Constructive Computer Architecture:

Control Hazards

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

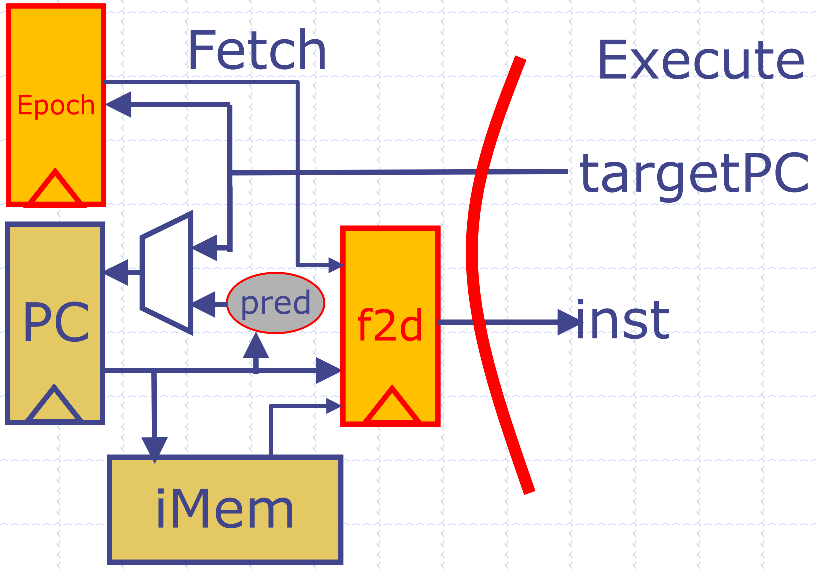
Killing fetched instructions

- In the simple design with combinational memory we have discussed so far, the mispredicted instruction was present in the f2d. So the Execute stage can atomically
  - Clear the f2d
  - Set the pc to the correct target
- In highly pipelined machines there can be multiple mispredicted and partially executed instructions in the pipeline; it will generally take more than one cycle to kill all such instructions

Need a more general solution then clearing the f2d FIFO

Epoch: a method for managing control hazards

- Add an epoch register in the processor state
- The Execute stage changes the epoch whenever the pc prediction is wrong and sets the pc to the correct value
- The Fetch stage associates the current epoch with every instruction when it is fetched
- The epoch of the instruction is examined when it is ready to execute. If the processor epoch has changed the instruction is thrown away



An epoch based solution

Can these rules execute concurrently ?

yes

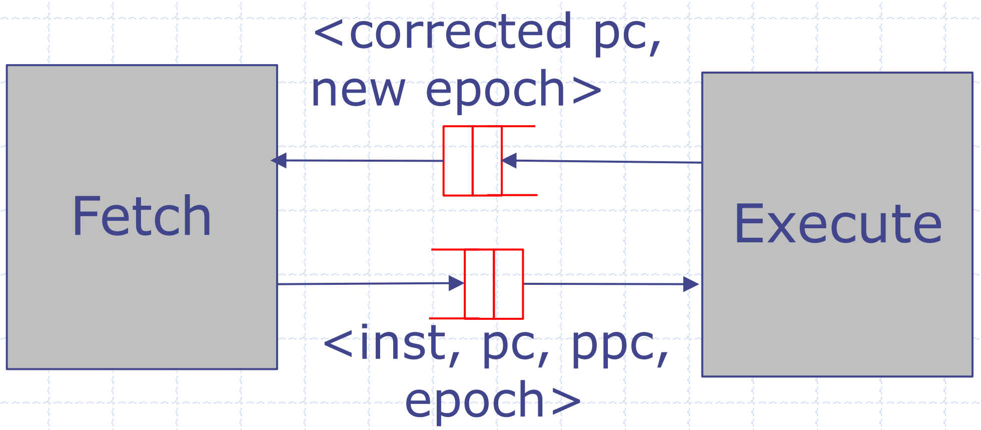
two values for epoch are sufficient

```
rule doFetch ;
  let instF=iMem.req(pc[0]);
  let ppcF=nextAddr(pc[0]); pc[0]<=ppcF;
  f2d.enq(Fetch2Decode{pc:pc[0],ppc:ppcF,epoch:epoch,
                       inst:instF});
endrule
rule doExecute;
   let x=f2d.first; let pcD=x.pc; let inEp=x.epoch;
   let ppcD = x.ppc; let instD = x.inst;
   if(inEp == epoch) begin
     let dInst = decode(instD); ... register fetch ...;
     let eInst = exec(dInst, rVal1, rVal2, pcD, ppcD);
     ...memory operation ...
     ...rf update ...
     if (eInst.mispredict)                          begin
         pc[1] <= eInst.addr; epoch <= epoch + 1; end
                          end
  f2d.deq; endrule
```

