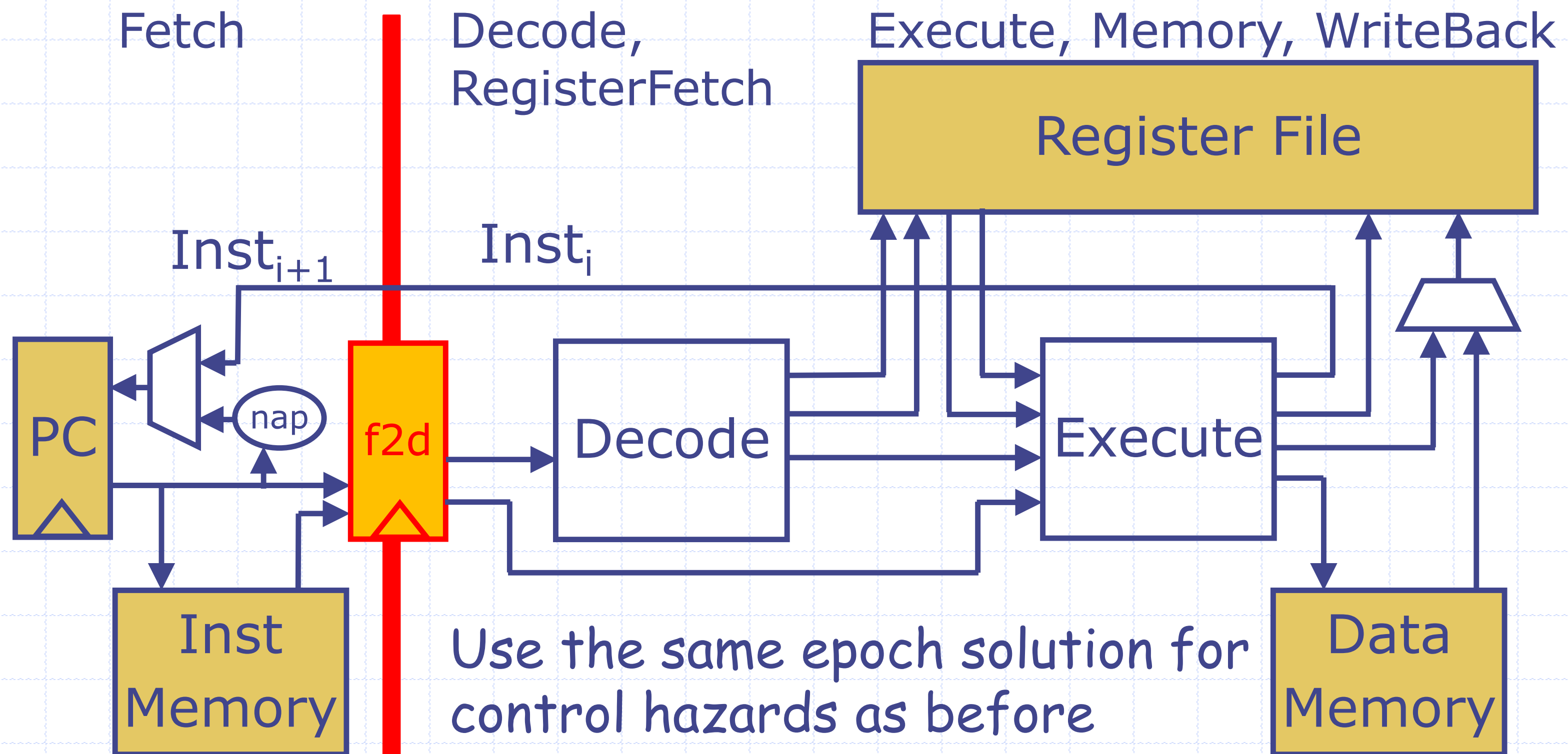
Constructive Computer Architecture:

