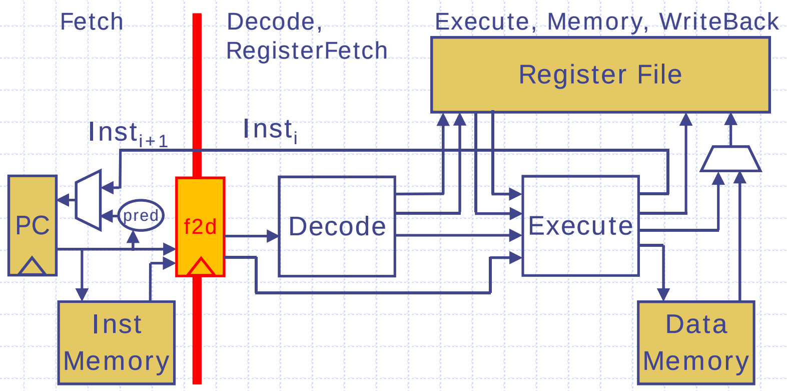
Constructive Computer Architecture:

Data Hazards in Pipelined Processors

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

Consider a different two-stage pipeline

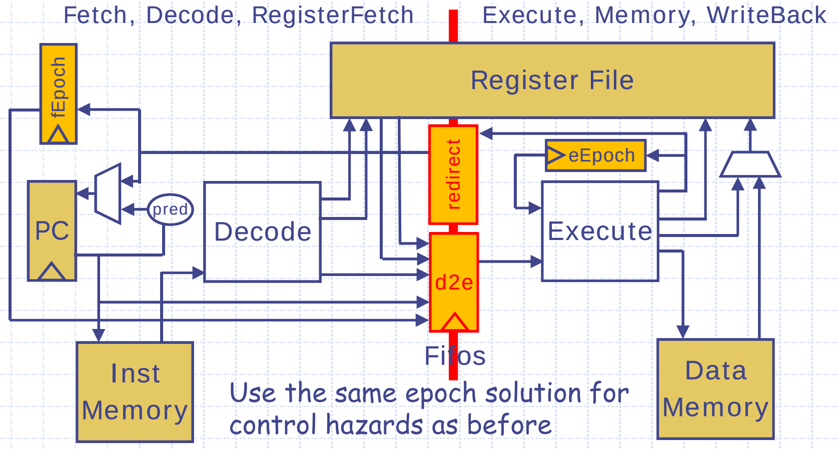


Suppose we move the pipeline stage from Fetch to after Decode and Register fetch for a better balance of work in two stages

Pipeline will still have control hazards

A different 2-Stage pipeline:

2-Stage-DH pipeline



Use the same epoch solution for control hazards as before

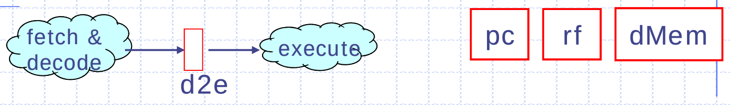
Converting the old pipeline into the new one

```
rule doFetch;
...    let instF = iMem.req(pc);
       f2d.enq(Fetch2Execute{... ~~inst: instF~~ ...}); ...
endrule

rule doExecute;               instF
...   let dInst = decode(~~instD~~);
      let rVal1 = rf.rd1(fromMaybe(?, dInst.src1));
      let rVal2 = rf.rd2(fromMaybe(?, dInst.src2));
      let eInst = exec(dInst, rVal1, rVal2, pcD, ppcD);
...
endrule
```

Not quite correct. Why?

