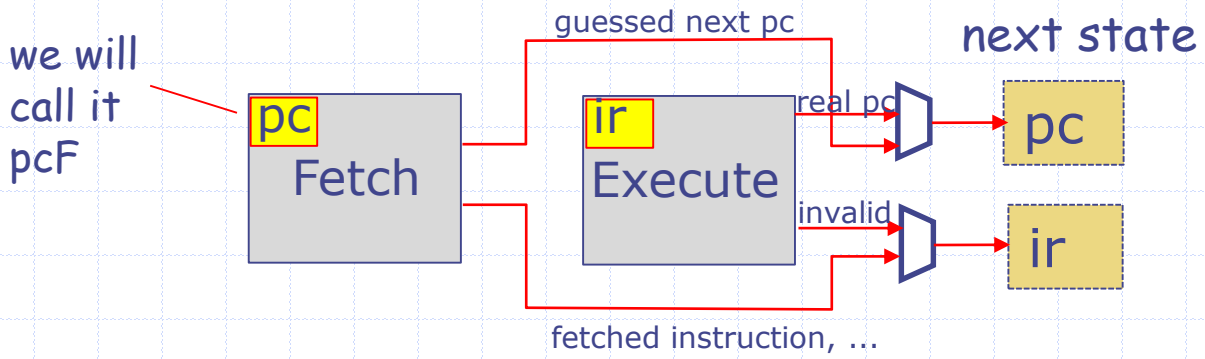
Constructive Computer Architecture:

Control Hazards

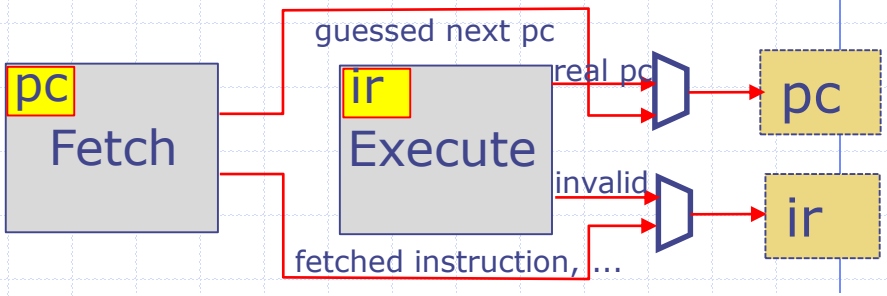
Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Synchronous 2-Stage Pipeline



Fetch and Execute are concurrently active on two different instructions; Fetch guesses the next pc and Execute corrects it when necessary

Synchronous 2-Stage Pipeline

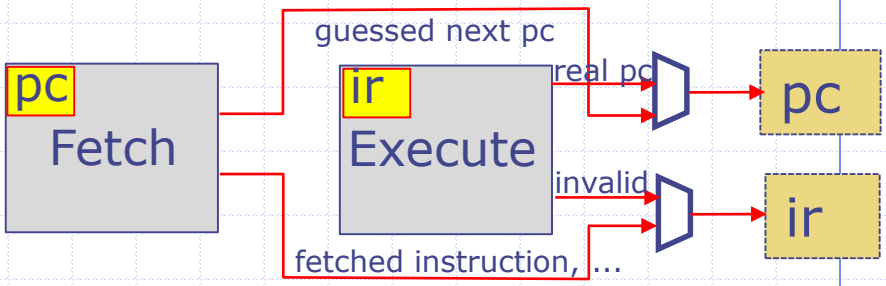



```
rule doPipeline ;
  let newInst = iMem.req(pcF);
  let newPcF = nap(pcF);
  let newIR= Valid(Fetch2Decode{pc:pcF,ppc:newPcF,
                                inst:newInst});
  if(isValid(ir)) begin
   let x = fromMaybe(?, ir); let pc = x.pc;
   let inst = x.inst;
   let dInst = decode(inst);
   ... register fetch, exec, memory op, rf update ...
   let nextPC = eInst.brTaken ? eInst.addr : pc + 4;
   if (x.ppc != nextPC) begin newIR = Invalid;
                              newPcF = nextPC; end
  end
  pcF <= newPcF; ir <= newIR;
endrule
```

