Constructive Computer Architecture:

Data Hazards in Pipelined Processors

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

Delivered by Andy Wright

Data Hazard

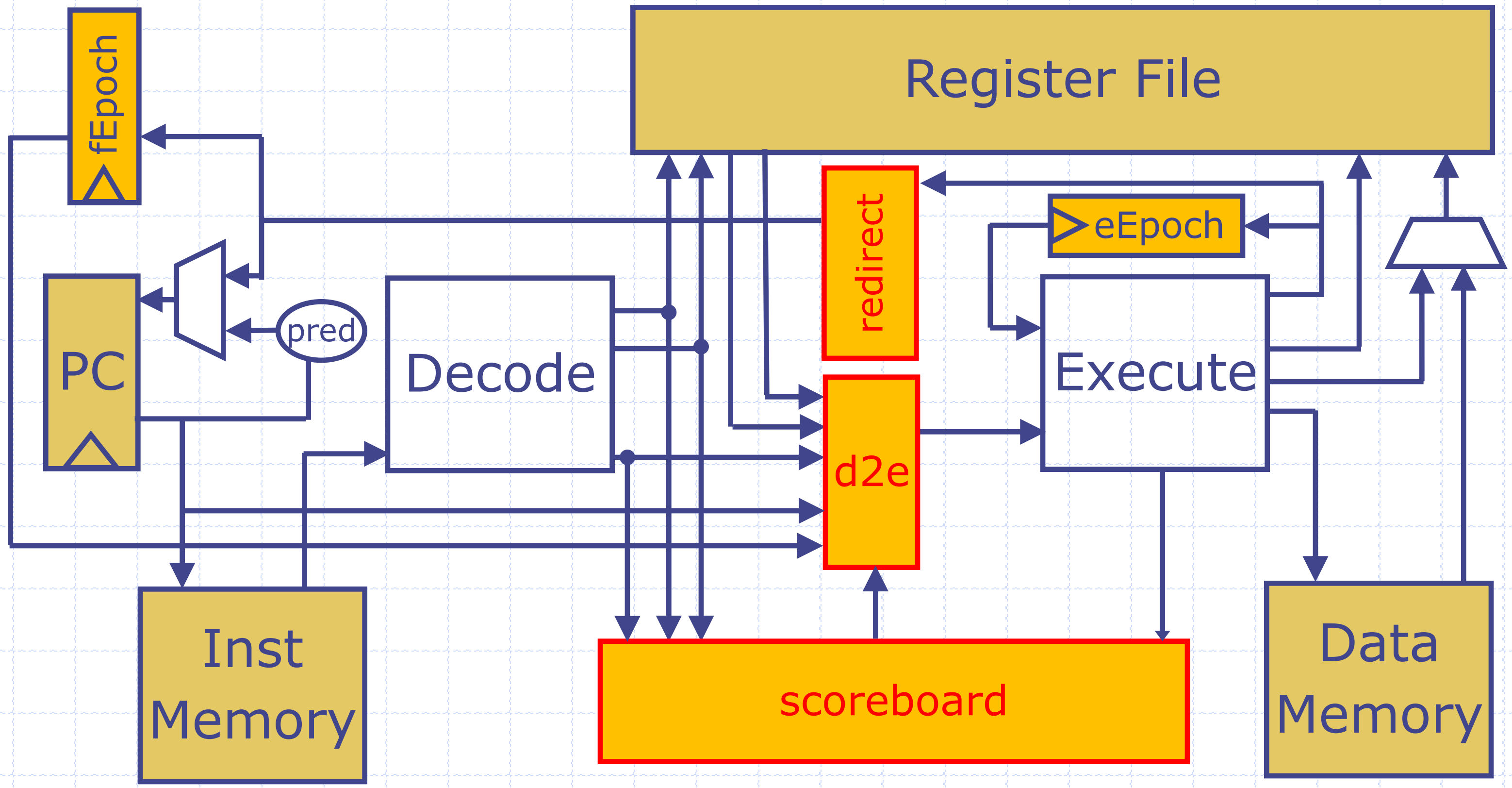
- Data hazard depends upon the match between the source registers of the fetched instruction and the destination register of an instruction already in the pipeline
- Both the source and destination registers must be Valid for a hazard to exist

```
function Bool isFound
    (Maybe#(FullIndx) x, Maybe#(FullIndx) 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 *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;
    if(execRedirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= execRedirect.first;
      execRedirect.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(validRegValue(dInst.src1));
         let rVal2 = rf.rd2(validRegValue(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*

