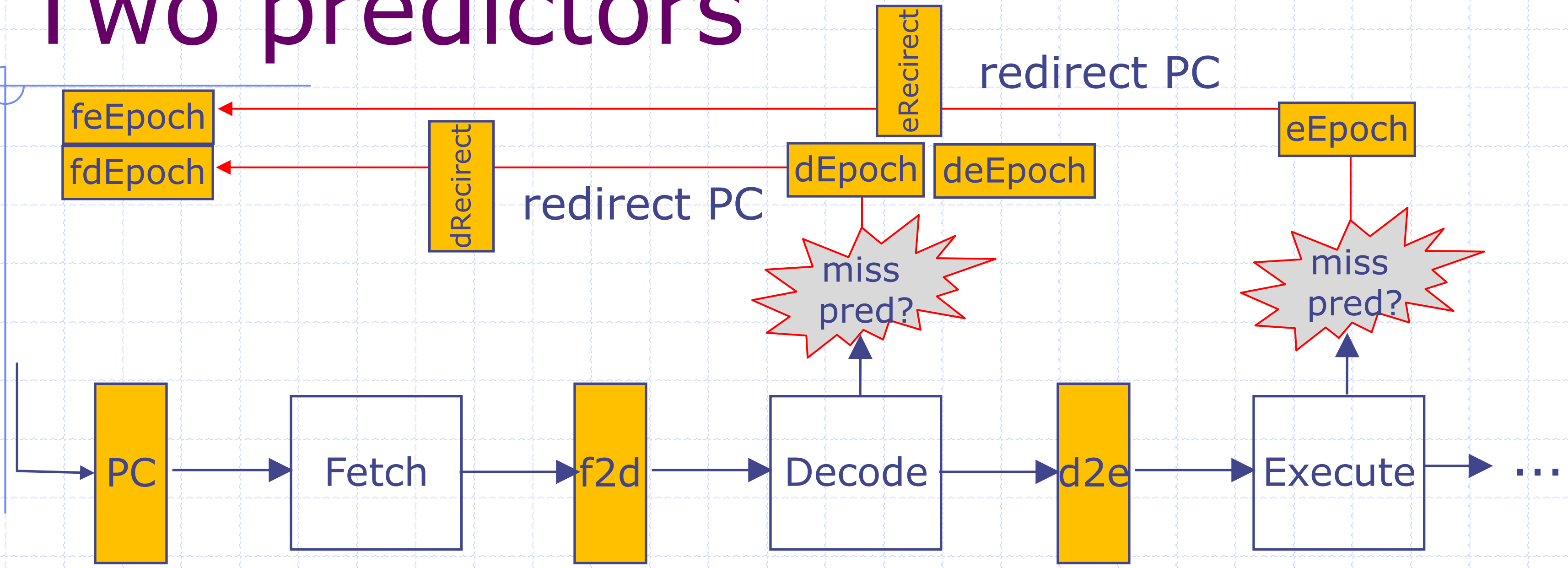
Constructive Computer Architecture

Tutorial 7: SMIPS Epochs

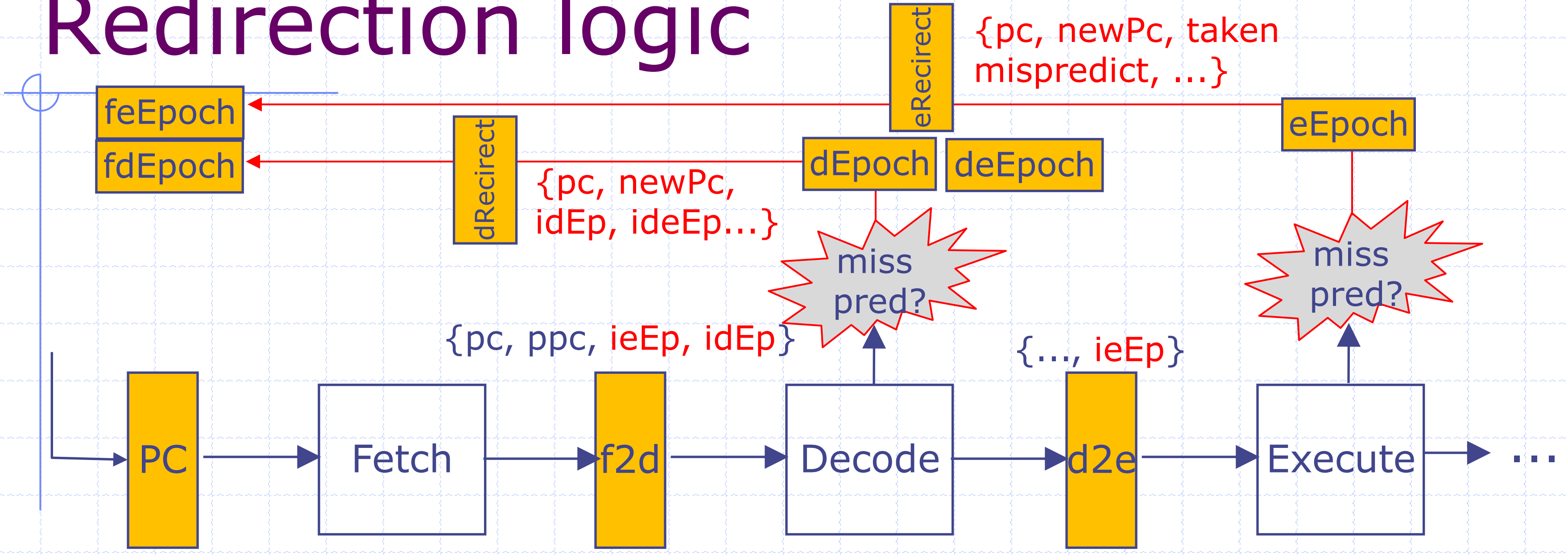
Andy Wright
6.175 TA

N-Stage pipeline: Two predictors



- Both Decode and Execute can redirect the PC; 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. deEpoch is updated by the incoming eEpoch
  - fdEpoch is Fetch's estimates of dEpoch
  - Initially set all epochs to 0
- Execute stage logic does not change

Decode stage Redirection logic



Is ieEp = deEp ?

yes

no

Is idEp = dEp ?

yes

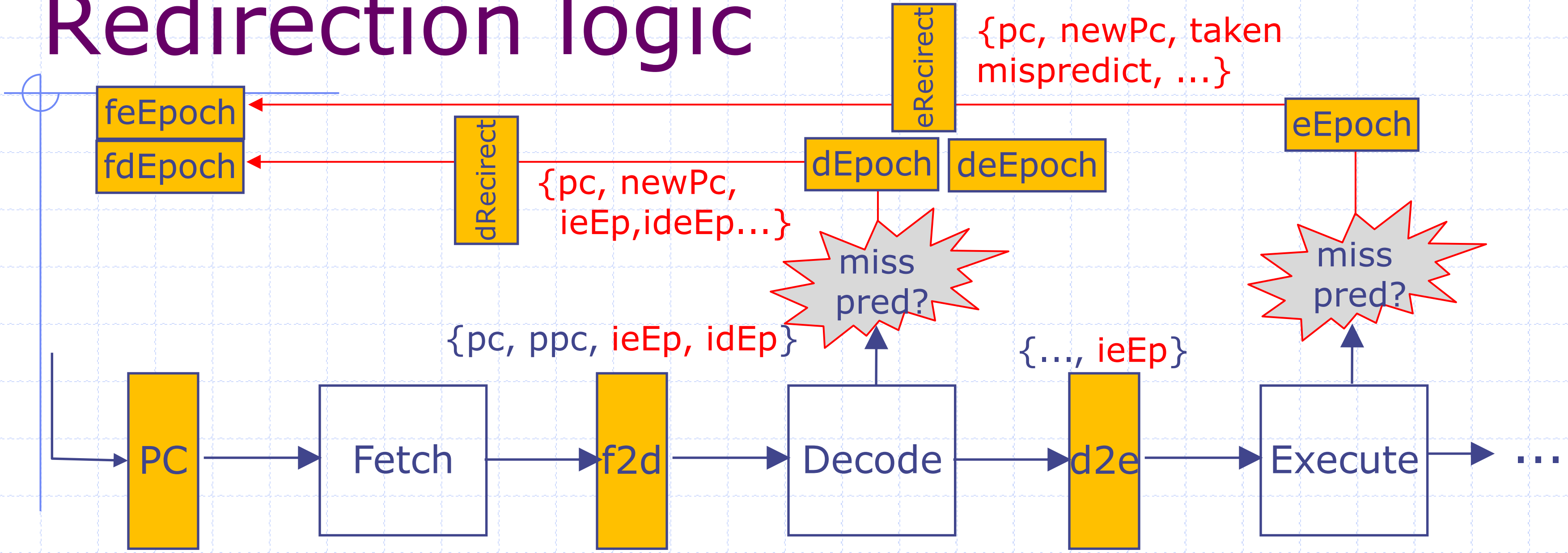
no

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

Wrong path instruction; drop it

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

N-Stage pipeline: Two predictors Redirection logic



- At execute:
  - (correct pc?) if (ieEp!=eEp) then poison the instruction
  - (correct ppc?) if (correct pc) & mispred then change eEp;
  - For every non-poisoned control instruction send <pc, newPc, taken, mispred, ...> to Fetch for training and redirection
- At fetch:
  - msg from execute: train btb & 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

now some coding ...

- 4-stage pipeline (F, D&R, E&M, W)
- Direction predictor training is incompletely specified

You will explore the effect of predictor training in the lab

4-Stage pipeline with Branch Prediction

```
module mkProc(Proc);
  Reg#(Addr)            pc <- mkRegU;
  RFile                 rf <- mkBypassRFile;
  IMemory             iMem <- mkIMemory;
  DMemory             dMem <- mkDMemory;
  Fifo#(1, Decode2Execute) d2e <- mkPipelineFifo;
  Fifo#(1, Exec2Commit)    e2c <- mkPipelineFifo;
  Scoreboard#(2) sb <- mkPipelineScoreboard;
  Reg#(Bool)     feEp <- mkReg(False);
  Reg#(Bool)     fdEp <- mkReg(False);
  Reg#(Bool)     dEp <- mkReg(False);
  Reg#(Bool)     deEp <- mkReg(False);
  Reg#(Bool)     eEp <- mkReg(False);
  Fifo#(ExecRedirect) redirect <- mkBypassFifo;
  Fifo#(DecRedirect) decRedirect <- mkBypassFifo;
  NextAddrPred#(16) btb <- mkBTB;
  DirPred#(1024) dirPred <- mkBHT;
```

4-Stage-BP pipeline Fetch rule: multiple predictors

```
rule doFetch;
     let inst = iMem.req(pc);
     if(redirect.notEmpty)
        begin redirect.deq; btb.update(redirect.first); end
     if(redirect.notEmpty && redirect.first.mispredict)
        begin pc <= redirect.first.nextPc; feEp <= !feEp; end
     else if(decRedirect.notEmpty) begin
            if(decRedirect.first.eEp == feEp)              begin
            fdEp <= !fdEp; pc <= decRedirect.first.nextPc;  end
          decRedirect.deq;              end;
     else begin
       let ppc = btb.predPc(pc);
       f2d.enq(Fetch2Decoode{pc: pc, ppc: ppc, inst: inst,
                                  eEp: feEp, dEp: fdEp});
       end
     endrule
```

Not enough information is being passed from Fetch to Decode to train BHT – lab problem

4-Stage-BP pipeline Decode&RegRead Action

```
function Action decAndRegFetch(DInst dInst, Addr pc, Addr ppc,
                                                           Bool eEp);
action
     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: ppc,
              dInst: dInst, epoch: eEp,
              rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.rDst);
      end
endaction
endfunction
```

4-Stage-BP pipeline Decode&RegRead rule

```
rule doDecode;
  let x = f2d.first; let inst = x.inst; let pc = x.pc;
  let ppc = x.ppc; let idEp = x.dEp; let ieEp = x.eEp;
  let dInst = decode(inst);
  let nextPc = dirPrec.predAddr(pc, dInst);
  if(ieEp != deEp) begin // change Decode's epochs and
                         // continue normal instruction execution
      deEp <= ieEp; let newdEp = idEp;
      decAndRegRead(inst, pc, nextPc, ieEp);
      if(ppc != nextPc)  begin  newdEp = !newdEp;
            decRedirect.enq(DecRedirect{pc: pc,
                                nextPc: nextPc, eEp: ieEp}); end
      dEp <= newdEp end
  else if(idEp == dEp) begin
      decAndRegRead(inst, pc, nextPc, ieEp);
      if(ppc != nextPc)                                 begin
        dEp <= !dEp; decRedirect.enq(DecRedirect{pc: pc,
                              newPc: newPc, eEp: ieEp}); end
                          end // if idEp!=dEp then drop,ie, no action
    f2d.deq;
endrule
```

BHT update is missing- lab problem

4-Stage-BP pipeline Execute rule: predictor training

```
rule doExecute;
    let x = d2e.first;
    let dInst = x.dInst; let pc    = x.pc;
    let ppc   = x.ppc;   let epoch = x.epoch;
    let rVal1 = x.rVal1; let rVal2 = x.rVal2;
    if(epoch == eEpoch)                begin
      let eInst = exec(dInst, rVal1, rVal2, pc, ppc);
      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});
      e2c.enq(Exec2Commit{dst:eInst.dst, data:eInst.data});
      if(eInst.mispredict) eEpoch <= !eEpoch
      if(eInst.iType == J || eInst.iType == Jr || eInst.iType == Br)
        redirect.enq(Redirect{pc: pc, nextPc: eInst.addr,
            taken: eInst.brTaken, mispredict: eInst.mispredict,
            brType: eInst.iType}); end
    else e2c.enq(Exec2Commit{dst:Invalid, data:?});
    d2e.deq;
endrule
```

4-Stage-BP pipeline Commit rule

```
rule doCommit;
  let dst  = eInst.first.dst;
  let data = eInst.first.data;
  if(isValid(dst))
    rf.wr(tuple2(validValue(dst), data);
  e2c.deq;
  sb.remove;
endrule
```

Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)

  BTB works well if the same case is used repeatedly

- Dynamic function call (jump to run-time function address)

  BTB works well if the same function is usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)

  BTB works well if return is usually to the same place

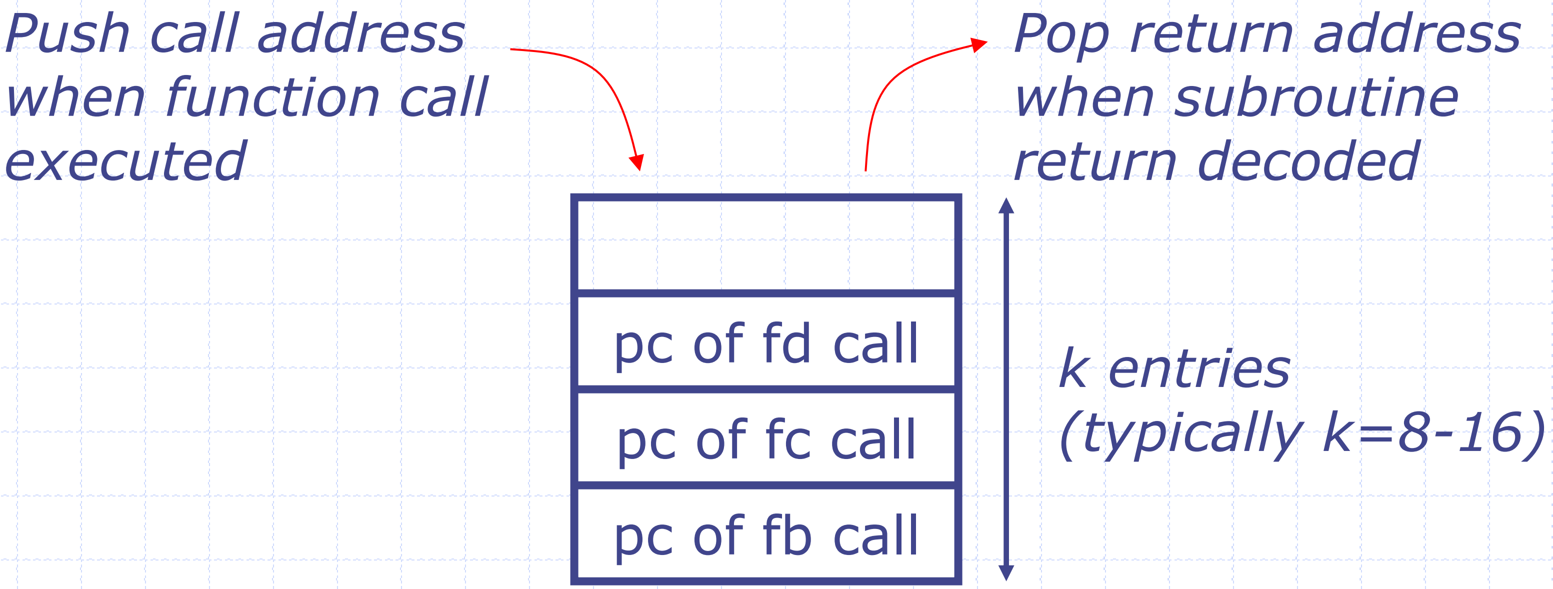
  However, often one function is called from many distinct call sites!

How well does BTB or BHT work for each of these cases?

