Constructive Computer Architecture:

Pipelined Processors

Thomas Bourgeat – EPFL

Slides prepared with Arvind (6.192 – Spring 23 MIT)

Ordre du jour

- Entrée: Pipelining
- Plat: Pipelining
- Dessert: Pipelining

So pipelining left and right, But first, why so much pipelining?

Las Meninas (The Maids of Honour) Diego Velázquez 1656

By some measures, the most important painting in the Western art history

Portrait of Infanta Margarita, the daughter of King Philip IV, in Royal Alcazar, Madrid

Different lighting

It is big!

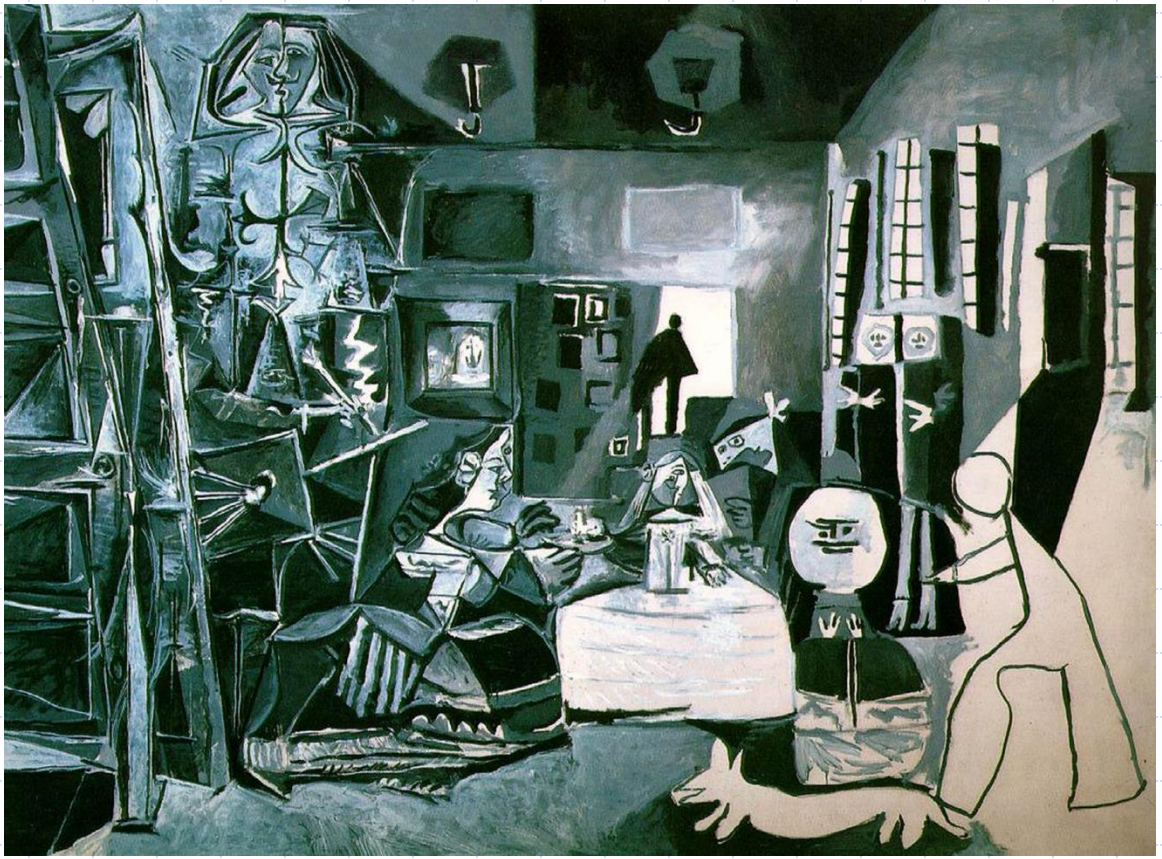
Museo del Prado, Madrid

Engages the viewer

Picasso

- Picasso had left Spain because of the Spanish civil war in nineteen thirties and had never seen Las Meninas
- In 1956, at the 300th anniversary of Diego Velázquez's Las Meninas, Picasso revisited Madrid to see the painting
- The story goes he came back and locked himself in his studio for three months and painted 58 versions of it – deconstructing and constructing – not copying
  - Can be seen at Museo Picasso in Barcelona

Deconstructing & Constructing: Las Meninas



Infanta Margarita

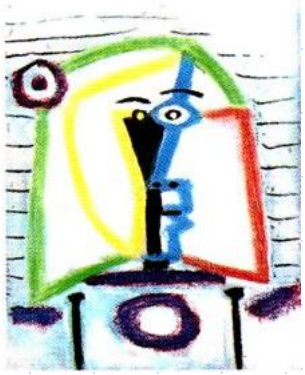
Perplexed? Distracted by sun light?

Deconstructing & Constructing:
Las Meninas – Infanta Margarita









Deconstructing & Constructing: Las Meninas

Deconstructing & Constructing: Las Meninas

Deconstructing & Constructing: Las Meninas

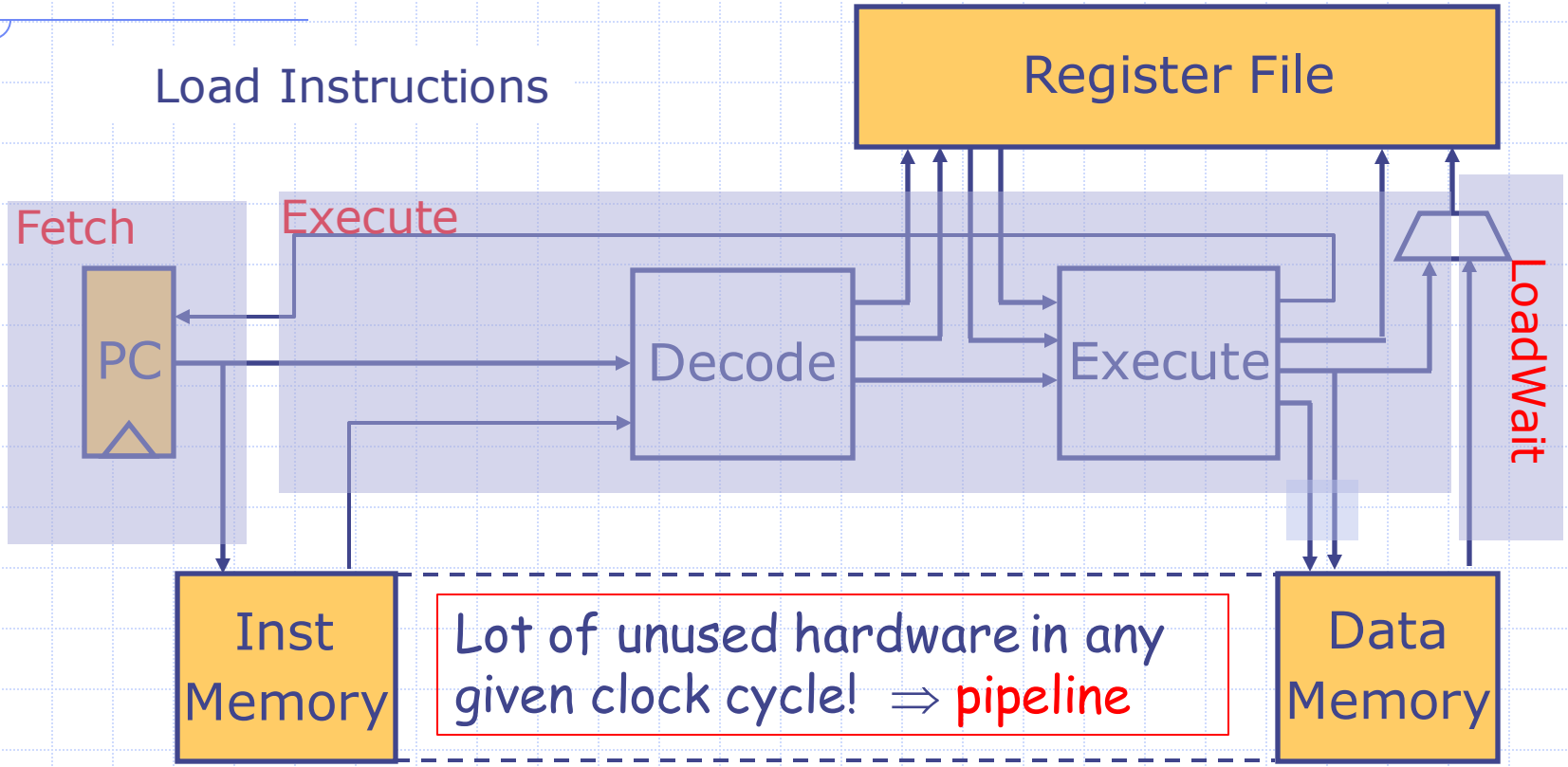
Why?