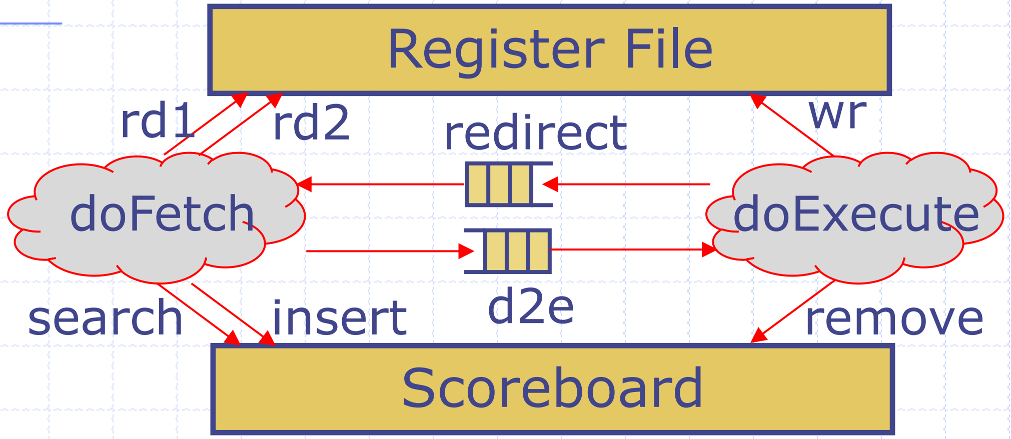
```
rule doFetch;
    if(execRedirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= execRedirect.first;
      execRedirect.deq;          end
    else
    begin
      let instF = iMem.req(pc);
      let ppcF = nextAddrPredictor(pc); ~~pc <= ppcF;~~
      let dInst = decode(instF);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
      if(!stall)                           begin
         let rVal1 = rf.rd1(validRegValue(dInst.src1));
         let rVal2 = rf.rd2(validRegValue(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 *corrected*

```
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(validRegValue(eInst.dst), eInst.data);
      if(eInst.mispredict) begin
        execRedirect.enq(eInst.addr); eEpoch <= !eEpoch; end
                                end
    d2e.deq; sb.remove;
endrule
```