Subroutine Return Stack

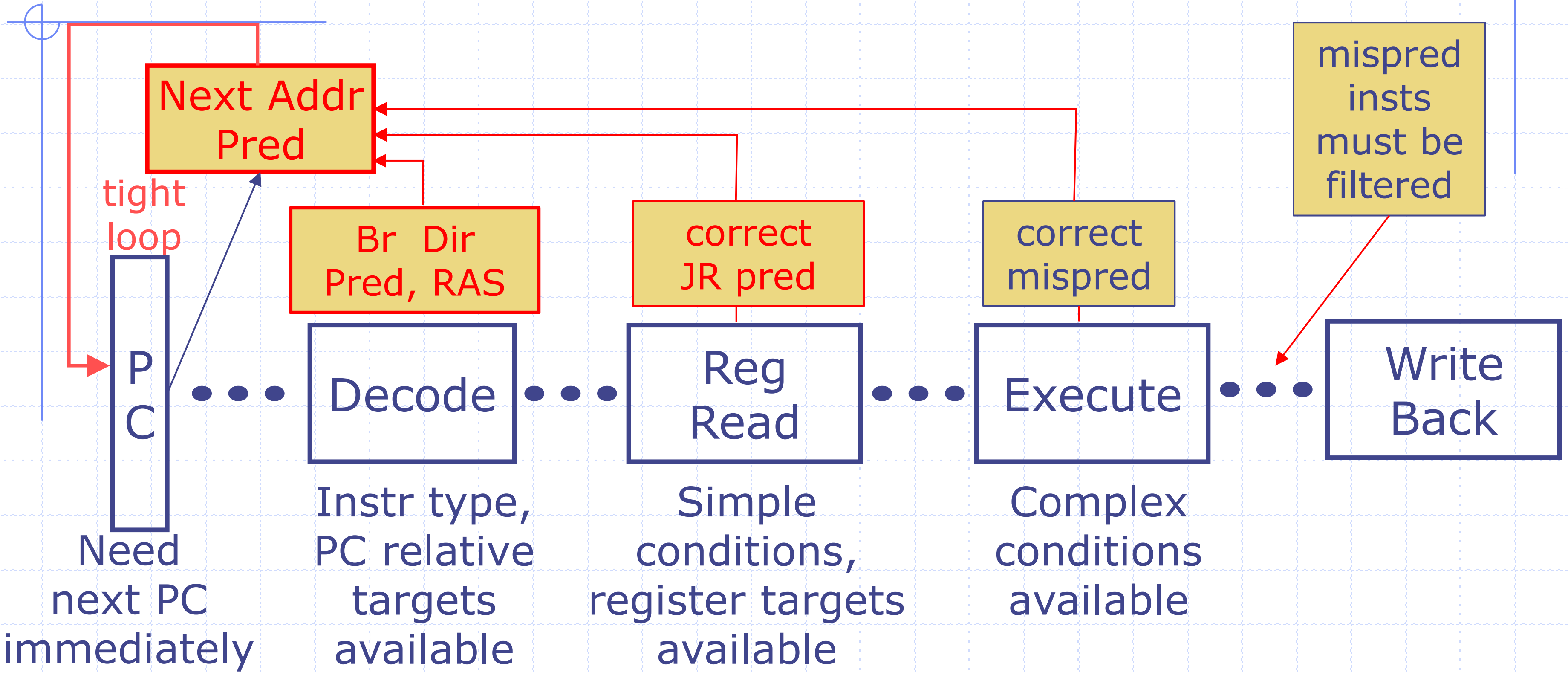
- A small structure to accelerate JR for subroutine returns is typically much more accurate than BTBs

```
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```



Push call address when function call executed
Pop return address when subroutine return decoded
pc of fd call
pc of fc call
pc of fb call
k entries (typically k=8-16)

Multiple Predictors: BTB + BHT + Ret Predictors



Next Addr Pred
tight loop
P C
Br Dir Pred, RAS
correct JR pred
correct mispred
mispred insts must be filtered
Decode
Reg Read
Execute
Write Back
Need next PC immediately
Instr type, PC relative targets available
Simple conditions, register targets available
Complex conditions available

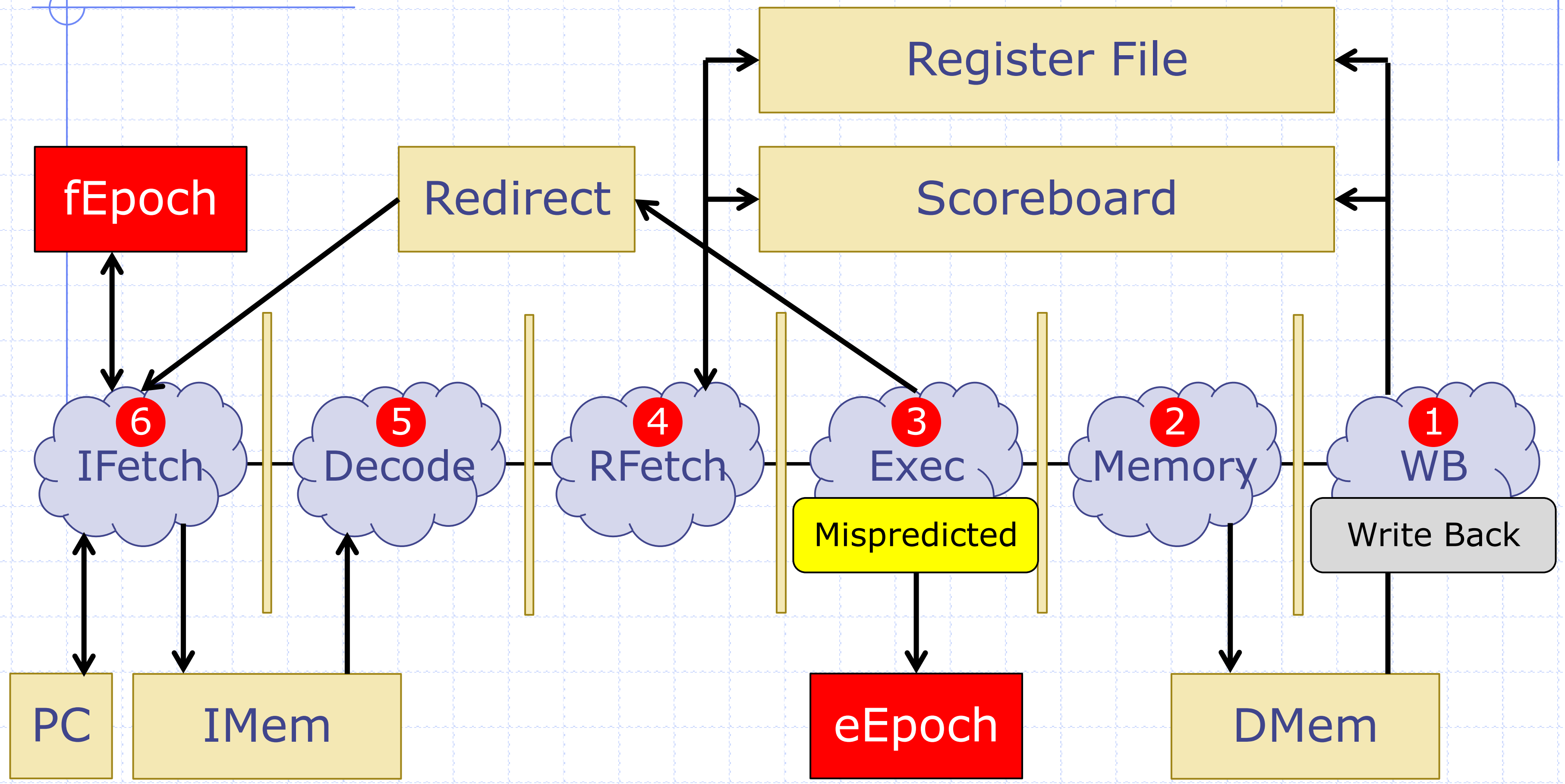
