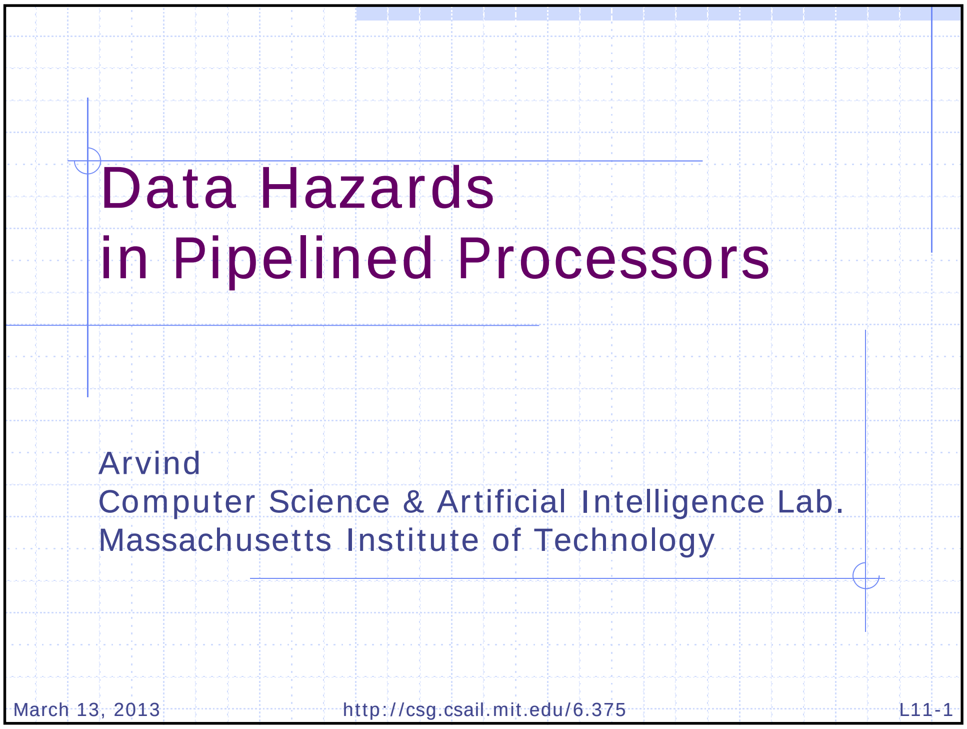
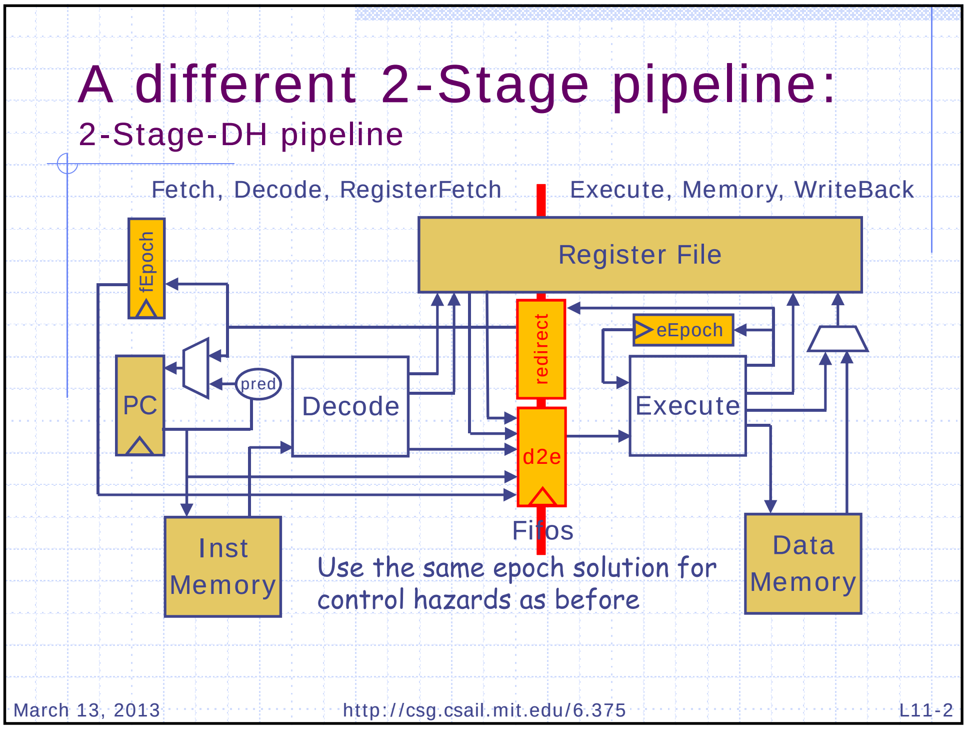


Data Hazards in Pipelined Processors

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

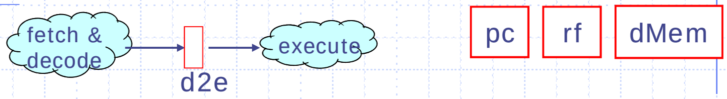


A different 2-Stage pipeline:

2-Stage-DH pipeline

Use the same epoch solution for control hazards as before

Data Hazards



time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
FDstage		FD_1	FD_2	FD_3	FD_4	FD_5			
EXstage			EX_1	EX_2	EX_3	EX_4	EX_5		

I_1 Add(R1,R2,R3)

I_2 Add(R4,R1,R2)

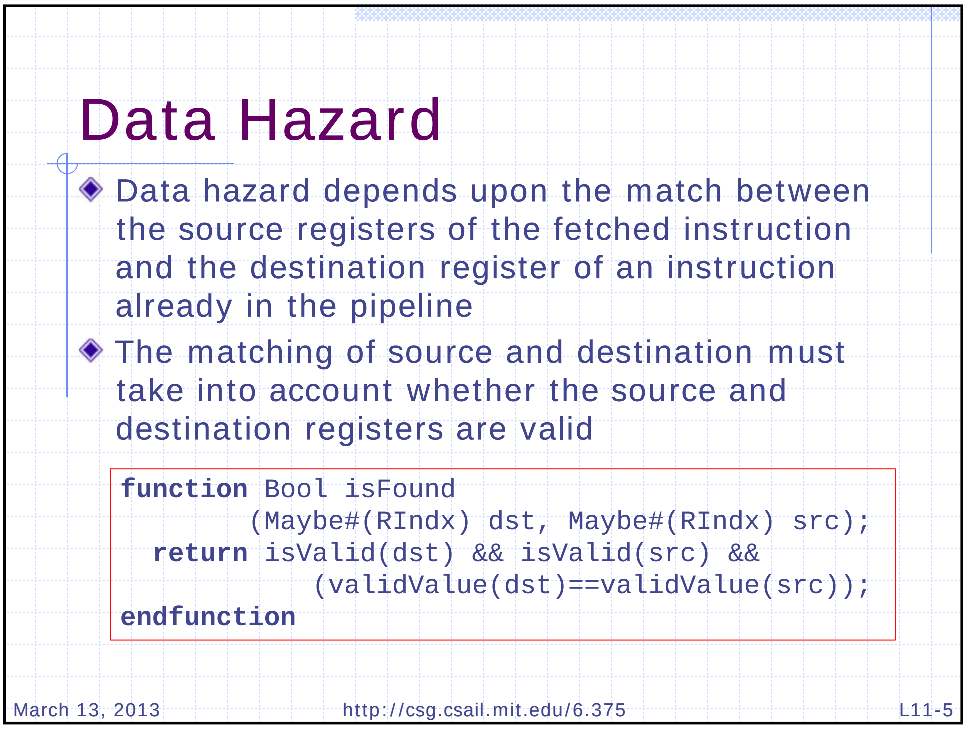
I_2 must be stalled until I_1 updates the register file

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
FDstage		FD_1	FD_2	FD_2	FD_3	FD_4	FD_5		
EXstage			EX_1		EX_2	EX_3	EX_4	EX_5	

Dealing with data hazards

- Keep track of instructions in the pipeline and determine if the register values to be fetched are stale, i.e., will be modified by some older instruction still in the pipeline. This condition is referred to as a read-after-write (RAW) hazard
- Stall the Fetch from dispatching the instruction as long as RAW hazard prevails
- RAW hazard will disappear as the pipeline drains

