Constructive Computer Architecture

Tutorial 7: SMIPS Labs and Epochs

Andy Wright
6.S195 TA

Introduction

- Lab 6
  - 6 Stage SMIPS Processor
  - Due today
- Lab 7
  - Complex Branch Predictors
  - Posted online
  - Due next Friday

Lab 6

- Does low IPC => low grade?
  - Only if low IPC is from a 'mistake' in your processor.
- What is a 'mistake'?
  - Not updating your BTB with redirect information
  - Using too small of a scoreboard
  - Having schedule conflicts between pipeline stages

Lab 6

- What questions do you have?

Lab 7

- Adding history bits to BTB
  - Combines target and direction prediction
- Implement BHT
  - Separates direction prediction from target prediction
- Synthesize for FPGA
  - Used to calculate IPS

Fixed slides from L16

Epoch Management

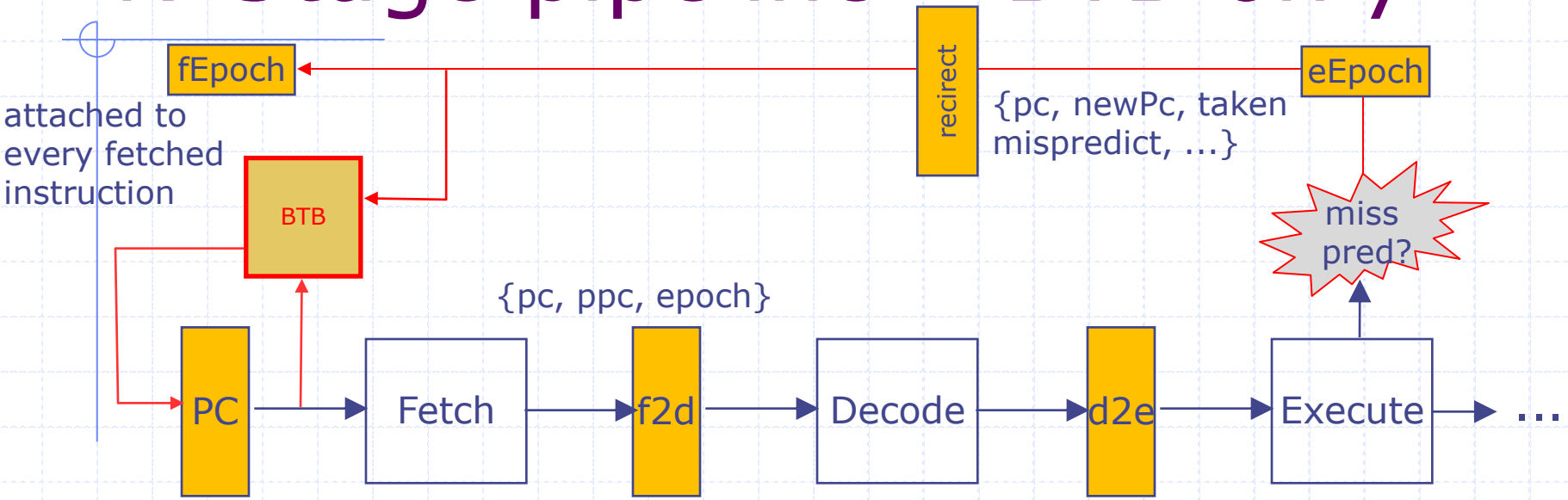
Multiple predictors in a pipeline

- At each stage we need to take two decisions:
  - Whether the current instruction is a *wrong path instruction*. Requires looking at epochs
  - Whether the prediction (ppc) following the current instruction is good or not. Requires consulting the prediction data structure (BTB, BHT, …)
- Fetch stage must correct the pc unless the redirection comes from a known wrong path instruction
- Redirections from Execute stage are always correct, i.e., cannot come from wrong path instructions