Fetch is potentially reading stale values from rf

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 $R1 \leftarrow R2+R3$

I_2 $R4 \leftarrow R1+R2$

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

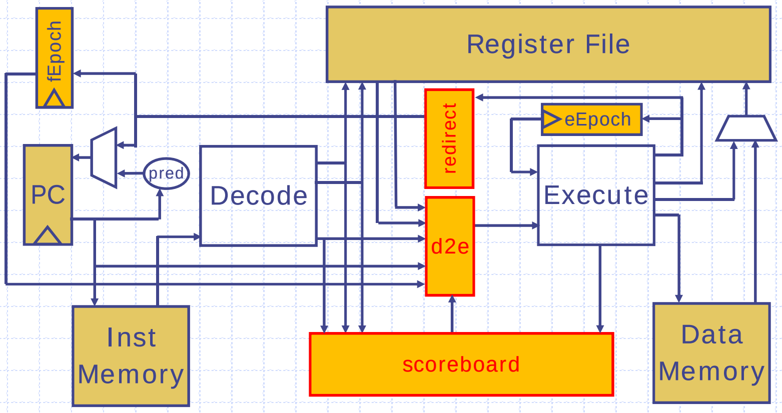
- Data hazard depends upon the match between the source registers of the fetched instruction and the destination register of an instruction already in the pipeline
- Both the source and destination registers must be Valid for a hazard to exist

```
function Bool isFound
    (Maybe#(RIndex) x, Maybe#(RIndex) y);
  if(x matches Valid .xv &&& y matches Valid .yv
                         &&& yv == xv)
       return True;
  else return False;
endfunction
```

Scoreboard: Keeping track of instructions in execution

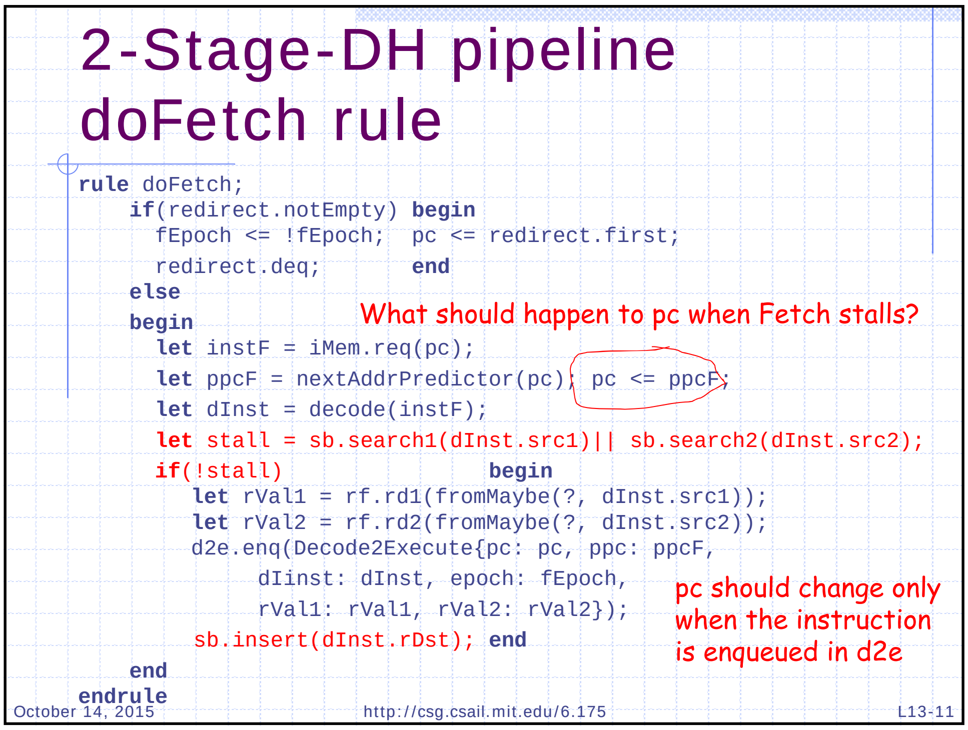
- Scoreboard: a data structure to keep track of the destination registers of the instructions beyond the fetch stage
  - *method insert:* inserts the destination (if any) of an instruction in the scoreboard when the instruction is decoded
  - *method search1(src):* searches the scoreboard for a data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

```
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) redirect <- mkFifo;

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

  rule doFetch …
  rule doExecute …
```