Discussion

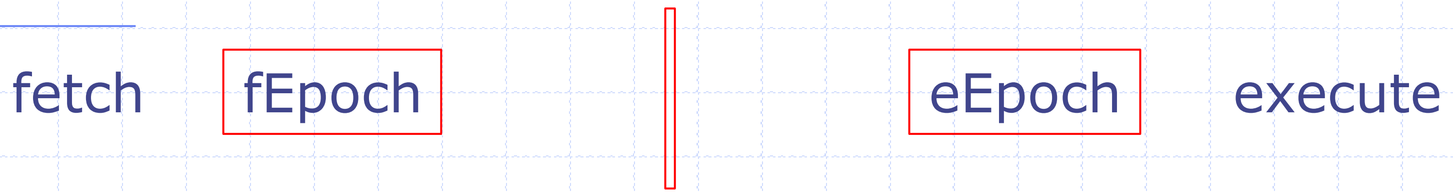
- Epoch based solution kills one wrong-path instruction at a time in the execute stage
- It may be slow, but it is more robust in more complex pipelines, if you have multiple stages between fetch and execute or if you have outstanding instruction requests to the iMem
- It requires the Execute stage to set the pc and epoch registers simultaneously which may result in a long combinational path from Execute to Fetch

Decoupled Fetch and Execute



- In decoupled systems a subsystem reads and modifies only local state atomically
  - In our solution, pc and epoch are read by both rules
- Properly decoupled systems permit greater freedom in independent refinement of subsystems