- Picasso was 75 and very aware of his Spanish heritage. Was he trying to improve upon the master's work?
- Picasso reportedly said:

  "I would say..what if you put them a little more to the right or left? I'll try to do it my way, forgetting about Velazquez. The test would surely bring me to modify or change the light because of having changed the position of a character. So, little by little, that would be a detestable Meninas for a traditional painter, but would be my Meninas."

Let’s build detestable pipelined processors!

Multicycle Processor: *Analysis*



- Assuming 20% load instructions, and memory latency of one, the average number of cycles per instruction:
  - 2 x .8 + 3 x .2 = 2.2

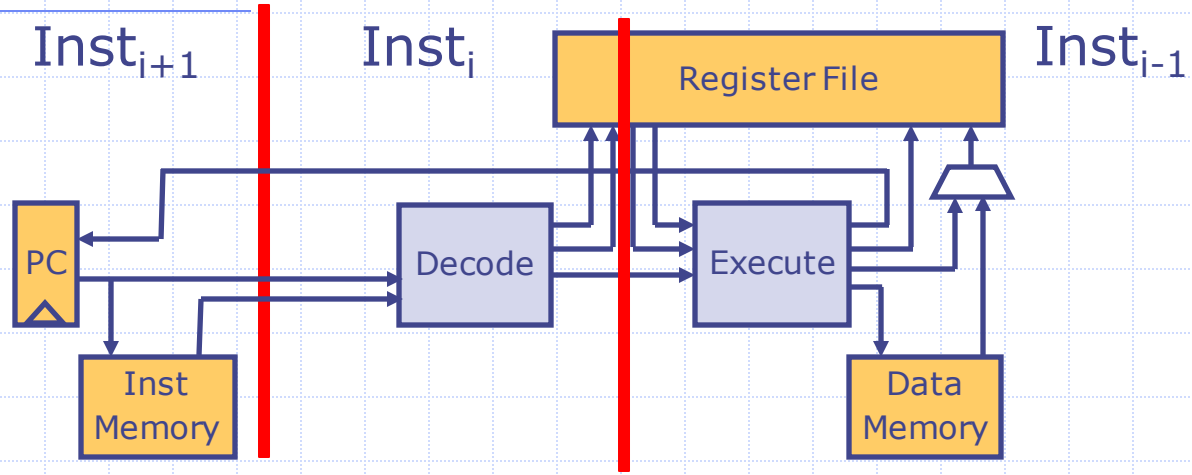
higher memory latency will make this number much worse

Processor pipelines

- Pipelining a processor encompasses many core challenges of computer architecture
  - Stringent correctness requirements
  - Requires speculative execution of instructions to pipeline at all!
  - Requires dealing with a variety of feedback in the pipeline

10. There are simple pipelined cores, there are also tremendously sophisticated pipelined cores

New problems in pipelining instructions (over arithmetic pipelines)



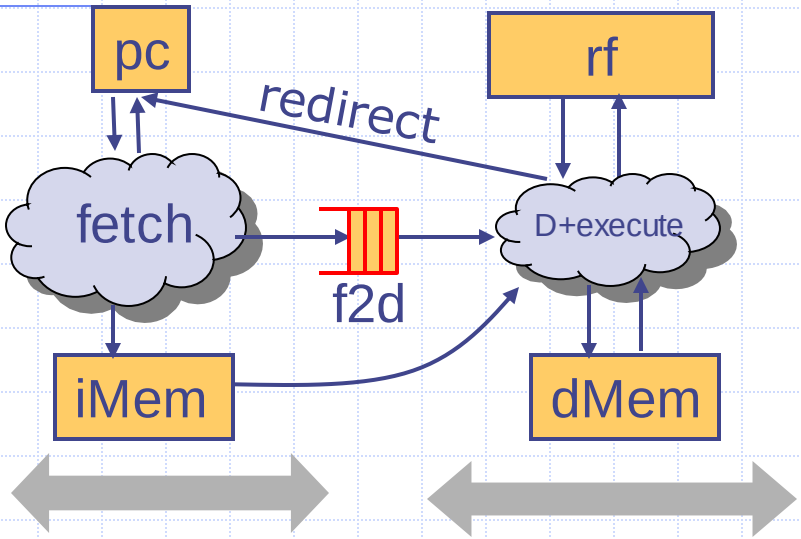
- *Control hazard:* pc for Inst$_{i+1}$ is not known until at least Inst$_i$ is decoded. So which instruction should be fetched?
  - Solution: *Speculate and squash later* if the prediction is wrong
- *Data hazard:* Inst$_i$ may be data dependent on Inst$_{i-1}$, and thus, it must wait for the effect of Inst$_{i-1}$ on the state of the machine (pc, rf, dMem) to take place
  - Solution: *Stall* instruction Inst$_i$ until the dependency is resolved
  - Number of stalls can be reduced by *bypassing,* that is by providing additional datapaths

Plan

1. Develop a two-stage pipeline by providing a solution for *control hazards*
2. Develop a two-stage pipeline by providing a solution for *data-hazard*
3. Develop a three-stage pipeline by providing a solution for *data hazards and control hazards*

To keep the explanations simple, we will first show the solutions with magic memory and then discuss pipelining multicycle processors

Control hazard

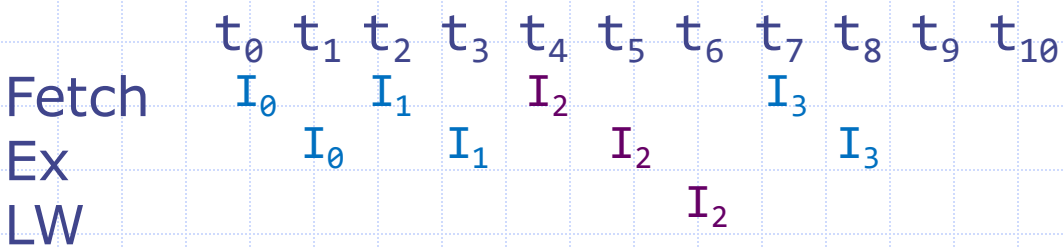


We will offer a solution that is independent of how many cycles each stage takes

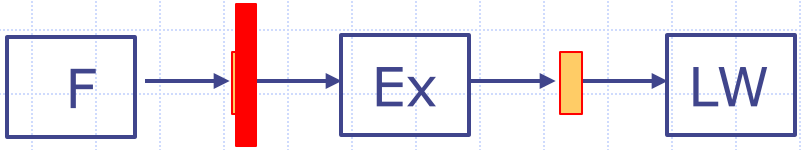
- Fetch stage initiates instruction fetch and sends the pc to Execute stage via f2d. It speculatively updates pc to pc+4
- Execute stage picks up instruction from f2d and executes it. It may take one or more cycles to do this
- These two stages operate independently except in case of a branch misprediction when Execute redirects the pc to the correct pc. All "wrong path" instructions have to be squashed