Data Hazards in Pipelined Processors

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

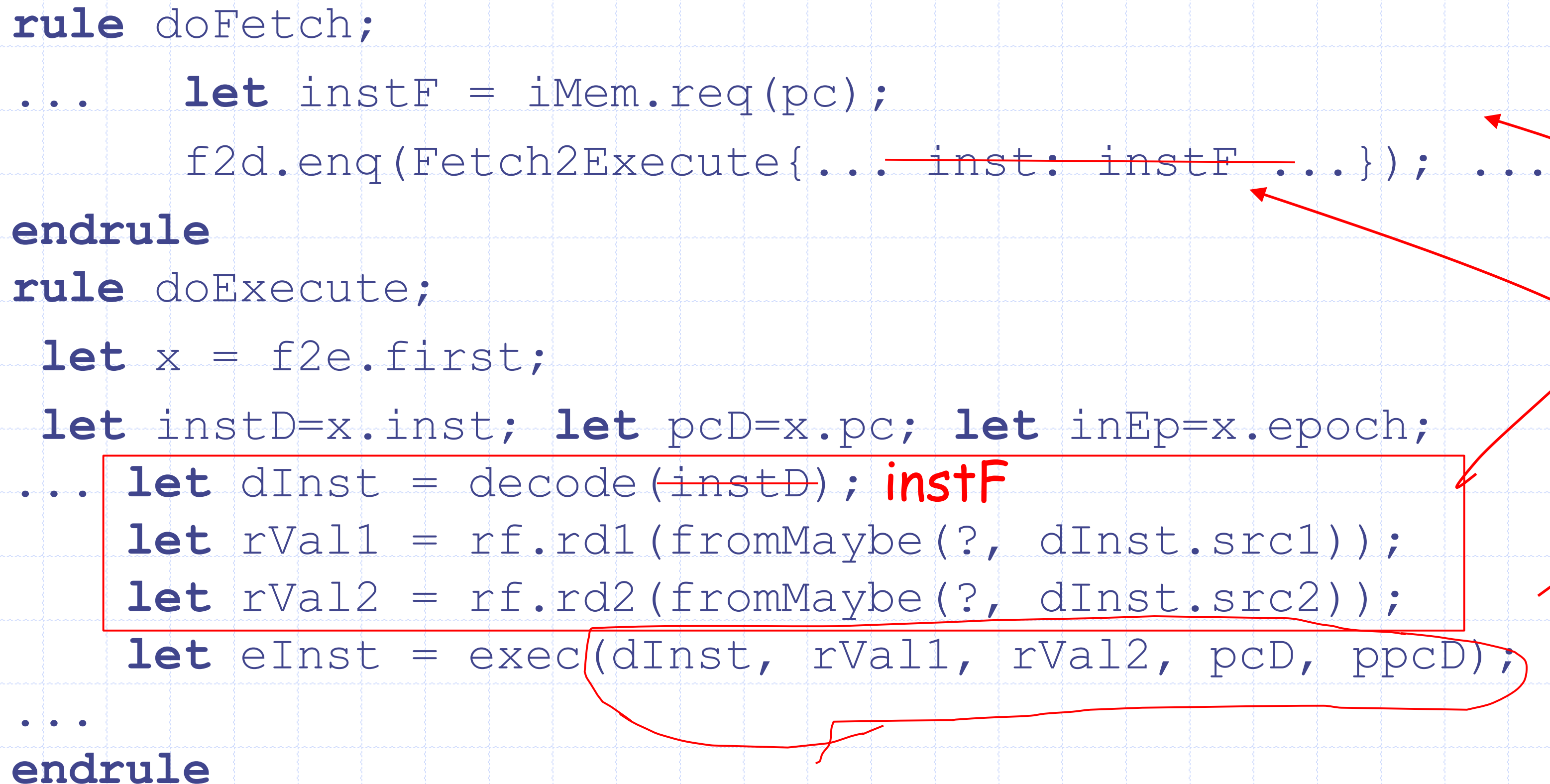
Consider a different two-stage pipeline



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 and we can use the epoch-based solution as before

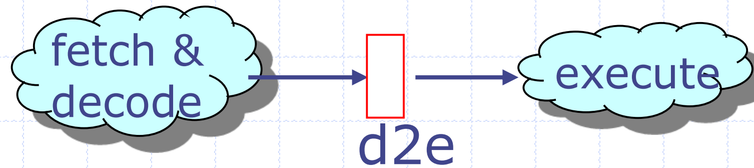
Converting the old pipeline into the new one

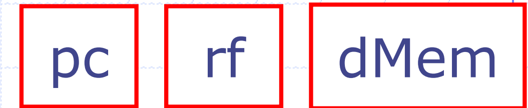


Not quite correct. Why?