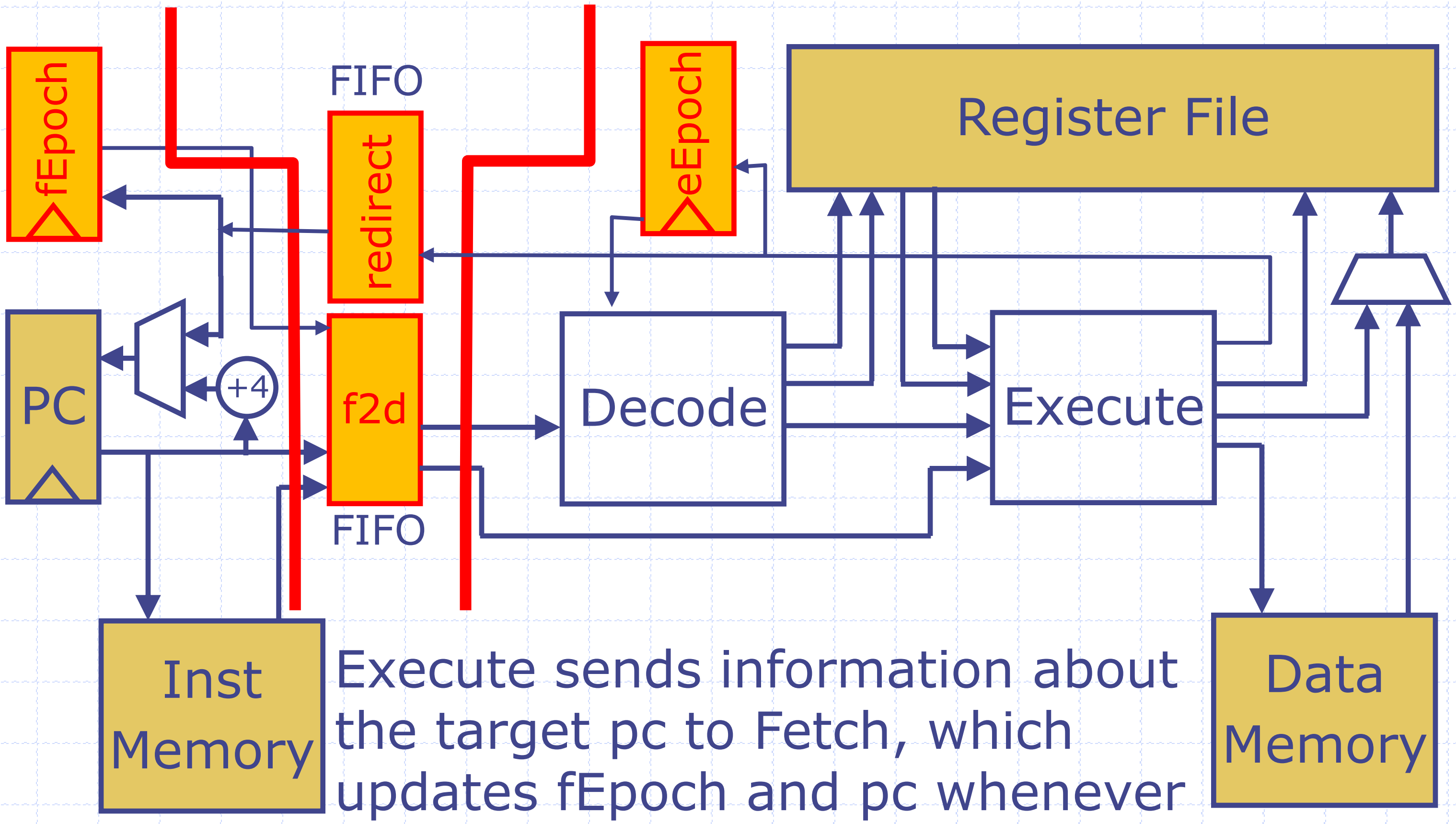
A decoupled solution using epochs



- Add fEpoch and eEpoch registers to the processor state; initialize them to the same value
- The epoch changes whenever Execute detects the pc prediction to be wrong. This change is reflected immediately in eEpoch and eventually in fEpoch via a message from Execute to Fetch
- Associate the fEpoch with every instruction when it is fetched
- In the execute stage, reject, i.e., kill, the instruction if its epoch does not match eEpoch

Control Hazard resolution

A robust two-rule solution



Execute sends information about the target pc to Fetch, which updates fEpoch and pc whenever it looks at the redirect PC fifo

Two-stage pipeline Decoupled *code structure*

```
module mkProc(Proc);
  Fifo#(Fetch2Execute) f2d <- mkFifo;
  Fifo#(Addr) execRedirect <- mkFifo;
  Reg#(Bool) fEpoch <- mkReg(False);
  Reg#(Bool) eEpoch <- mkReg(False);

  rule doFetch;
    let instF = iMem.req(pc);
    ...
    f2d.enq(... instF ..., fEpoch);
  endrule
  rule doExecute;
    if(inEp == eEpoch) begin
      Decode and execute the instruction; update state;
      In case of misprediction,  execRedirect.enq(correct pc);
                                        end
    f2d.deq;
  endrule
endmodule
```