Timing diagrams and bubbles

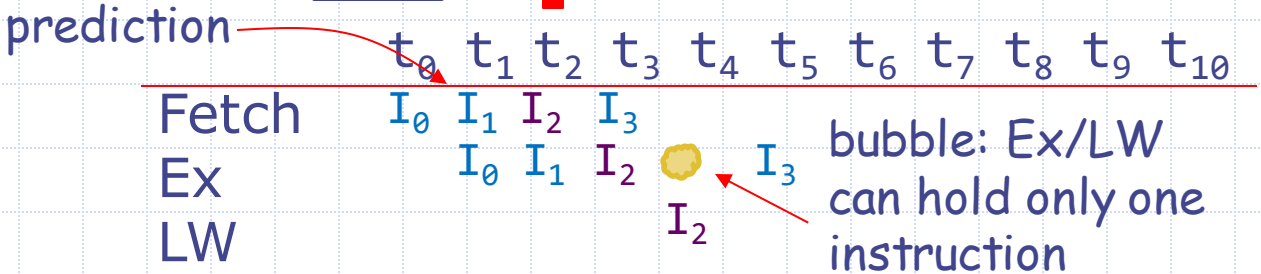
Multicycle Processor



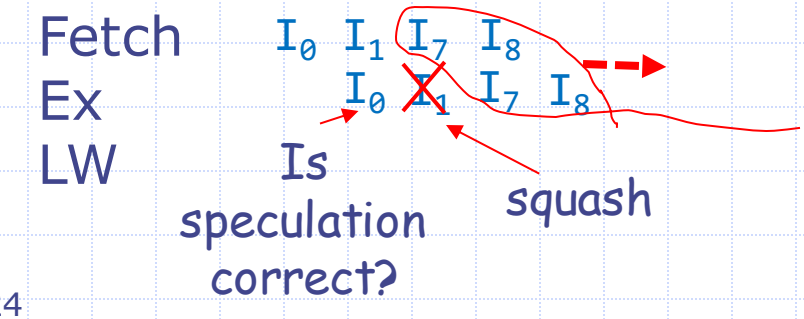
Execution of I_0; I_1; I_2; I_3; … only I_2 is a load instruction



Two-stage Pipeline



Suppose I_0 is a branch instruction which jumps to I_7 instead of I_1



How do we detect a misprediction?

- We initiate a fetch for the instruction at pc, and make a prediction for the next pc (ppc)
- The instruction at pc carries the prediction (ppc) with it as it flows through the pipeline
- At the Execute stage we know the real next pc. It is a misprediction if the next pc ≠ ppc

What does it mean to squash a partially executed instruction?

- The instruction should have no effect on the processor state
  - must not update register file or pc
  - must not launch a Store
- These conditions are easy to ensure in our two-stage processor because there is at most one instruction in the Ex/LW state

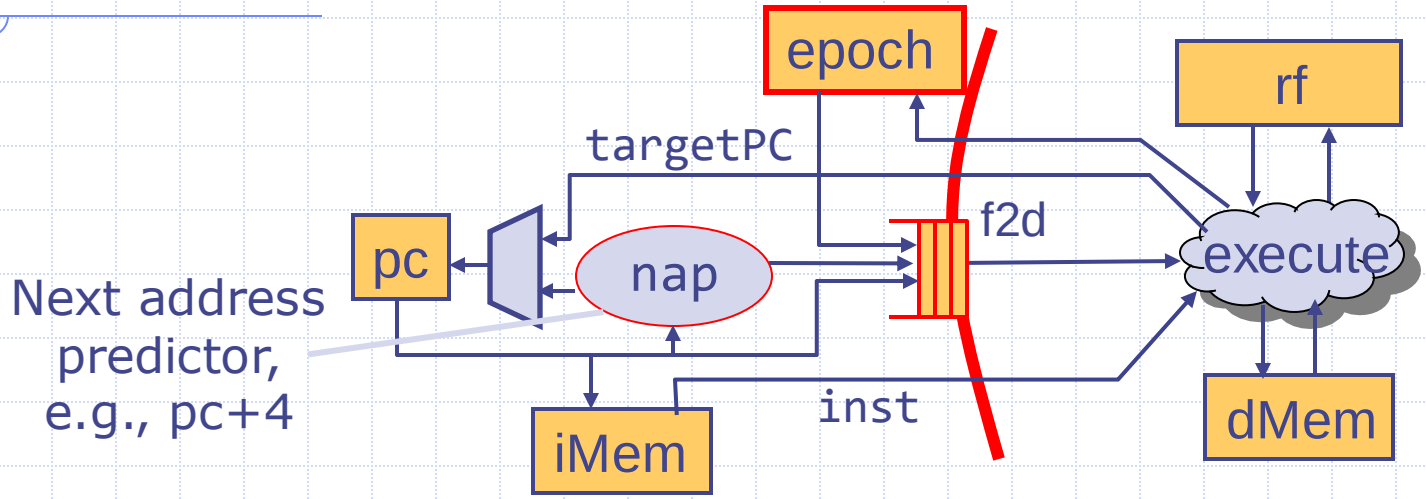
Killing fetched instructions

- In a simple 2-stage design, all the mispredicted instructions were present in f2d. So, the Execute stage can *atomically:*
  - Clear f2d
  - Set pc to the correct target
  - Hmmm, actually, can it clear f2d?

- In highly pipelined machines there can be multiple mispredicted and partially executed instructions in the pipeline; it generally takes more than one cycle to kill all such instructions

Need a more general solution then clearing the f2d FIFO

Epoch: a method to manage control hazards



- Add an *epoch* register to 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* to every instruction sent to the Execute stage
- 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

```
rule doFetch ;
  let instF=iMem.req(pcF);
  let ppcF=nap(pcF); pcF<=ppcF;
  f2d.enq(Fetch2Decode{pc:pcF,ppc:ppcF,epoch:epoch,
                       inst:instF});
endrule
rule doExecute;
   let x=f2d.first; let pc=x.pc; let inEp=x.epoch;
   let inst = x.inst;
   if(inEp == epoch) begin
     ...decode, register fetch, exec, memory op,
        rf update nextPC ...
   if (x.ppc != nextPC) begin pcF <= eInst.addr;
                              epoch <= next(epoch); end
                       end
   f2d.deq; endrule
```

Can these rules execute concurrently ?

How many epoch values are sufficient?

An epoch-based solution
For concurrency, turn pcF in an EHR

```
rule doFetch ;
  let instF=iMem.req(pcF[0]);
  let ppcF=nap(pcF[0]); pcF[0]<=ppcF;
  f2d.enq(Fetch2Decode{pc:pcF[0],ppc:ppcF,epoch:epoch,
                       inst:instF});
endrule
rule doExecute;
   let x=f2d.first; let pc=x.pc; let inEp=x.epoch;
   let inst = x.inst;
   if(inEp == epoch) begin
     ...decode, register fetch, exec, memory op,
        rf update nextPC ...
   if (x.ppc != nextPC) begin pcF[1] <= eInst.addr;
                              epoch <= next(epoch); end
                       end
  f2d.deq; endrule
```

two values for epoch are sufficient!