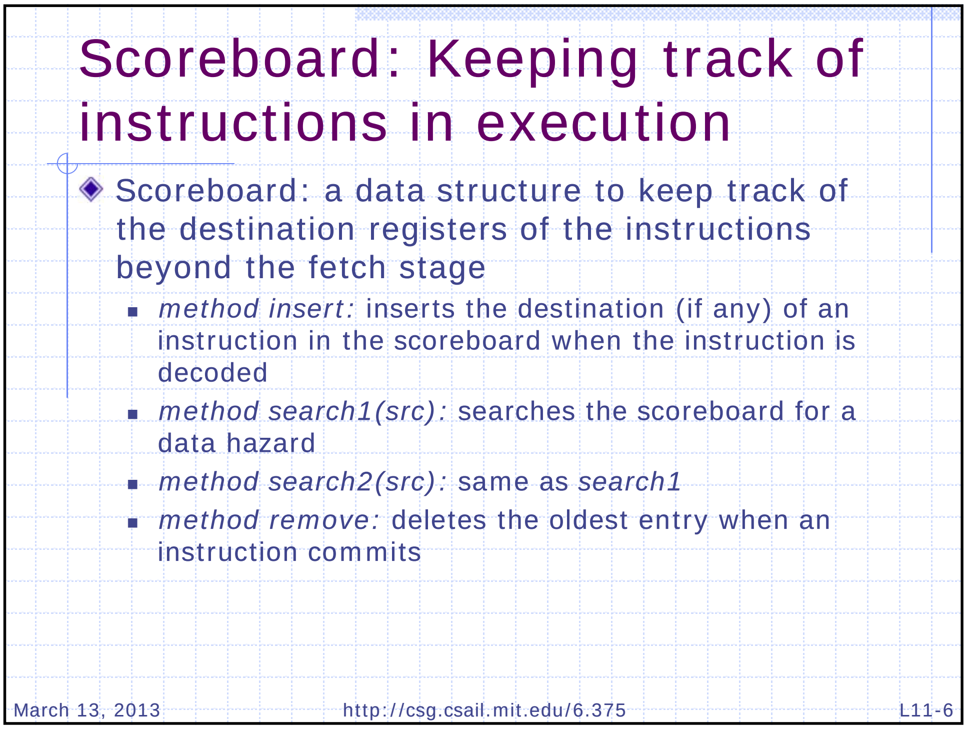
Scoreboard: A data structure to keep track of the instructions in the pipeline beyond the Fetch stage



Data Hazard

- Data hazard depends upon the match between the source registers of the fetched instruction and the destination register of an instruction already in the pipeline
- The matching of source and destination must take into account whether the source and destination registers are valid

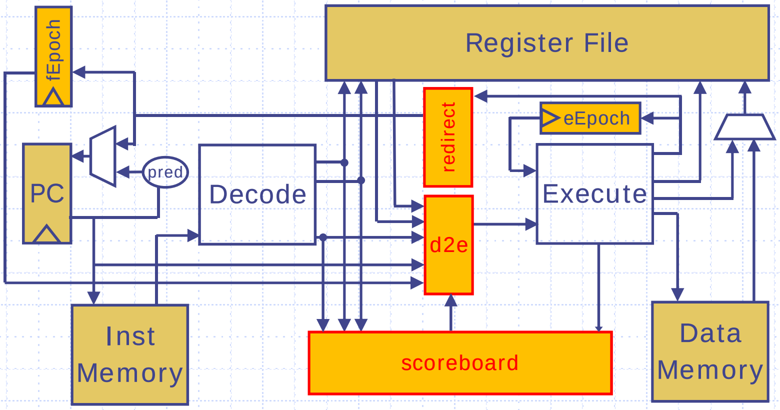
```
function Bool isFound
        (Maybe#(RIndx) dst, Maybe#(RIndx) src);
  return isValid(dst) && isValid(src) &&
            (validValue(dst)==validValue(src));
endfunction
```



Scoreboard: Keeping track of instructions in execution

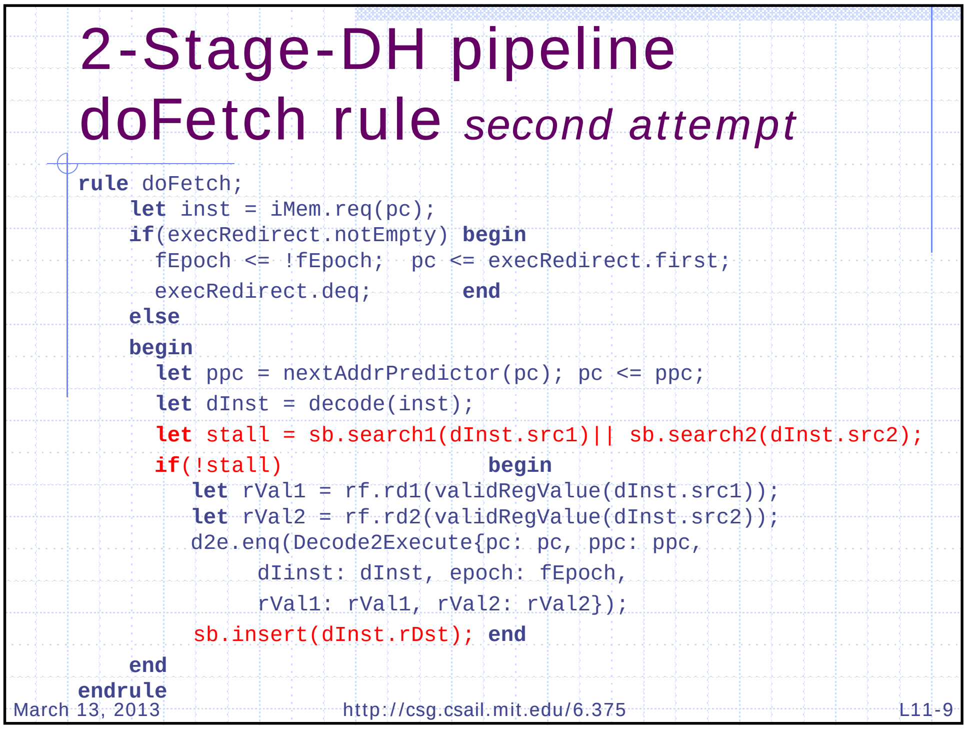
- Scoreboard: a data structure to keep track of the destination registers of the instructions beyond the fetch stage
  - *method insert:* inserts the destination (if any) of an instruction in the scoreboard when the instruction is decoded
  - *method search1(src):* searches the scoreboard for a data hazard
  - *method search2(src):* same as *search1*
  - *method remove:* deletes the oldest entry when an instruction commits

2-Stage-DH pipeline: Scoreboard and Stall logic



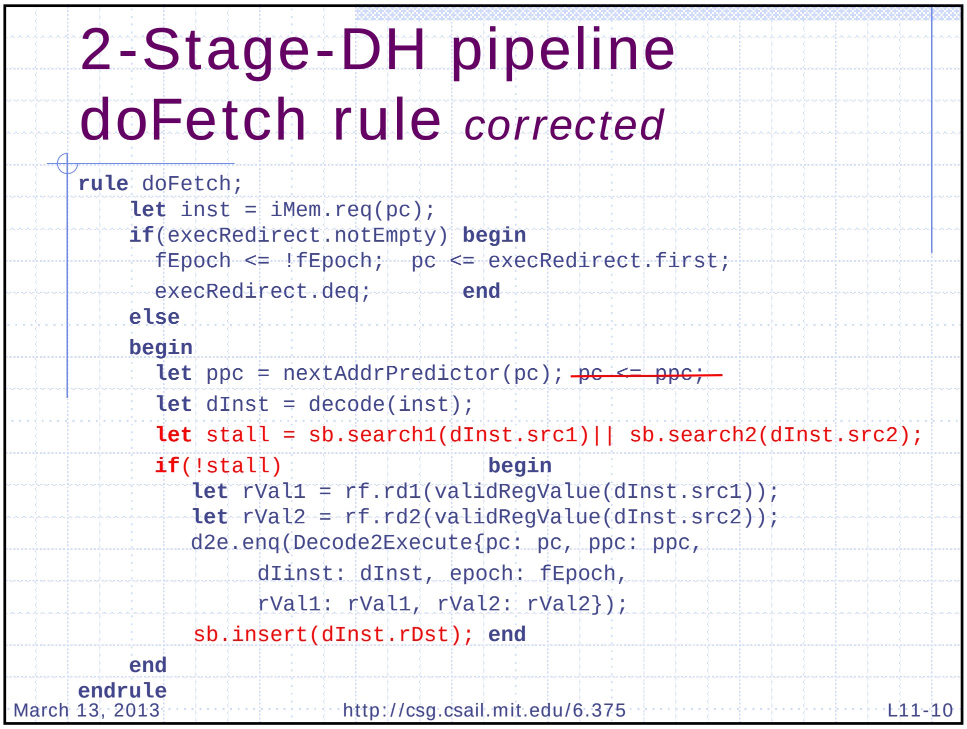
2-Stage-DH pipeline *corrected*

```
module mkProc(Proc);
  Reg#(Addr)          pc <- mkRegU;
  RFile               rf <- mkRFile;
  IMemory           iMem <- mkIMemory;
  DMemory           dMem <- mkDMemory;
  Fifo#(Decode2Execute) d2e <- mkFifo;
  Reg#(Bool)      fEpoch <- mkReg(False);
  Reg#(Bool)      eEpoch <- mkReg(False);
  Fifo#(Addr) execRedirect <- mkFifo;

  Scoreboard#(1) sb <- mkScoreboard;
      // contains only one slot because Execute
      // can contain at most one instruction

  rule doFetch …
  rule doExecute …
```