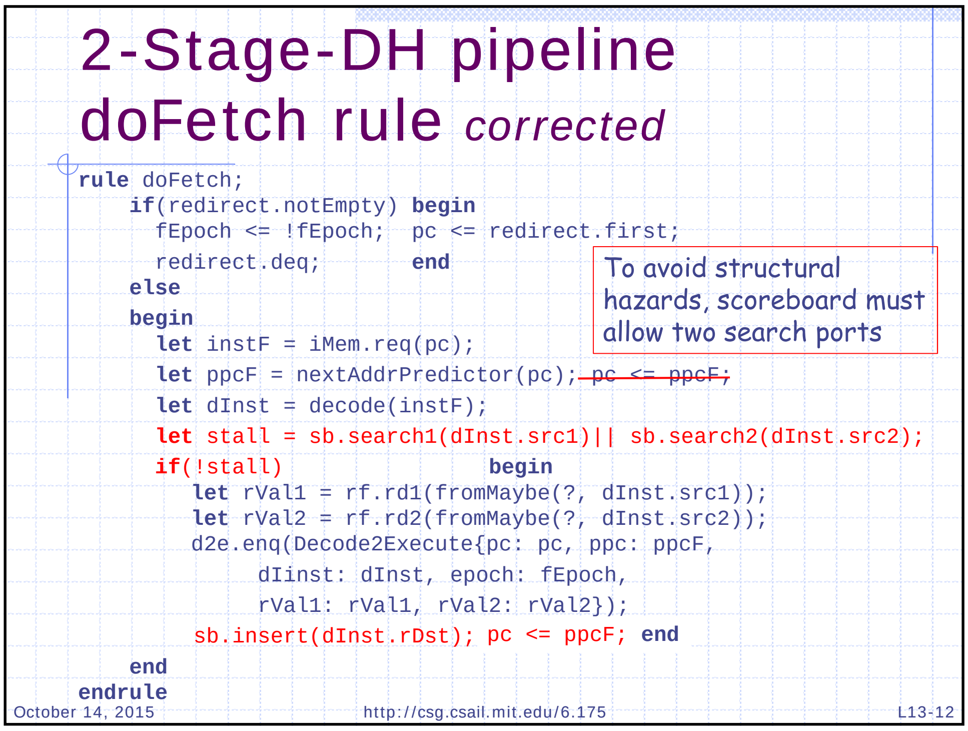


2-Stage-DH pipeline doFetch rule

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

What should happen to pc when Fetch stalls?

pc should change only when the instruction is enqueued in d2e



2-Stage-DH pipeline doFetch rule *corrected*

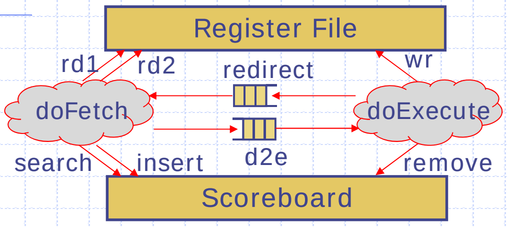
```
rule doFetch;
    if(redirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= redirect.first;
      redirect.deq;       end
    else
    begin
      let instF = iMem.req(pc);
      let ppcF = nextAddrPredictor(pc);~~ pc <= ppcF;~~
      let dInst = decode(instF);
      let stall = sb.search1(dInst.src1)||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

To avoid structural hazards, scoreboard must allow two search ports

2-Stage-DH pipeline doExecute rule