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

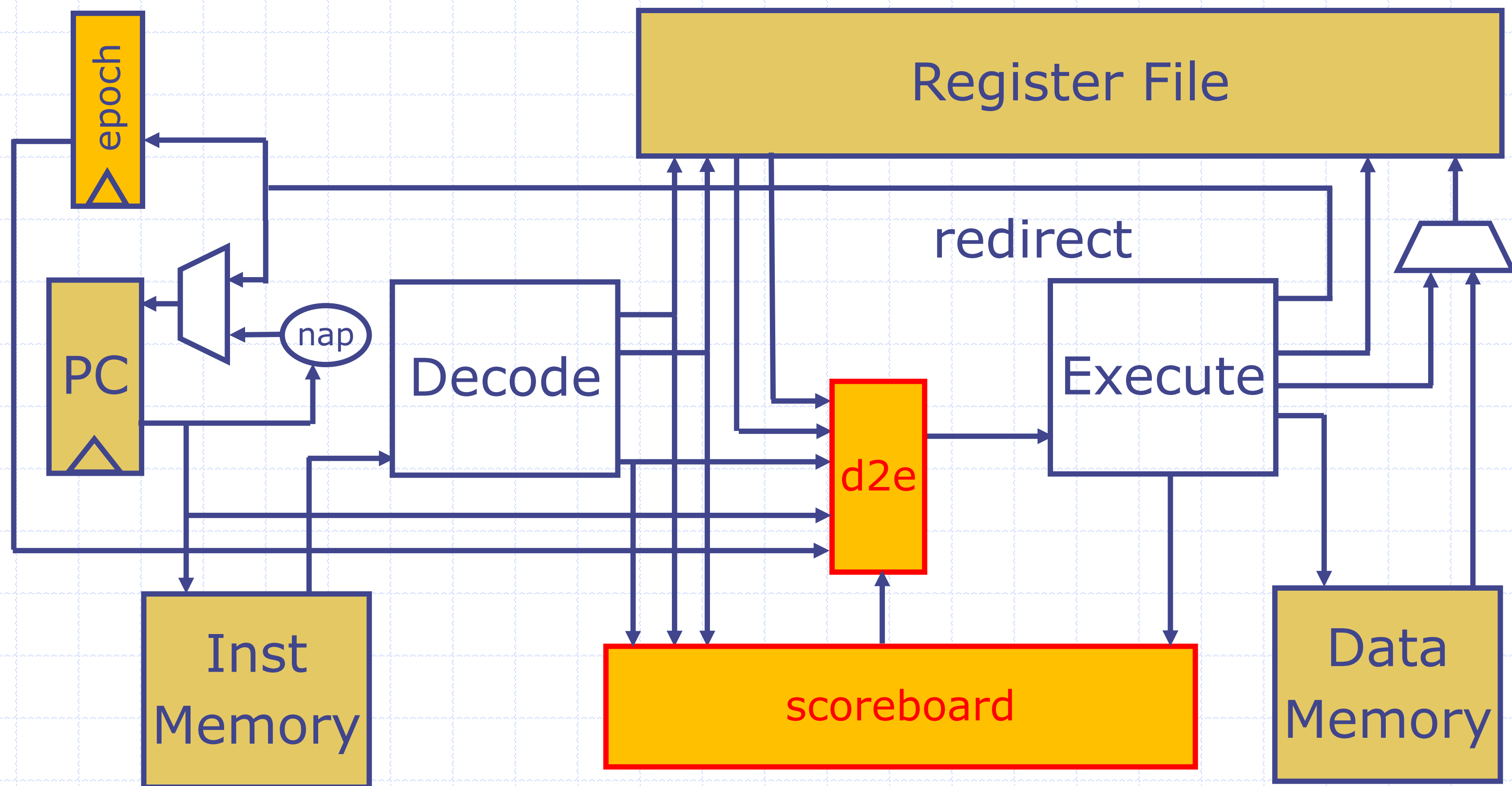
- Data hazard depends upon the match between the source registers of the fetched instruction and the destination register of an instruction already in the pipeline
- Both the source and destination registers must be Valid for a hazard to exist

```
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);
  EHR#(2,Addr)           pc <- mkEHR(U);
  RFile                rf <- mkRFile;
  IMemory            iMem <- mkIMemory;
  DMemory            dMem <- mkDMemory;
  Fifo#(Decode2Execute) d2e <- mkFifo;
  Reg#(Bool)      epoch <- mkReg(False);
  Scoreboard#(n) sb <- mkScoreboard;
      // n, the number of slots in the sb must be ≥
      // the number of instructions in the Execute
      // phse (including d2e)

  rule doFetch …
  rule doExecute …
```