Dropping or poisoning an instruction

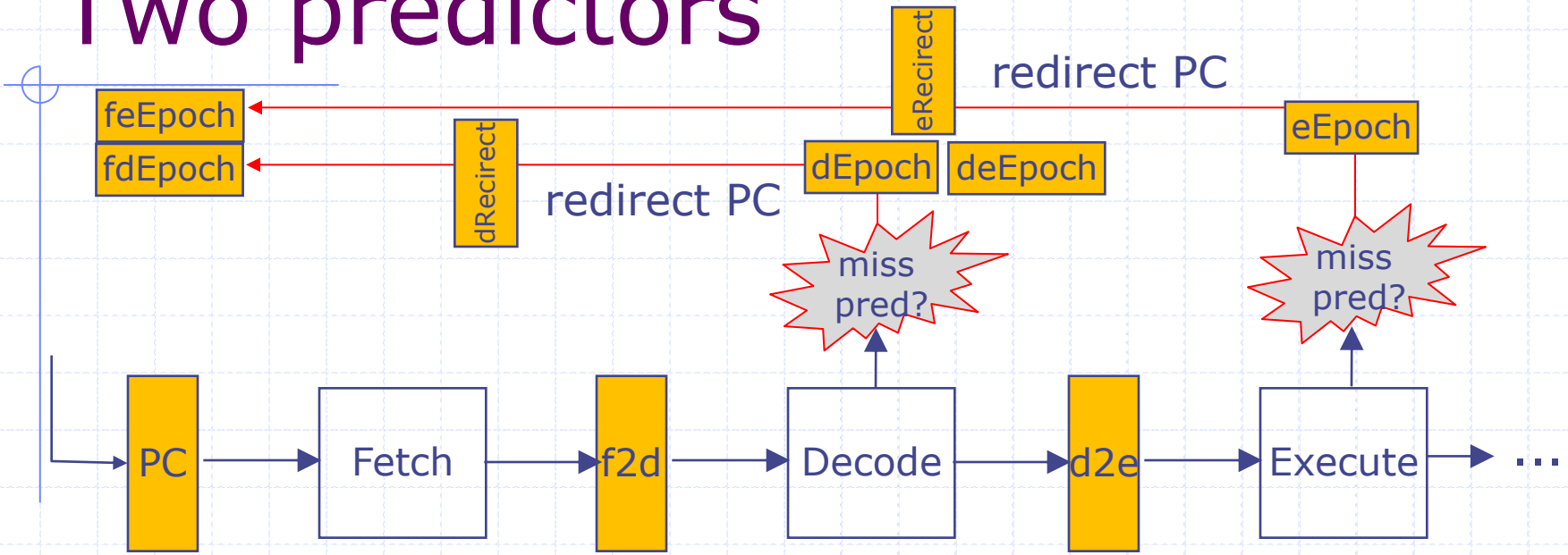
- Once an instruction is determined to be on the wrong path, the instruction is either dropped or poisoned
- Drop: If the wrong path instruction has not modified any book keeping structures (e.g., Scoreboard) then it is simply removed
- Poison: If the wrong path instruction has modified book keeping structures then it is poisoned and passed down for book keeping reasons (say, to remove it from the scoreboard)
- Subsequent stages know not to update any architectural state for a poisoned instruction

N-Stage pipeline – BTB only



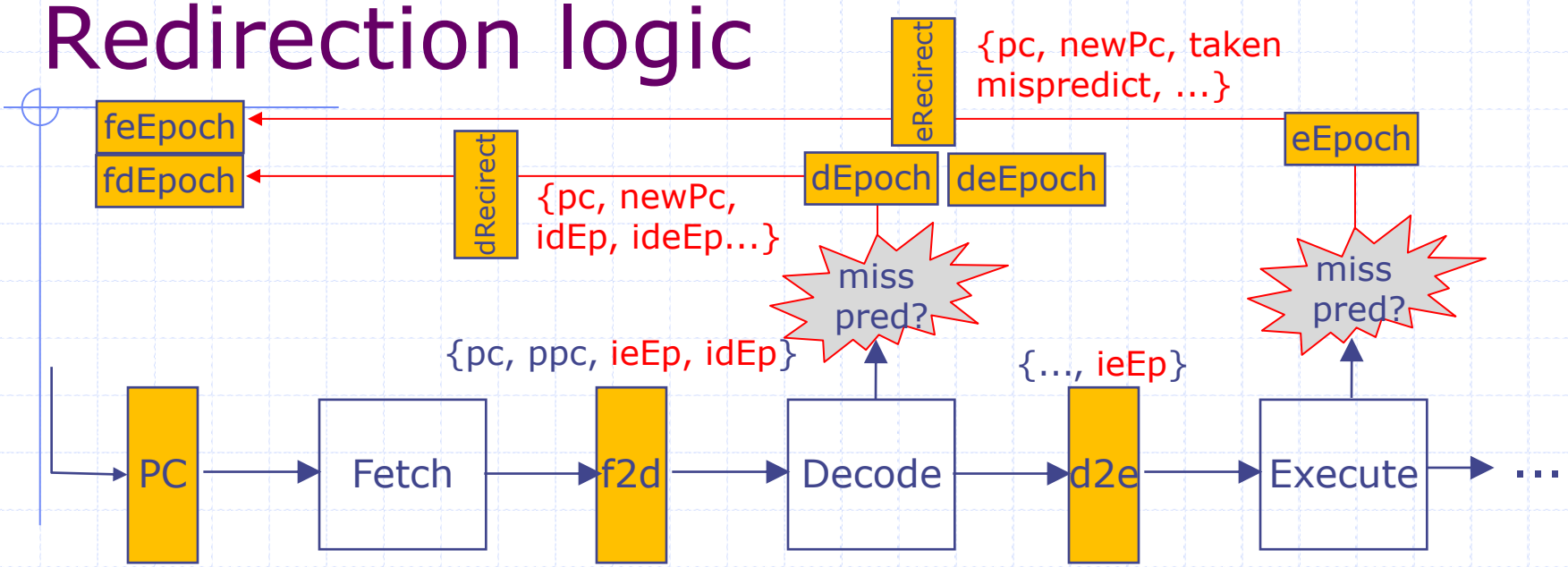
- At Execute:
  - (pc) if (epoch!=eEpoch) then mark instruction as poisoned
  - (ppc) if (no poisoning) & mispred then change eEpoch; send <pc, newPc, ...> to Fetch
- At Fetch:
  - msg from execute: train BTB with <pc, newPc, taken, mispredict>
  - if msg from execute indicates misprediction then set pc, change fEpoch