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

Epoch mechanism is independent of the sophisticated branch prediction schemes that we will study later

4-Cycle Harvard Processor

into a 2-stage pipeline

```
module mkProcHarvard3cycle(Empty);
    Code to instantiate pc, rf, mem, and registers that hold
    the state of a partially executed instruction
    rule doFetch if (state == Fetch);
       Code to initiate instruction fetch; hold pc in a reg;
       go to Decode
    rule doDecode if (state == Decode);
       let inst <- mem.resp;
       Code to decode and read the operands from rf; hold
       partially executed inst in a reg; go to execute
    rule doExecute if (state == Execute);
       Code to execute all instructions and initiate dMem request;
       hold the (partial) results in a reg; go to LoadWait
    rule doLoadWait if (state == LoadWait);
    (if Load then wait for the load value), update rf and pc,
    go to Fetch
endmodule
```

$Inst_{i+1}$

$Inst_i$

4-Cycle Harvard Processor

into a 2-stage pipeline

```
module mkProcHarvard3cycle(Empty);
    Code to instantiate pc, rf, mem, and registers that hold
    the state of a partially executed instruction
    rule doFetch if (state == Fetch);
      Initiate iMem req; let ppcF=nap(pcF); pcF<=ppcF;
       f2d.enq(Fetch2Decode{pc:pcF,ppc:ppcF,epoch:epoch});
    endrule
    rule doDecode if (state == Decode);
      let x=f2d.first; let pc=x.pc; let inEp=x.epoch;
      let inst <- iMem.res(); f2d.deq;
      if(inEp == epoch) // correct path instruction
        then ...decode, register fetch, set d2e, go to Execute
        else go to Decode

    rule doExecute if (state == Execute); …
    rule doLoadWait if (state == LoadWait); …
endmodule
```

4-Cycle Harvard Processor

into a 2-stage pipeline

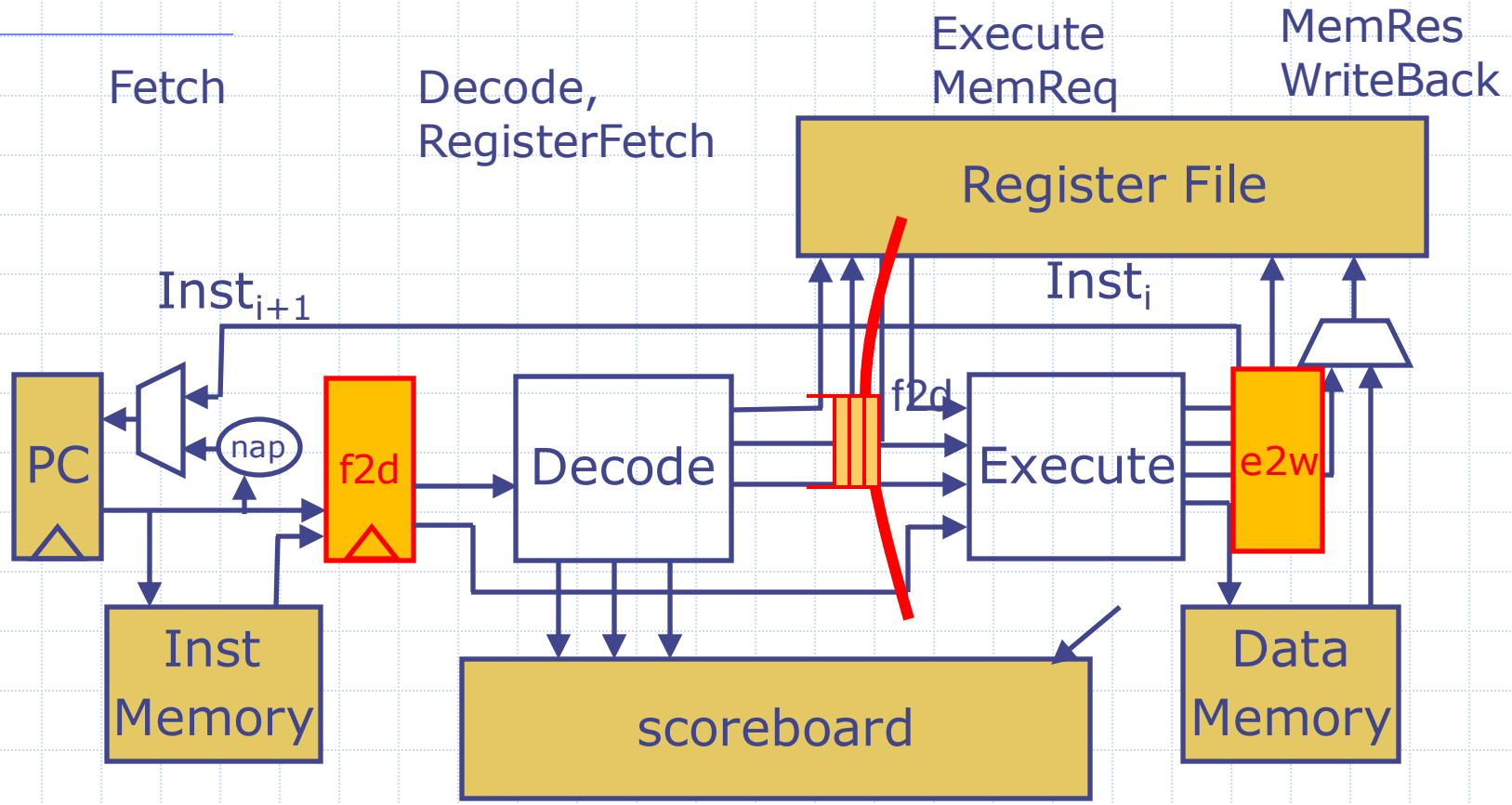
```
rule doDecode if (state == Decode); …
rule doExecute if (state == Execute);
   Extract fields from d2e, execute and compute nextPC
   if prediction is correct
   then initiate memory op, set e2w, goto LoadWait
   else update pcF; epoch <= next(epoch); goto Decode

rule LoadWait if (state == LoadWait);
   Extract fields from e2w;
   If necessary, wait for dMem response;
   update rf; goto Decode
```

Can doDecode, doExecute and LoadWait rules fire concurrently?

Can any of these rules fire concurrently with doFetch?

