc,
                              epoch:epoch.r1});
  pc.w1(ppc);
endrule

rule doFetch2 (fr.notEmpty && ir.notFull);
  let frpc    = fr.first.pc;
  let frppc   = fr.first.ppc;
  let frepoch = fr.first.epoch;
  let inst = iCache.resp; iCache.respDeq;
  ir.enq(TypeFetch2Decode{pc:frpc, ppc:frppc,
                               epoch:frepoch, inst:inst});
  fr.deq;
endrule
```

Different architectures require different scheduling



- if ir is a pipelined FIFO (deq<enq) then reg file has to be a bypass register file (wr < {rd1,rd2})
- if ir is normal FIFO (deq CF enq) then reg file can be either ordinary or bypass register file

We can build the interfaces we want using EHRs and achieve the desired concurrency in a systematic way

next lecture - non-blocking caches