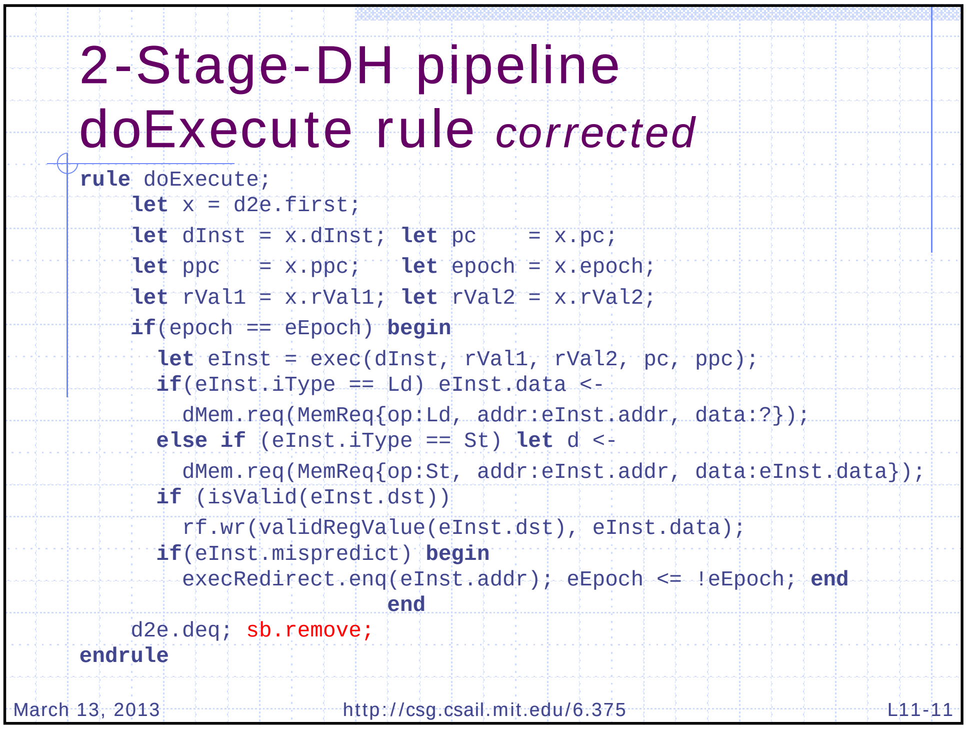
2-Stage-DH pipeline doFetch rule *second attempt*

```
rule doFetch;
    let inst = iMem.req(pc);
    if(execRedirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= execRedirect.first;
      execRedirect.deq;       end
    else
    begin
      let ppc = nextAddrPredictor(pc); pc <= ppc;
      let dInst = decode(inst);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
      if(!stall)                  begin
         let rVal1 = rf.rd1(validRegValue(dInst.src1));
         let rVal2 = rf.rd2(validRegValue(dInst.src2));
         d2e.enq(Decode2Execute{pc: pc, ppc: ppc,
              dIinst: dInst, epoch: fEpoch,
              rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.rDst); end
    end
endrule
```



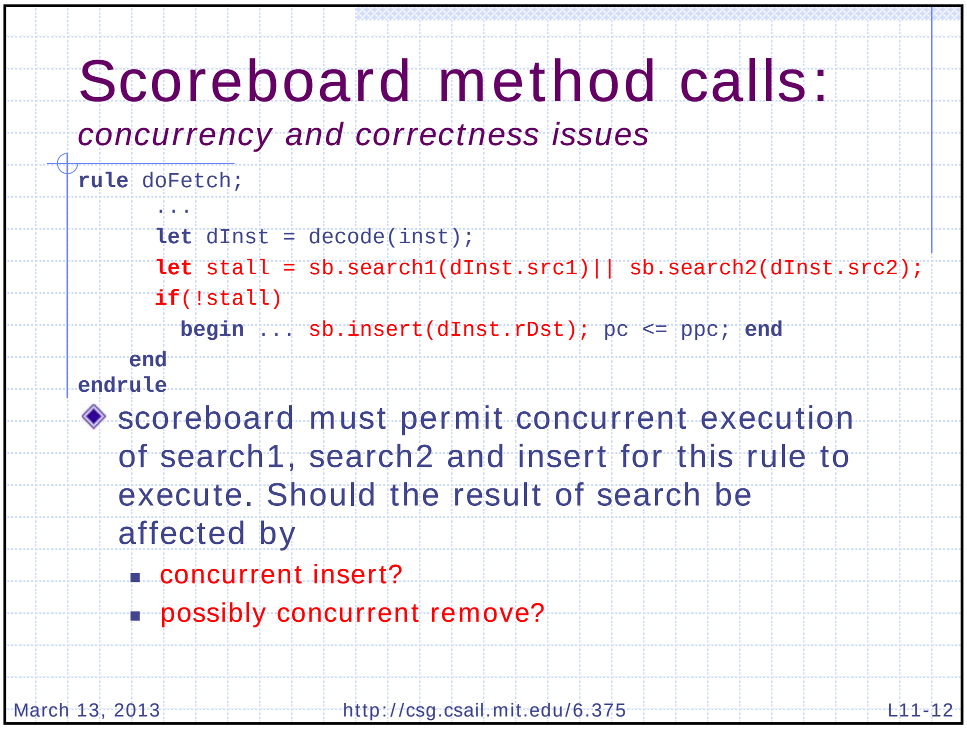
2-Stage-DH pipeline doFetch rule *corrected*

```
rule doFetch;
    let inst = iMem.req(pc);
    if(execRedirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= execRedirect.first;
      execRedirect.deq;       end
    else
    begin
      let ppc = nextAddrPredictor(pc); ~~pc <= ppc;~~
      let dInst = decode(inst);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
      if(!stall)                  begin
         let rVal1 = rf.rd1(validRegValue(dInst.src1));
         let rVal2 = rf.rd2(validRegValue(dInst.src2));
         d2e.enq(Decode2Execute{pc: pc, ppc: ppc,
              dIinst: dInst, epoch: fEpoch,
              rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.rDst); end
    end
endrule
```



2-Stage-DH pipeline doExecute rule *corrected*

```
rule doExecute;
    let x = d2e.first;
    let dInst = x.dInst; let pc    = x.pc;
    let ppc   = x.ppc;   let epoch = x.epoch;
    let rVal1 = x.rVal1; let rVal2 = x.rVal2;
    if(epoch == eEpoch) begin
      let eInst = exec(dInst, rVal1, rVal2, pc, ppc);
      if(eInst.iType == Ld) eInst.data <-
        dMem.req(MemReq{op:Ld, addr:eInst.addr, data:?});
      else if (eInst.iType == St) let d <-
        dMem.req(MemReq{op:St, addr:eInst.addr, data:eInst.data});
      if (isValid(eInst.dst))
        rf.wr(validRegValue(eInst.dst), eInst.data);
      if(eInst.mispredict) begin
        execRedirect.enq(eInst.addr); eEpoch <= !eEpoch; end
                        end
    d2e.deq; sb.remove;
endrule
```



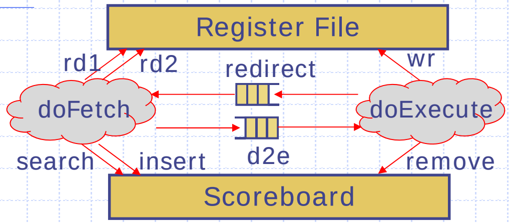
Scoreboard method calls:

concurrency and correctness issues

```
rule doFetch;
      ...
      let dInst = decode(inst);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
      if(!stall)
        begin ... sb.insert(dInst.rDst); pc <= ppc; end
    end
endrule
```

- scoreboard must permit concurrent execution of search1, search2 and insert for this rule to execute. Should the result of search be affected by
  - concurrent insert?
  - possibly concurrent remove?

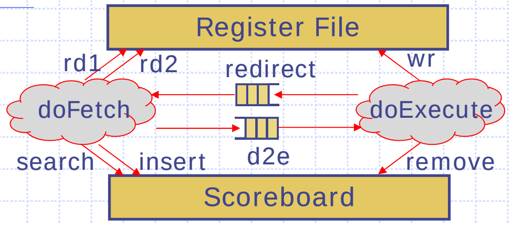
Assumptions about method calls



- Within a rule
  - d2e.first ? d2e.deq
  - redirect.first ? redirect.deq
  - sb.search ? sb.insert