A 4-cycle, 2-stage pipeline – detestable machine



4-Cycle Harvard Processor

into a 2-stage pipeline

```
module mkProcHarvard2stagePipeline(Empty);
    instantiate pc, rf, mem, and registers that hold
    the state of a partially executed instruction
    rule doFetch if (state == Fetch);
      initiate instruction fetch; hold pc in a reg; go to Decode
    rule doDecode if (state == Decode);
       let inst <- mem.resp;
       decode inst and read operands from rf;
       hold partially executed inst in a reg; go to Execute
    rule doExecute if (state == Execute);
       Code to execute all instructions and initiate dMem request;
       hold the (partial) results in a reg; go to WB
    rule doWB if (state == WB);
    (if Load then wait for the load value), update rf and pc,
    go to Fetch
endmodule
```

$Inst_{i+1}$

$Inst_i$

4-Cycle Harvard Processor

into a 2-stage pipeline 1 (correct?)

```
module mkProcHarvard3cycle(Empty);
    instantiate pc, rf, mem, instantiate d2e, epoch and regs ..;
    rule doFetch if (stateFD == Fetch);
       Initiate iMem req(pc); ppc=nap(pc); pc<=ppc;
       f2d <= Fetch2Decode{pc:pc, ppc:ppc, epoch:epoch}
       stateFD <= Decode
    rule doDecode if (stateFD == Decode);
       let inst <- mem.resp; decode inst; read operands from rf;
       d2e.enq(Decode2Execute{pc:pc, ppc:ppc, epoch:epoch
       v1:rvalv1, v2:rval2}); stateFD <= Fetch
    rule doExecute if (stateEW == Execute);
       execute all instructions; if mem inst, initiate dMem req;
       hold the (partial) results in a reg; go to WB
    rule doWB if (stateEW == WB);
    (if Load then wait for the load value), update rf,
    (if needed, redirect pc;) go to Execute; endmodule
```

$Inst_{i+1}$

$Inst_i$

4-Cycle Harvard Processor

into a 2-stage pipeline 2a (correct?)

```
module mkProcHarvard3cycle(Empty);
    instantiate pc, rf, mem, instantiate d2e, registers…;
    rule doFetch if (stateFD == Fetch);
        Initiate inst fetch (pc); ppc=nap(pc); pc<=ppc;
        f2d <= Fetch2Decode{pc:pc, ppc:ppc, epoch:epoch}
        stateFD <= Decode
    rule doDecode if (stateFD == Decode);
        let inst <- mem.resp; // suppose the epoch has changed?
        let x=f2d; let pc=x.pc; let inEp=x.epoch;
        if(inEp == epoch) // correct path instruction
        then decode inst; read operands from rf;
          d2e.enq(Decode2Execute{pc:pc, ppc:ppc, epoch:epoch
          v1:rvalv1, v2:rval2});
        else  // wrong path instruction and do nothing
          stateFD <= Fetch
```

$Inst_{i+1}$

$Inst_i$

4-Cycle Harvard Processor

into a 2-stage pipeline 2b (correct?)

```
rule doFetch if (stateFD == Fetch); …
rule doDecode if (stateFD == Decode); …
rule doExecute if (stateEW == Execute);
   Extract fields from d2e, execute and compute nextPC
   if prediction is correct
   then
        initiate memory op; nextEp = epoch
        set e2w; stateEW <= WB;
   else correct pc; nextEp = next(epoch);
rule doWB if (stateEW == WB);
   Extract fields from e2w;
   If necessary, wait for dMem response;
   update rf; update pc; update epoch
```

Are these rules correct?

Data Hazards!

4-Cycle Harvard Processor

into a 2-stage pipeline 2a (correct?)

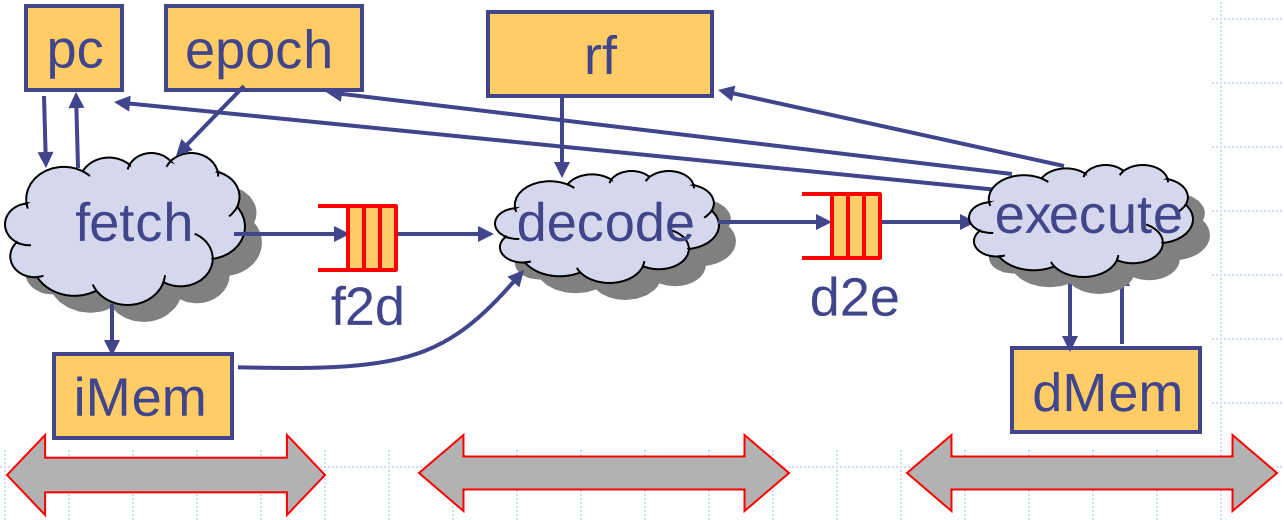
```
        rule doFetch if (stateFD == Fetch);
          Initiate inst fetch (pc); ppc=nap(pc); pc<=ppc;
          f2d <= Fetch2Decode{pc:pc, ppc:ppc, epoch:epoch}
          stateFD <= Decode
Inst_i+1  rule doDecode if (stateFD == Decode);
          let inst <- mem.resp;
          let x=f2d; let pc=x.pc; let inEp=x.epoch;
          if(inEp == epoch) // correct path instruction
          then decode inst; read operands from rf;
               d2e.enq(Decode2Execute{pc:pc, ppc:ppc,
epoch:epoch
               v1:rvalv1, v2:rval2});
          else  // wrong path instruction and do nothing
          stateFD <= Fetch
Inst_i    rule doExecute if (stateEW == Execute); …
          rule doWB if (stateEW == WB); …
```