pass pcF and predicted pc to the execute stage

execute

Performance?




```
rule doPipeline ;
  let newInst = iMem.req(pcF);
  let newPcF = nap(pcF);
  let newIR=Valid(Fetch2Decode{pc:pcF,
                               ppc:newPcF,
                               inst:newInst});
  if(isValid(ir)) begin
   let x = fromMaybe(?, ir); let pc = x.pc;
   let inst = x.inst;
   let dInst = decode(inst);
   ... register fetch, exec, memory op,
       rf update, nextPC ...
   if (x.ppc != nextPC)
     begin newIR = Invalid;
           newPcF = nextPC; end
  end
  pcF <= newPcF; ir <= newIR;
endrule
```

fetch

execute

- Notice there is always a bubble (dead cycle) after every miss-prediction
- The critical path:

  $\max\{t_{newPcF}, t_{newIr}\}$

  $\approx \max\{$

  $\max\{t_{nap}, t_{exec}\},$

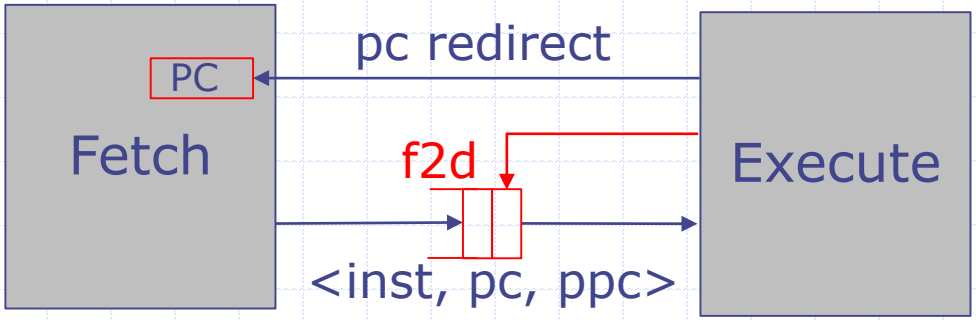
  $\max\{t_{iMem}, t_{exec}\}\}$

  $\approx \max\{t_{iMem}, t_{exec}\}$

t_{exec} includes t_{decode} etc.

The critical path is *not* $(t_{iMem} + t_{exec})$

Elastic two-stage pipeline



- We replace f2d register by a FIFO to make the machine more elastic, that is, Fetch keeps putting instructions into f2d and Execute keeps removing and executing instructions from f2d
- Fetch passes the pc and predicted pc in addition to the inst to Execute; Execute redirects the PC in case of a miss-prediction

An elastic Two-Stage pipeline

```
rule doFetch ;
  let inst = iMem.req(pcF);
  let newPcF = nap(pcF); pcF <= newPcF;
  f2d.enq(Fetch2Decode{pc:pcF, ppc:newPcF, inst:inst});
endrule

rule doExecute ;
   let x = f2d.first; let pc = x.pc;
   let inst = x.inst;
    ... register fetch, exec, memory op, rf update,
        nextPC ...
   if (x.ppc != nextPC) begin pcF <= eInst.addr;
                              f2d.clear; end
   else f2d.deq;
endrule
```

Can these rules execute concurrently assuming the FIFO allows concurrent enq, deq and clear?

No,
double writes in pc

These rules can execute in any order, however, the execution of doExecute may throw away fetched instructions

clear vs deq ?

For concurrency make pc into an EHR design 1

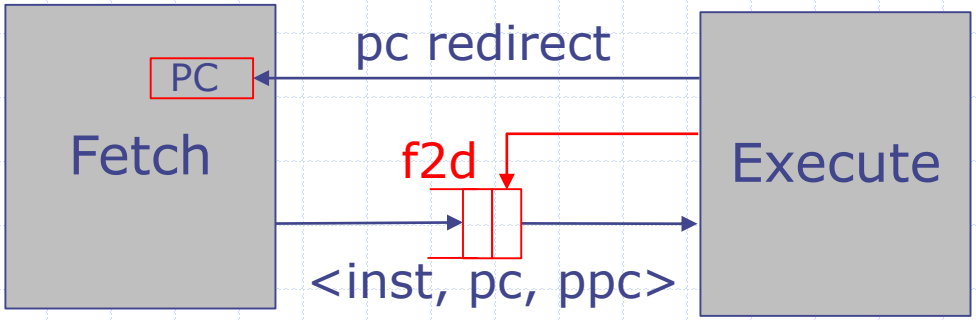
```
rule doFetch ;
  let inst = iMem.req(pcF[0]);
  let newPcF = nap(pcF[0]);
  pcF[0] <= newPcF;
  f2d.enq(Fetch2Decode{pc:pcF[0], ppc:newPcF, inst:inst});
endrule

rule doExecute ;
   let x = f2d.first; let pc = x.pc;
   let inst = x.inst;
    ... register fetch, exec, memory op, rf update,
        nextPC ...
   if (x.ppc != nextPC) begin pcF[1] <= eInst.addr;
                              f2d.clear; end
   else f2d.deq;
endrule
```