```
rule doExecute;
    let x = d2e.first;
    let dInstE = x.dInst; let pcE    = x.pc;
    let ppcE   = x.ppc;   let epoch  = x.epoch;
    let rVal1E = x.rVal1; let rVal2E = x.rVal2;
    if(epoch == eEpoch) begin
      let eInst = exec(dInstE, rVal1E, rVal2E, pcE, ppcE);
      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(fromMaybe(?, eInst.dst), eInst.data);
      if(eInst.mispredict) begin
        redirect.enq(eInst.addr); eEpoch <= !eEpoch; end
                    end
    d2e.deq; sb.remove;
endrule
```

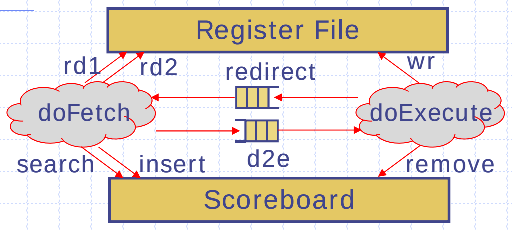
A correctness issues



- If the search by Decode does not see an instruction in the scoreboard, then its effect must have taken place. This means that any updates to the register file by that instruction must be visible to the subsequent register reads ⇒
  - remove and wr should happen atomically
  - search and rd1, rd2 should happen atomically

Fetch and Execute can execute in any order

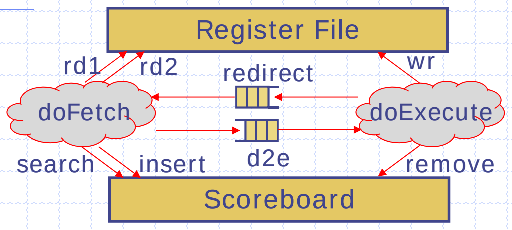
Concurrently executable Fetch and Execute



which is better?

- Case 1: doExecute < dofetch ⇒
  - rf: wr < rd (bypass rf)
  - sb: remove < {search, insert}
  - d2e: {first, deq} {<, CF} enq (pipelined or CF Fifo)
  - redirect: enq {<, CF} {deq, first} (bypass or CF Fifo)
- Case 2: doFetch < doExecute ⇒
  - rf: rd < wr (normal rf)
  - sb: {search, insert} < remove
  - d2e: enq {<, CF} {deq, first} (bypass or CF Fifo)
  - redirect: {first, deq} {<, CF} enq (pipelined or CF Fifo)

Performance issues



- To avoid a stall due to a RAW hazard between successive instructions
  - sb: remove < search
  - rf: wr < rd (bypass rf)
- To minimize stalls due to control hazards
  - redirect: bypass fifo
- What kind of fifo should be used for d2e ?
  - Either a pipeline or CF fifo would do fine

2-Stage-DH pipeline

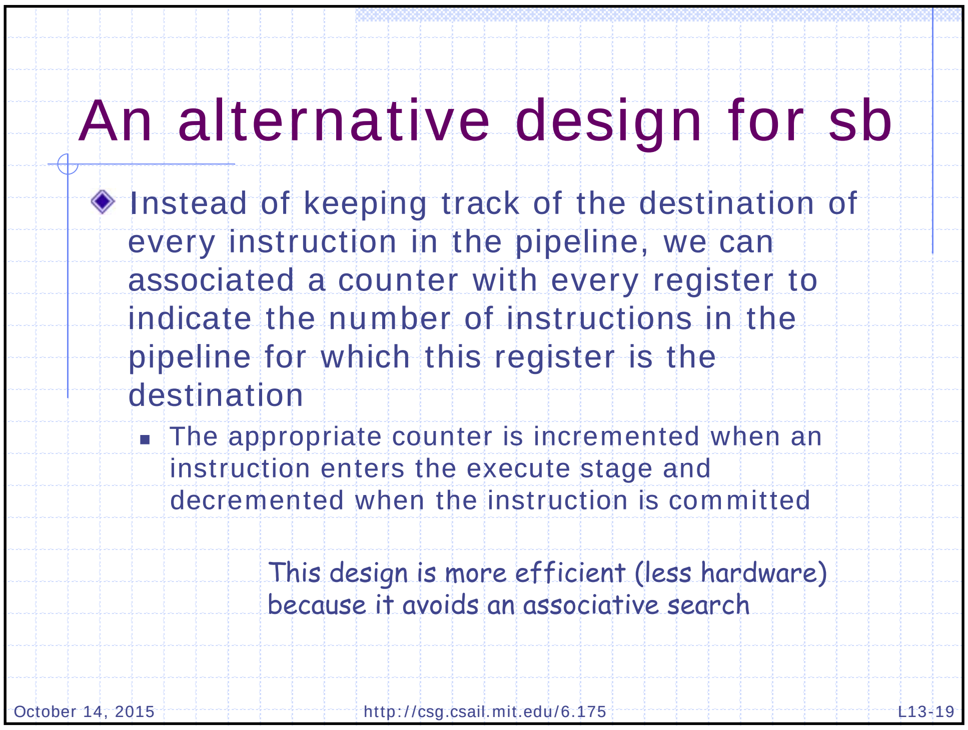
with proper specification of Fifos, rf, scoreboard