N-Stage pipeline: Two predictors



- Suppose both Decode and Execute can redirect the PC; Execute redirect should have priority, i.e., Execute redirect should never be overruled
- We will use separate epochs for each redirecting stage
  - feEpoch and deEpoch are estimates of eEpoch at Fetch and Decode, respectively
  - fdEpoch is Fetch's estimates of dEpoch
  - Initially set all epochs to 0

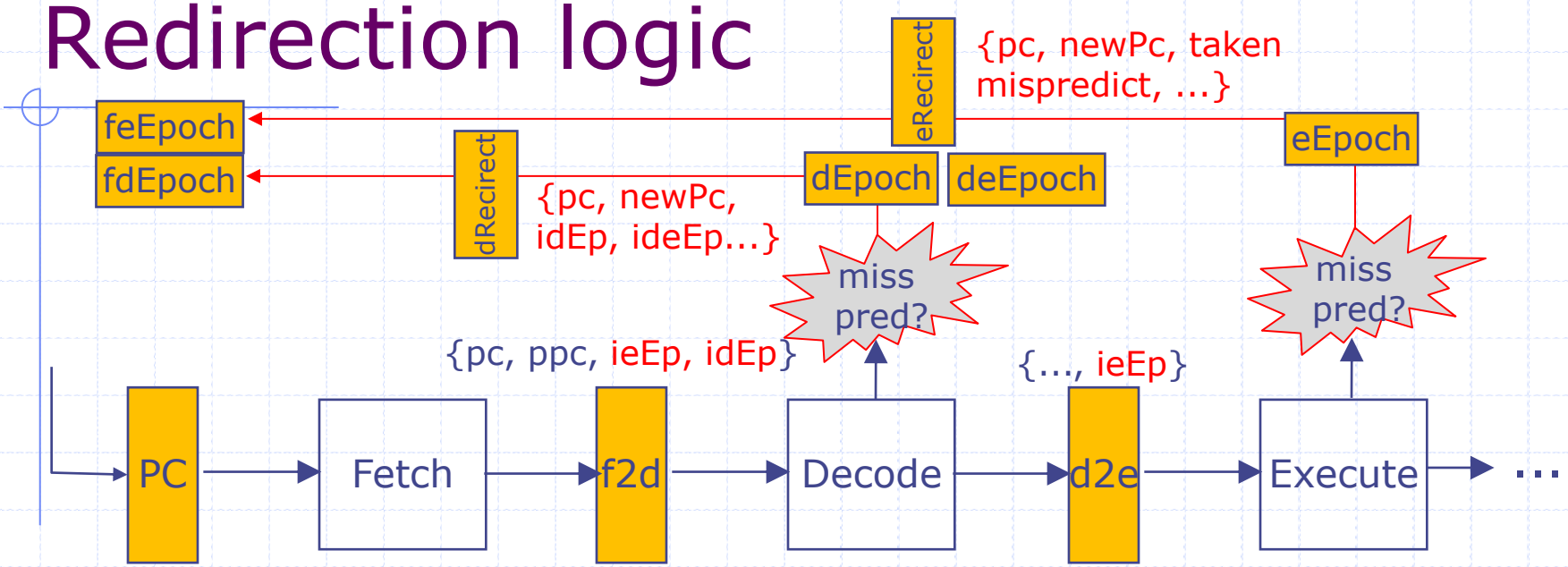
N-Stage pipeline: Two predictors Redirection logic



- At execute:
  - (pc) if (ieEp!=eEp) then poison the instruction
  - (ppc) if (no poisoning) & mispred then change eEp;
  - (ppc) for every control instruction send <pc, target pc, taken, mispred…> to fetch
- At fetch:
  - msg from execute: if (mispred) set pc, change feEp,
  - msg from decode: If (no redirect message from Execute)
  - if (ideEp=feEp) then set pc, change fdEp to idEp
- At decode: …

make sure that the msg from Decode is not from a wrong path instruction

Decode stage Redirection logic



Is ieEp = deEp ?

yes

Is idEp = dEp ?