- One of the PowerPCs has all the three predictors
- Performance analysis is quite difficult – depends upon the sizes of various tables and program behavior
- Correctness: The system must work even if every prediction is wrong

Epoch Tutorial

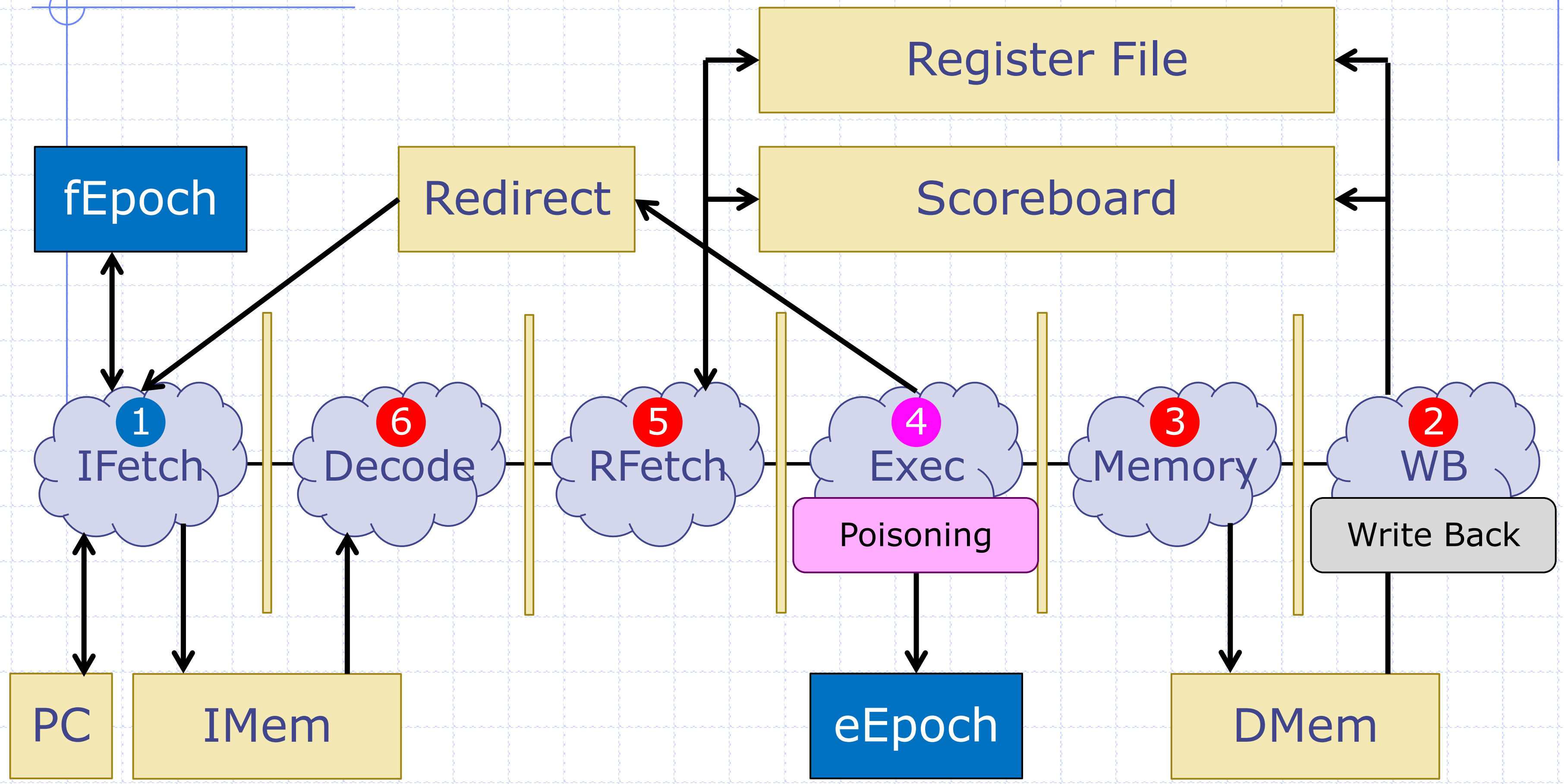
Handling Multiple Epochs

- If only one epoch changes, it acts just like the case where there is only one epoch.
- First we are going to look at the execute epoch and the decode epoch separately.

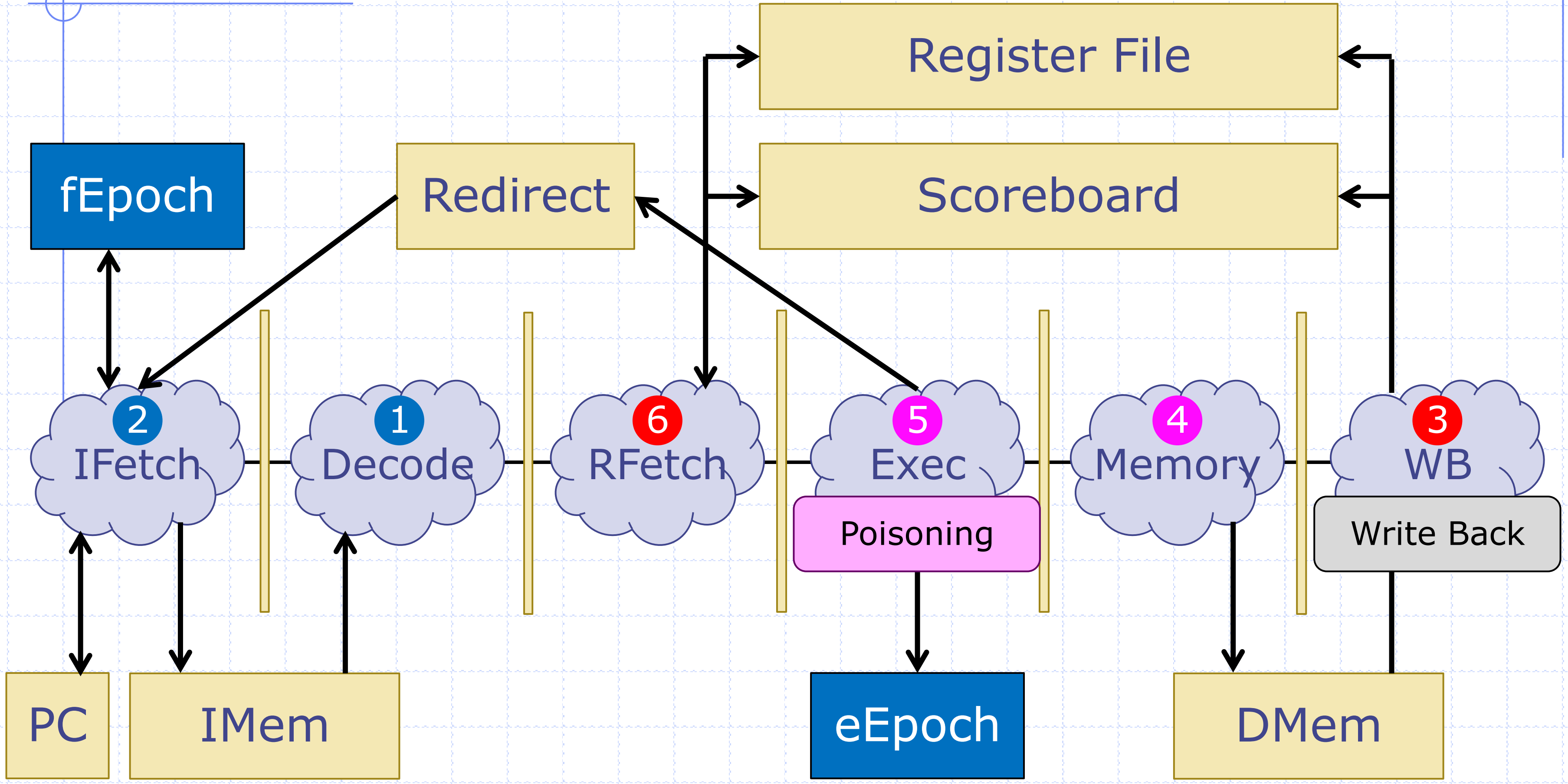
Correcting PC in Execute



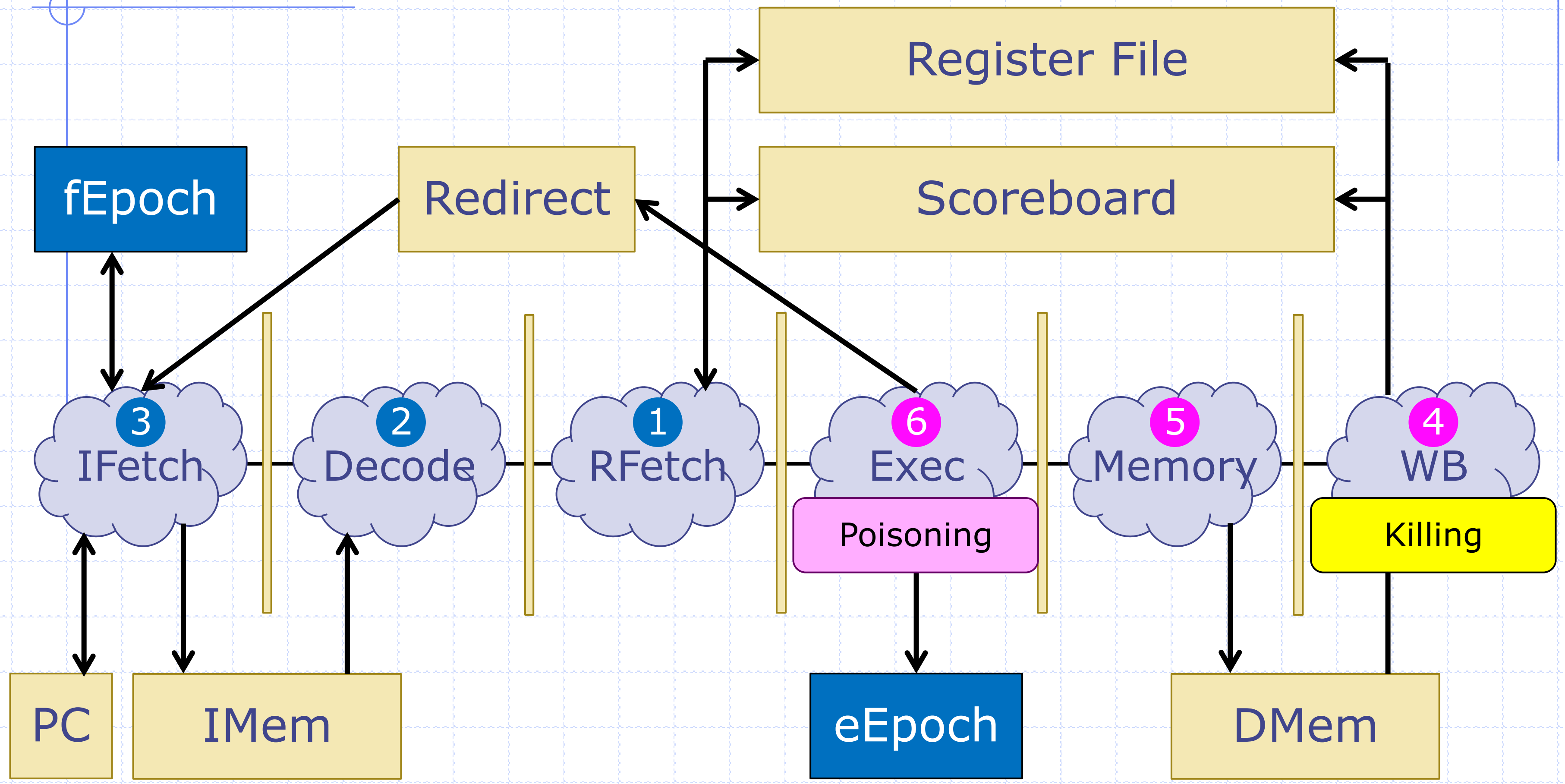
Correcting PC in Execute



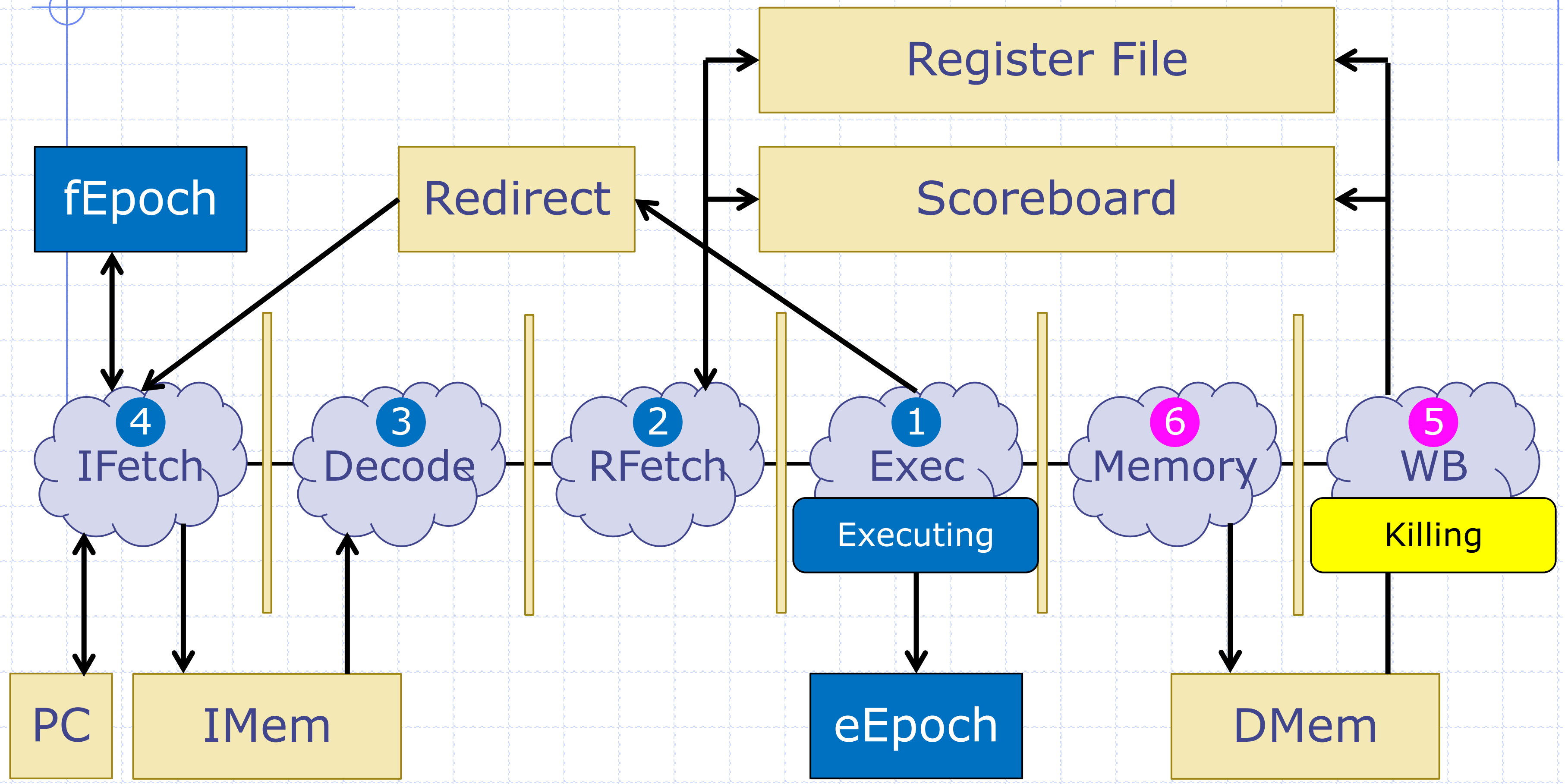
Correcting PC in Execute



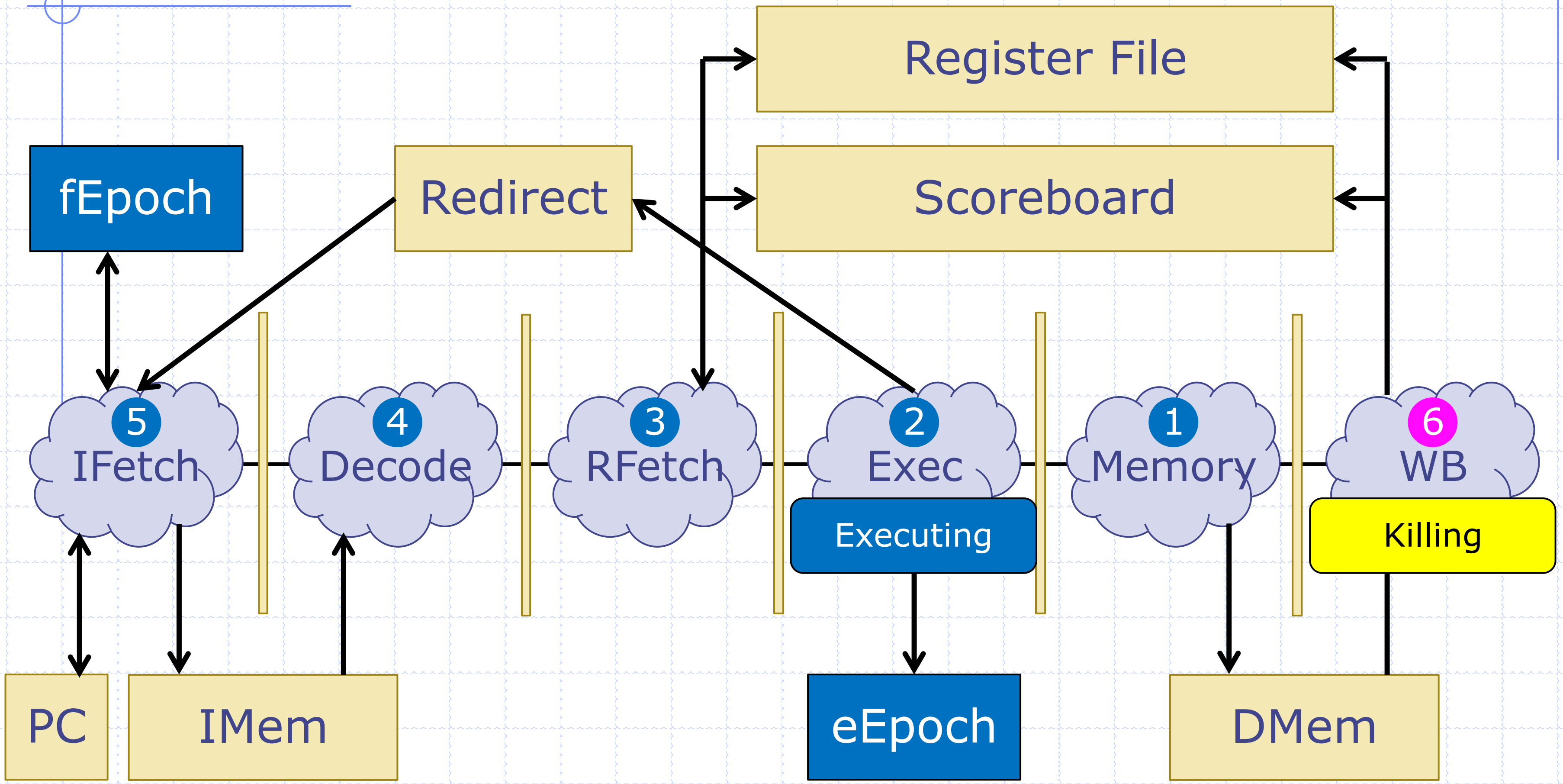
Correcting PC in Execute



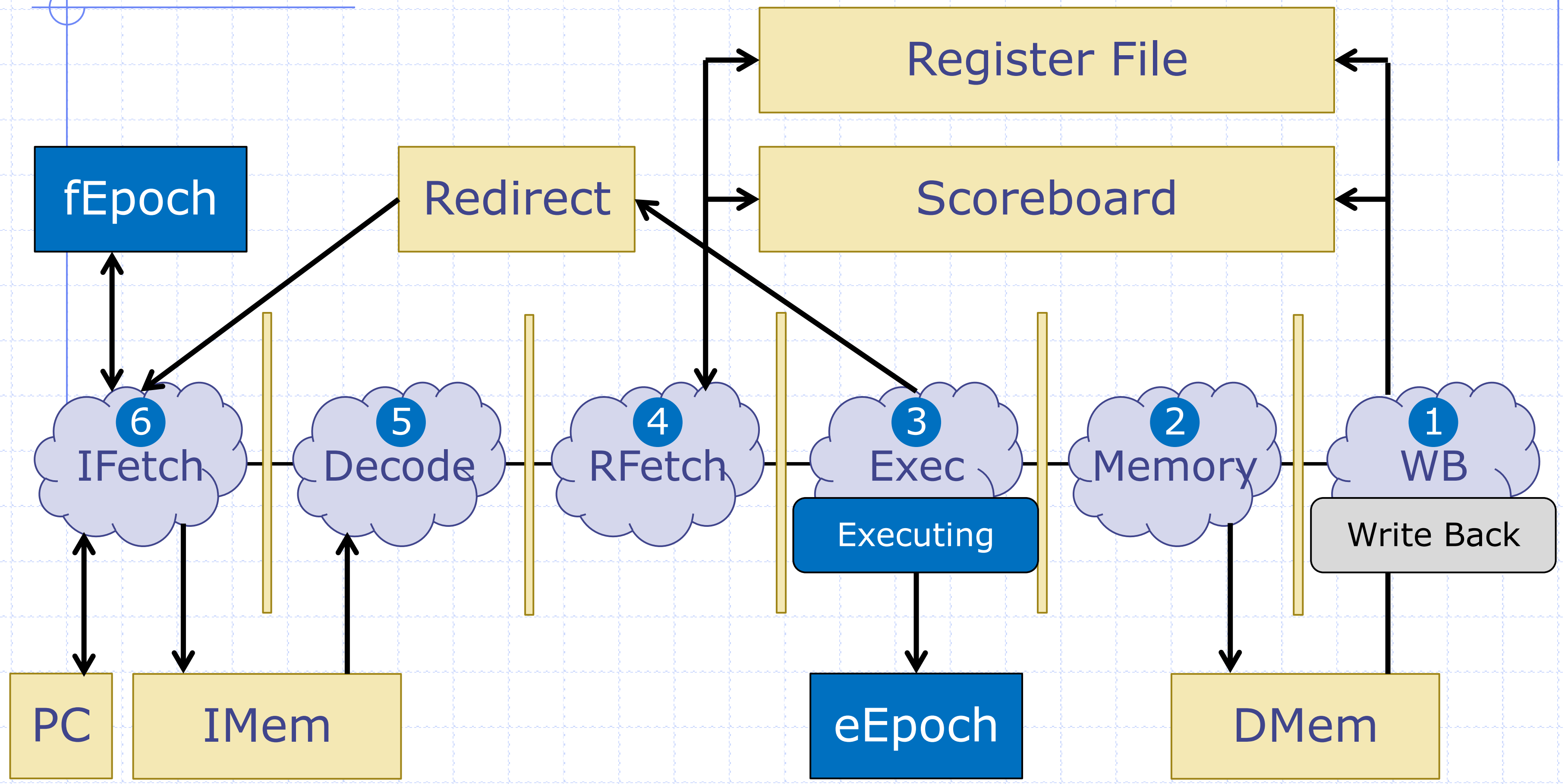
Correcting PC in Execute