doFetch < doExecute

Notice, for concurrency, f2d implementation must guarantee that (enq < clear)

Concurrency and Correctness

Fetch < Execute



- Once Execute redirects the PC,
  - no wrong path instruction should be executed
  - the next instruction executed must be the redirected one

Thus, concurrent execution requires (enq < clear)

Performance?

A dead-cycle or pipeline bubble after each miss prediction

Design 1 Performance

```
rule doFetch ;
  let inst = iMem.req(pcF[0]);
  let newPcF = nap(pcF[0]);
  pcF[0] <= newPcF;
  f2d.enq(Fetch2Decode{pc:pcF[0], ppc:newPcF, inst:inst});
endrule
```

doFetch < doExecute
enq < clear

f2d is guaranteed to be empty after each misprediction (the same as the synchronous design)

```
rule doExecute ;
   let x = f2d.first; let pc = x.pc;
   let inst = x.inst;
    ... register fetch, exec, memory op, rf update,
        nextPC ...
   if (x.ppc != nextPC) begin pcF[1] <= eInst.addr;
                              f2d.clear; end
   else f2d.deq;
endrule
```

Design 2

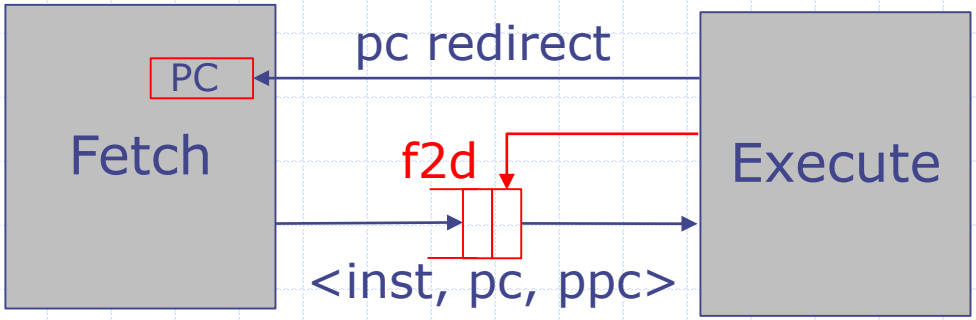
doExecute < doFetch

```
rule doFetch ;
  let inst = iMem.req(pcF[1]);
  let newPcF = nap(pcF[1]);
  pcF[1] <= newPcF;
  f2d.enq(Fetch2Decode{pc:pcF[1], ppc:newPcF, inst:inst});
endrule

rule doExecute ;
   let x = f2d.first; let pc = x.pc;
   let inst = x.inst;
    ... register fetch, exec, memory op, rf update,
        nextPC ...
   if (x.ppc != nextPC) begin pcF[0] <= eInst.addr;
                              f2d.clear; end
   else f2d.deq;
endrule
```

1. Concurrency: should (clear < enq) ?
2. Does this design have better performance?

Design 2 correctness/concurrency

Execute < Fetch



- Once Execute redirects the PC,
  - no wrong path instruction should be executed
  - the next instruction executed must be the redirected one

Thus, concurrent execution requires (clear < enq)

Performance? No dead-cycle but the critical path length is (t_{iMem} + t_{exec})

Slower clock means every instruction will take longer!