The Fetch rule

```
rule doFetch;
 let instF = iMem.req(pc);
 if(!execRedirect.notEmpty)
    begin
      let ppcF = nextAddrPredictor(pc);
      pc <= ppcF;
      f2d.enq(Fetch2Execute{pc: pc, ppc: ppcF,
                           inst: instF, epoch: fEpoch});

    end
  else
    begin
      fEpoch <= !fEpoch;  pc <= execRedirect.first;
      execRedirect.deq;
    end
endrule
```

pass the pc and predicted pc to the execute stage

Notice: In case of PC redirection, nothing is enqueued into f2d

The Execute rule

exec returns a flag if there was a fetch misprediction

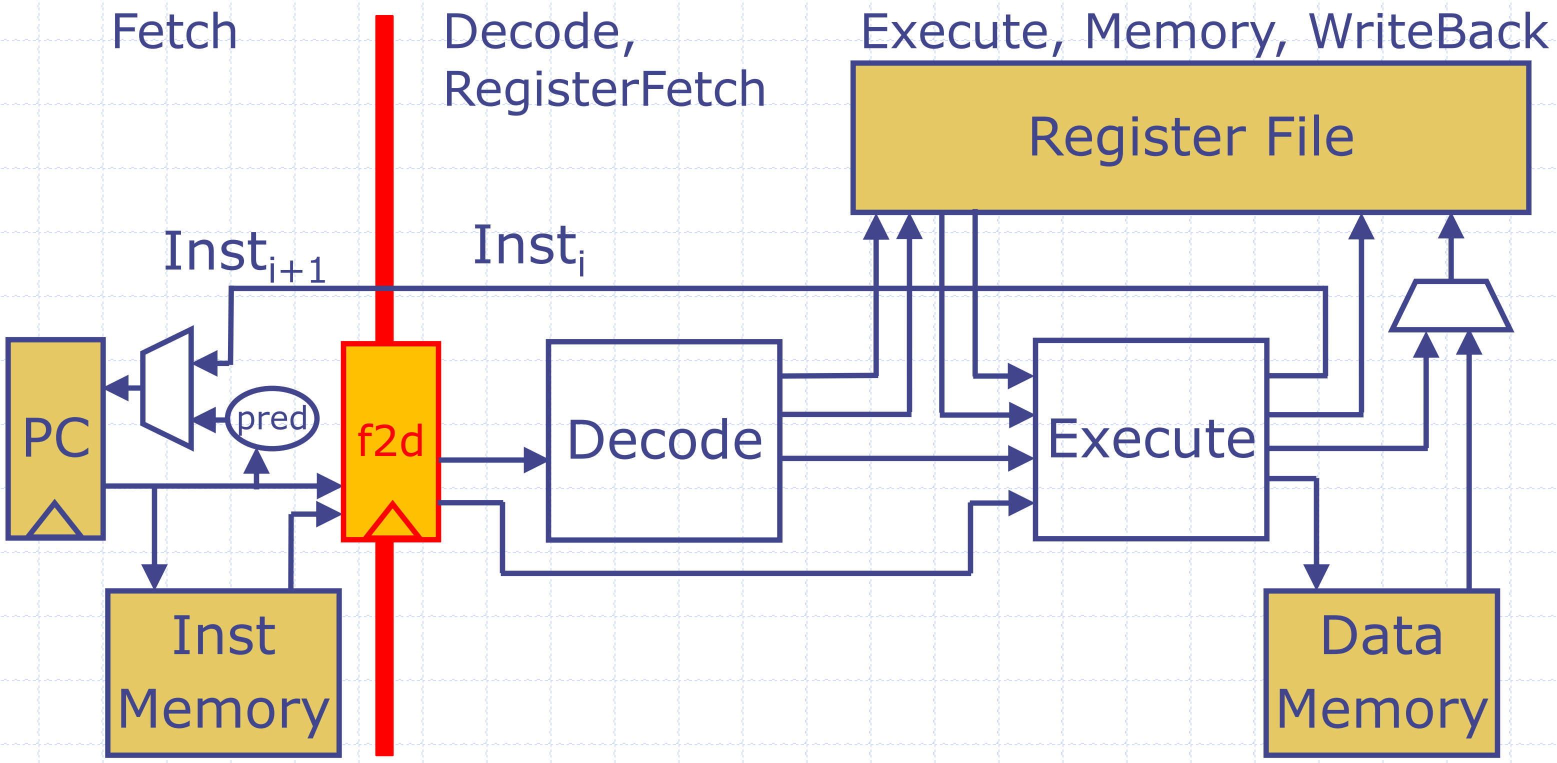
```
rule doExecute;
  let instD  = f2d.first.inst; let pcF    = f2d.first.pc;
  let ppcD   = f2d.first.ppc;  let inEp   = f2d.first.epoch;
  if(inEp == eEpoch) begin
    let dInst = decode(instD);
    let rVal1 = rf.rd1(validRegValue(dInst.src1));
    let rVal2 = rf.rd2(validRegValue(dInst.src2));
    let eInst = exec(dInst, rVal1, rVal2, pcD, ppcD);
    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 <= !inEp;
    end
  end
  f2d.deq;
endrule
```