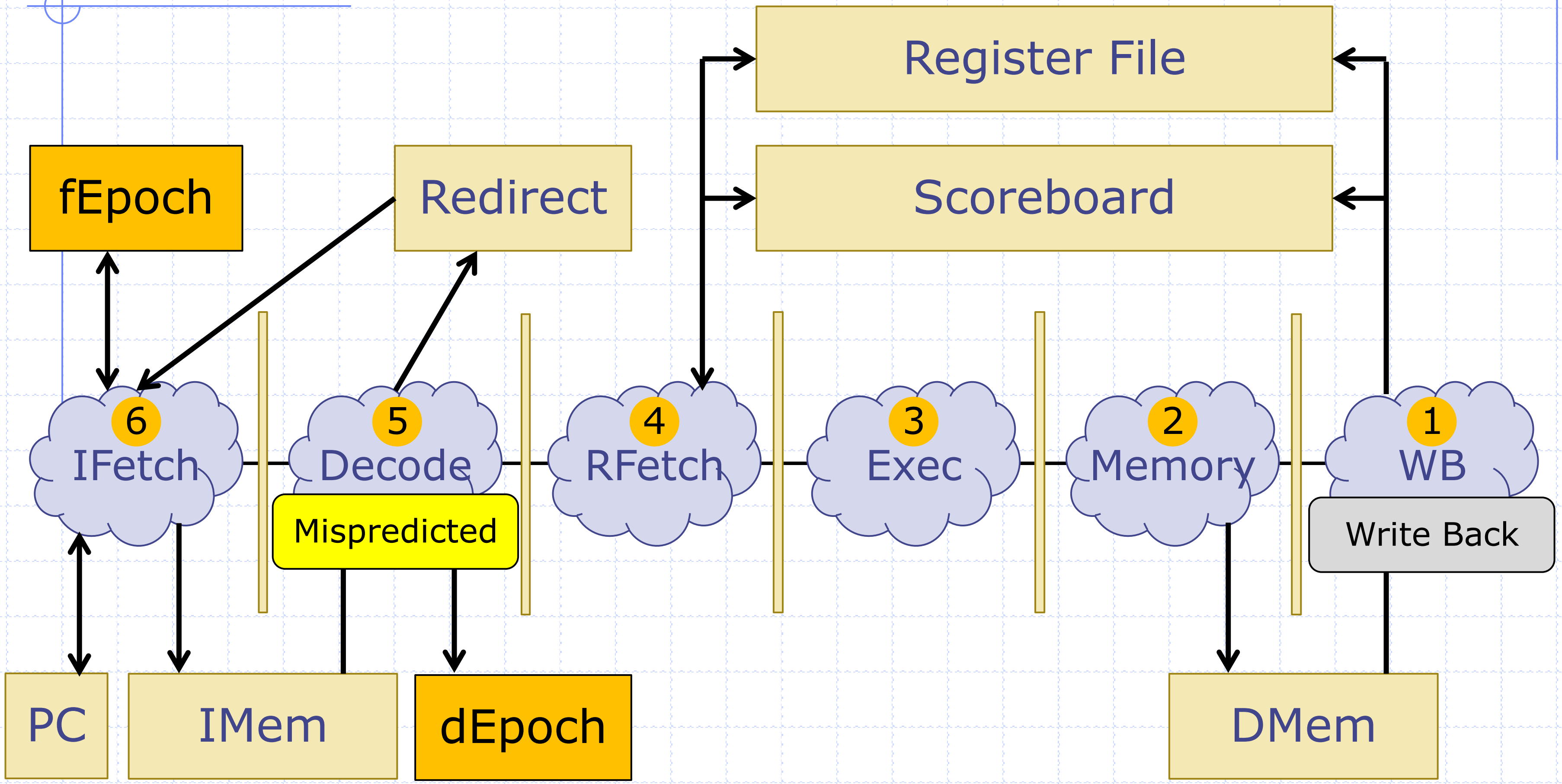
Correcting PC in Execute



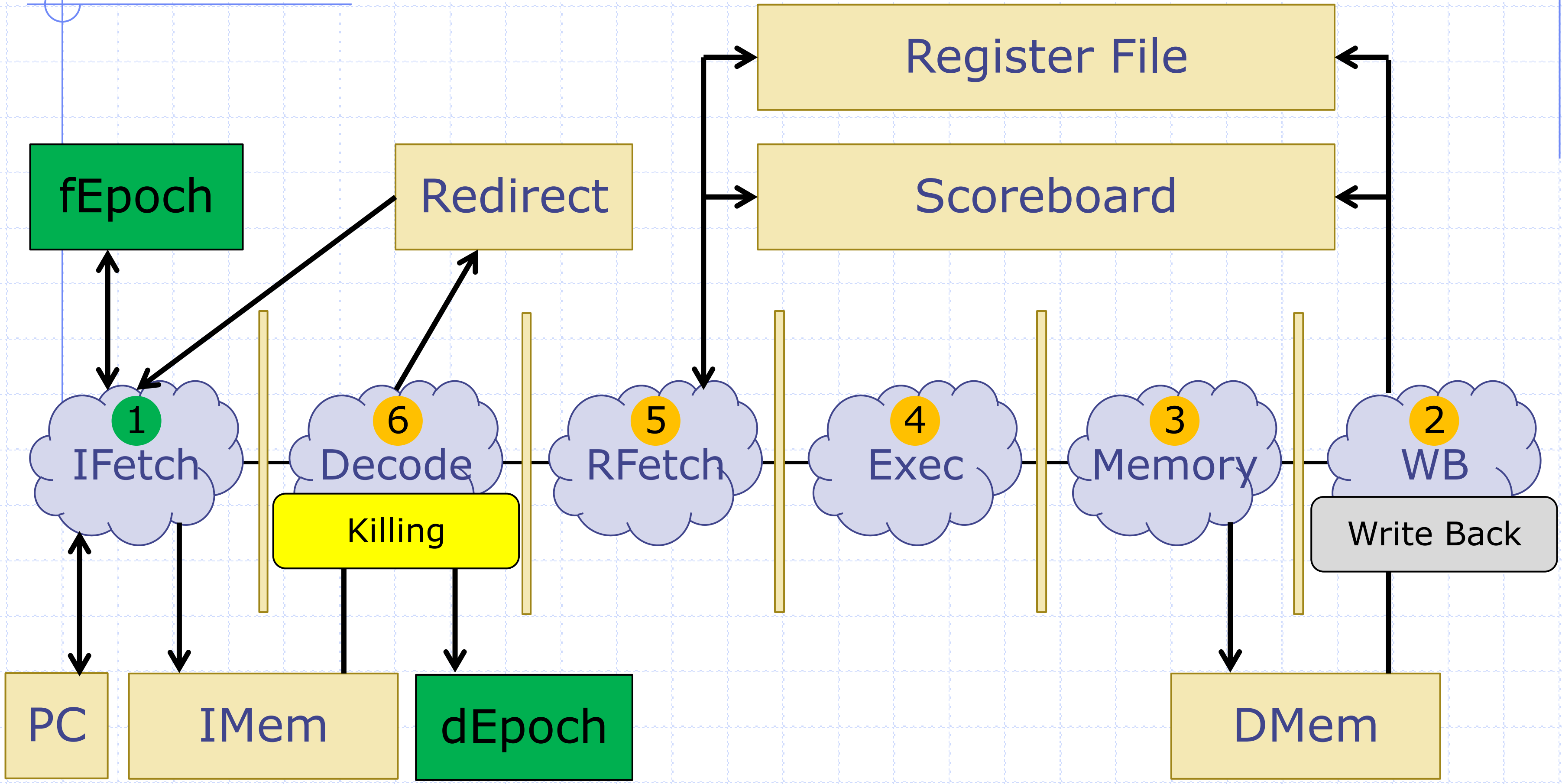
Correcting PC in Execute



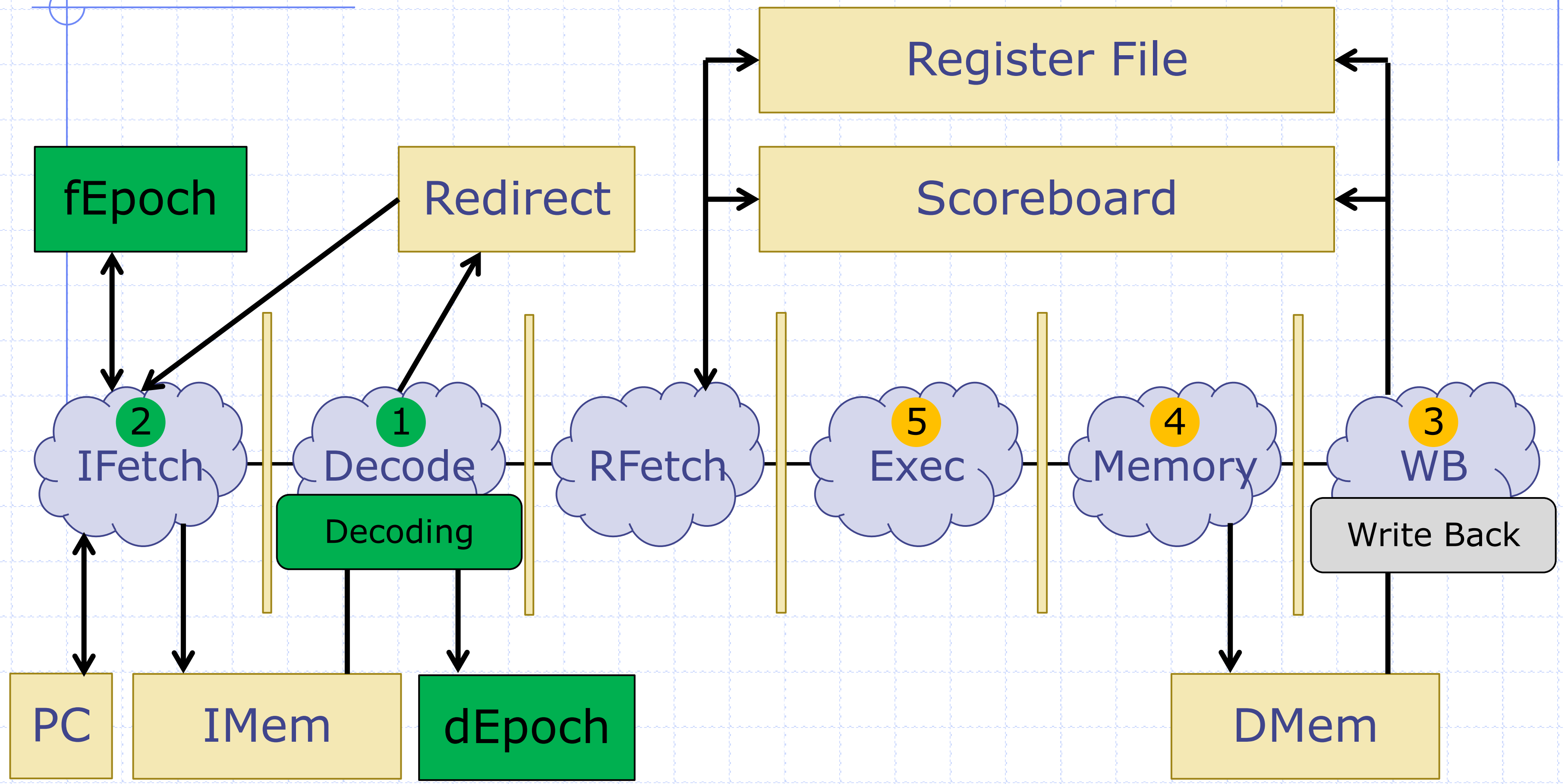
Correcting PC in Decode



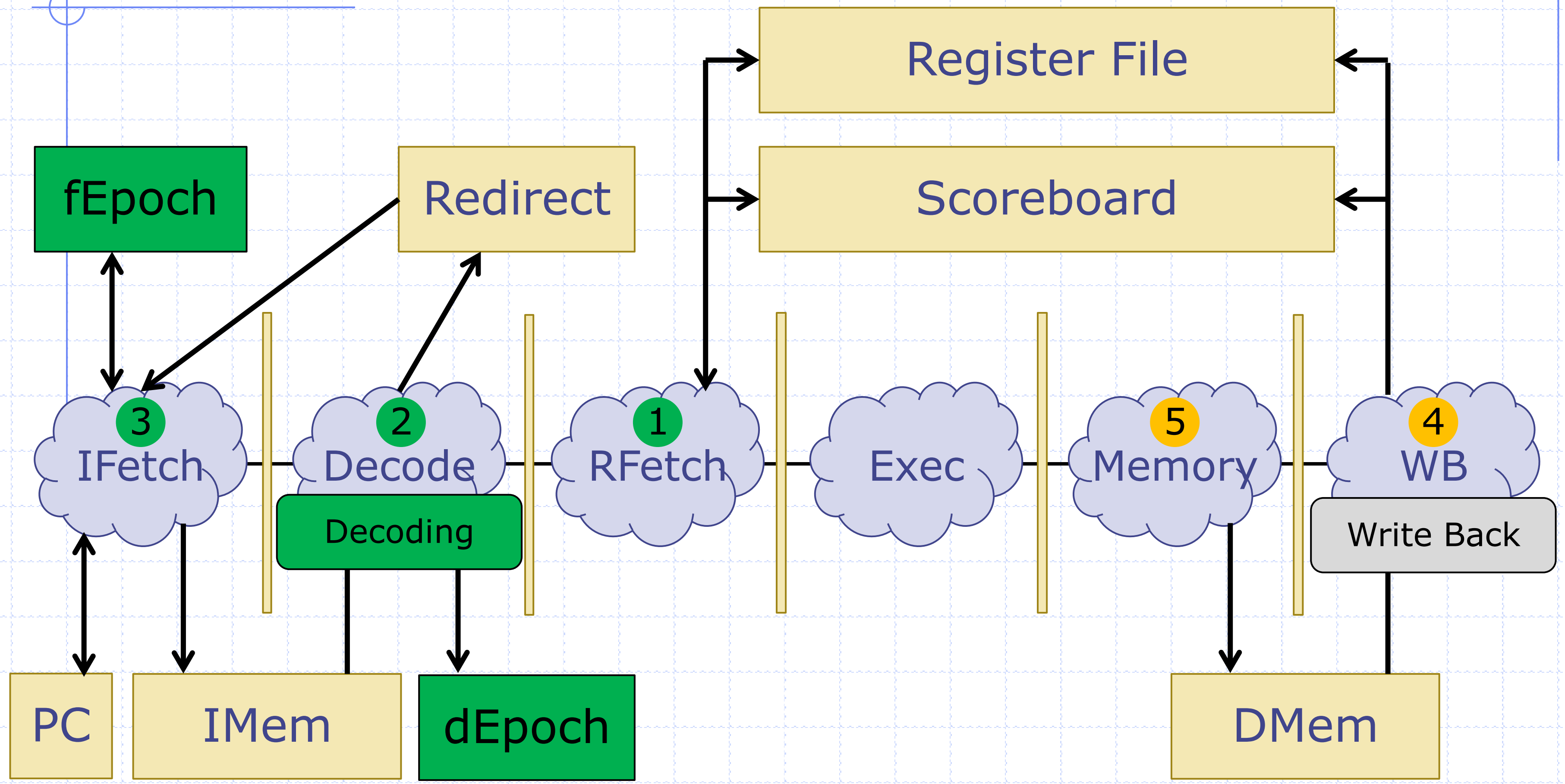
Correcting PC in Decode



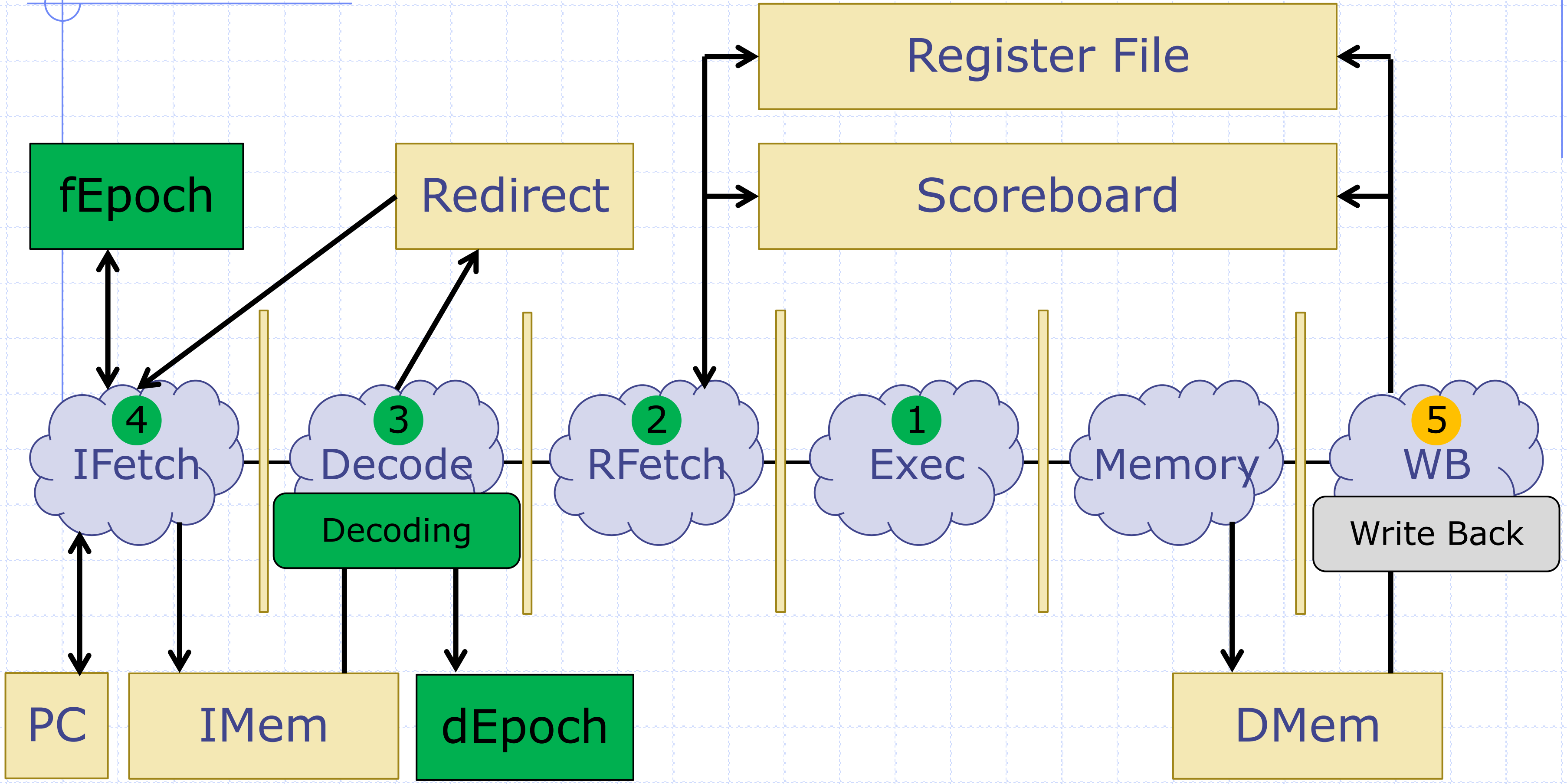
Correcting PC in Decode



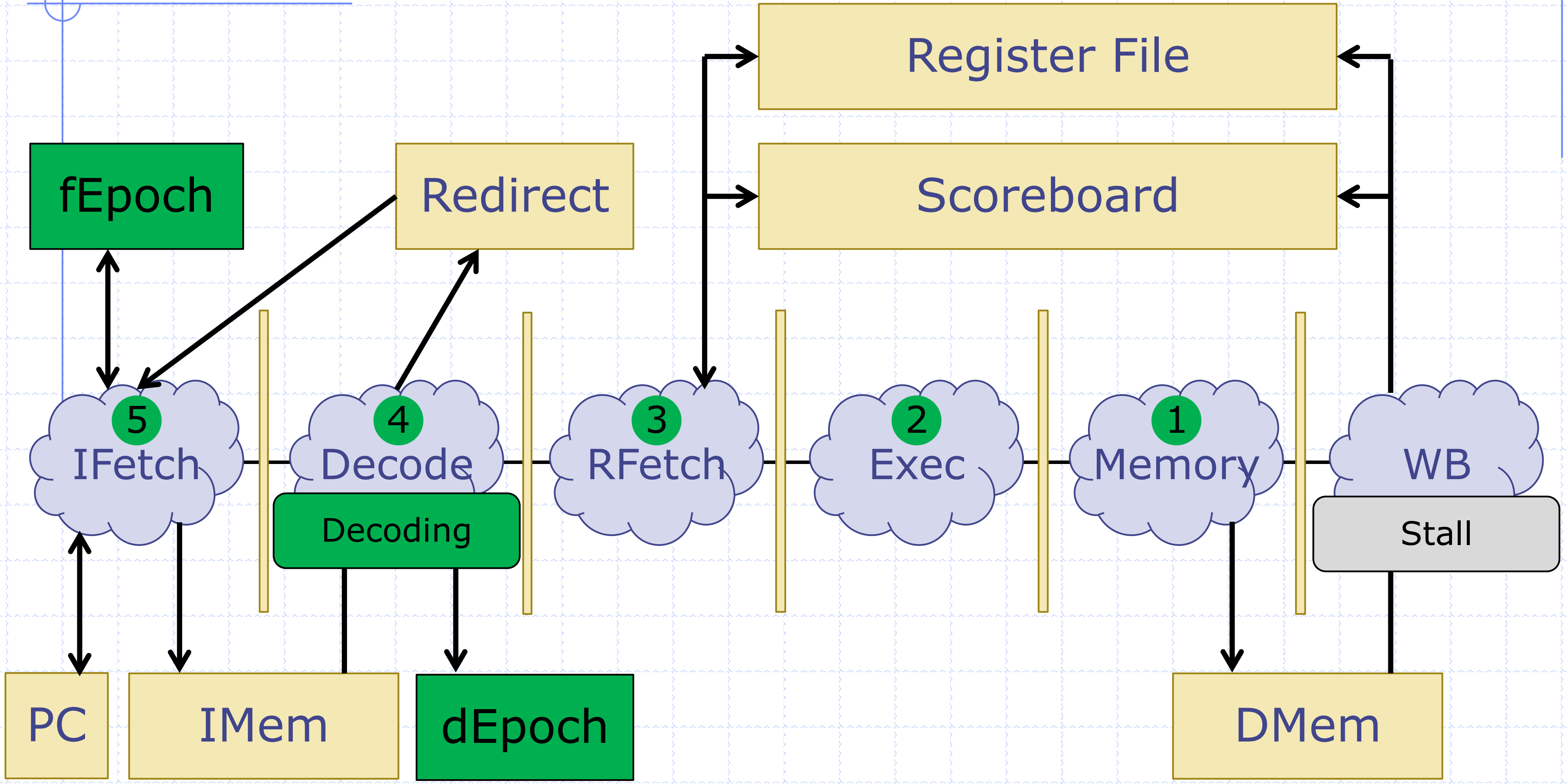
Correcting PC in Decode



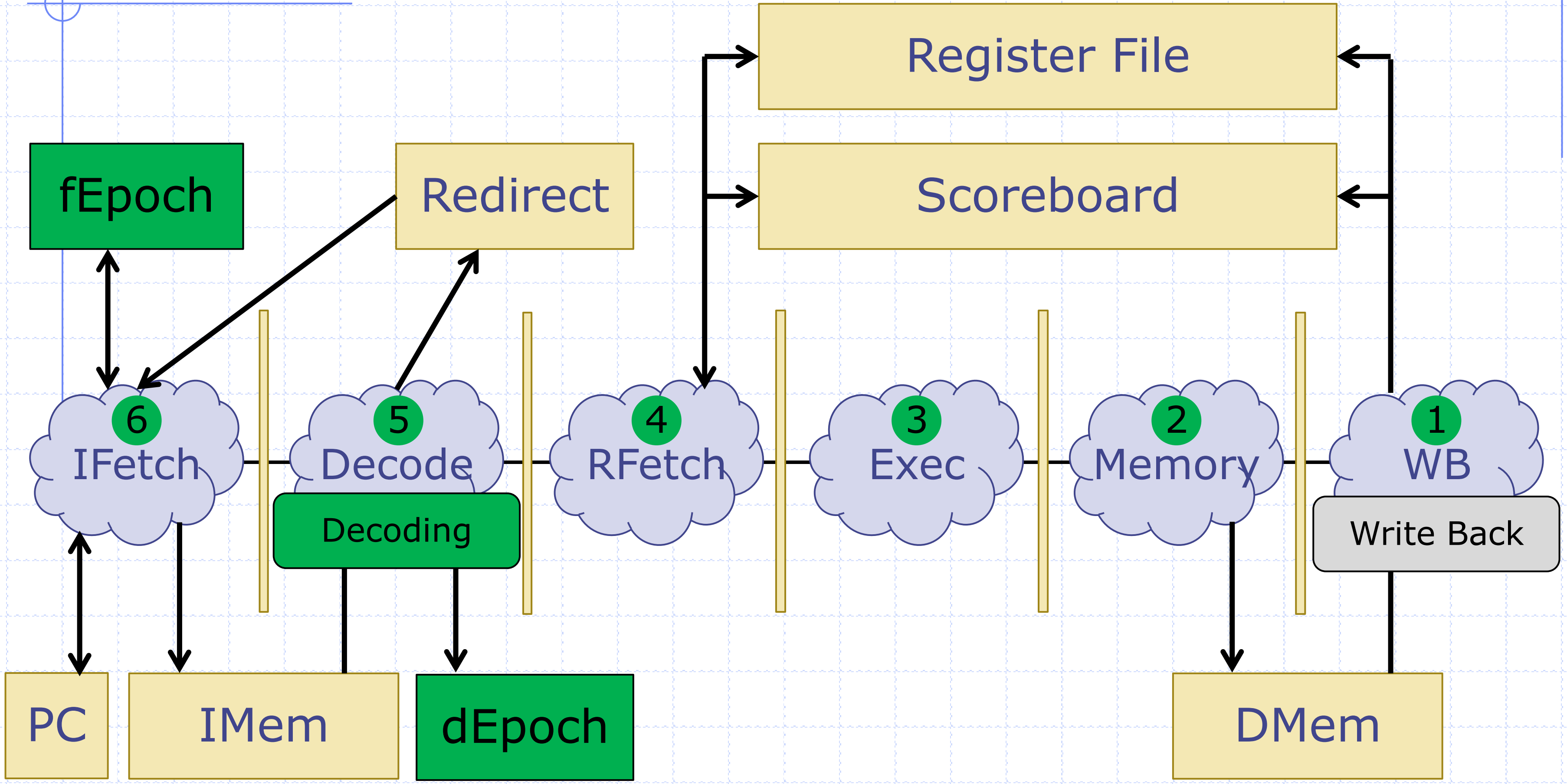
Correcting PC in Decode



Correcting PC in Decode

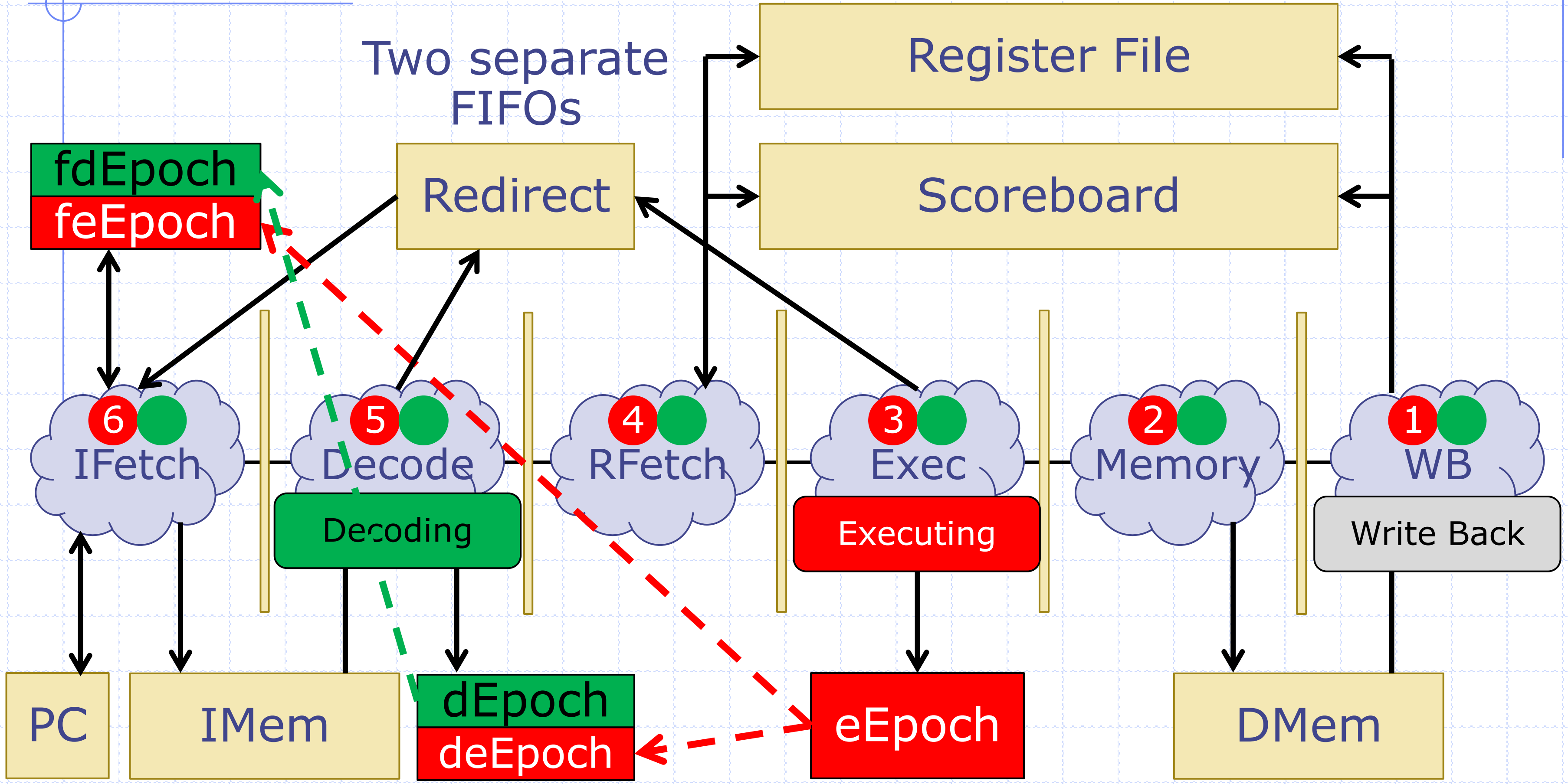


Correcting PC in Decode



Correcting PC in Both Decode and Execute

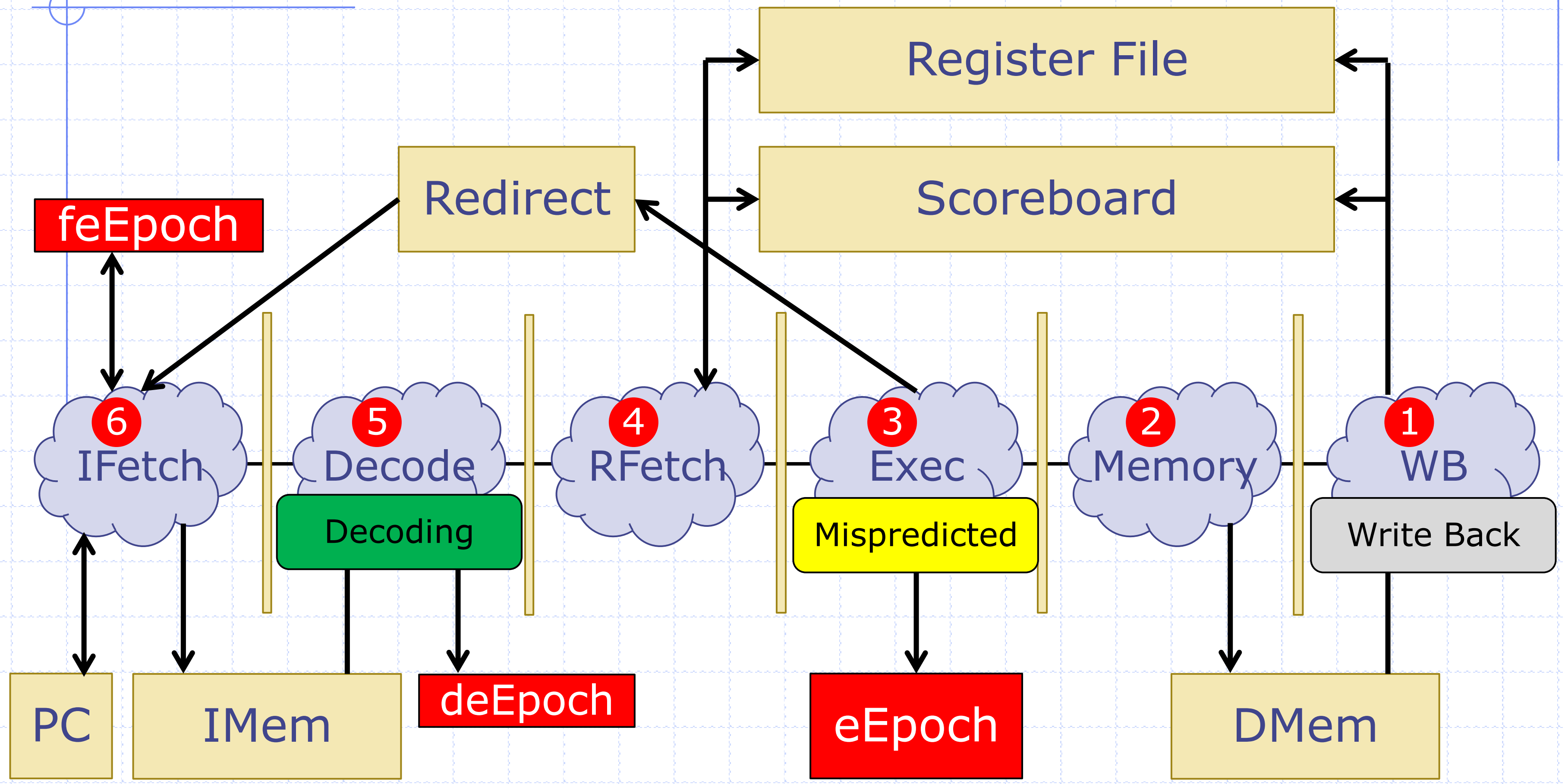
Correcting PC in Both Decode and Execute



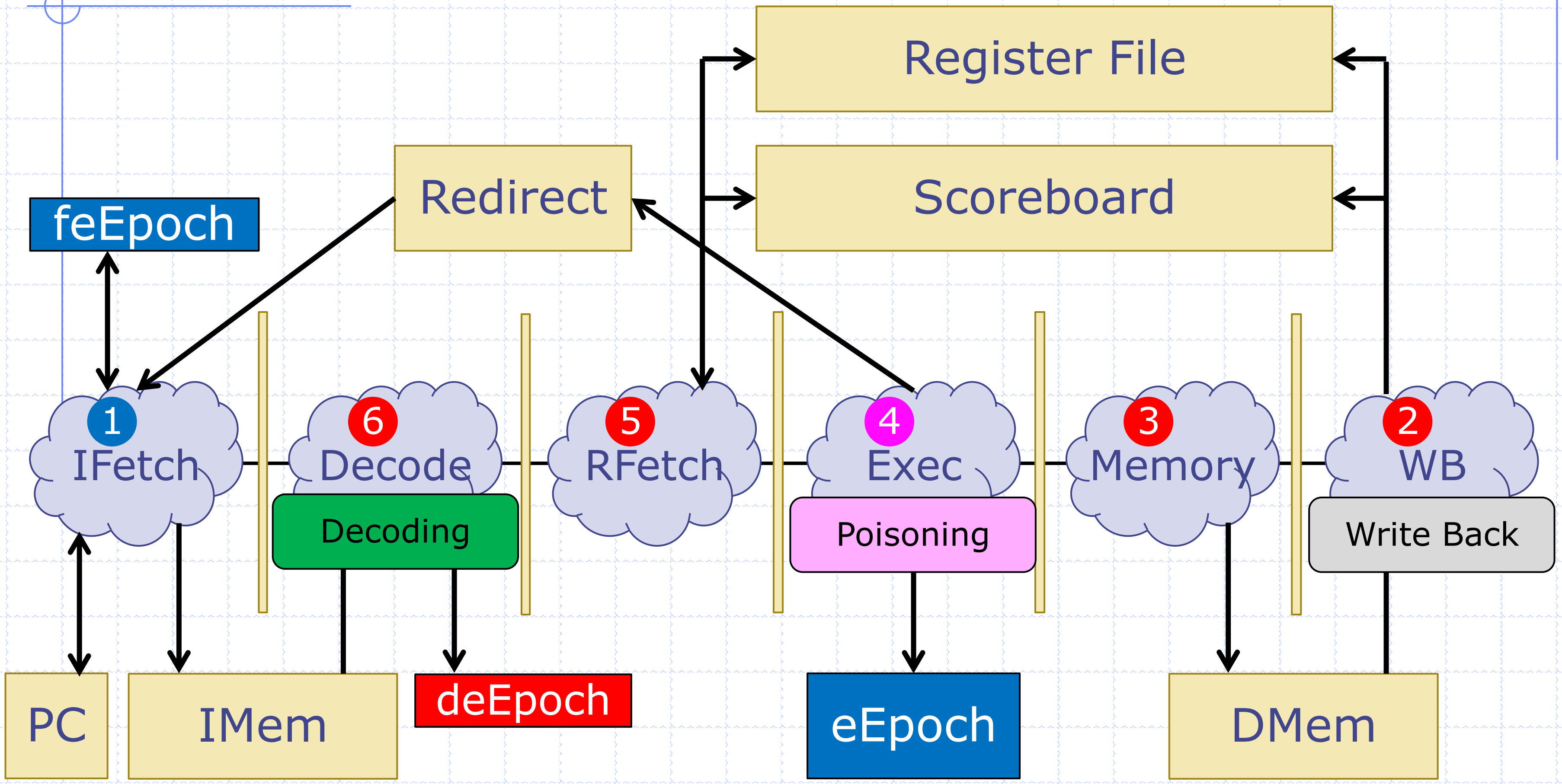
Fetch has local estimates of eEpoch and dEpoch