yes

Current instruction is OK; check the ppc prediction via BHT, Switch dEp if misprediction

no

Wrong path instruction; drop it

no

Current instruction is OK but Execute has redirected the pc; Set <deEp, dEp> to <ieEp, idEp> check the ppc prediction via BHT, Switch dEp if misprediction

Another way to manage epochs

Write the rules as simple as possible (guarded atomic actions), then add EHRs if necessary

Fetch Rule

```
fInst.pc = pc;
fInst.ppc = prediction( pc );
fInst.eEpoch = eEpoch;
fInst.dEpoch = dEpoch;
...
pc <= fInst.ppc;
f2dFifo.enq( fInst );
```

Decode Rule

```
if( dInst.eEpoch != eEpoch )
        kill fInst
else if( dInst.dEpoch != dEpoch )
        kill fInst
else begin
        let newpc = prediction( dInst );
        if( newpc != dInst.ppc ) begin
                pc <= newpc
                dEpoch <= !dEpoch;
        end
        …
end
```

Execute Rule

```
if( eInst.eEpoch != eEpoch )
        poison eInst
else begin
        if( mispredict ) begin
                pc <= newpc;
                eEpoch <= !eEpoch;
                train branch predictors
        end
        …
end
```

Conflicts

- PC read < PC write
  - fetch < {decode, execute}
- dEpoch read < dEpoch write
  - fetch < decode
- eEpoch read < eEpoch write
  - {fetch, decode} < execute
- PC write C PC write
  - fetch C decode C execute C fetch

None of these stages can execute in the same clock cycle!

Now add EHRs

1) Choose an ordering between the rules and assign the corresponding EHR ports

   (fetch, decode, execute)

2) Change conflicting registers into EHRs

   (pc)

```
Ehr#(3, Addr) pc -> mkEhr(?);
```

Fetch Rule – port 0

```
fInst.pc = pc[0];
fInst.ppc = prediction( pc[0] );
fInst.eEpoch = eEpoch;
fInst.dEpoch = dEpoch;
...
pc[0] <= fInst.ppc;
f2dFifo.enq( fInst );
```

Decode Rule – port 1

```
if( dInst.eEpoch != eEpoch )
        kill fInst
else if( dInst.dEpoch != dEpoch )
        kill fInst
else begin
        let newpc = prediction( dInst );
        if( newpc != dInst.ppc ) begin
                pc[1] <= newpc;
                dEpoch <= !dEpoch;
        end
        …
end
```

Execute Rule – port 2

```
if( eInst.eEpoch != eEpoch )
        poison eInst
else begin
        if( mispredict ) begin
                pc[2] <= newpc;
                eEpoch <= !eEpoch;
                train branch predictors
        end
        …
end
```

Another Ordering

1) Choose an ordering between the rules and assign the corresponding EHR ports
   (execute, decode, fetch)
2) Change conflicting registers into EHRs
   (pc, dEpoch, eEpoch)

```
Ehr#(3, Addr) pc -> mkEhr(?);
Ehr#(3, Bool) dEpoch -> mkEhr(False);
Ehr#(3, Bool) eEpoch -> mkEhr(False);
```

Fetch Rule – port 2

```
fInst.pc = pc[2];
fInst.ppc = prediction( pc[2] );
fInst.eEpoch = eEpoch[2];
fInst.dEpoch = dEpoch[2];
...
pc[2] <= fInst.ppc;
f2dFifo.enq( fInst );
```

Decode Rule – port 1

```
if( dInst.eEpoch != eEpoch[1] )
        kill fInst
else if( dInst.dEpoch != dEpoch[1] )
        kill fInst
else begin
        let newpc = prediction( dInst );
        if( newpc != dInst.ppc ) begin
                pc[1] <= newpc;
                dEpoch[1] <= !dEpoch[1];
        end
        …
end
```