Concurrency analysis

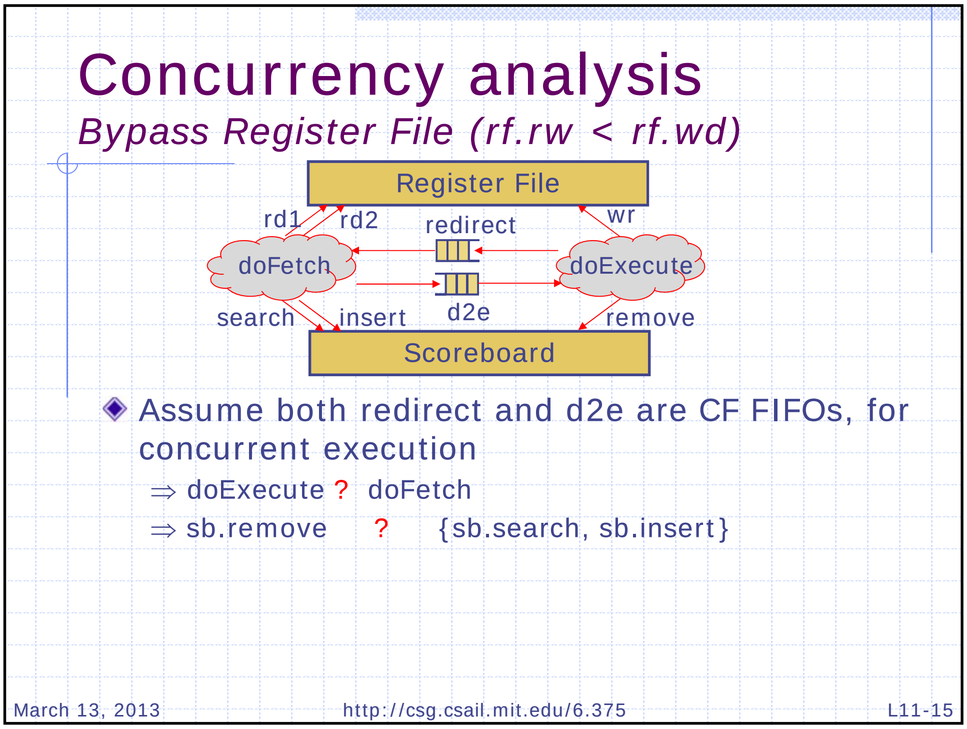
normal Register File (rf.rd < rf.wr)



- Assume both redirect and d2e are CF FIFOs, for concurrent execution

  ⇒ doFetch ? doExecute

  ⇒ {sb.search, sb.insert} ? sb.remove



Concurrency analysis

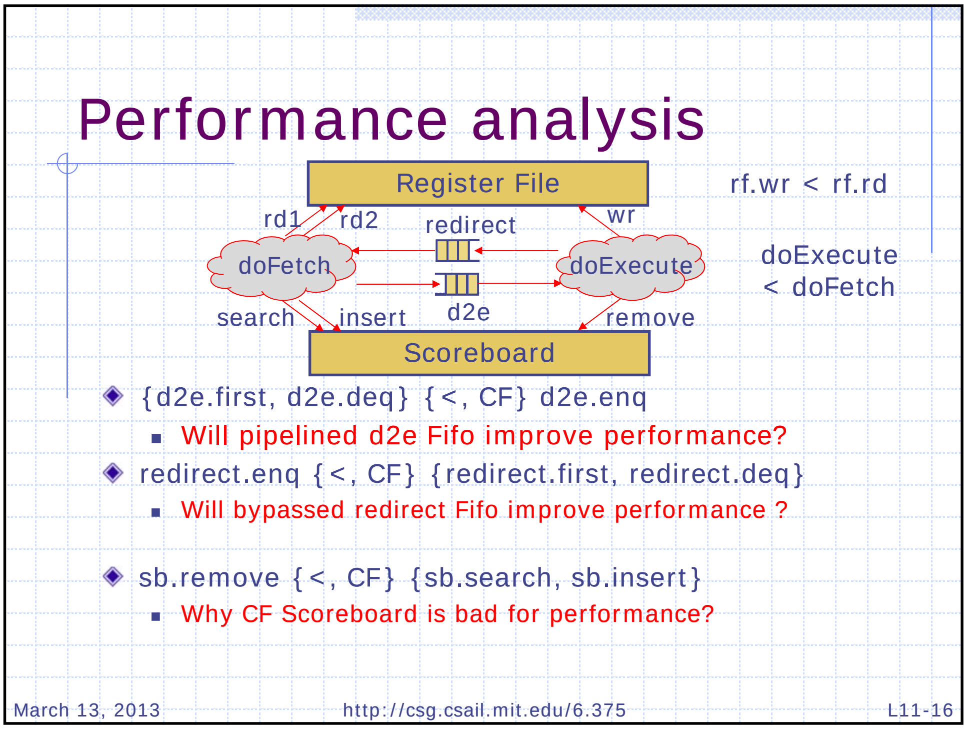
Bypass Register File (rf.rw < rf.wd)

Register File
rd1
rd2
redirect
wr
doFetch
doExecute
search
insert
d2e
remove
Scoreboard

- Assume both redirect and d2e are CF FIFOs, for concurrent execution

  ⇒ doExecute ? doFetch

  ⇒ sb.remove ? {sb.search, sb.insert}



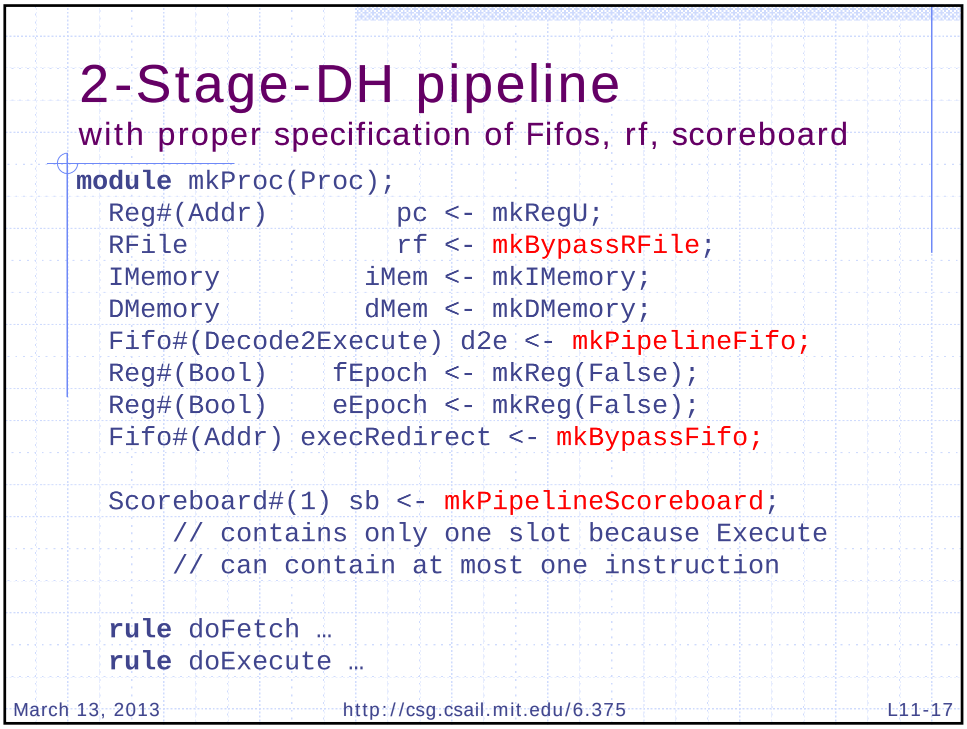
Performance analysis

Register File
rd1
rd2
redirect
wr
doFetch
doExecute
search
insert
d2e
remove
Scoreboard

rf.wr < rf.rd

doExecute
< doFetch

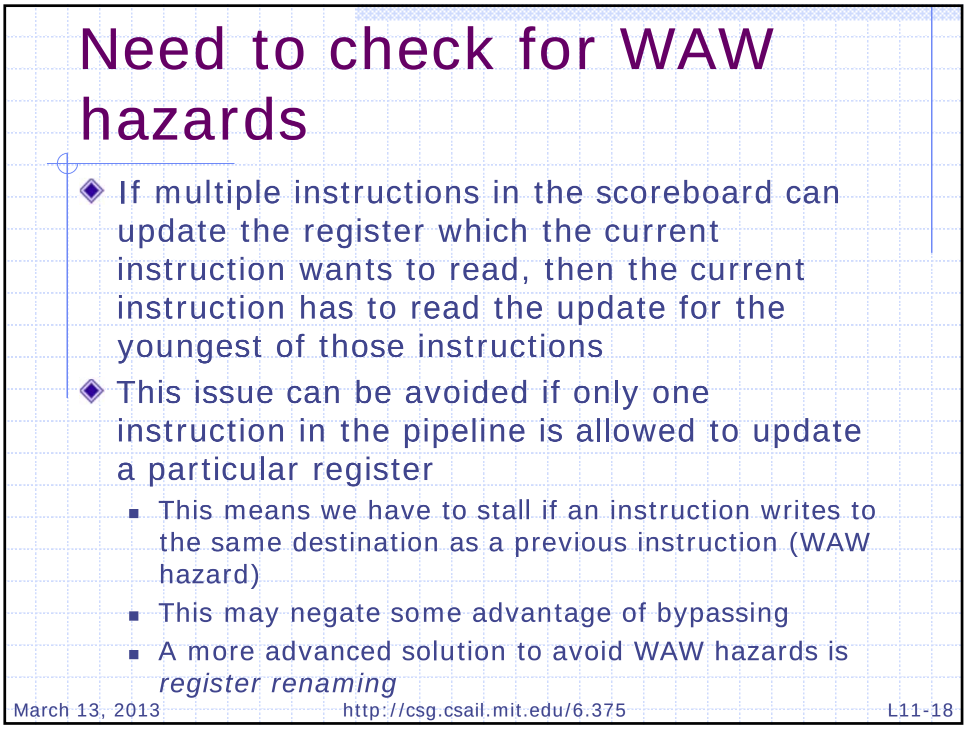
- {d2e.first, d2e.deq} {<, CF} d2e.enq
  - Will pipelined d2e Fifo improve performance?
- redirect.enq {<, CF} {redirect.first, redirect.deq}
  - Will bypassed redirect Fifo improve performance ?
- sb.remove {<, CF} {sb.search, sb.insert}
  - Why CF Scoreboard is bad for performance?