Decode has a local estimate of eEpoch ← How does this work?

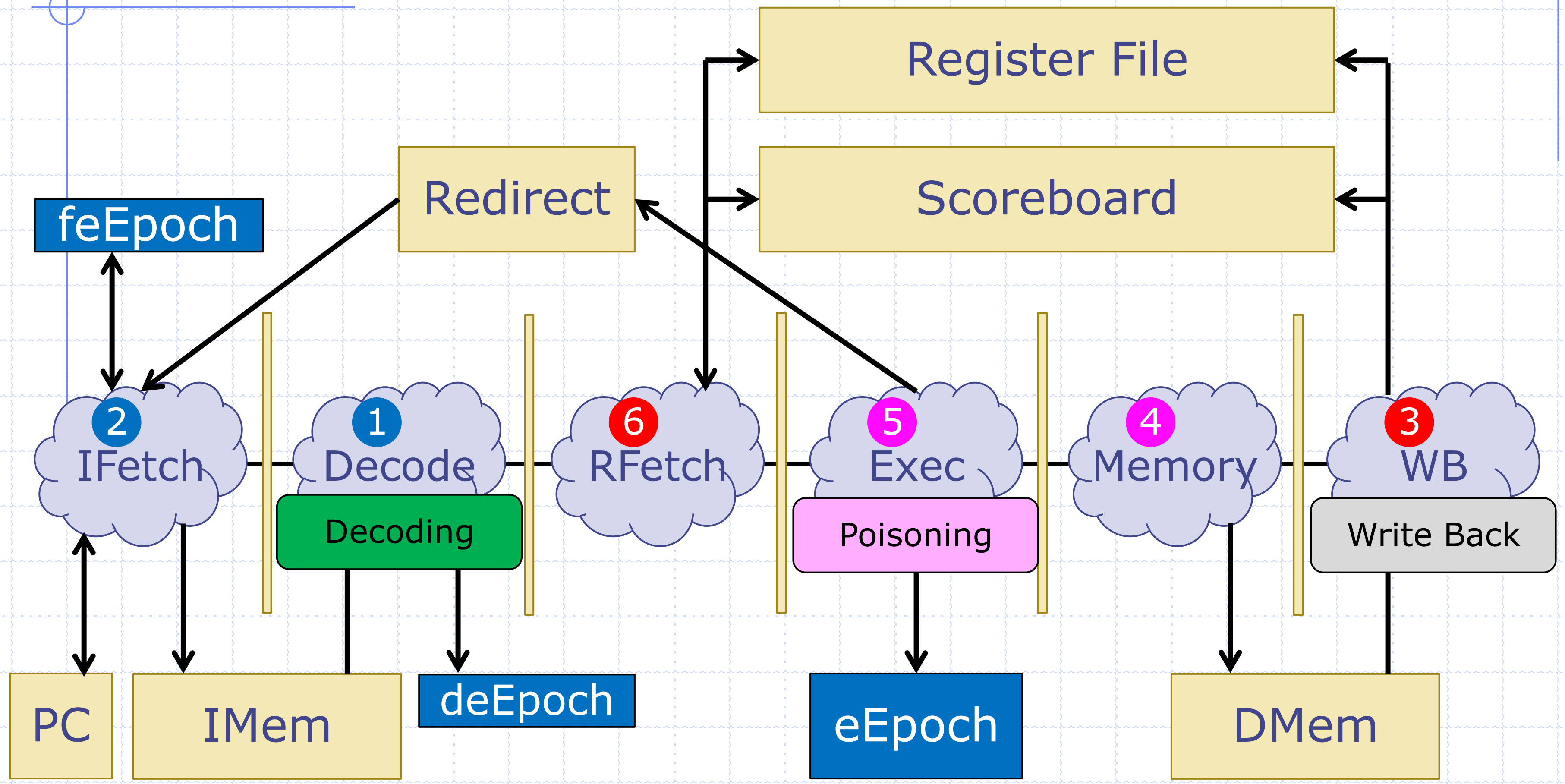
Estimating eEpoch in Decode



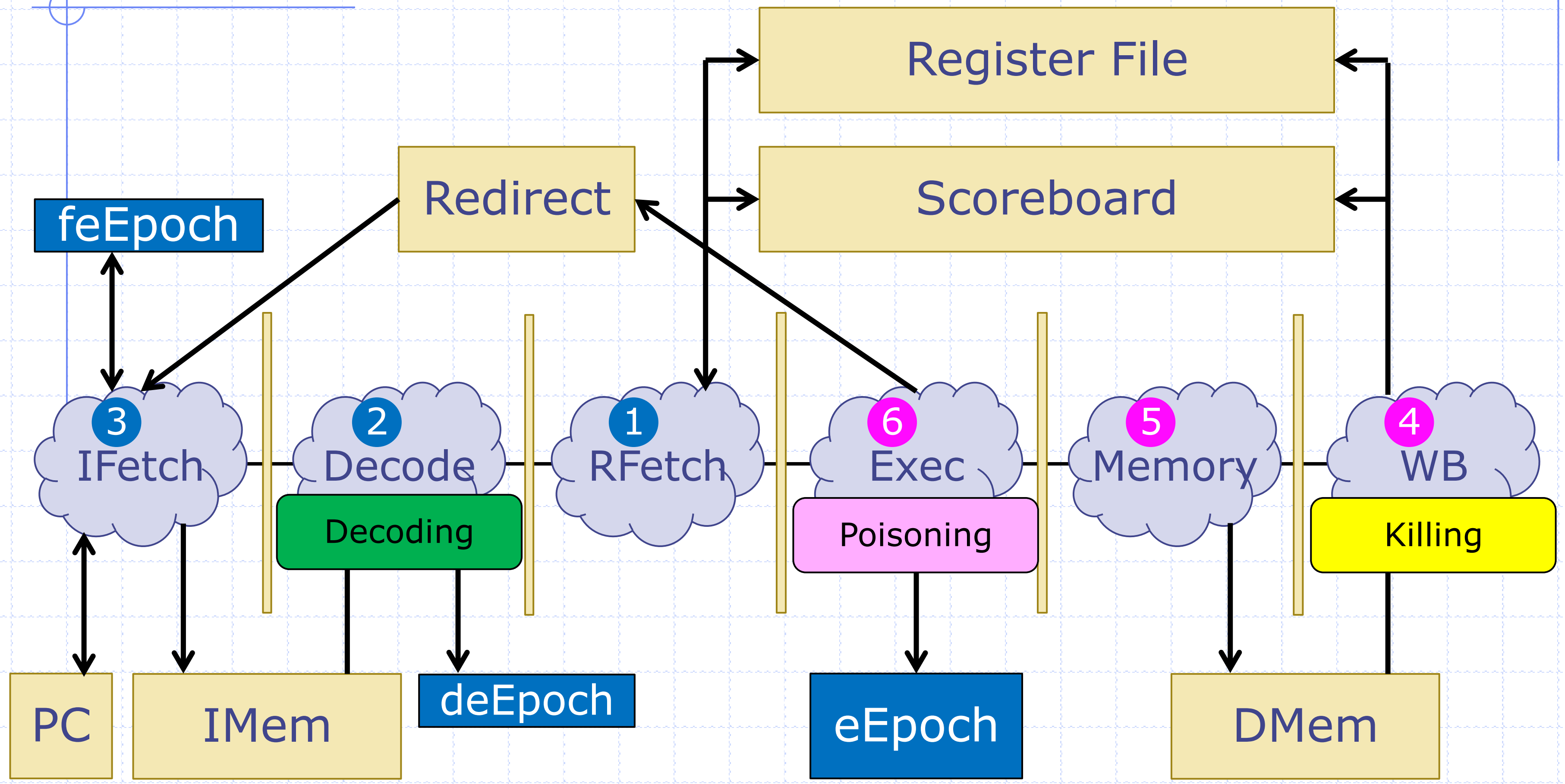
Estimating eEpoch in Decode



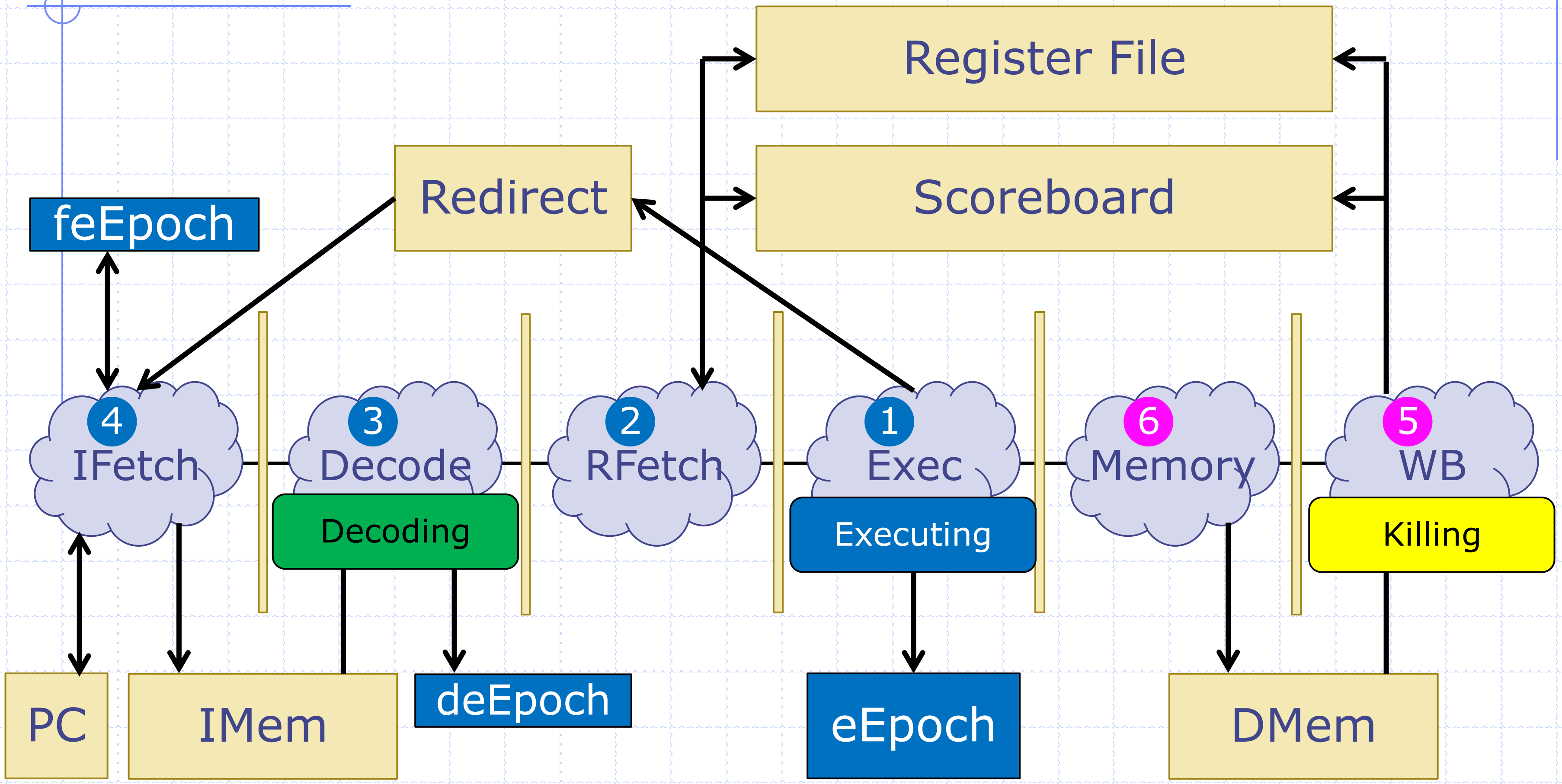
Estimating eEpoch in Decode



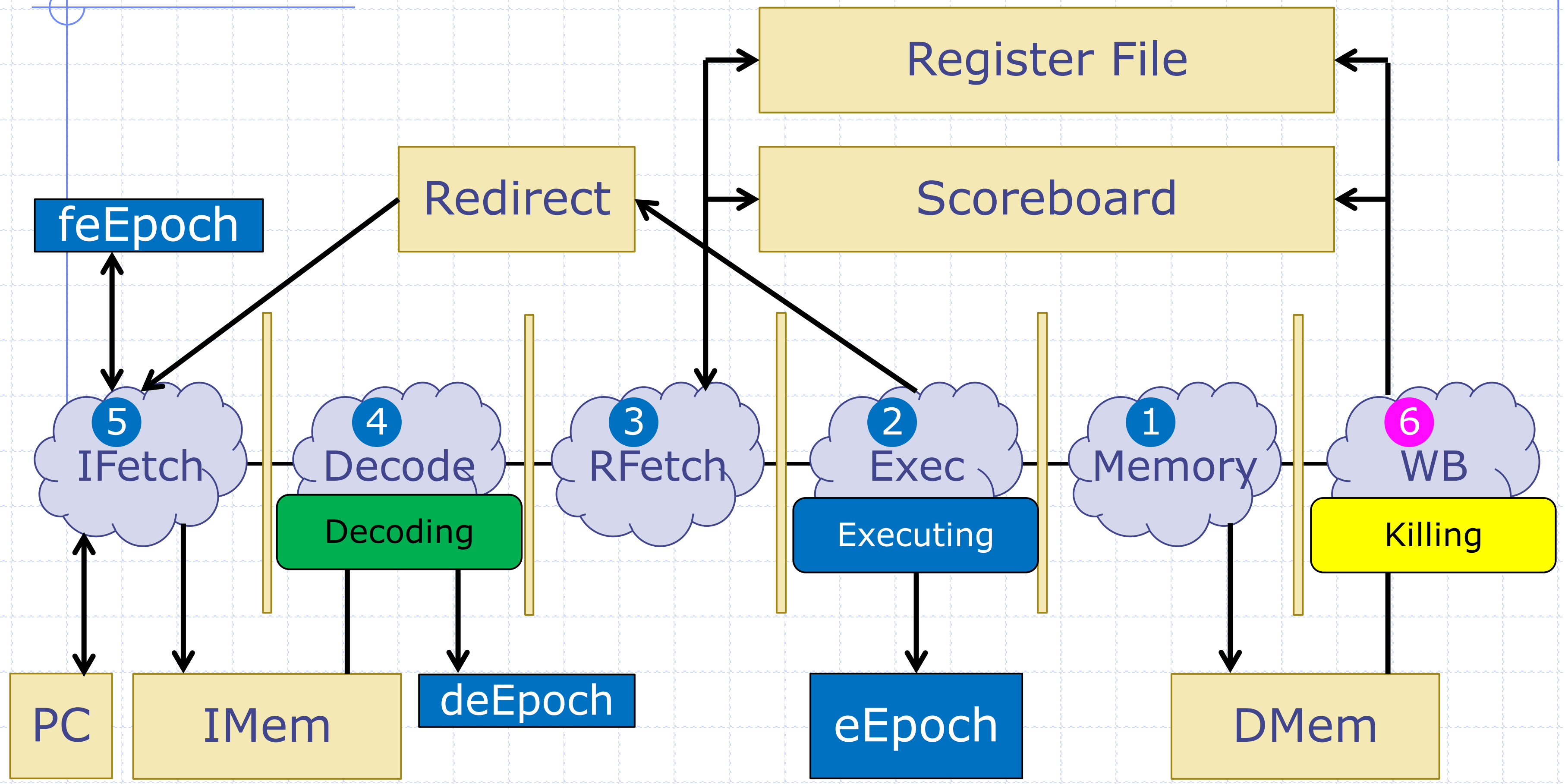
Estimating eEpoch in Decode



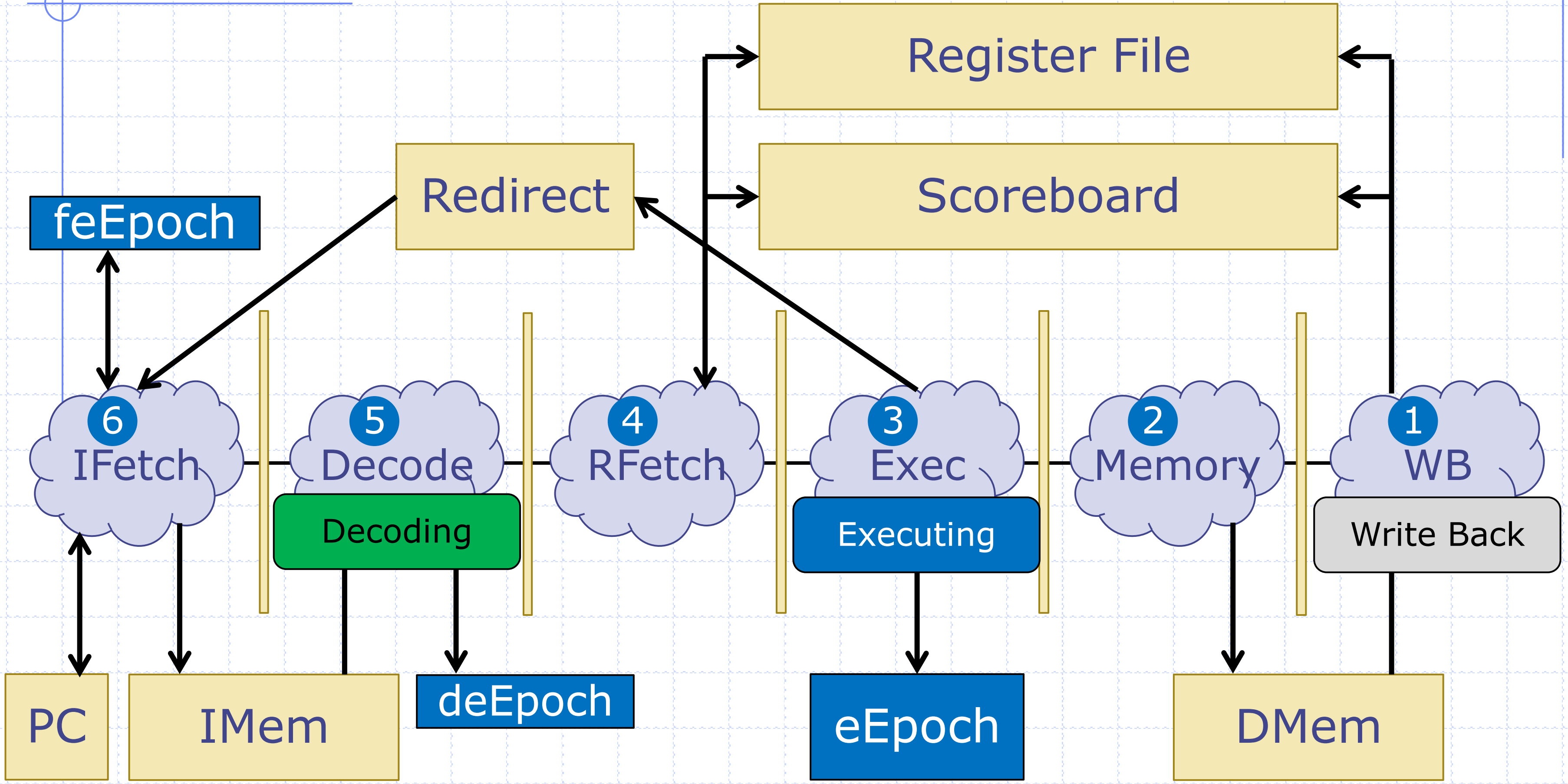
Estimating eEpoch in Decode



Estimating eEpoch in Decode



Estimating eEpoch in Decode



Estimating eEpoch in Decode

- Decode has no way to know what the execute epoch really is.
  - Its best guess is whatever execute epoch is coming in.
  - It only keeps track of the old epoch to know if there was a change.
    - In that case, it needs to change its decode epoch to match the incoming instruction

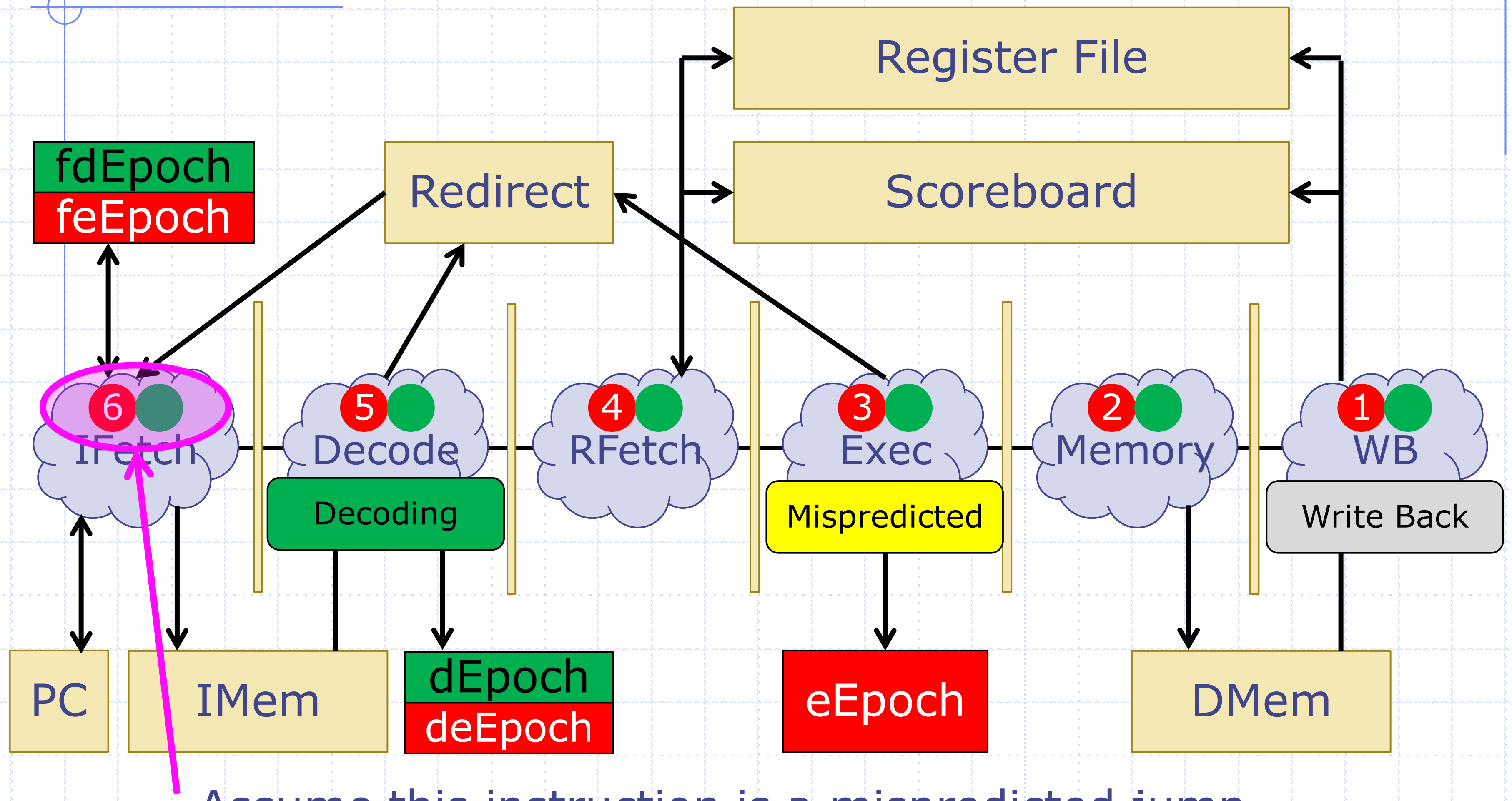