Execute Rule – port 0

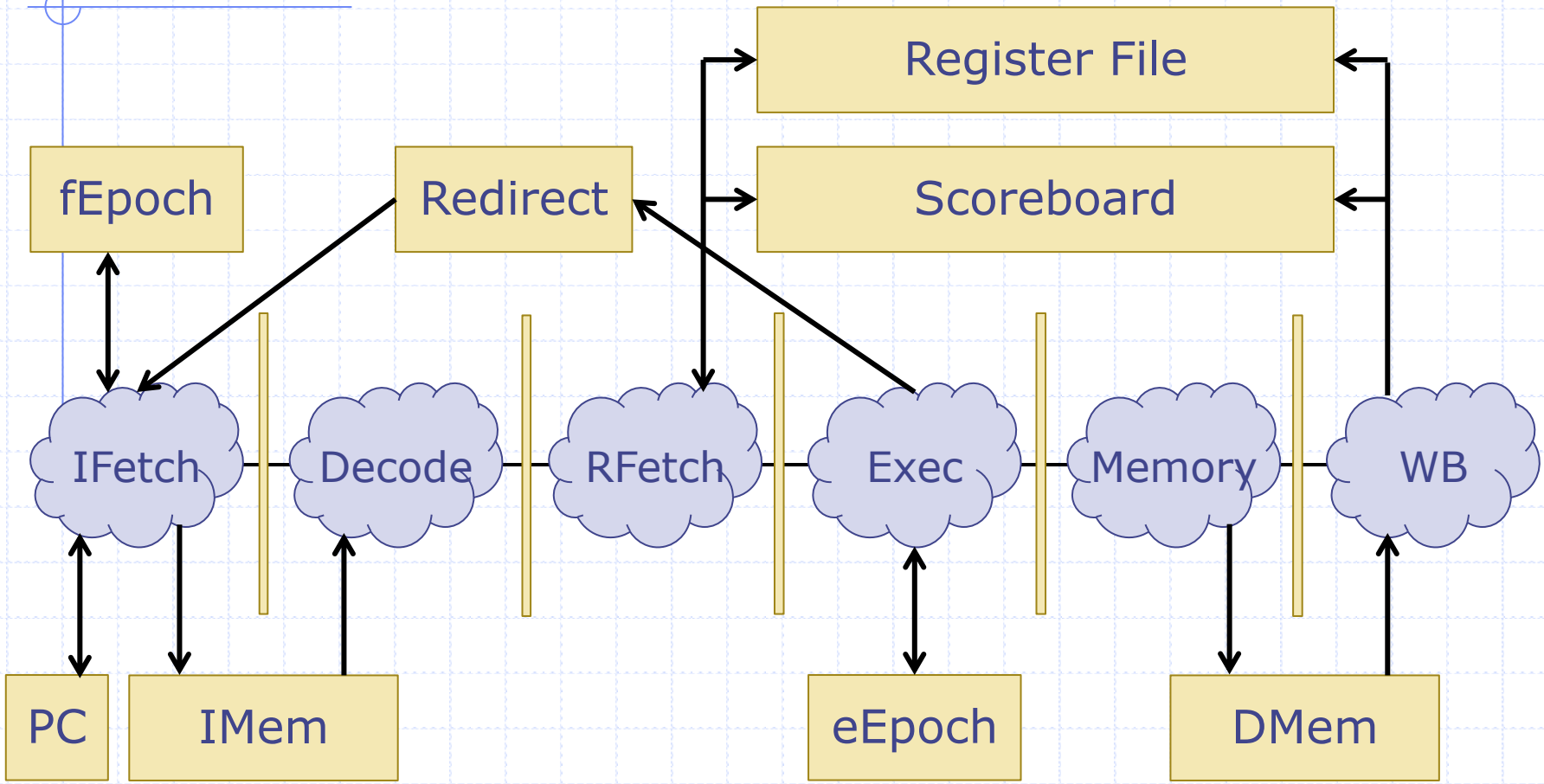
```
if( eInst.eEpoch != eEpoch[0] )
        poison eInst
else begin
        if( mispredict ) begin
                pc[0] <= newpc;
                eEpoch[0] <= !eEpoch[0];
                train branch predictors
        end
        …
end
```