2-Stage-DH pipeline

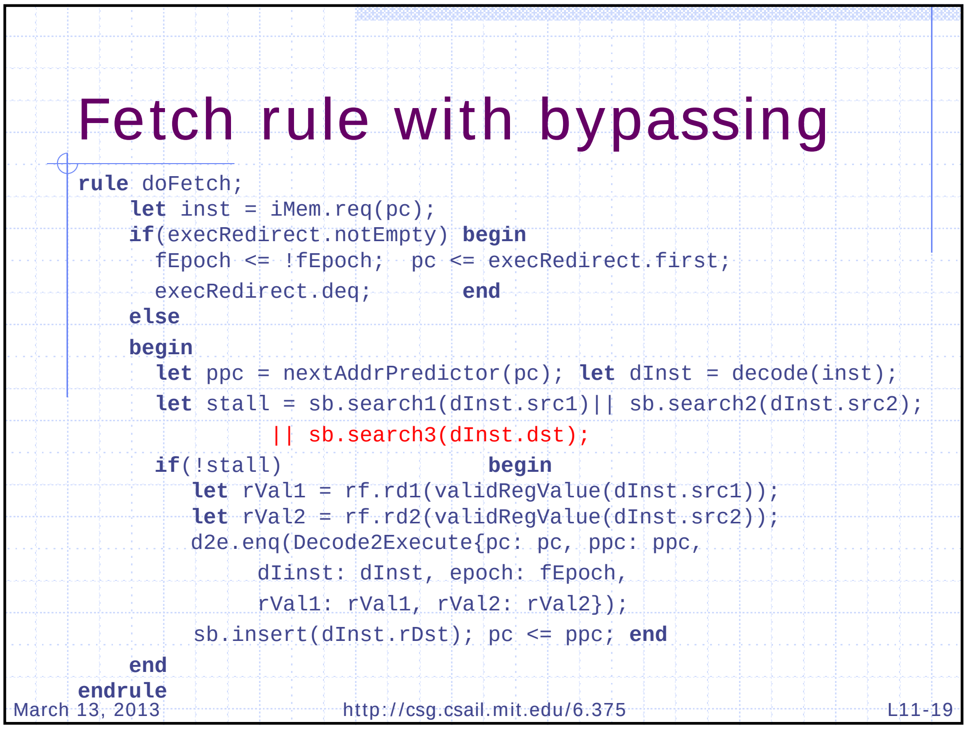
with proper specification of Fifos, rf, scoreboard

```
module mkProc(Proc);
  Reg#(Addr)          pc <- mkRegU;
  RFile               rf <- mkBypassRFile;
  IMemory           iMem <- mkIMemory;
  DMemory           dMem <- mkDMemory;
  Fifo#(Decode2Execute) d2e <- mkPipelineFifo;
  Reg#(Bool)      fEpoch <- mkReg(False);
  Reg#(Bool)      eEpoch <- mkReg(False);
  Fifo#(Addr) execRedirect <- mkBypassFifo;

  Scoreboard#(1) sb <- mkPipelineScoreboard;
      // contains only one slot because Execute
      // can contain at most one instruction

  rule doFetch …
  rule doExecute …
```



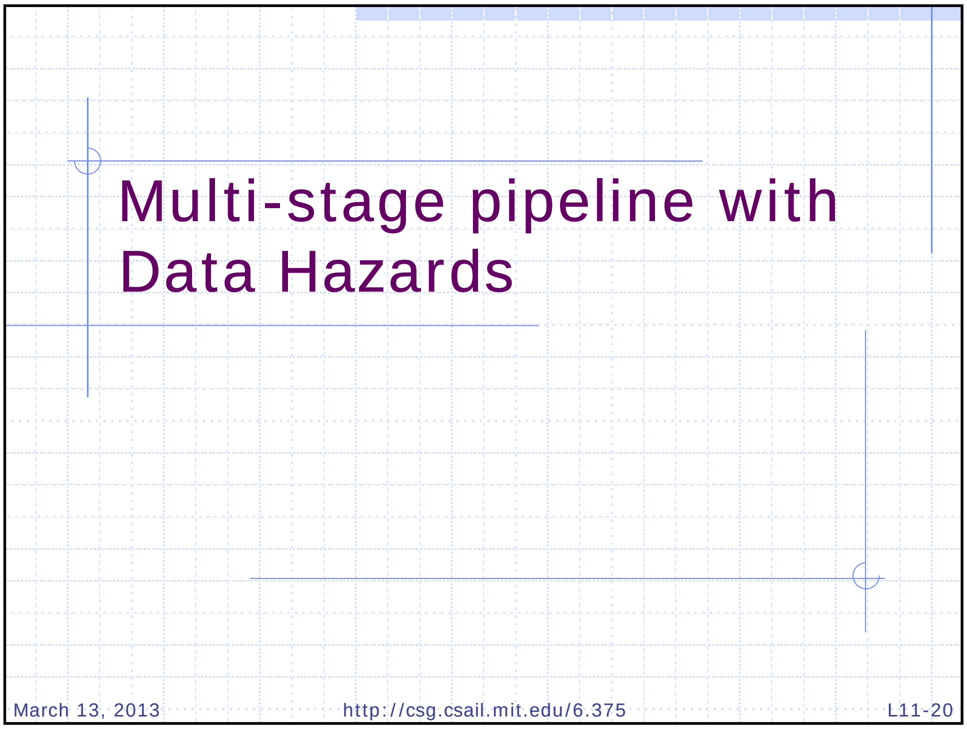
Need to check for WAW hazards

- If multiple instructions in the scoreboard can update the register which the current instruction wants to read, then the current instruction has to read the update for the youngest of those instructions
- This issue can be avoided if only one instruction in the pipeline is allowed to update a particular register
  - This means we have to stall if an instruction writes to the same destination as a previous instruction (WAW hazard)
  - This may negate some advantage of bypassing
  - A more advanced solution to avoid WAW hazards is *register renaming*

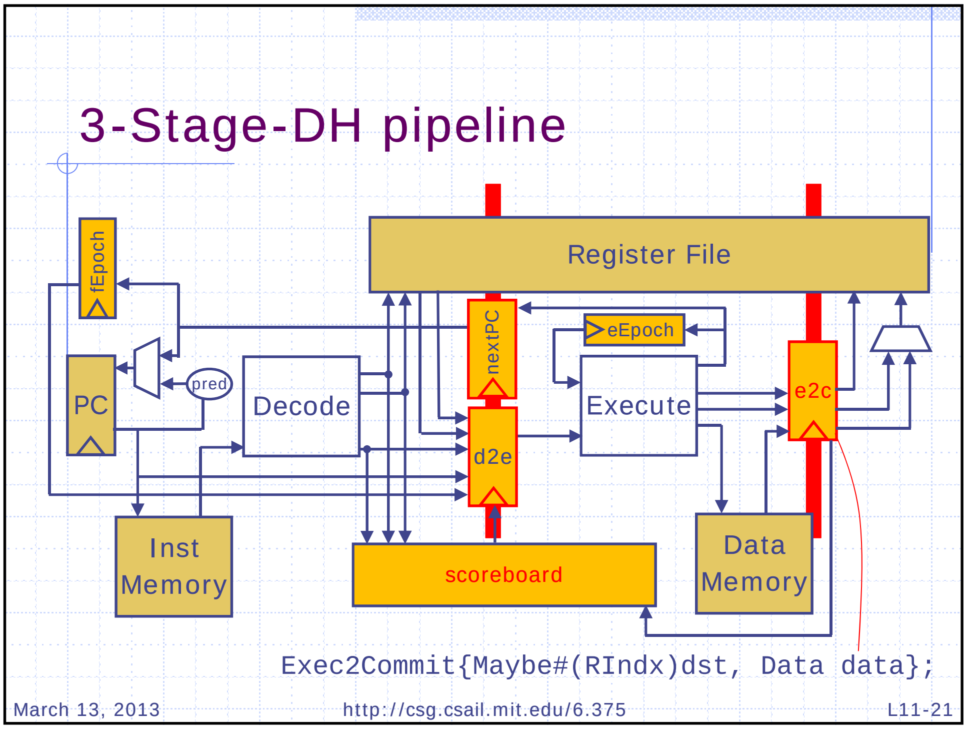


Fetch rule with bypassing

```
rule doFetch;
    let inst = iMem.req(pc);
    if(execRedirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= execRedirect.first;
      execRedirect.deq;       end
    else
    begin
      let ppc = nextAddrPredictor(pc); let dInst = decode(inst);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
               || sb.search3(dInst.dst);
      if(!stall)                begin
         let rVal1 = rf.rd1(validRegValue(dInst.src1));
         let rVal2 = rf.rd2(validRegValue(dInst.src2));
         d2e.enq(Decode2Execute{pc: pc, ppc: ppc,
              dIinst: dInst, epoch: fEpoch,
              rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.rDst); pc <= ppc; end
    end
endrule
```