Correcting PC from Multiple Stages Concurrently

- What if decode and execute see mispredictions in the same cycle?
  - If execute sees a misprediction, then the decode instruction is a wrong path instruction. The redirect coming from decode should be ignored by the fetch stage.

Redirection in Execute then Decode

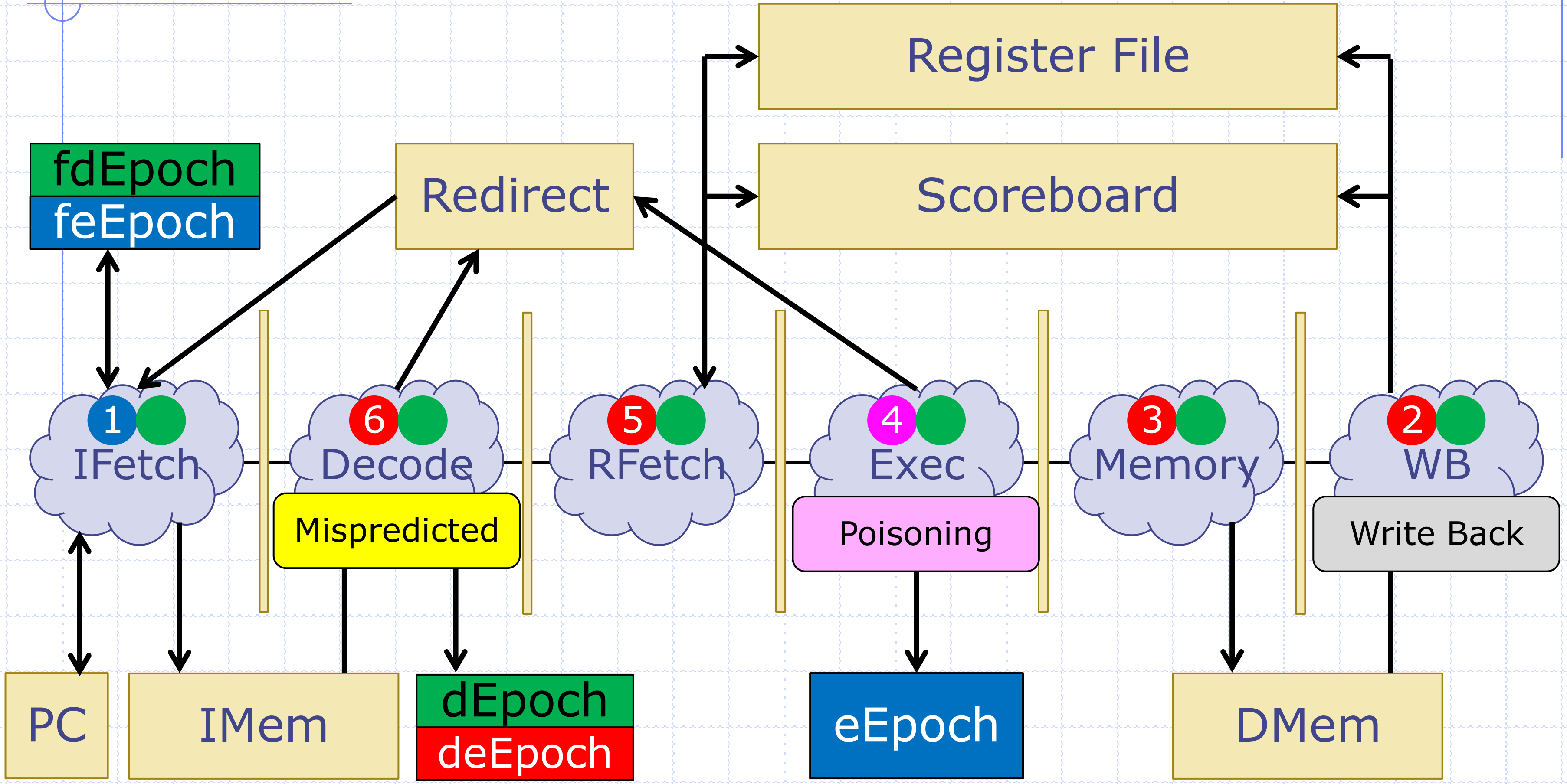
- What if execute sees a misprediction, then decode sees one in the next cycle?
  - The decode instruction will be a wrong path instruction, but the decode stage will not no it, so it will send a redirect message. The fetch stage should ignore this message.

Redirection in Execute then Decode



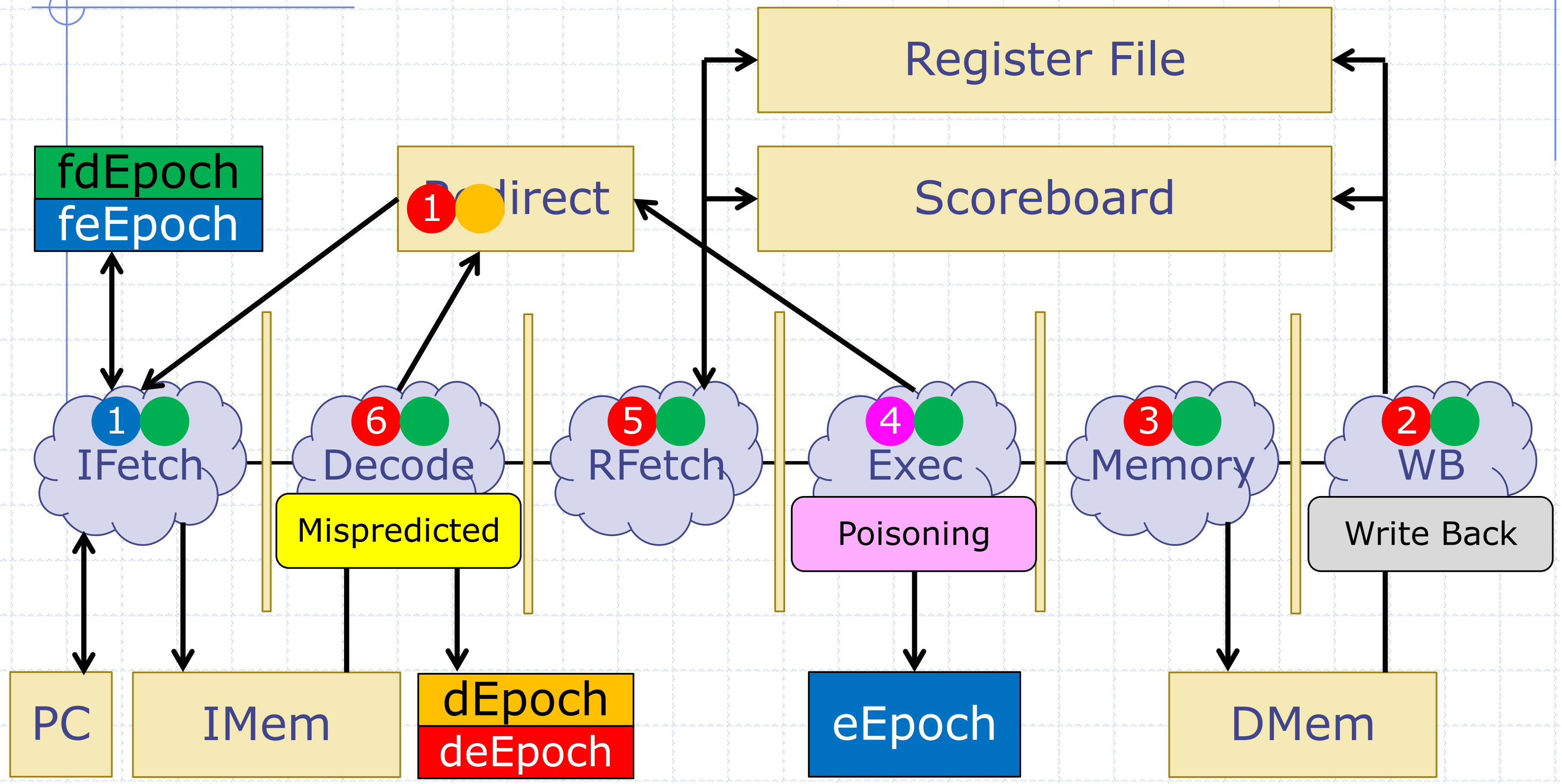
Assume this instruction is a mispredicted jump instruction. It will be in the decode stage next cycle