Assume doFetch < doExecute

2-Stage-DH pipeline doFetch rule

What should happen to pc when Fetch stalls?

```
rule doFetch;
      let instF = iMem.req(pc[0]);
      let ppcF = nap(pc[0]); pc[0] <= ppcF;
      let dInst = decode(instF);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
      if(!stall)                    begin
         …fetch register values
         d2e.enq(Decode2Execute{pc: pc, ppc: ppcF,
                dIinst: dInst, epoch: epoch,
                rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.rDst); end
endrule
```

pc should change only when the instruction is enqueued in d2e

2-Stage-DH pipeline doFetch rule *corrected*

```
rule doFetch;
      let instF = iMem.req(pc[0]);
      let ppcF = nap(pc[0]); pc[0] <= ppcF;
      let dInst = decode(instF);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
      if(!stall)                     begin
         ...fetch register values
         d2e.enq(Decode2Execute{pc: pc, ppc: ppcF,
               dIinst: dInst, epoch: fEpoch,
               rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.rDst); pc[0] <= ppcF; end
endrule
```

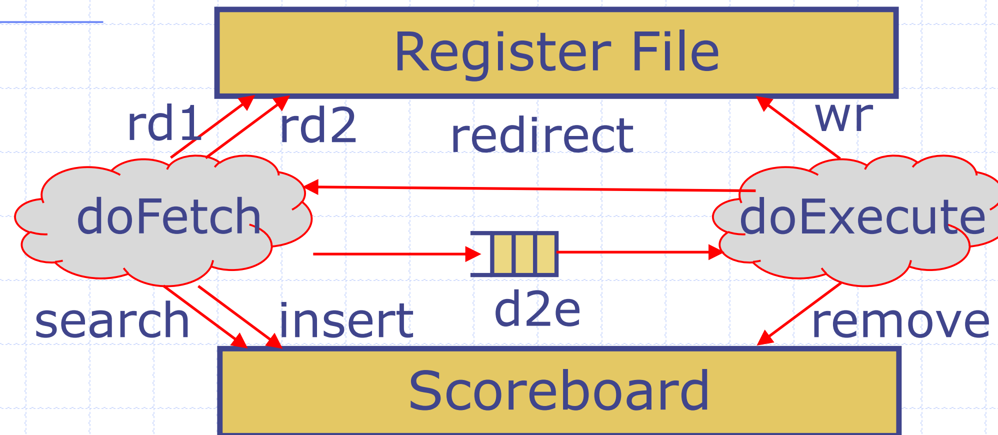
To avoid structural hazards, scoreboard must allow two search ports

2-Stage-DH pipeline doExecute rule

```
rule doExecute;
    let x = d2e.first;
    let dInstE = x.dInst; let pcE = x.pc; let inEp  = x.epoch;
    let rVal1E = x.rVal1; let rVal2E = x.rVal2;
    if(epoch == inEp) begin
      let eInst = exec(dInstE, rVal1E, rVal2E, pcE);
      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);
      let nextPC = eInst.brTaken ? eInst.addr : pcE + 4;
      if (x.ppc != nextPC) begin pc[1] <= eInst.addr;
                                 epoch <= !epoch; end
                        end
    d2e.deq; sb.remove;
endrule
```

The same as before

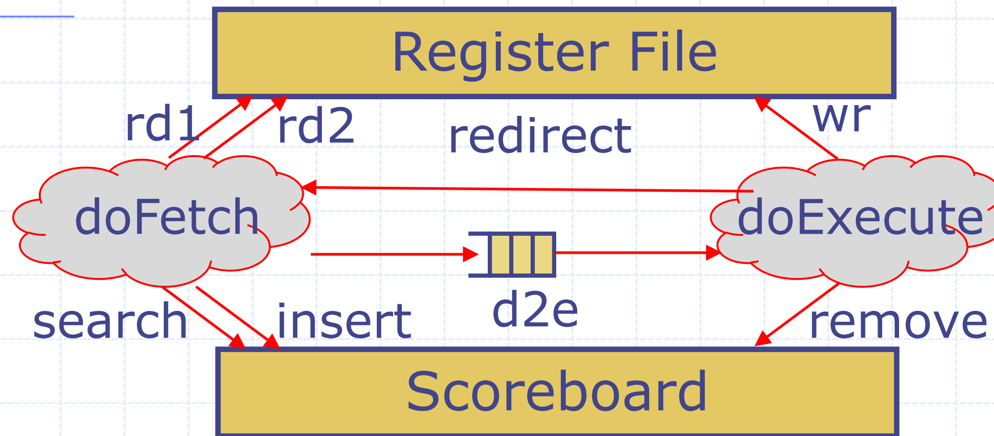
A correctness issue



- 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

Concurrency and Performance

doFetch < doExecute



Bypass FIFO does not make sense here

- For correctness:
  - rf: rd < wr (normal rf)
  - sb: {search, insert} < remove
  - d2e: enq {<, CF} {deq, first} (CF Fifo)
- performance ?
  - Dead cycle after each misprediction
  - Dead cycle after each RAW hazard