Different View of EHR

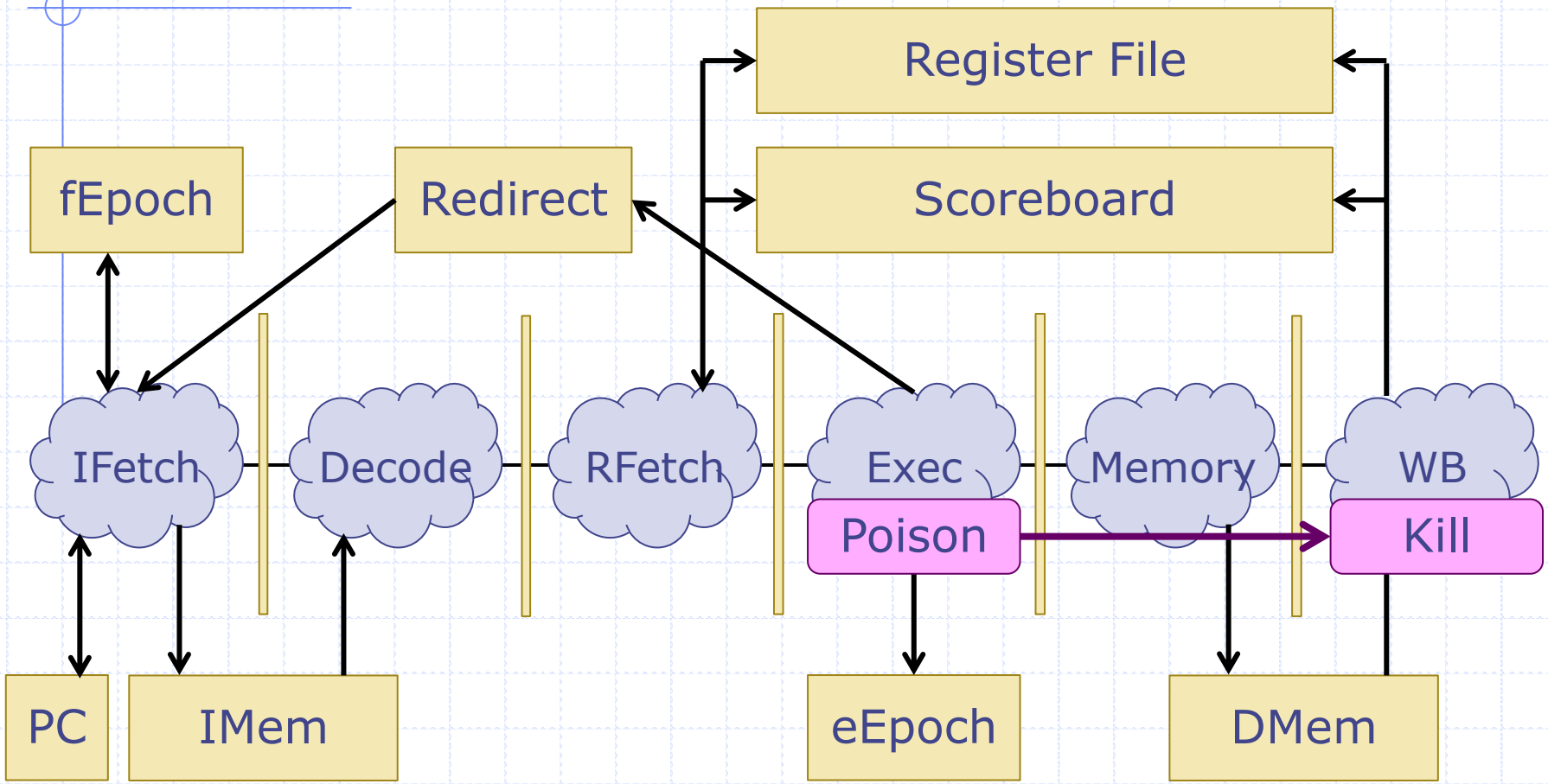
- This transformation makes more sense when you think of an EHR as sub-cycle register.
- This is explained more in the paper "The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs" by Daniel L. Rosenband

Questions?