Redirection in Execute then Decode



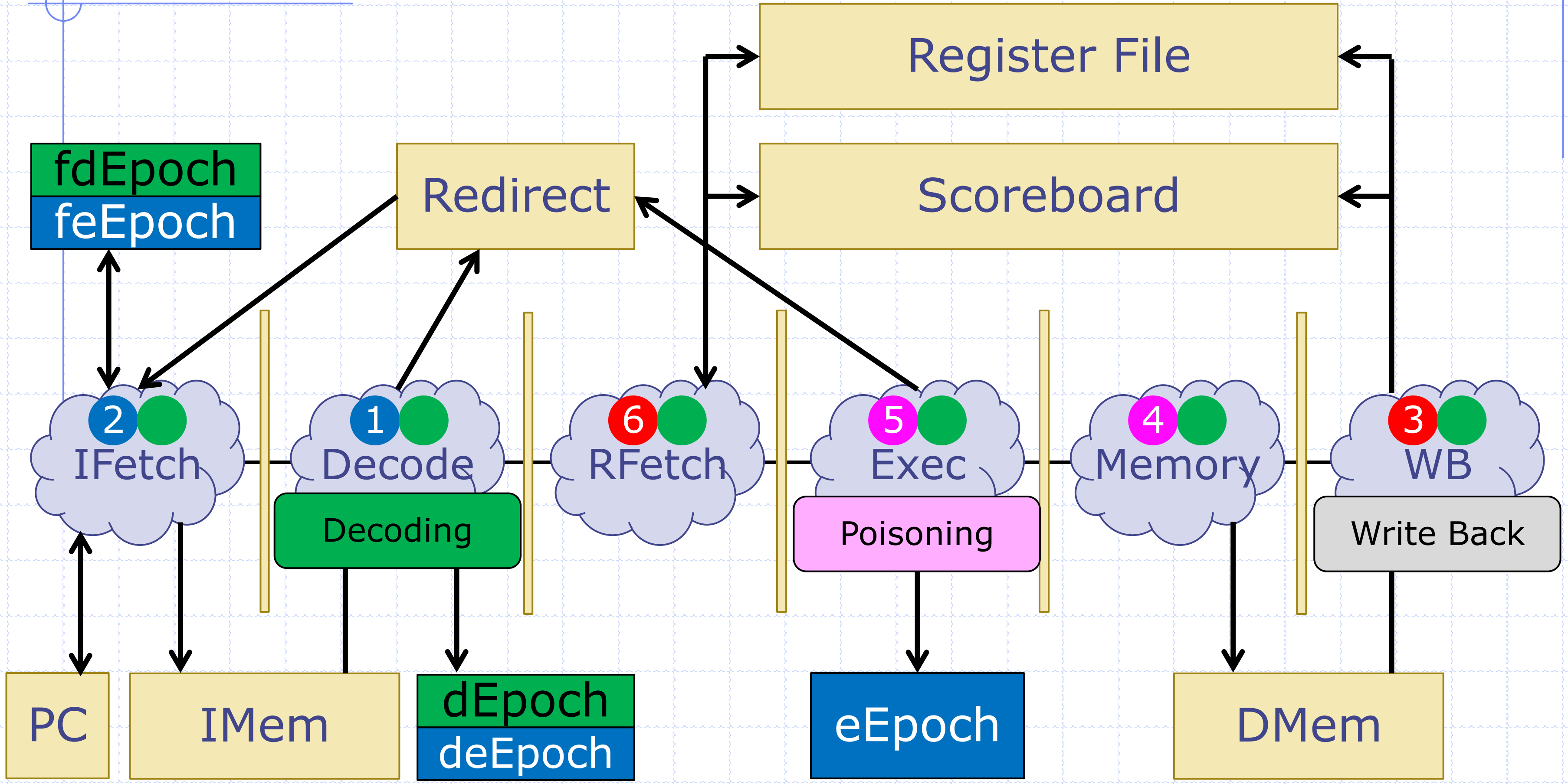
The decode stage's estimate of eEpoch is old, so it isn't able to recognize it is decoding a wrong path instruction.

Redirection in Execute then Decode



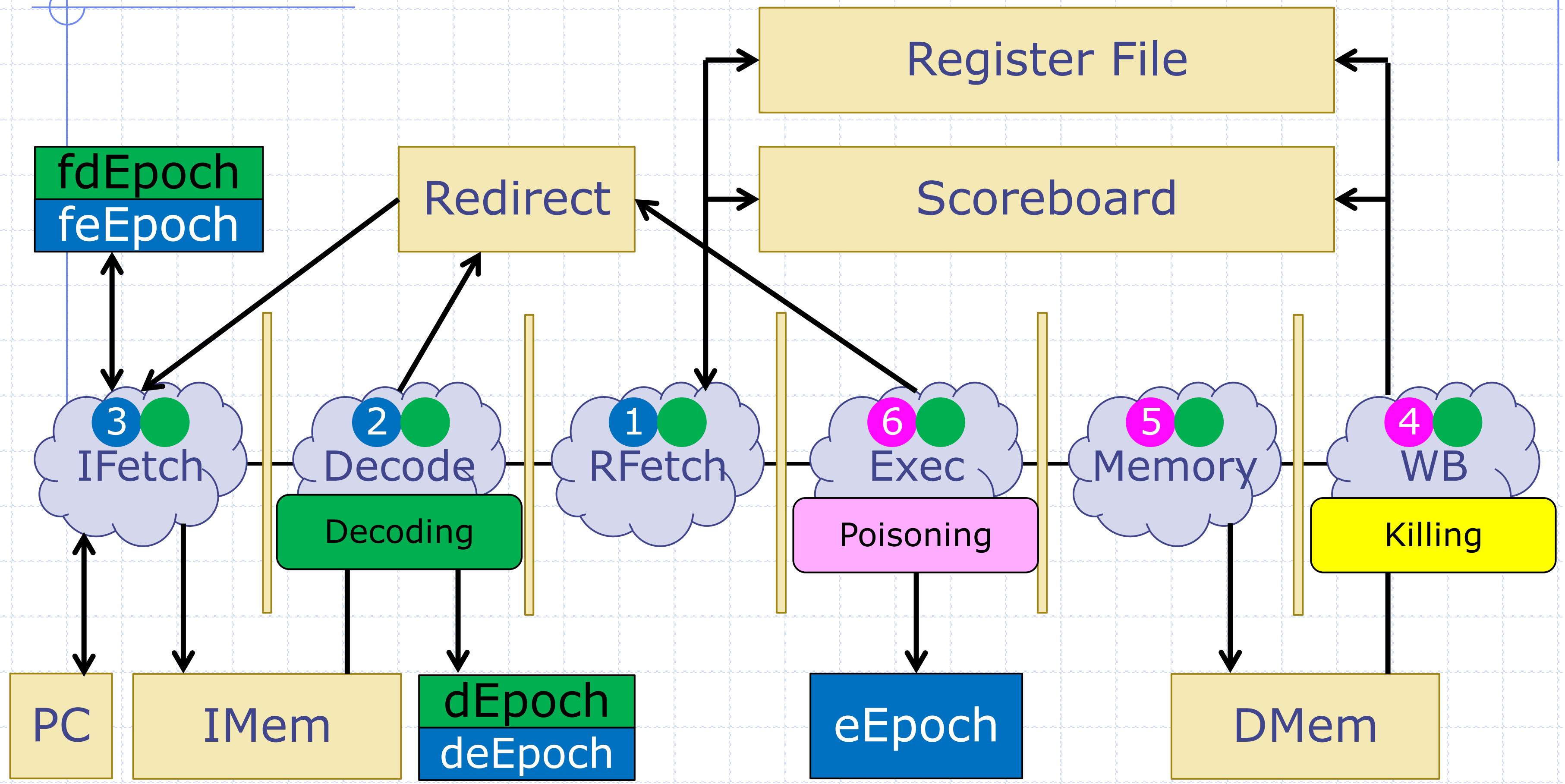
Fetch will remove the redirection from the redirect FIFO without changing the PC because the execute epochs don’t match.

Redirection in Execute then Decode

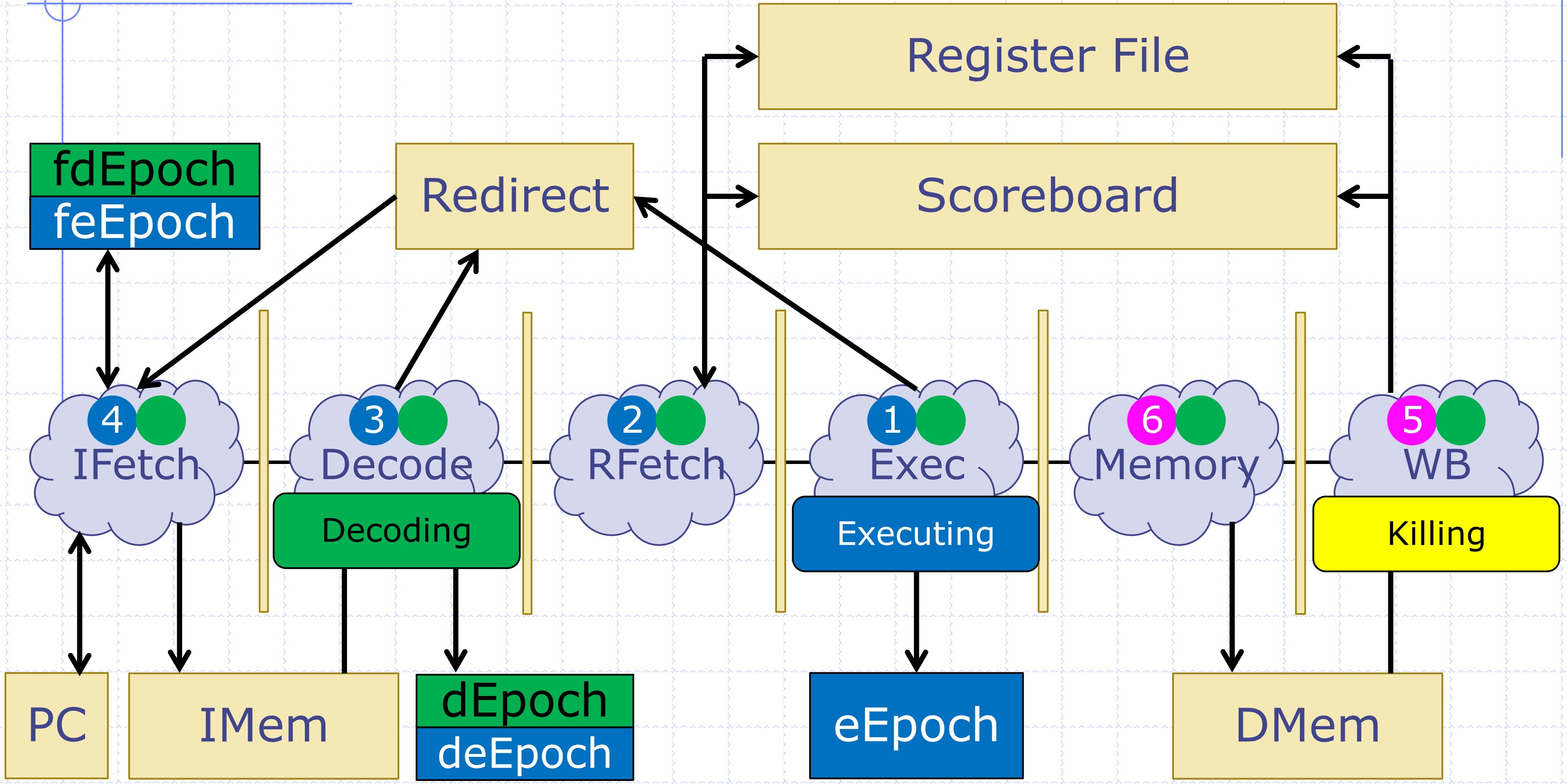


The Decode stage sees the change in the execute epoch and corrects its decode epoch to match the incoming instruction

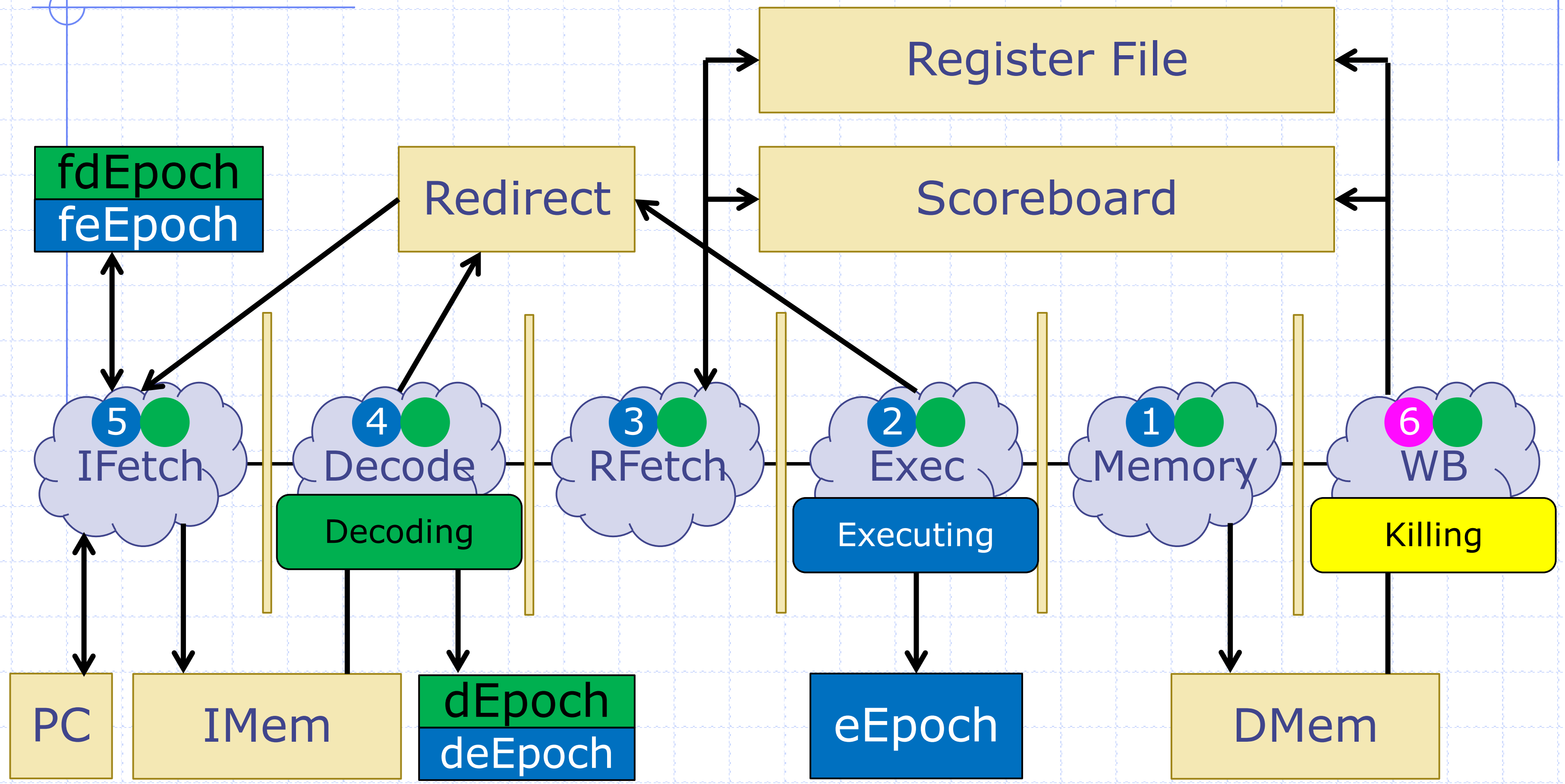
Redirection in Execute then Decode



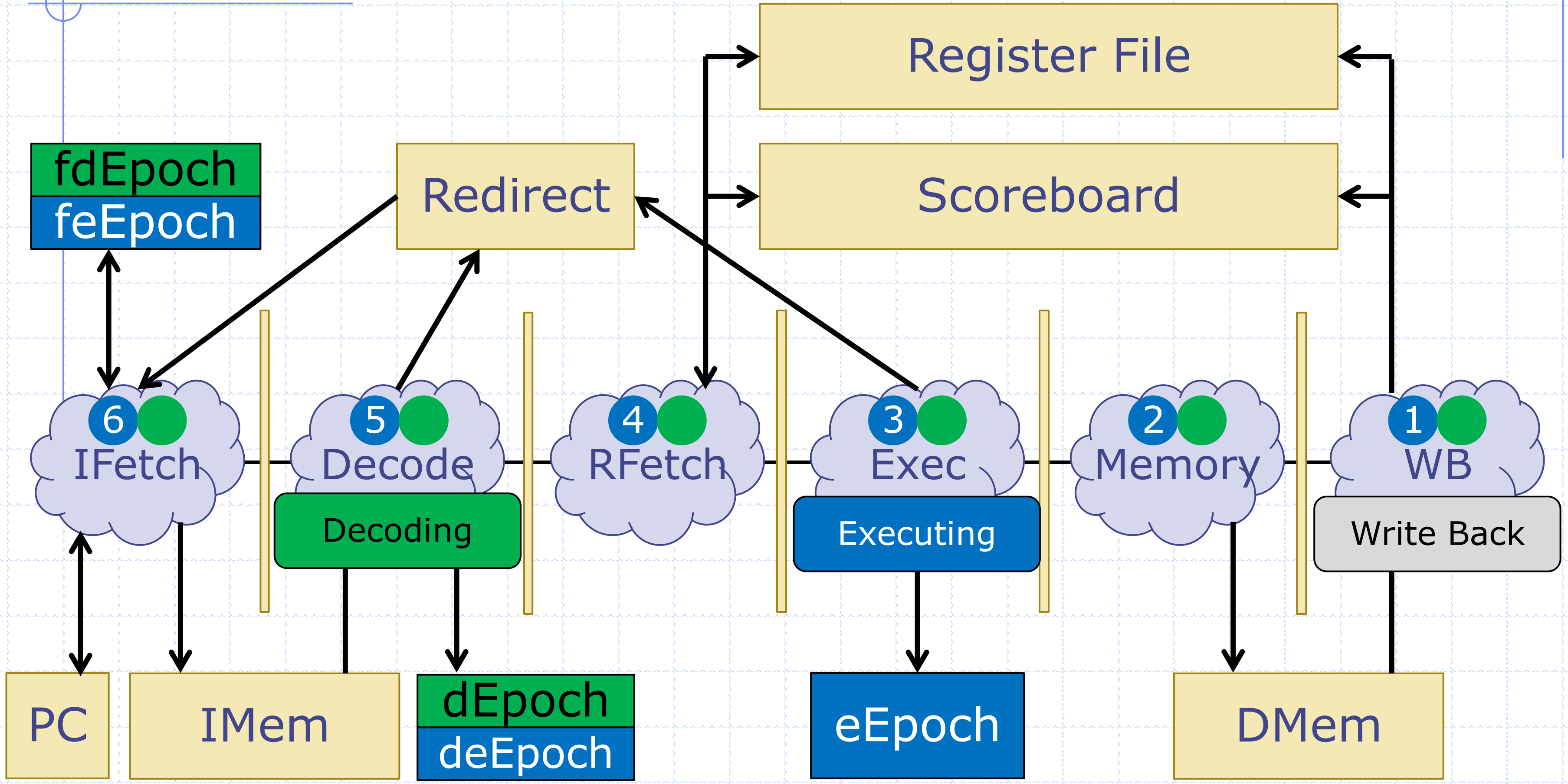
Redirection in Execute then Decode



Redirection in Execute then Decode