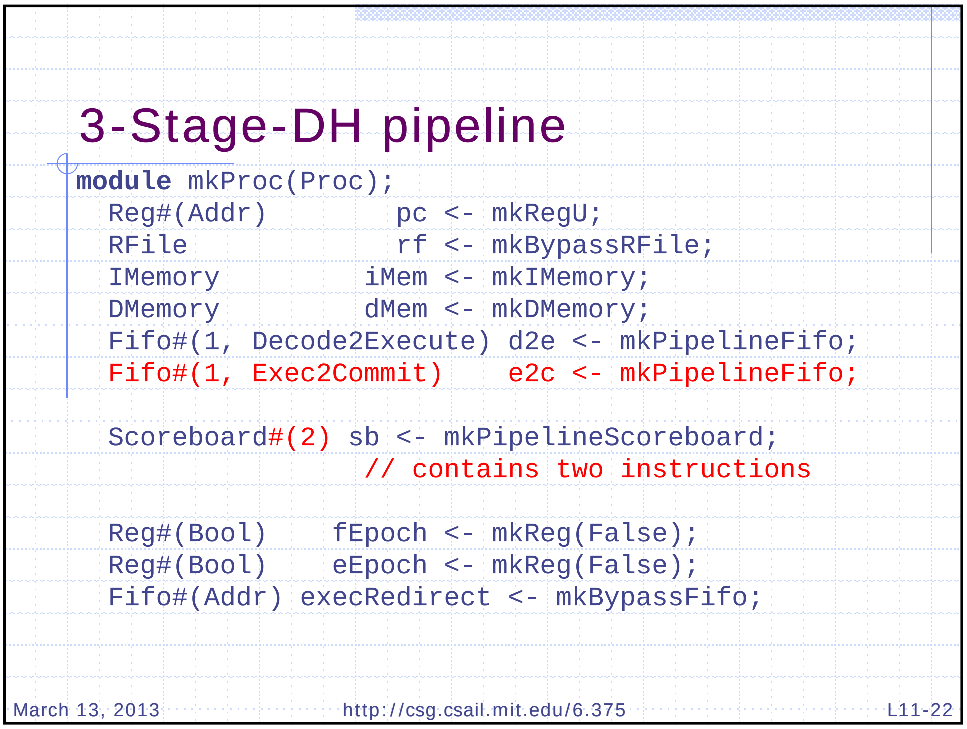


Multi-stage pipeline with Data Hazards



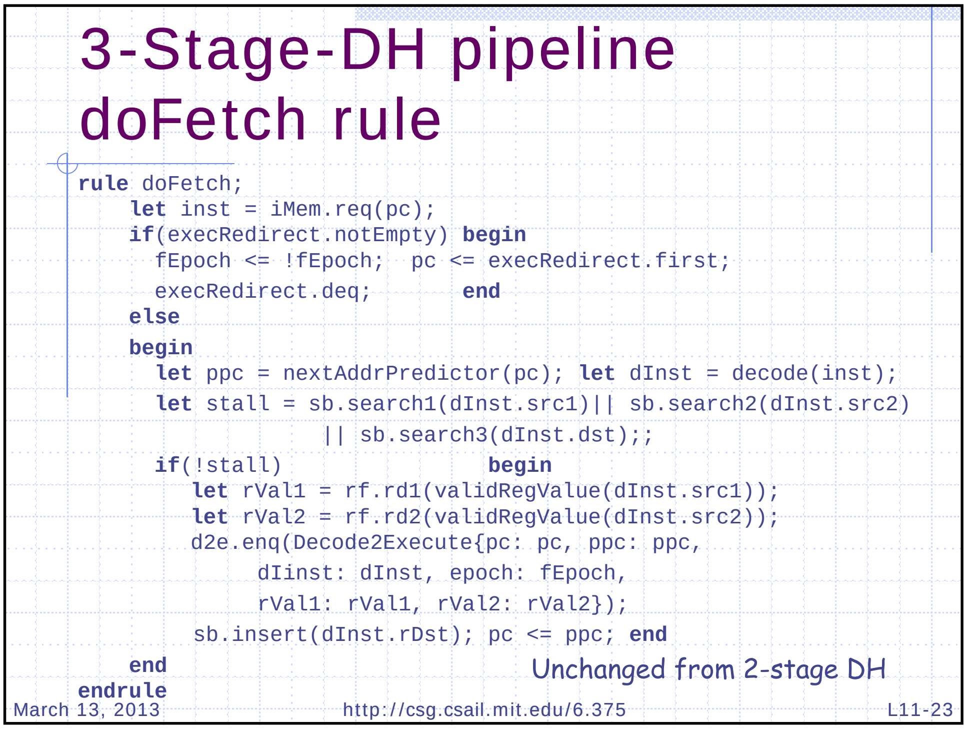
3-Stage-DH pipeline

Exec2Commit{Maybe#(RIndx)dst, Data data};



3-Stage-DH pipeline

```
module mkProc(Proc);
  Reg#(Addr)          pc <- mkRegU;
  RFile               rf <- mkBypassRFile;
  IMemory           iMem <- mkIMemory;
  DMemory           dMem <- mkDMemory;
  Fifo#(1, Decode2Execute) d2e <- mkPipelineFifo;
  Fifo#(1, Exec2Commit)    e2c <- mkPipelineFifo;

  Scoreboard#(2) sb <- mkPipelineScoreboard;
                // contains two instructions

  Reg#(Bool)     fEpoch <- mkReg(False);
  Reg#(Bool)     eEpoch <- mkReg(False);
  Fifo#(Addr) execRedirect <- mkBypassFifo;
```

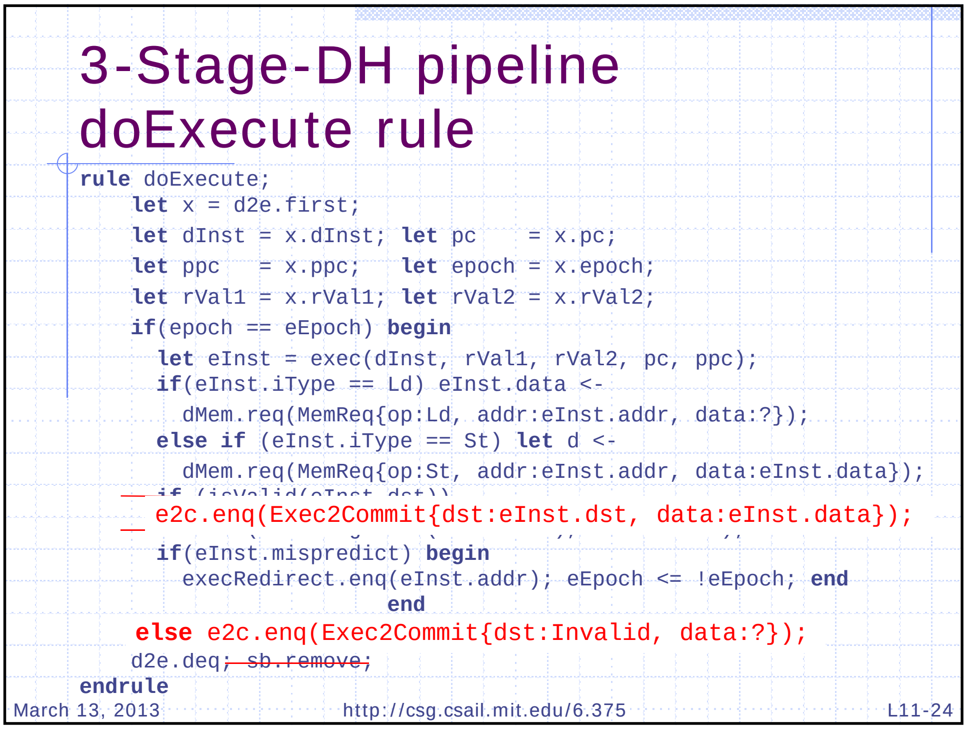


3-Stage-DH pipeline doFetch rule

```
rule doFetch;
    let inst = iMem.req(pc);
    if(execRedirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= execRedirect.first;
      execRedirect.deq;       end
    else
    begin
      let ppc = nextAddrPredictor(pc); let dInst = decode(inst);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2)
                   || sb.search3(dInst.dst);;
      if(!stall)                  begin
         let rVal1 = rf.rd1(validRegValue(dInst.src1));
         let rVal2 = rf.rd2(validRegValue(dInst.src2));
         d2e.enq(Decode2Execute{pc: pc, ppc: ppc,
              dIinst: dInst, epoch: fEpoch,
              rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.rDst); pc <= ppc; end
    end
endrule
```