If data-hazard then stall

Data Hazards

An instruction read a register that is updated by another recent instruction

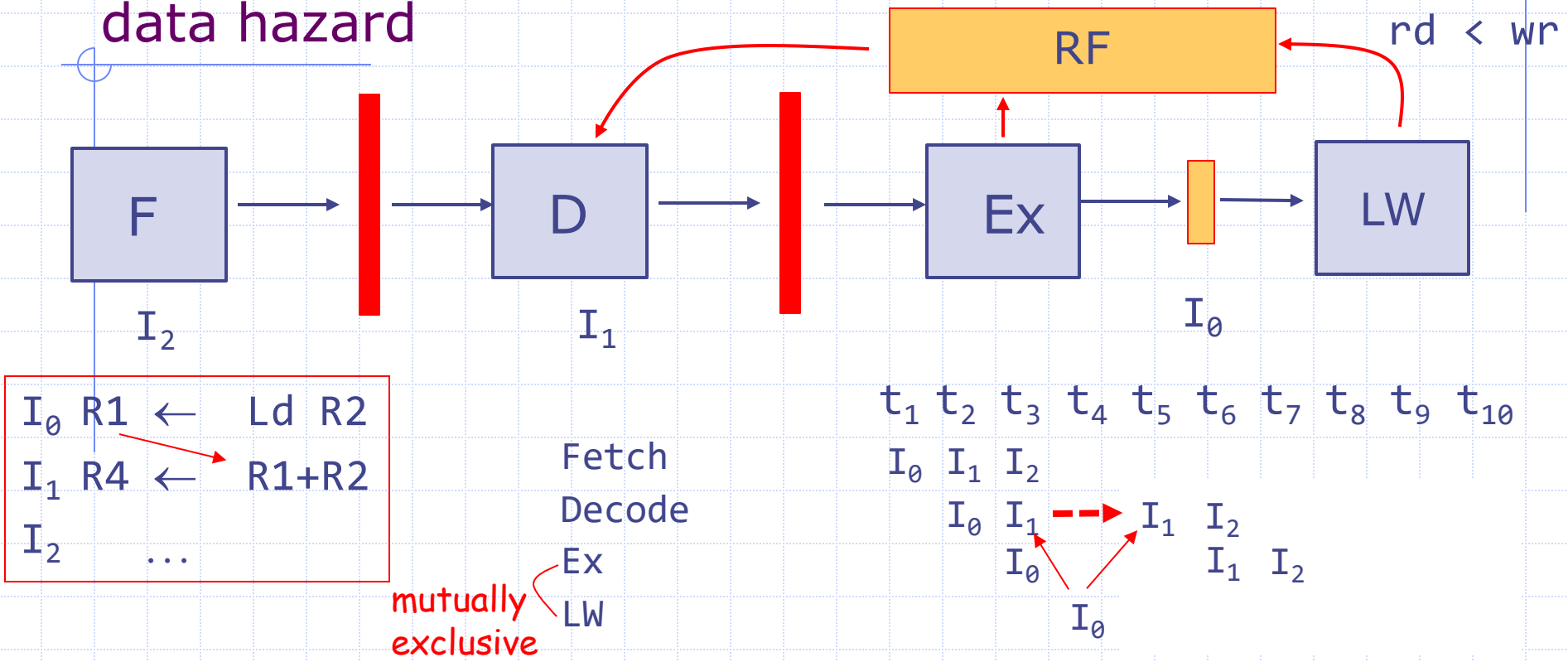
Pipelining Decode and Execute



- Execute step probably has the longest propagation delay (decode + register-file read + execute)
- Separate Execute into two stages:
  - Decode and register-file-read
  - Execute – including the initiation of memory instructions
- This introduces a new problem known as a *Data Hazard*, that is, the register file, when it is read, may have stale values

Three stage pipeline

data hazard



- I_1 must be stalled until I_0 updates the register file, i.e., the data hazard disappears ⇒ need a mechanism to stall
- The data hazard will disappear as pipeline drains

Complication: the stalled instruction may be a wrong-path instruction

Data Hazard

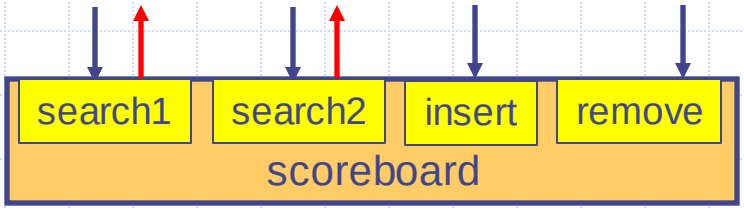
- Data hazard arises when a source register of the fetched instruction matches the destination register of an instruction already in the pipeline
- Both the source and destination registers must be valid for a hazard to exist

Dealing with data hazards
(aka read-after-write (RAW) hazard)

- Introduce a *Scoreboard* -- a data structure to keep track of the destinations of the instructions in the pipeline beyond the Decode stage
  - Initially the scoreboard is empty
- Compare sources of an instruction when it is decoded with the destinations in the scoreboard
- Stall the Decode from dispatching the instruction to Execute if there is a RAW hazard
- When the instruction is dispatched, enter its destination in the scoreboard
- When an instruction completes, delete its source from the scoreboard

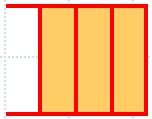
A stalled instruction will be unstalled when the RAW hazard disappears. This is guaranteed to happen as the pipeline drains.

Scoreboard



- *method insert(dst):* inserts the destination of an instruction or Invalid in the scoreboard when the instruction is decoded
- *method search1(src):* searches the scoreboard for a data hazard, i.e., a dst that matches src
- *method search2(src):* same as *search1*
- *method remove:* deletes the oldest entry when an instruction commits

Two designs for scoreboard



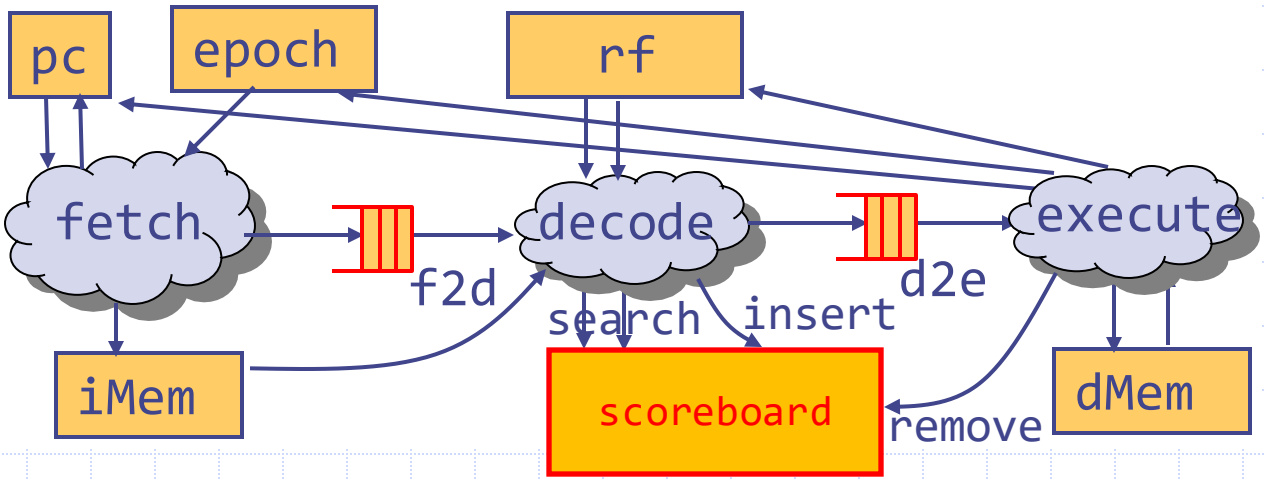
versus

- A fifo of depth equal to the number of pipeline stages in Execute
- *Insert:* enq (dst)
- *Remove:* deq
- *Search:* compare source against each entry

- One Boolean flag for each register (Initially all False)
- *Insert:* set the flag for register rd to True (block if it is already True)
- *Remove:* set the flag for register rd to False
- *Search:* Return the value of the flag for the source register

Counter design takes less hardware, especially for deep pipelines, and is more efficient because it avoids searching each element of the fifo

Scoreboard in the pipeline



- If search by Decode does not see an instruction in scoreboard, then that instruction must have updated the state
- Thus, when an instruction is removed from the scoreboard, its updates to Register File must be visible to the subsequent register reads in Decode
  - `remove` and `wr` should happen simultaneously
  - `search`, and `rd1`, `rd2` should happen simultaneously

This will require a bypass register file

WAW hazards

- Can a destination register name appear more than once in the scoreboard ?
- If multiple instructions in the pipeline 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; two solutions
  - avoid WAW hazard by stalling the decode if the destination is already present in sb
  - Use a more complex sb and make sure a destination stays in sb as long as necessary