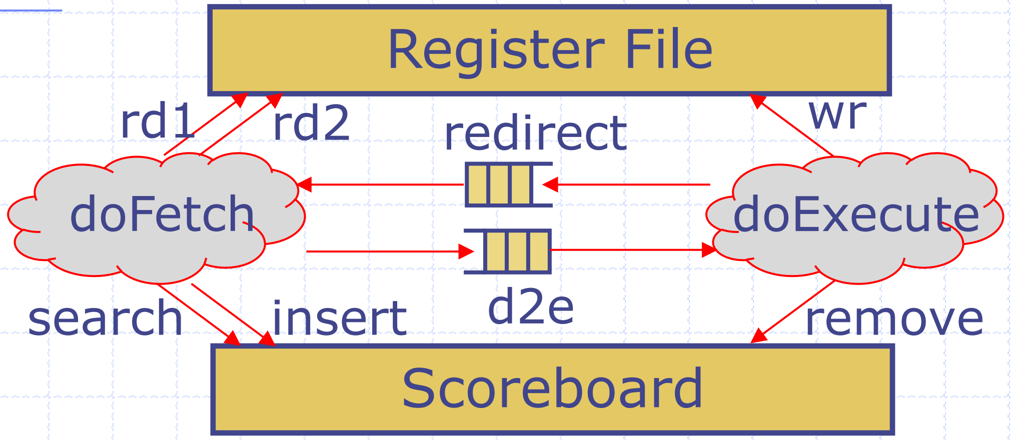
A correctness issues



- 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

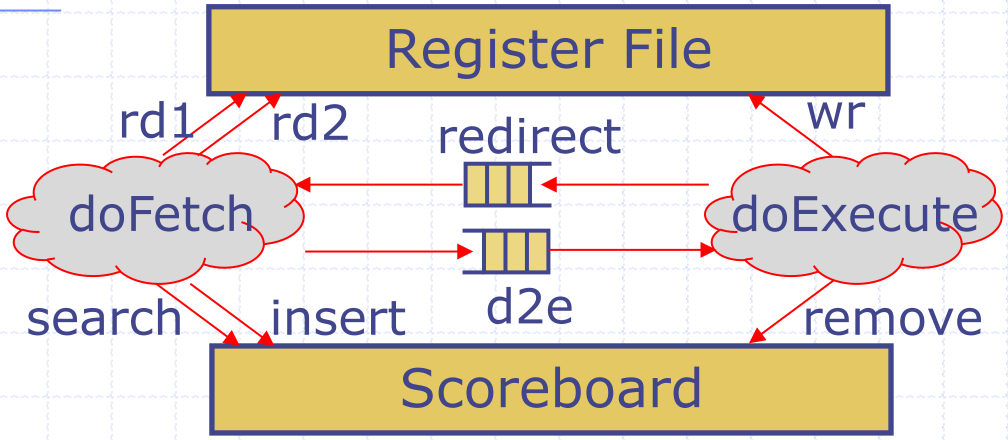
Concurrently executable Fetch and Execute



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) execRedirect <- 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 bit with every register to indicate if that register is the destination of some instruction in the pipeline
  - Appropriate register bit is set when an instruction enters the execute stage and cleared when the instruction is committed
- The design will not work if multiple instructions in the pipeline have the same destination
  - don’t let an instruction with WAW hazard enter the pipeline

Fetch rule to avoid WAW hazard

```
rule doFetch;
    if(execRedirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= execRedirect.first;
      execRedirect.deq;        end
    else
    begin
      let instF = iMem.req(pc);
      let ppcF = nextAddrPredictor(pc); let dInst = decode(instF);
      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: ppcF,
             dIinst: dInst, epoch: fEpoch,
             rVal1: rVal1, rVal2: rVal2});
        sb.insert(dInst.rDst); pc <= ppcF; end
    end
endrule
```

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 paths which can slow down the clock

next – module implementations and multistage pipelines

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
```

$\boxed{\{rd1, rd2\} < wr}$

Bypass Register File using EHR

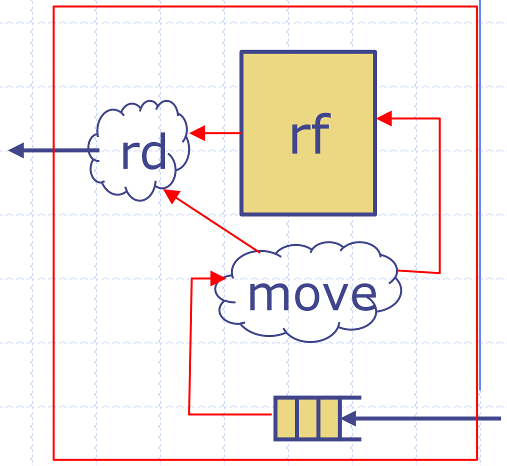
```
module mkBypassRFile(RFile);
  Vector#(32,Ehr#(2, Data)) rfile <-
                            replicateM(mkEhr(0));

  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
```

$\boxed{wr < \{rd1, rd2\}}$

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
```