Unchanged from 2-stage DH

March 13, 2013 http://csg.csail.mit.edu/6.375 L11-23



3-Stage-DH pipeline doExecute rule

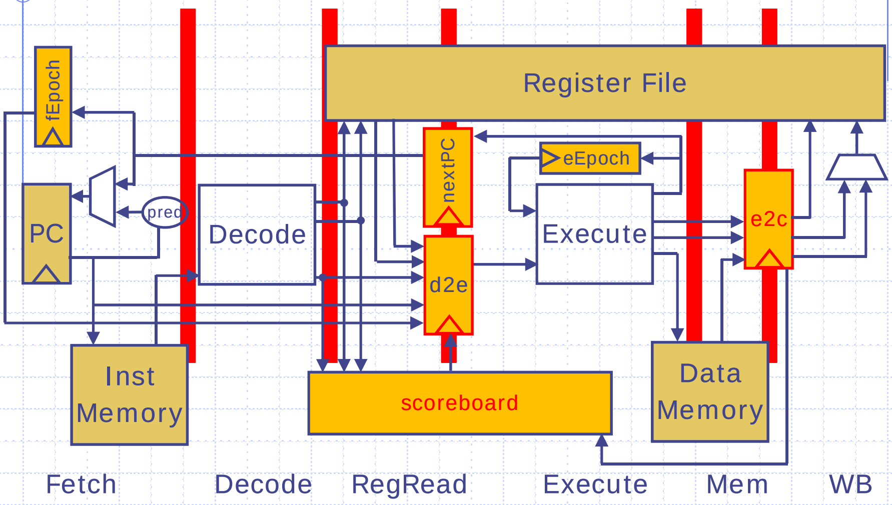
```
rule doExecute;
    let x = d2e.first;
    let dInst = x.dInst; let pc    = x.pc;
    let ppc   = x.ppc;   let epoch = x.epoch;
    let rVal1 = x.rVal1; let rVal2 = x.rVal2;
    if(epoch == eEpoch) begin
      let eInst = exec(dInst, rVal1, rVal2, pc, ppc);
      if(eInst.iType == Ld) eInst.data <-
        dMem.req(MemReq{op:Ld, addr:eInst.addr, data:?});
      else if (eInst.iType == St) let d <-
        dMem.req(MemReq{op:St, addr:eInst.addr, data:eInst.data});
      ~~if (isValid(eInst.dst))~~
      e2c.enq(Exec2Commit{dst:eInst.dst, data:eInst.data});
      if(eInst.mispredict) begin
        execRedirect.enq(eInst.addr); eEpoch <= !eEpoch; end
                    end
   else e2c.enq(Exec2Commit{dst:Invalid, data:?});
    d2e.deq;~~ sb.remove;~~
endrule
```

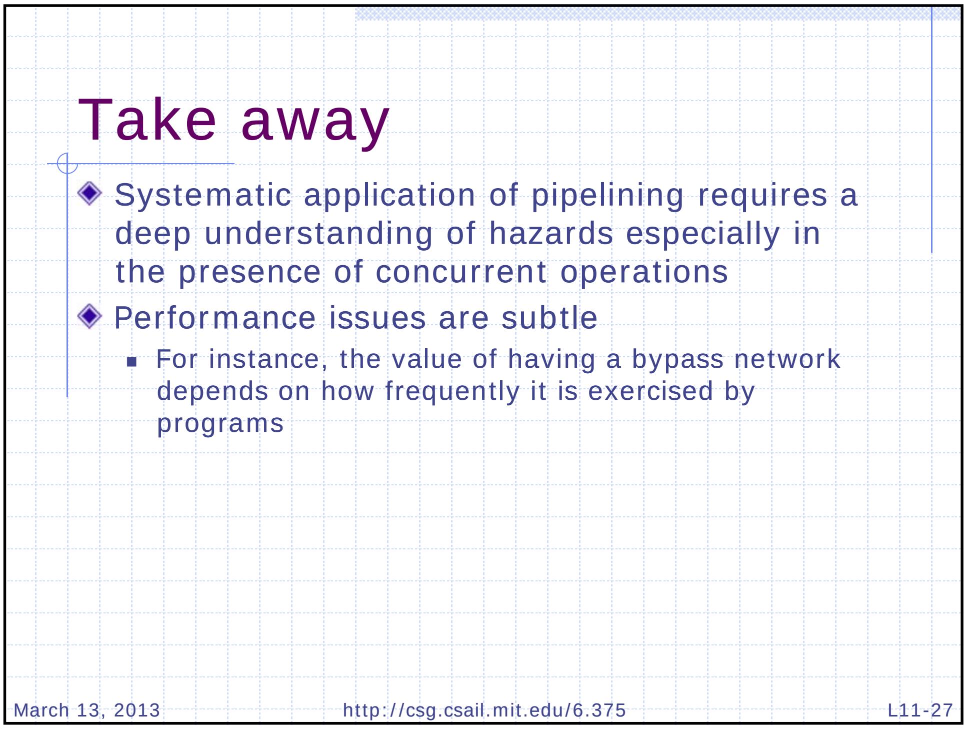
March 13, 2013 http://csg.csail.mit.edu/6.375 L11-24

3-Stage-DH pipeline doCommit rule

```
rule doCommit;
  let dst = eInst.first.dst;
  let data = eInst.first.data;
  if(isValid(dst))
    rf.wr(tuple2(validValue(dst), data);
  e2c.deq;
  sb.remove;
endrule
```

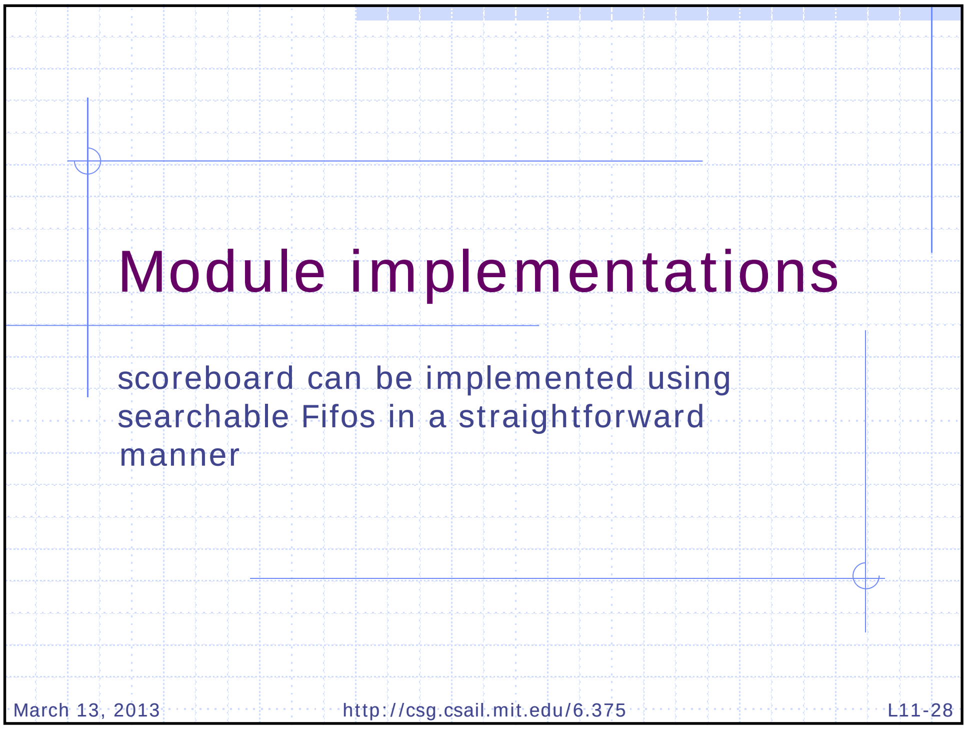
6-Stage-DH pipeline





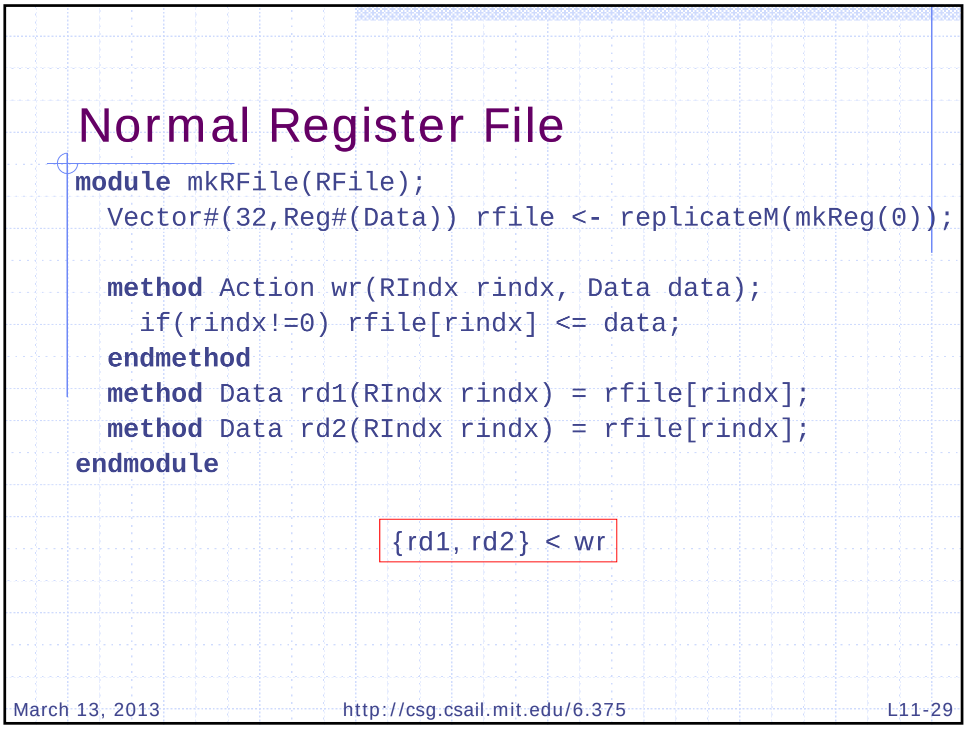
Take away

- Systematic application of pipelining requires a deep understanding of hazards especially in the presence of concurrent operations
- Performance issues are subtle
  - For instance, the value of having a bypass network depends on how frequently it is exercised by programs