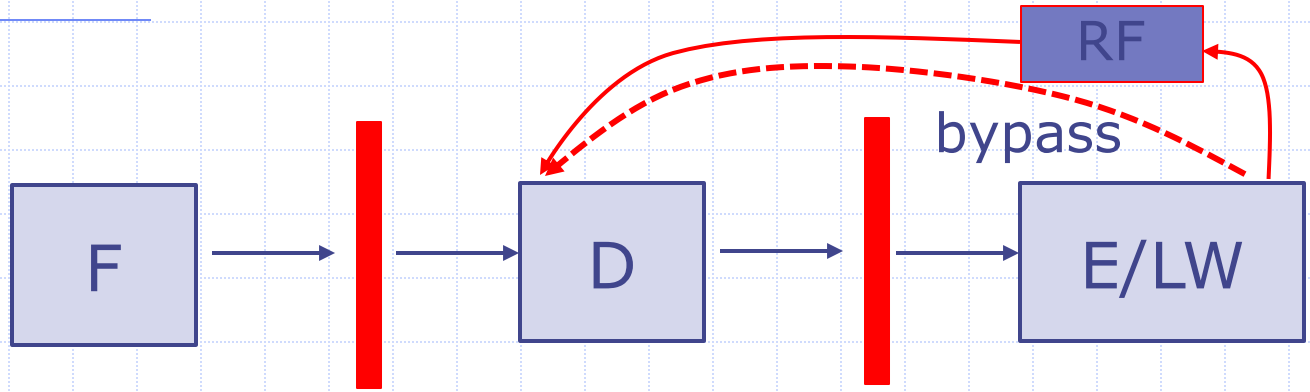
Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

CPI t_{Clk}

- Pipelining lowers t_{Clk}. What about CPI?
- $CPI = CPI_{ideal} + CPI_{hazard}$
  - CPI_{ideal}: cycles per instruction if no stall
- CPI_{hazard} contributors
  - Data hazards: long operations, cache misses
  - Control hazards: branches, jumps, exceptions

Bypassing



- Bypassing is a technique to reduce the number of stalls (that is, the number of cycles) by providing extra data paths between the producer of a value and its consumer
- Bypassing introduces new combinational paths, and this can increase combinational delay (and hence the clock period) and area
- The effectiveness of a bypass is determined by how often it is used

Normal vs Bypass 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

Can we design a bypass register file so that:

wr < {rd1, rd2}

Bypass Register File using EHR

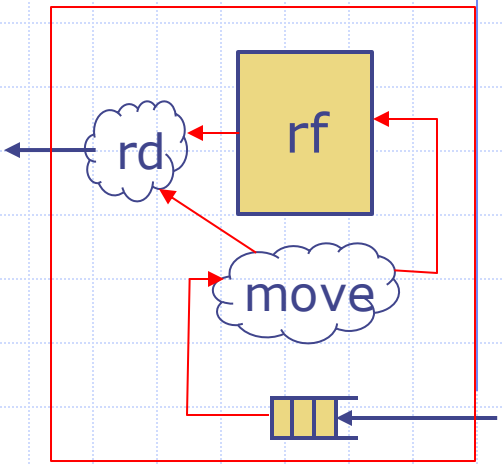
```
module mkBypassRFile(RFile);
  Vector#(32,Ehr#(2, Data)) rfile <- replicateM(mkEhr(0));

  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

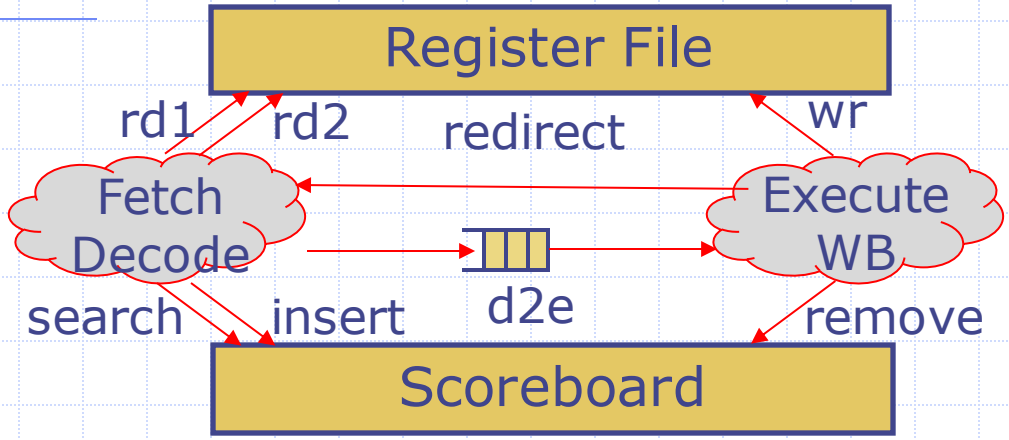
with external bypassing