Takeaway

- Get the functionality right before worrying about concurrency
- Introduce EHRs systematically to avoid rule conflicts; analyze various designs for dead cycles and critical path lengths
  - BSV compiler produces information about conflicts
  - Dead cycles can be estimated by running suitable benchmark programs
  - Estimation of critical paths is often difficult and requires hardware synthesis tools

Killing fetched instructions

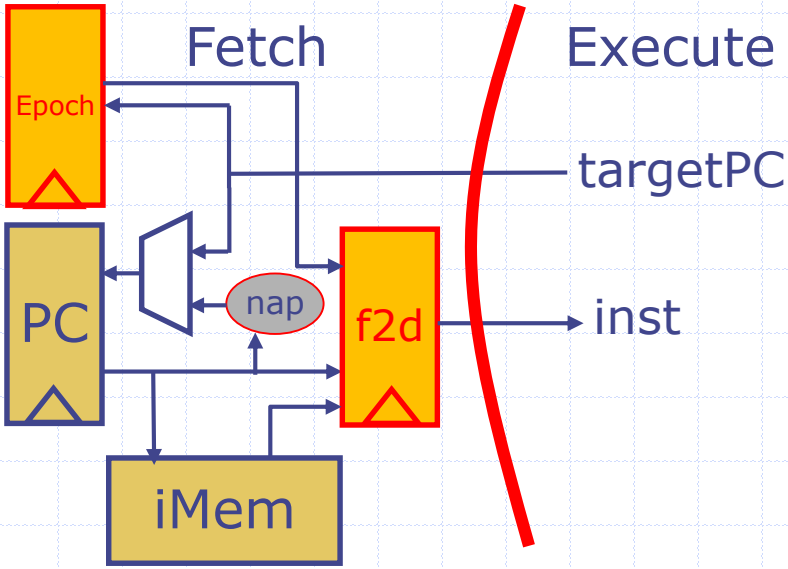
- In the simple design with combinational memory we have discussed so far, all the mispredicted instructions were present in f2d. So the Execute stage can *atomically:*
  - Clear f2d
  - Set pc to the correct target

- 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 to manage 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

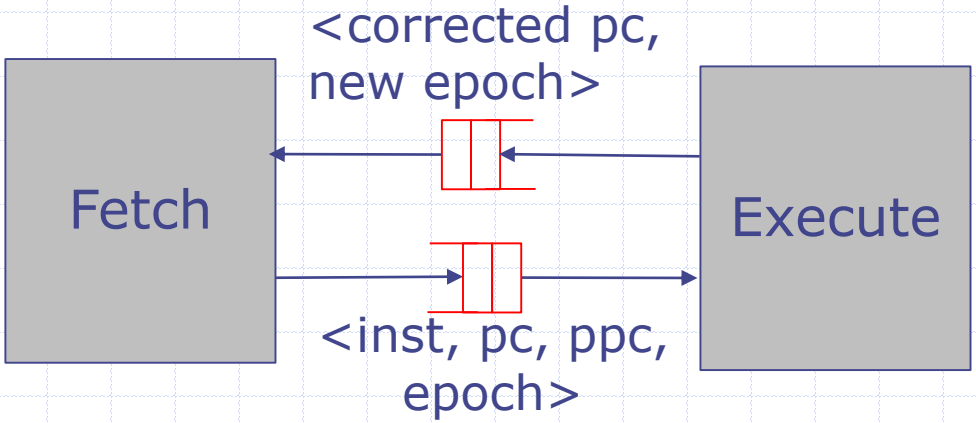
two values for epoch are sufficient

```
rule doFetch ;
  let instF=iMem.req(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
```

Discussion

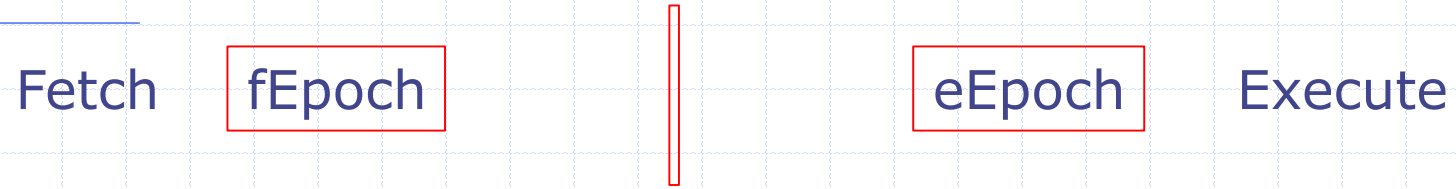
- Epoch based solution kills one wrong-path instruction at a time in the execute stage
- It may be slow, but it is more robust in more complex pipelines, if you have multiple stages between fetch and execute or if you have outstanding instruction requests to the iMem
- It requires the Execute stage to 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