Can these rules execute concurrently?

yes, assuming CF FIFOs

Epoch mechanism is independent of the branch prediction scheme used. We will study sophisticated branch prediction schemes later

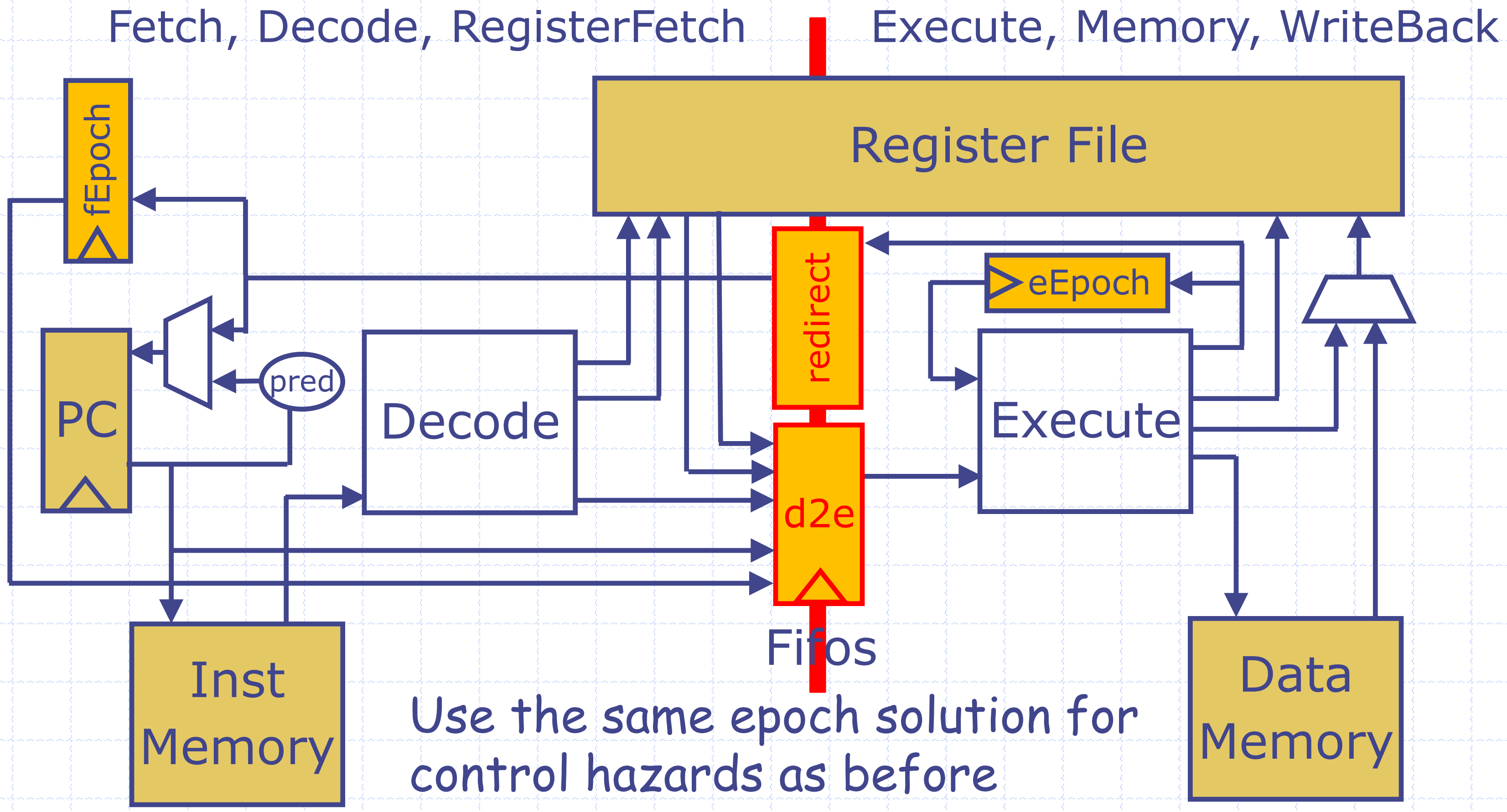
Consider a different two-stage pipeline



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

Type Decode2Execute

- The Fetch stage, in addition to fetching the instruction, also decodes the instruction and fetches the operands from the register file. It passes these operands to the Execute stage

```
typedef struct {
    Addr pc; Addr ppc; Bool epoch;
    DecodedInst dInst; Data rVal1; Data rVal2;
} Decode2Execute deriving (Bits, Eq);
```

values instead of register names