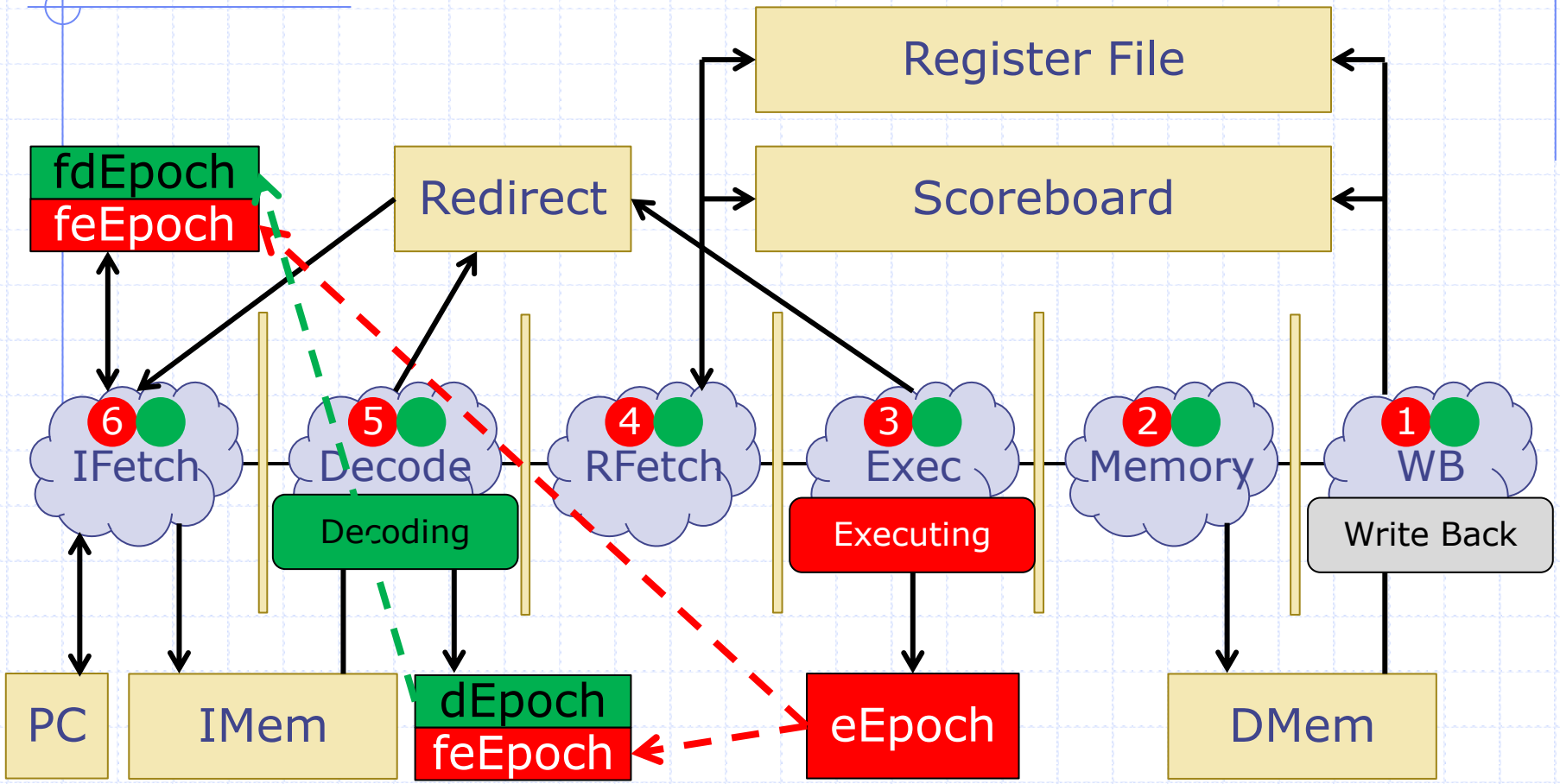
6 stage SMIPS pipeline



Poisoning Pipeline



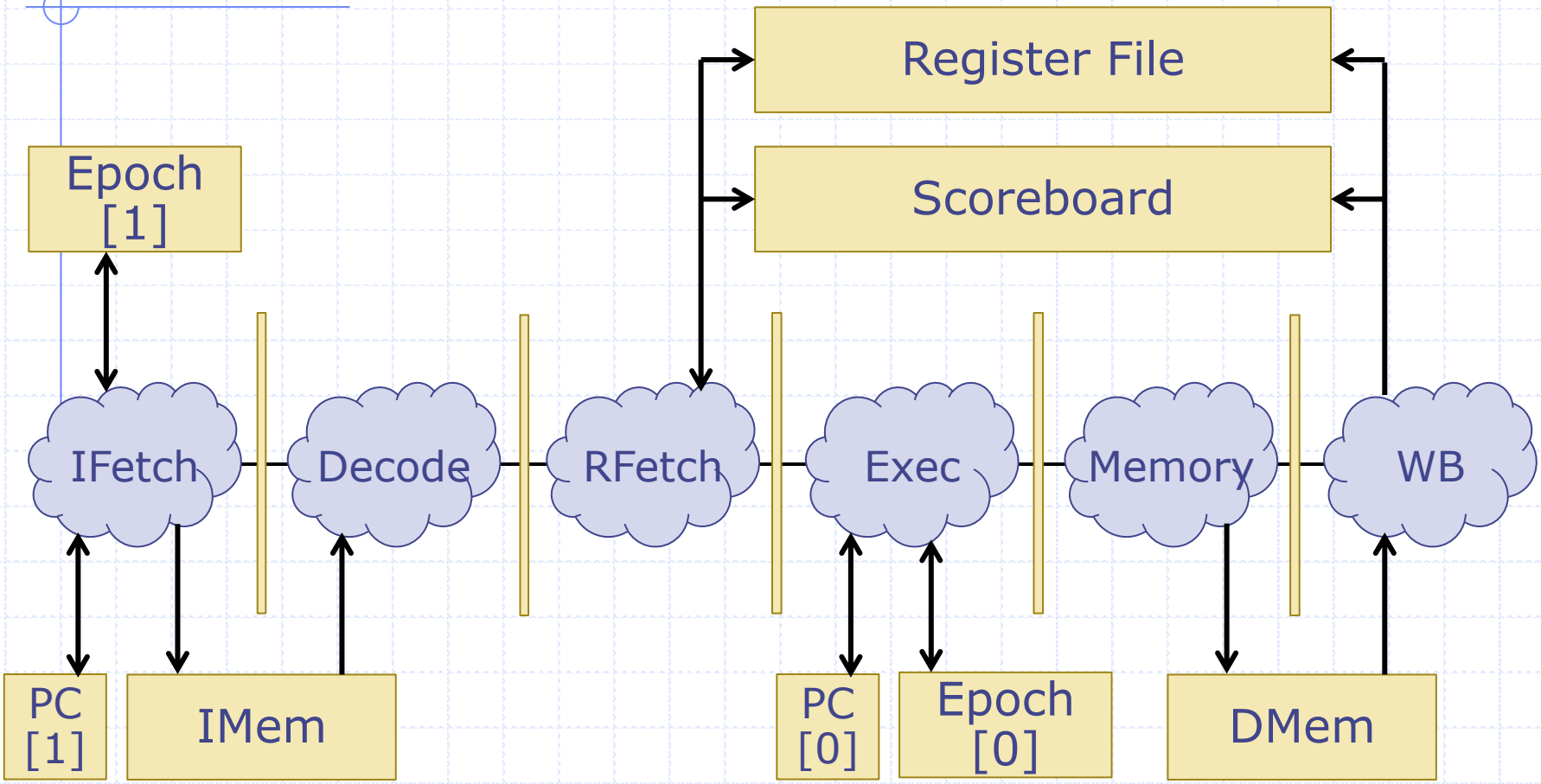
Correcting PC in Decode and Execute



Fetch has local estimates of eEpoch and dEpoch

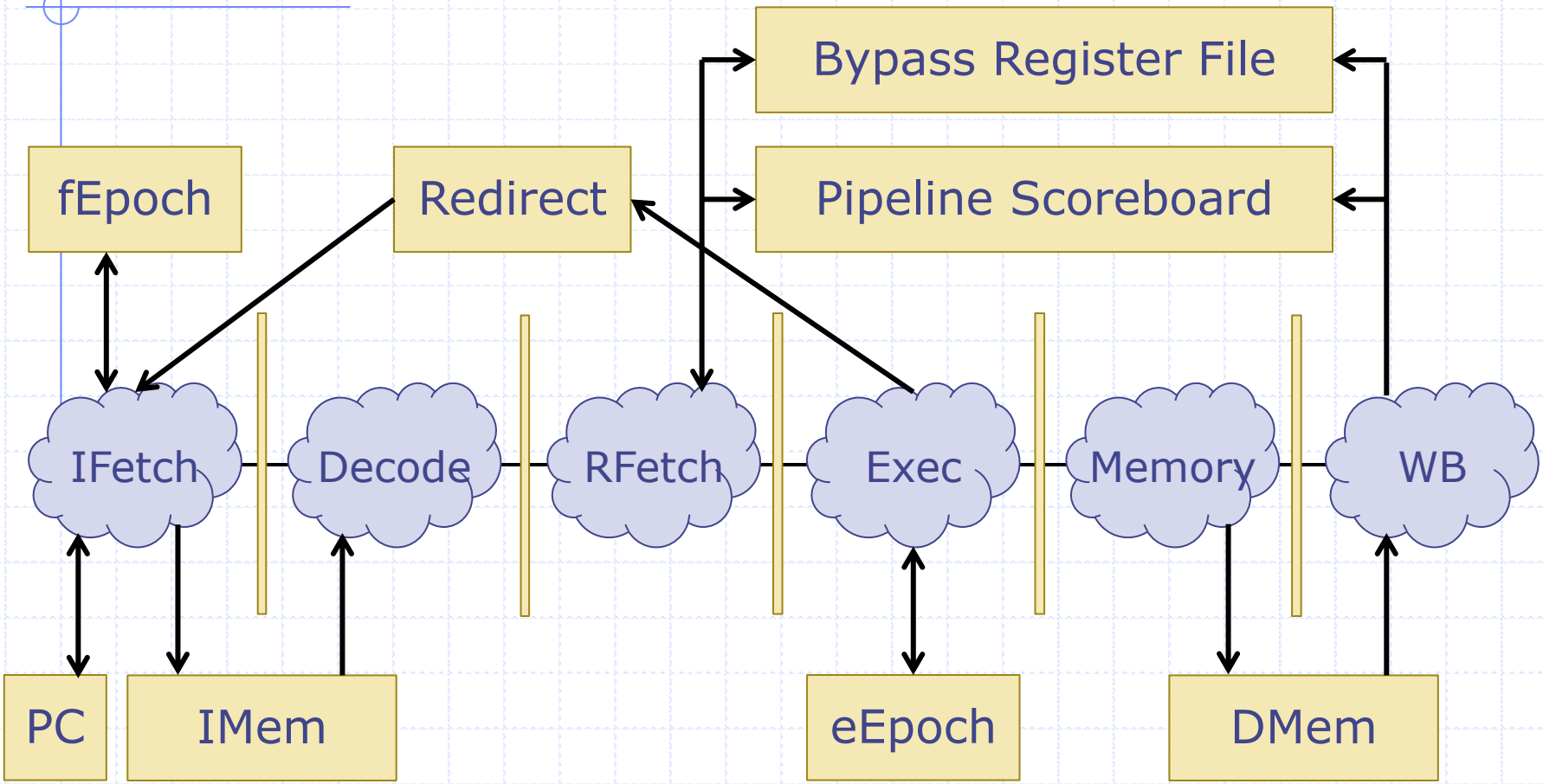
Decode has a local estimate of eEpoch

fEpoch and PC feedback



Make the PC an EHR too! Whenever Execute sees a misprediction, IFetch reads the correct next instruction *in the same cycle*!

RFile and SB feedback



You can use a scoreboard that removes before searching (called a pipeline scoreboard because it is similar to pipeline fifo's deq<enq behavior)