```
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
```

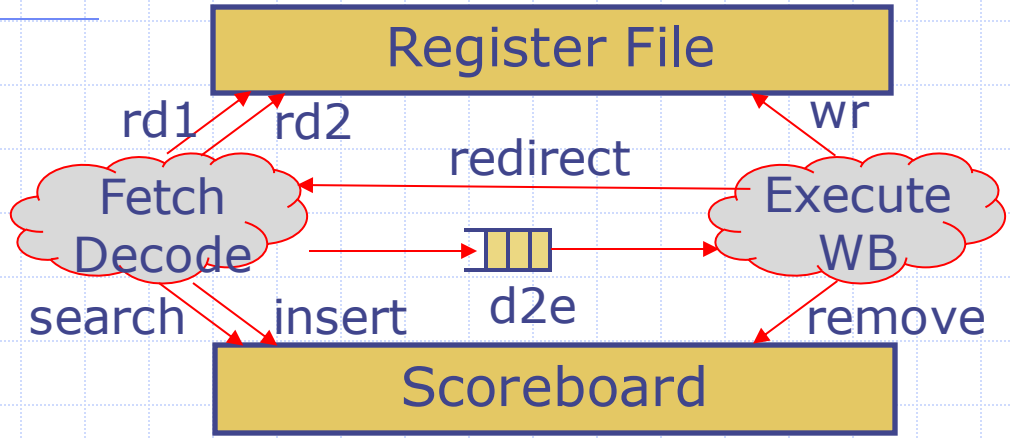
wr < {rd1, rd2}

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

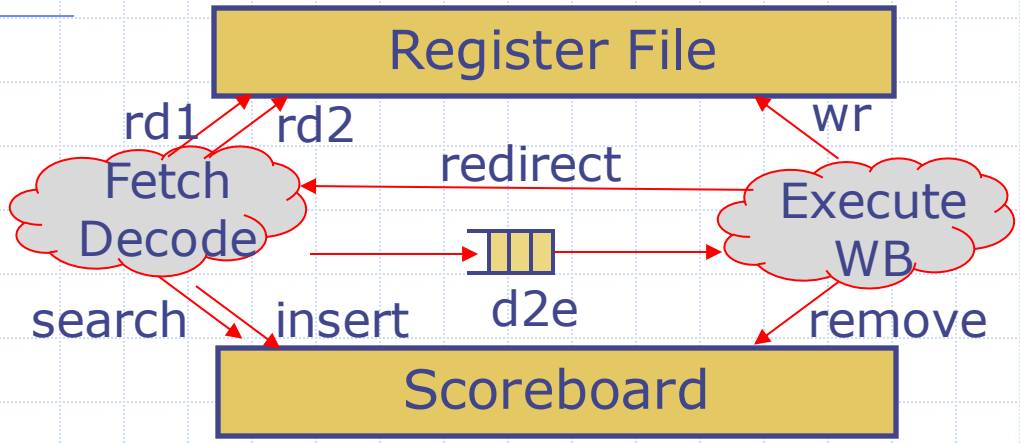
Concurrency and Performance
Suppose: doFetch < doExecute



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

Concurrency and Performance suppose: doExecute < doFetch



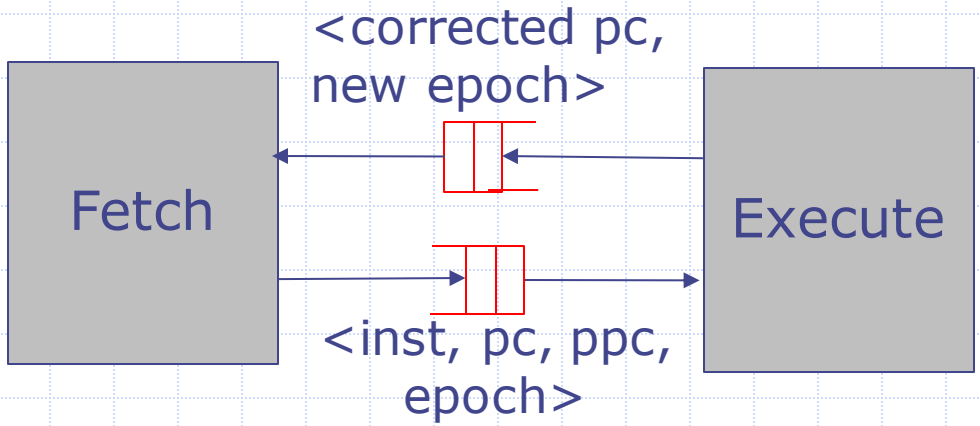
- For correctness;
  - rf: wr < rd (bypass rf)
  - sb: remove < {search, insert}
  - d2e: {first, deq} {<, CF} enq (pipelined or CF Fifo)
- To avoid a stall due to a RAW hazard between successive instructions
  - sb: remove < search
  - rf: wr < rd (bypass rf)
- Also, no dead cycle after a misprediction

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
- Concurrency and performance issues are subtle
  - For instance, bypassing necessarily increases combinational path lengths which can slow down the clock

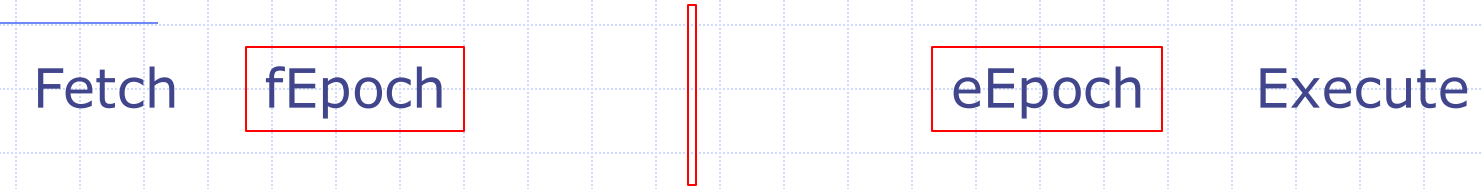
Non-atomic prediction correction

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

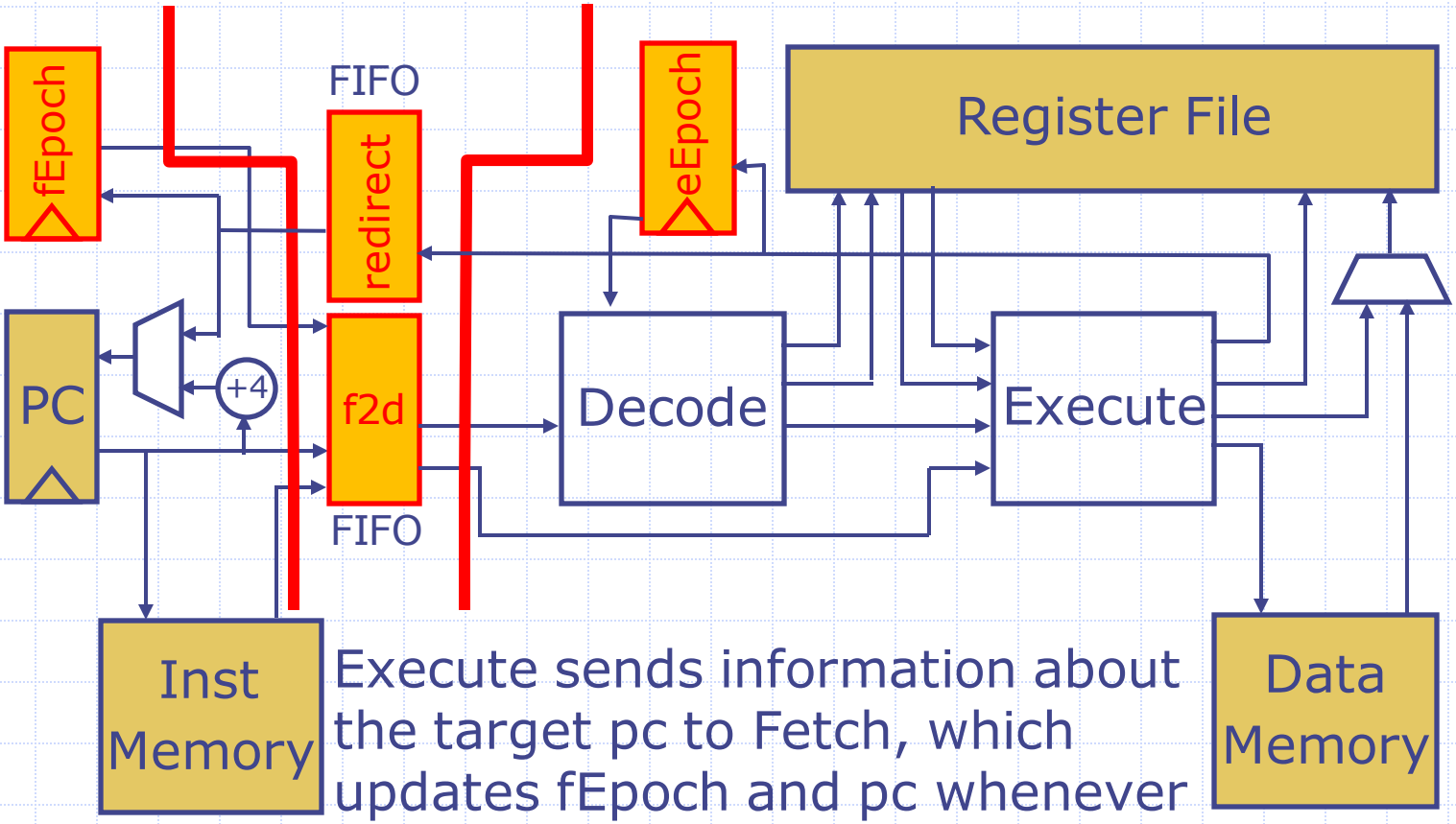
A decoupled solution using epochs



- 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 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



Two-stage pipeline Decoupled *code structure*

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

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

The Fetch rule

```
rule doFetch;
 let inst = iMem.req(pcF);
 if(redirect.empty)
    begin
      let newPcF = nap(pcF);
      pcF <= newPcF;
      f2d.enq(Fetch2Execute{pc: pcF, ppc: newPcF,
                            inst: inst, epoch: fEpoch});
    end
 else
    begin
      fEpoch <= !fEpoch;  pcF <= redirect.first;
      redirect.deq;
    end
endrule
```

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

The Execute rule

```
rule doExecute;
   let x = f2d.first;
   let inst = x.inst; let pc = x.pc; let inEp = x.epoch;
   if(inEp == eEpoch) begin
      ...decode, register fetch, exec, memory op,
         rf update nextPC ...
   if (x.ppc != nextPC) begin redirect.enq(eInst.addr);
                              eEpoch <= !inEp; end
                        end
  f2d.deq;
endrule
```

Can doFetch and doExecute execute concurrently?

yes, assuming CF FIFOs