```
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) redirect <- mkBypassFifo;

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

  rule doFetch …
  rule doExecute …
```

Can a destination register name appear more than once in the scoreboard ?

WAW hazards

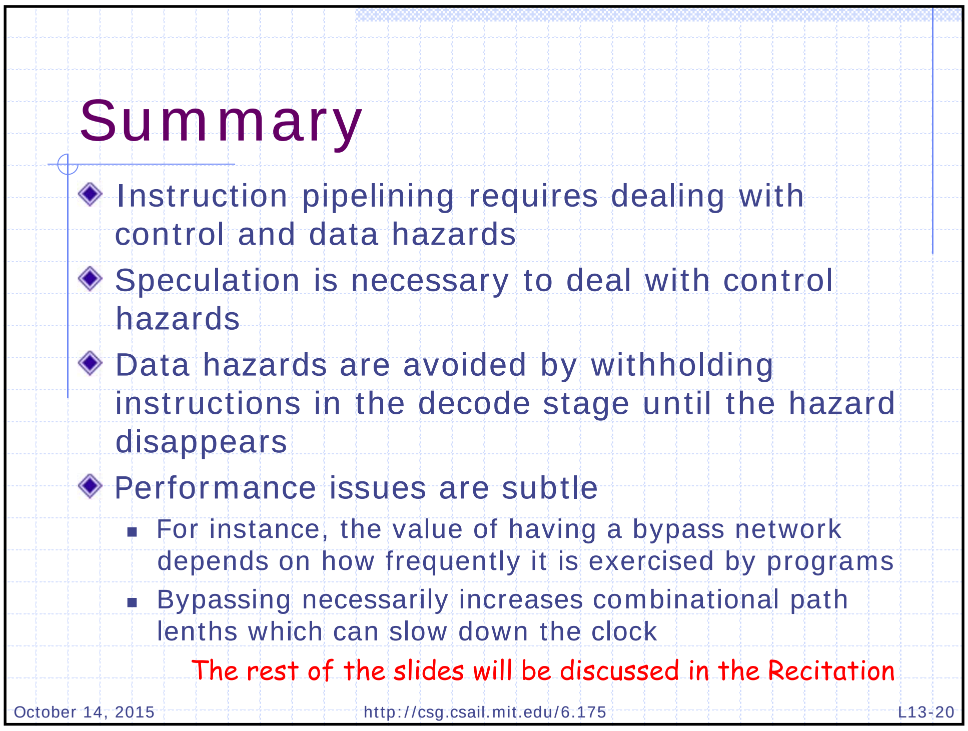
- If multiple instructions in the scoreboard can update the register which the current instruction wants to read, then the current instruction has to read the update for the youngest of those instructions
- This is not a problem in our design because
  - instructions are committed in order
  - the RAW hazard for the instruction at the decode stage will remain as long as the any instruction with the required destination is present in sb



An alternative design for sb

- Instead of keeping track of the destination of every instruction in the pipeline, we can associated a counter with every register to indicate the number of instructions in the pipeline for which this register is the destination
  - The appropriate counter is incremented when an instruction enters the execute stage and decremented when the instruction is committed

This design is more efficient (less hardware) because it avoids an associative search



Summary

- Instruction pipelining requires dealing with control and data hazards
- Speculation is necessary to deal with control hazards
- Data hazards are avoided by withholding instructions in the decode stage until the hazard disappears
- Performance issues are subtle
  - For instance, the value of having a bypass network depends on how frequently it is exercised by programs
  - Bypassing necessarily increases combinational path lenths which can slow down the clock

The rest of the slides will be discussed in the Recitation

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

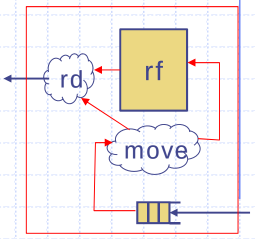
Bypass Register File using EHR

```
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
```

wr < {rd1, rd2}

Scoreboard implementation

using searchable Fifos

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