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) execRedirect <- mkFifo;

  rule doFetch …
  rule doExecute …
```

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

```
rule doFetch;
    let instF = iMem.req(pc);
    if(execRedirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= execRedirect.first;
      execRedirect.deq;         end
    else
    begin
      let ppcF = nextAddrPredictor(pc); pc <= ppcF;
      let dInst = decode(instF);
      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});
    end
endrule
```

moved from Execute

2-Stage-DH pipeline doExecute rule *first attempt*

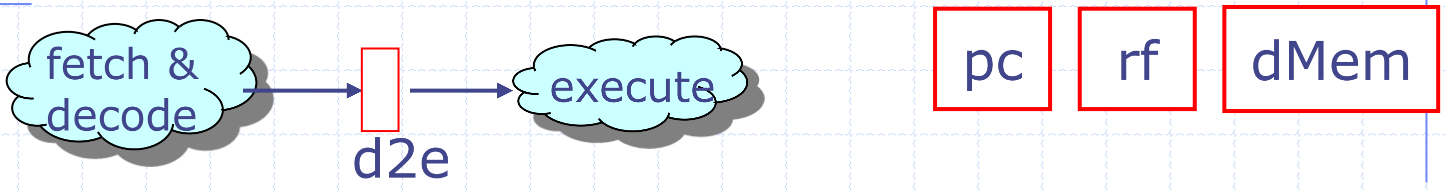
Not quite correct. Why?

Fetch is potentially reading stale values from rf