Module implementations

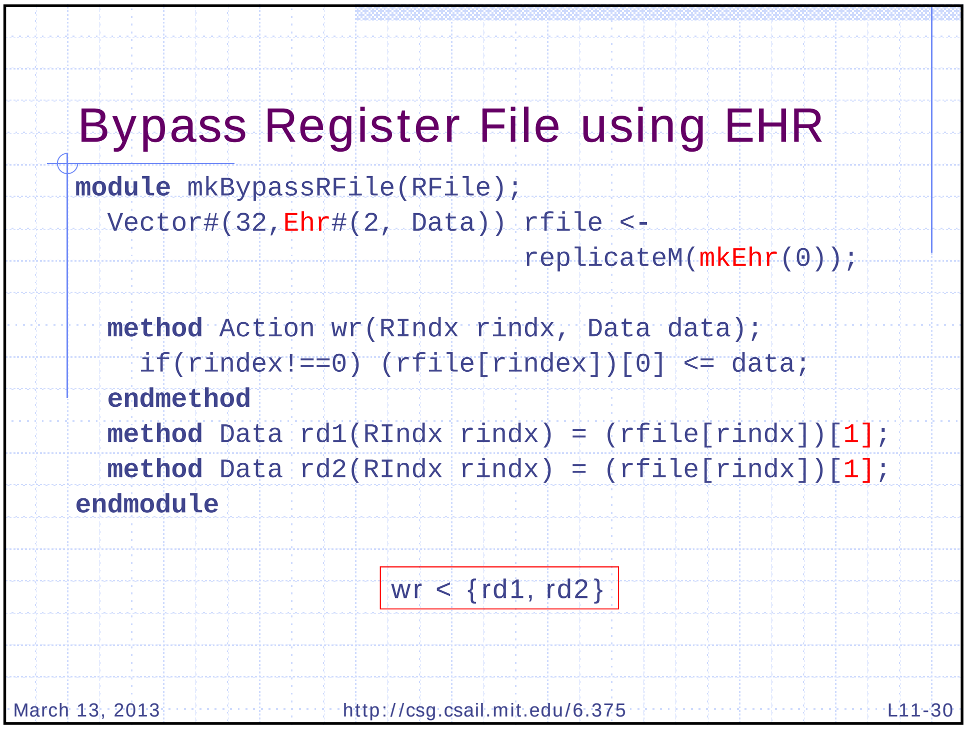
scoreboard can be implemented using searchable Fifos in a straightforward manner



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
```

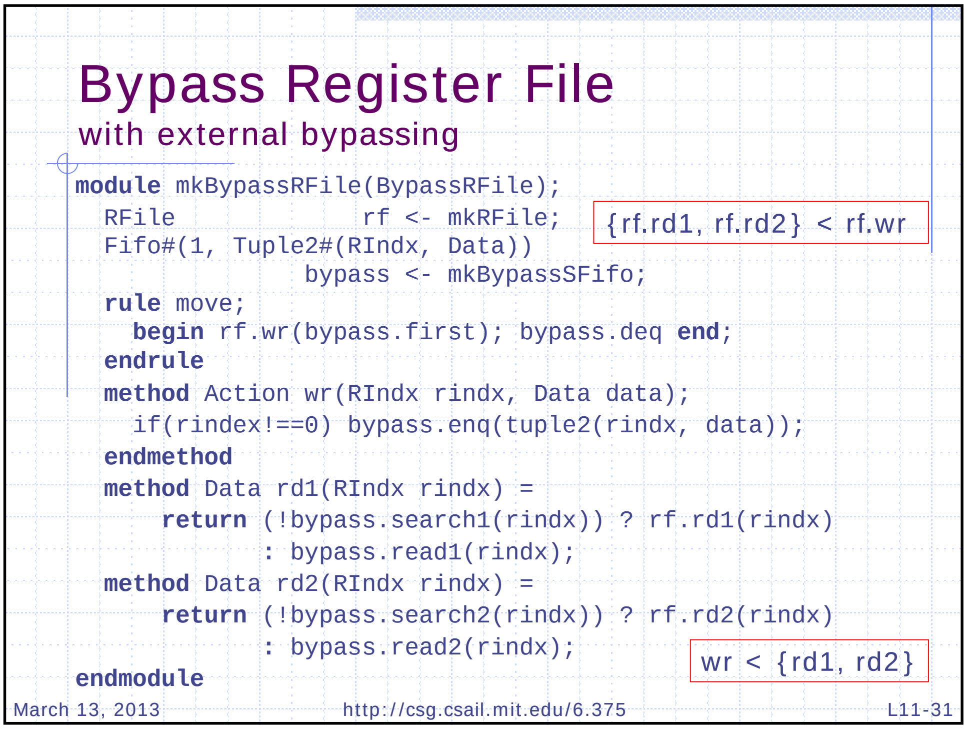
{rd1, rd2} < wr



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}



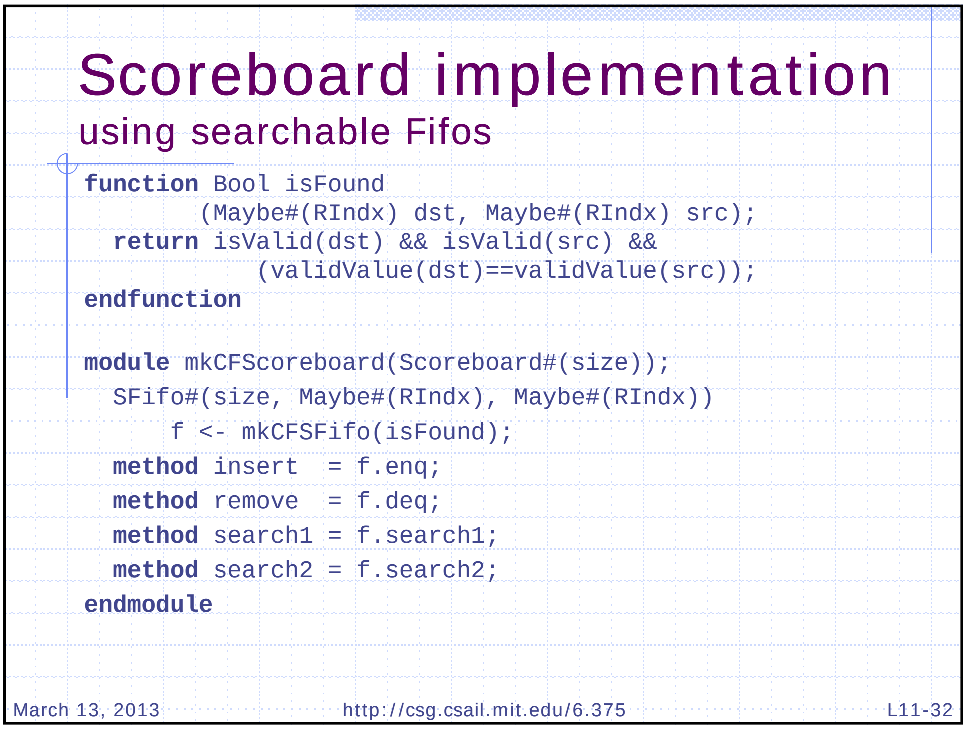
Bypass Register File

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}



Scoreboard implementation

using searchable Fifos

```
function Bool isFound
        (Maybe#(RIndx) dst, Maybe#(RIndx) src);
  return isValid(dst) && isValid(src) &&
            (validValue(dst)==validValue(src));
endfunction

module mkCFScoreboard(Scoreboard#(size));
  SFifo#(size, Maybe#(RIndx), Maybe#(RIndx))
      f <- mkCFSFifo(isFound);
  method insert  = f.enq;
  method remove  = f.deq;
  method search1 = f.search1;
  method search2 = f.search2;
endmodule
```