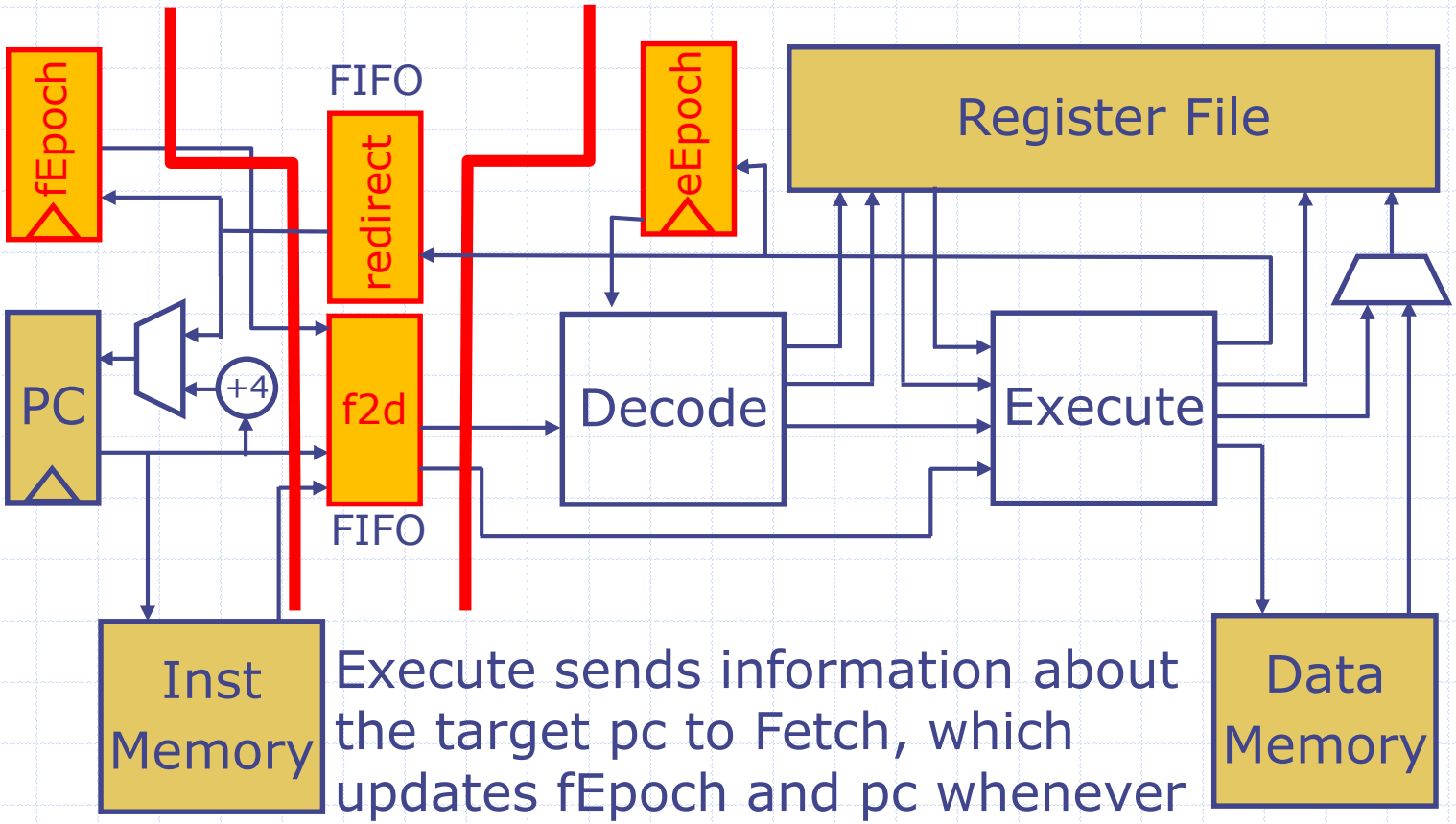
A decoupled solution using epochs



- 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 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 examines the redirect (PC) fifo

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.notEmpty)
    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

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