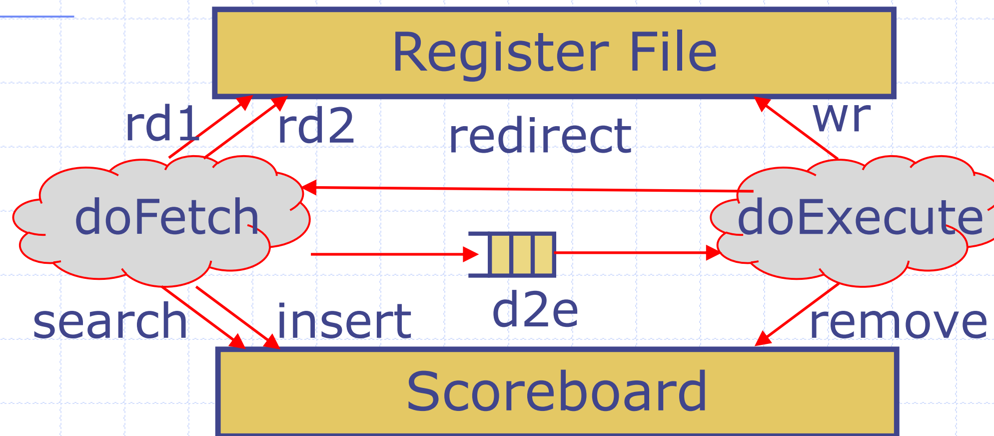
Maybe we should consider doExecute < doFetch even though the clock cycle may be a bit longer

2-Stage-DH pipeline doExecute < doFetch

```
rule doFetch;
      let instF = iMem.req(pc[1]);
      let ppcF = nap(pc[1]);;
      let dInst = decode(instF);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
      if(!stall)                     begin
         …fetch register values
         d2e.enq(Decode2Execute{pc: pc[1], ppc: ppcF,
                   dIinst: dInst, epoch: fEpoch,
                   rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.rDst); pc[1] <= ppcF end
endrule
rule doExecute;
      the same as before …
      if (x.ppc != nextPC) begin pc[0] <= eInst.addr;
                                  epoch <= !epoch; end
      end    d2e.deq; sb.remove;
endrule
```

Concurrency and Performance

doFetch < doExecute



- For correctness;
  - rf: wr < rd (bypass rf)
  - sb: remove < {search, insert}
  - d2e: {first, deq} {<, CF} enq (pipelined or CF Fifo)
- To avoid a stall due to a RAW hazard between successive instructions
  - sb: remove < search
  - rf: wr < rd (bypass rf)
- Also no dead cycle after a misprediction

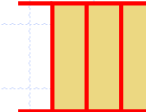
WAW hazards

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

One counter for each register in rf (Initially 0)

vs



One slot to hold rd for each instruction in the pipeline

- Insert: increment the counter for register rd
- Remove: decrement the counter for register rd
- Search: If the counter for the source register is >0, return True

This design takes less hardware for deep pipelines and is more efficient because it avoids associative searches

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 lengths 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

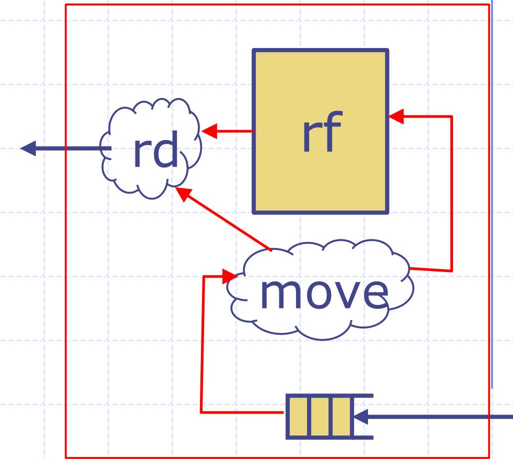
```
module mkBypassRFile(RFile);
  Vector#(32,Ehr#(2, Data)) rfile <-
                             replicateM(mkEhr(0));

  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
```