```
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) &&
        validValue(eInst.dst).regType == Normal)
        rf.wr(validRegValue(eInst.dst), eInst.data);
      if(eInst.mispredict) begin
        execRedirect.enq(eInst.addr); eEpoch <= !eEpoch; end
                              end
    d2e.deq;
endrule
```

no change

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

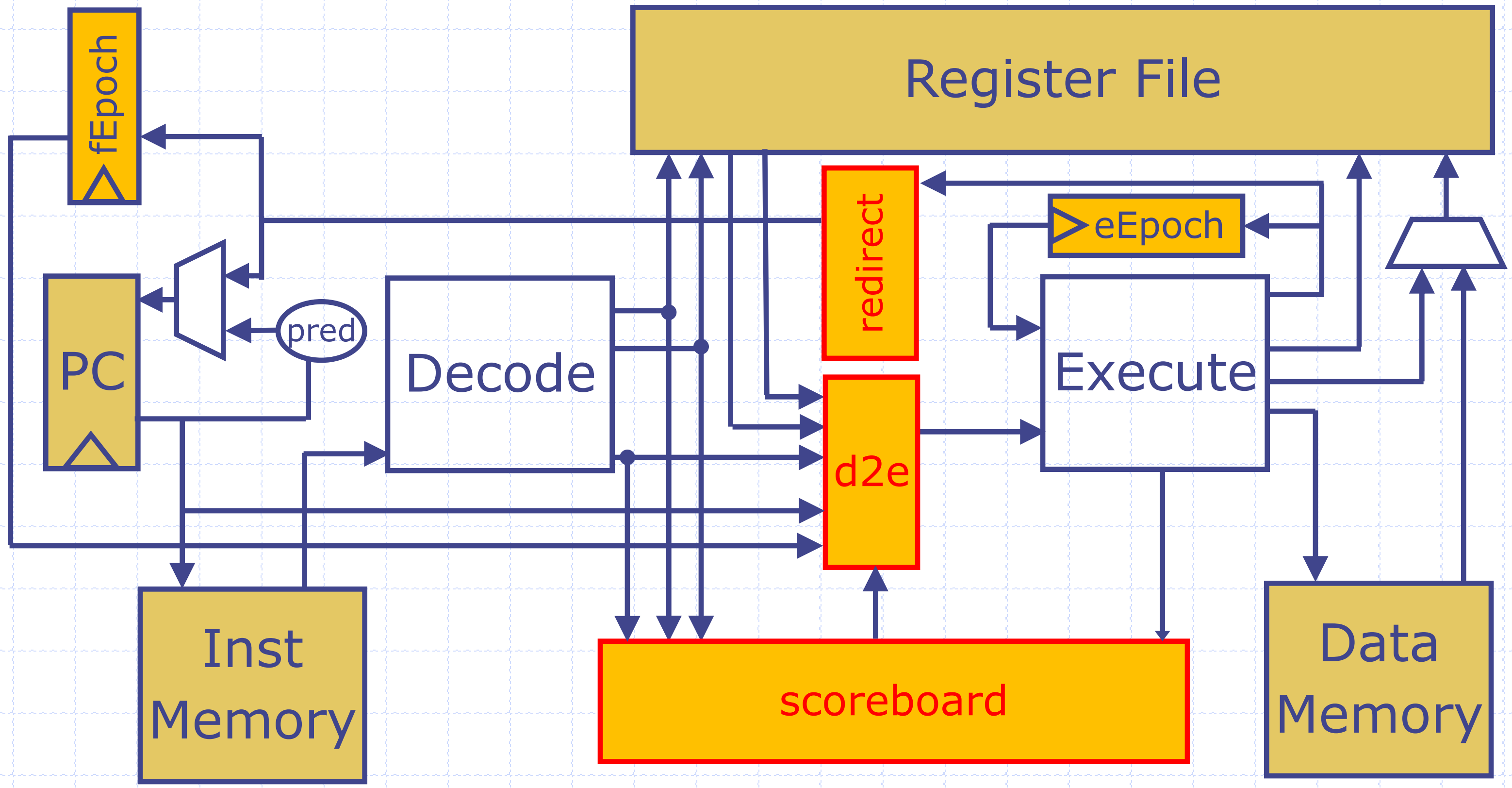
- Data hazard depends upon the match between the source registers of the fetched instruction and the destination register of an instruction already in the pipeline
- Both the source and destination registers must be Valid for a hazard to exist

```
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


fEpoch
Register File
redirect
eEpoch
PC
pred
Decode
Execute
d2e
Inst Memory
scoreboard
Data Memory