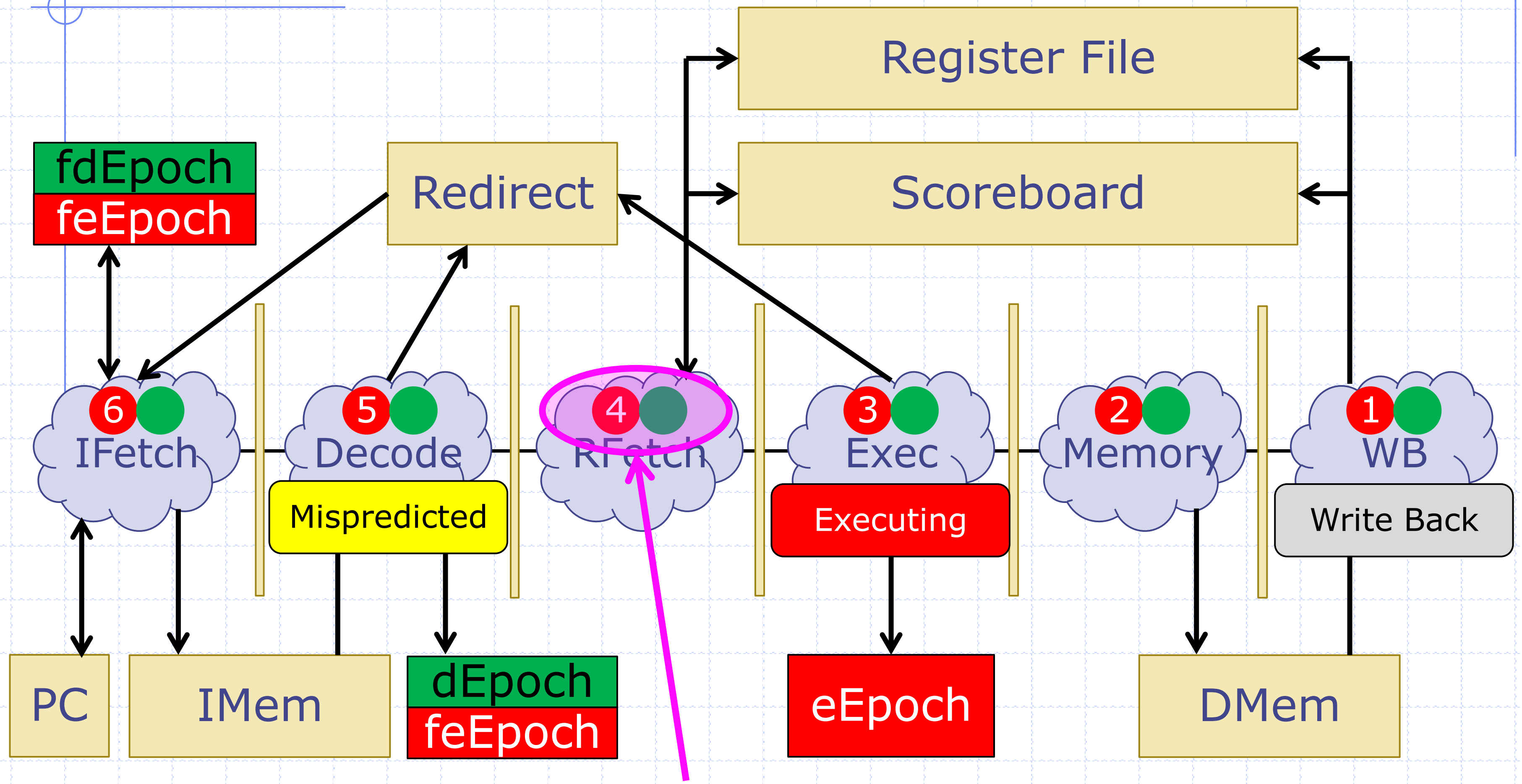
Redirection in Execute then Decode



Redirection in Decode then Execute

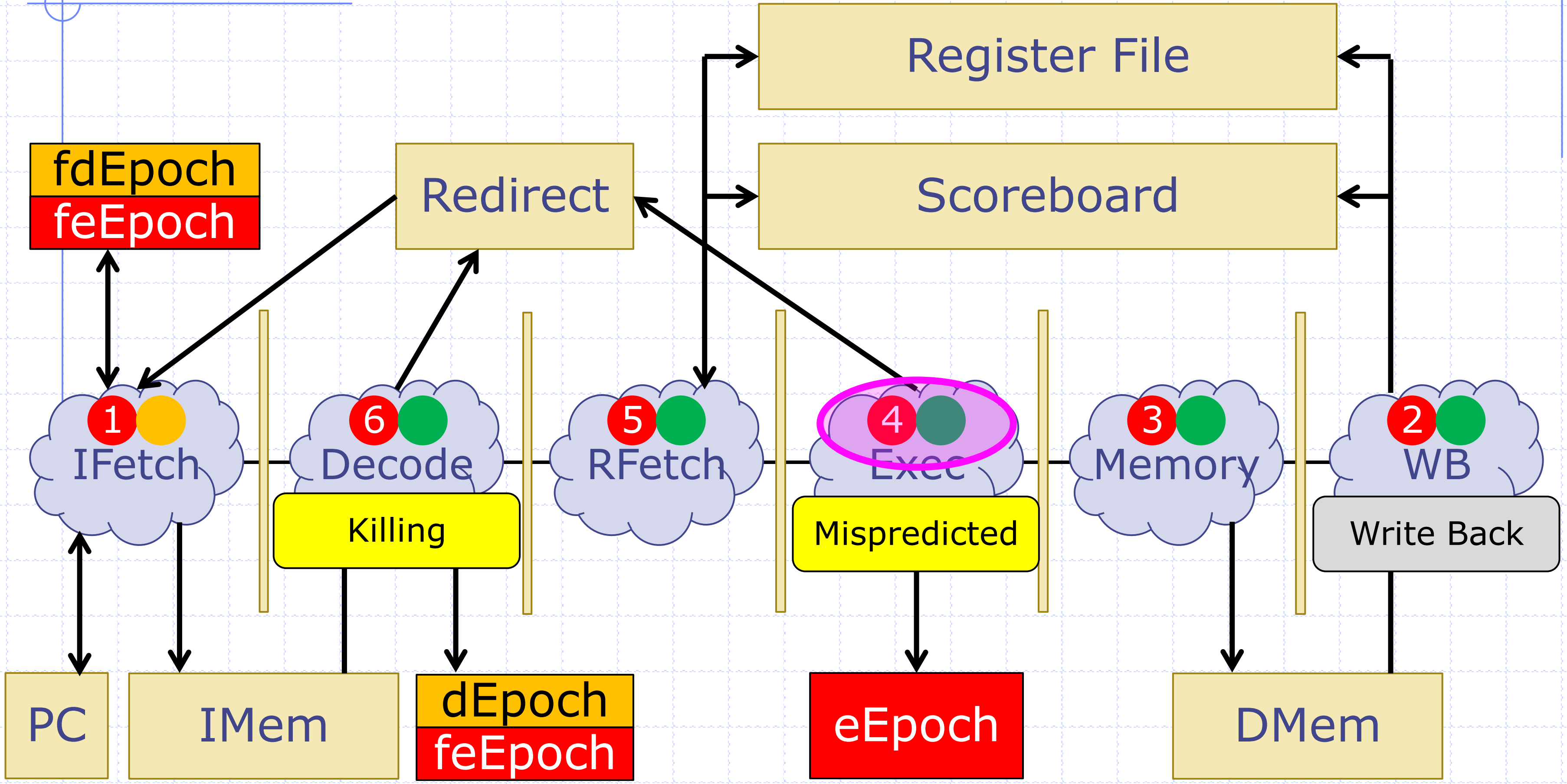
- What if decode sees a misprediction, then execute sees one in the next cycle?
  - The decode instruction will be a wrong path instruction, but it won’t be known to be wrong path until later

Redirection in Decode then Execute



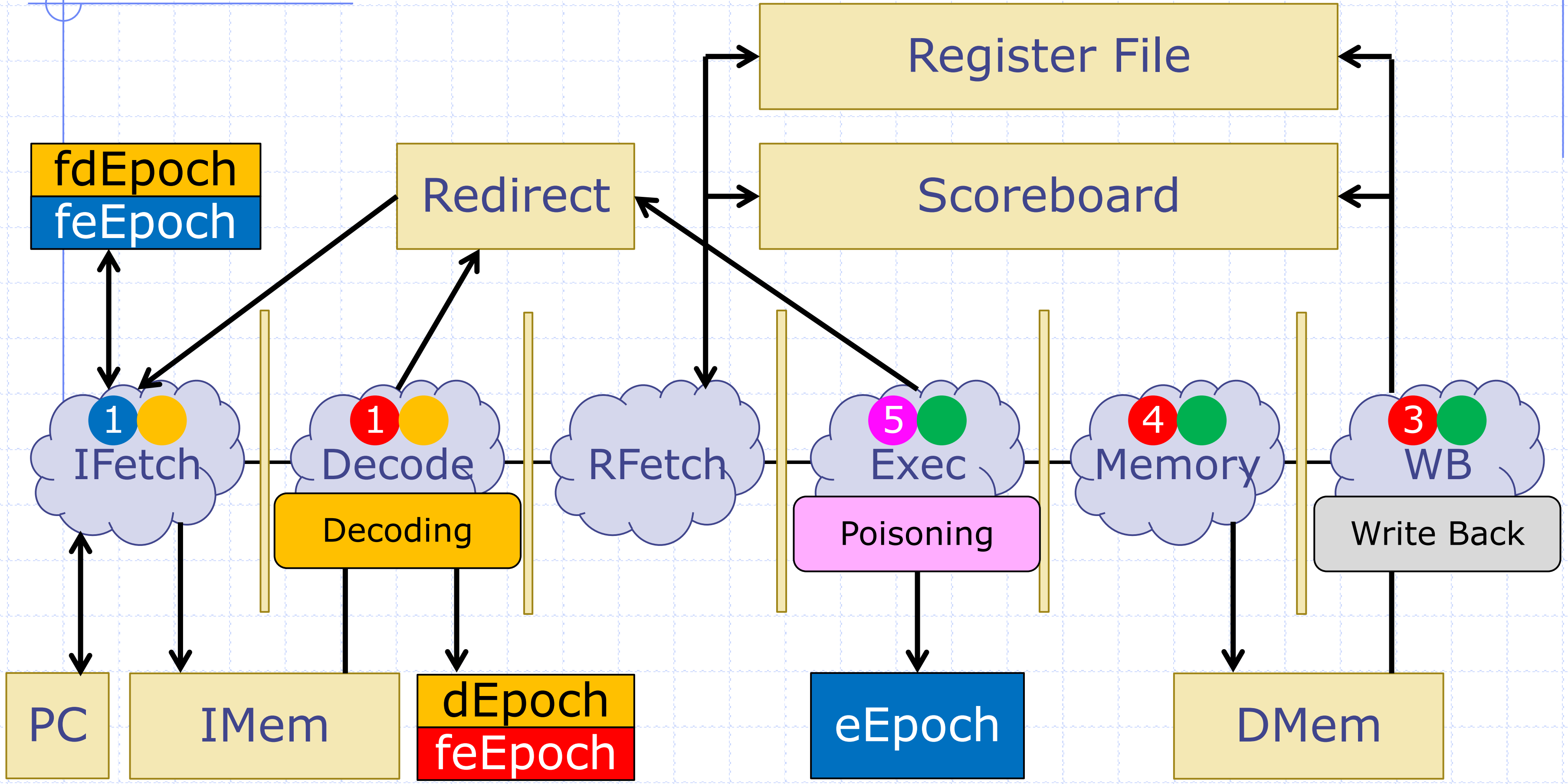
Assume this instruction is a mispredicted branch instruction. It will be in the execute stage next cycle

Redirection in Decode then Execute



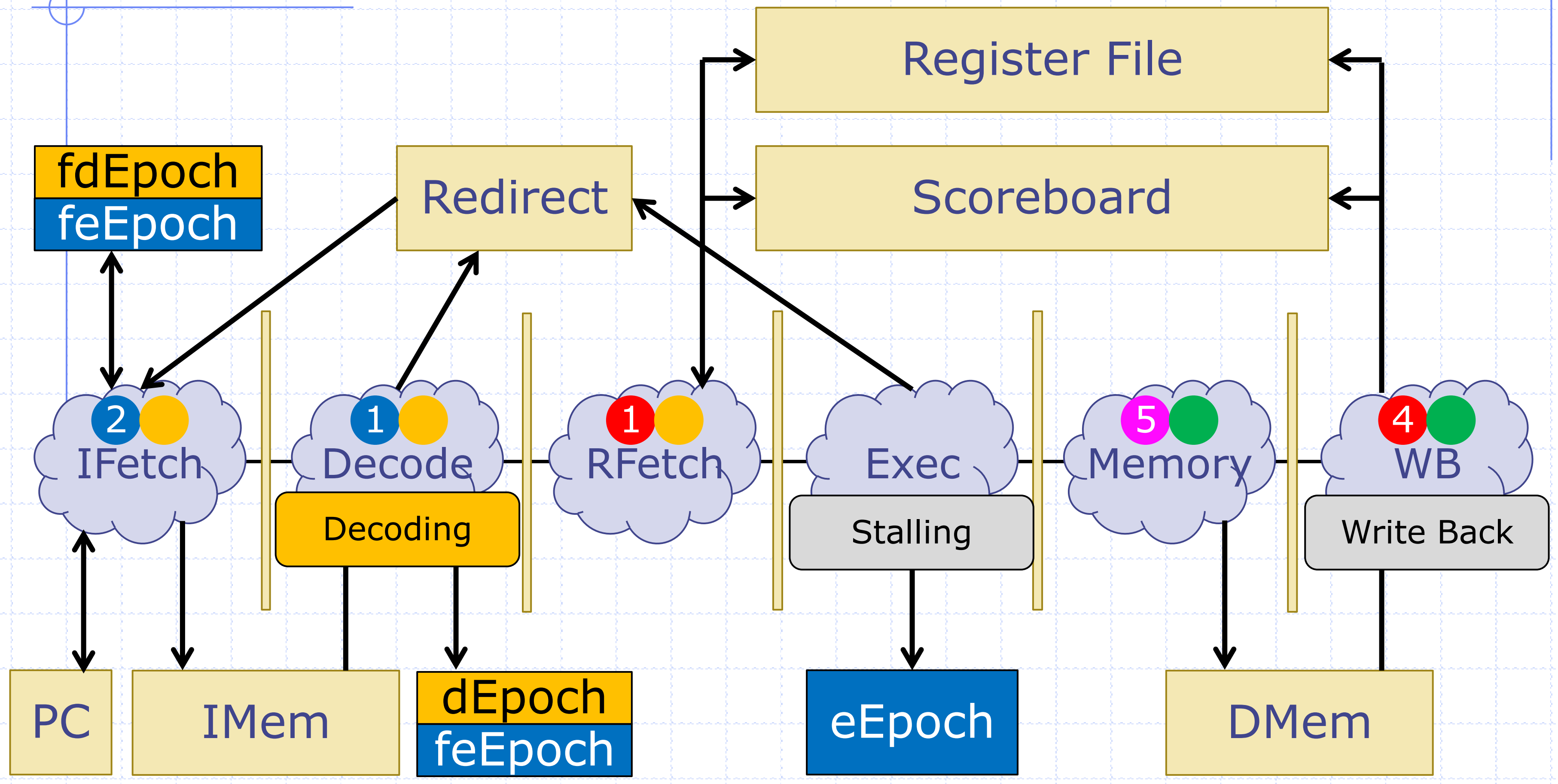
The PC was just “corrected” to a different wrong path instruction

Redirection in Decode then Execute

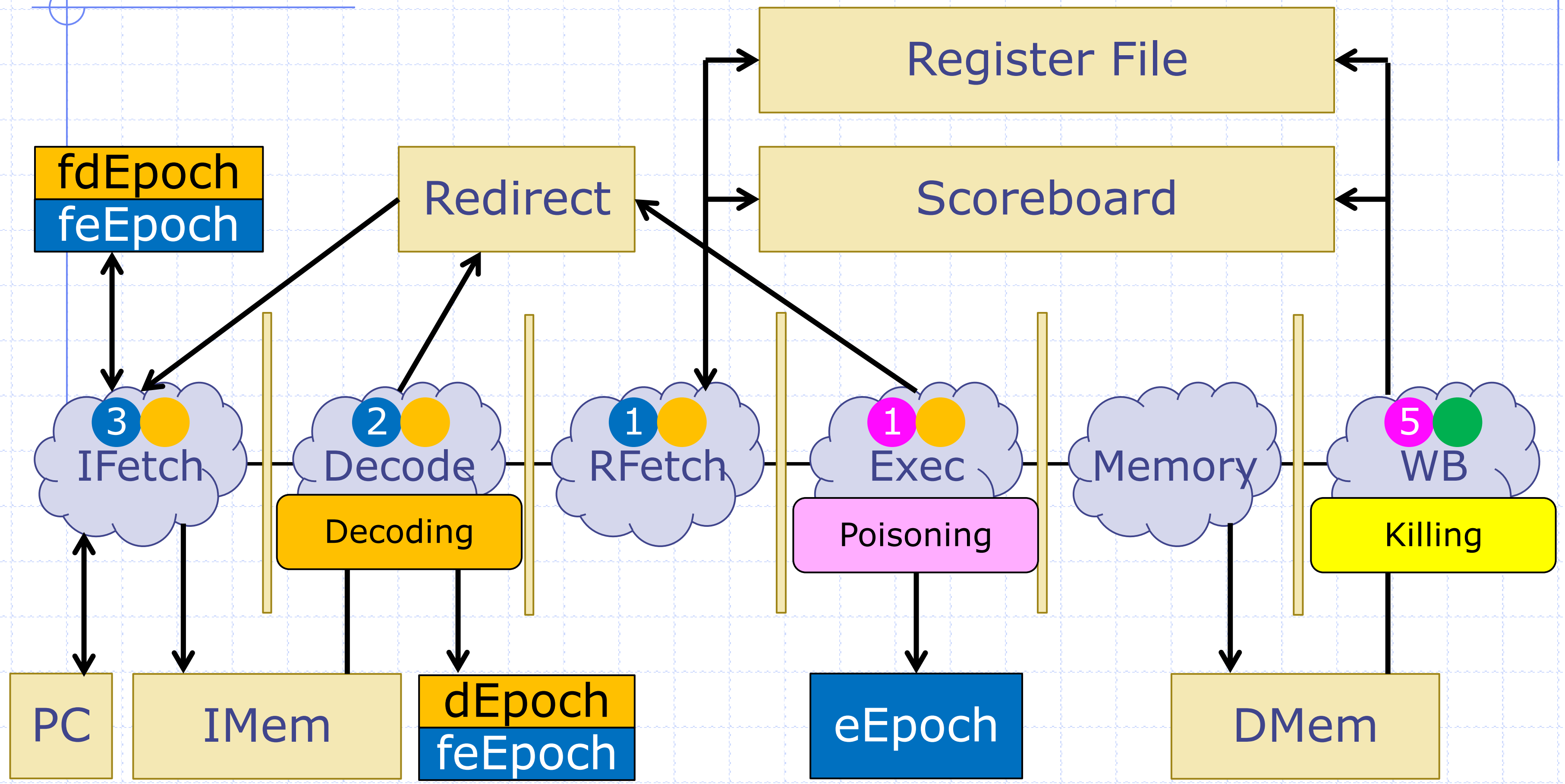


The PC was just corrected to a correct path instruction

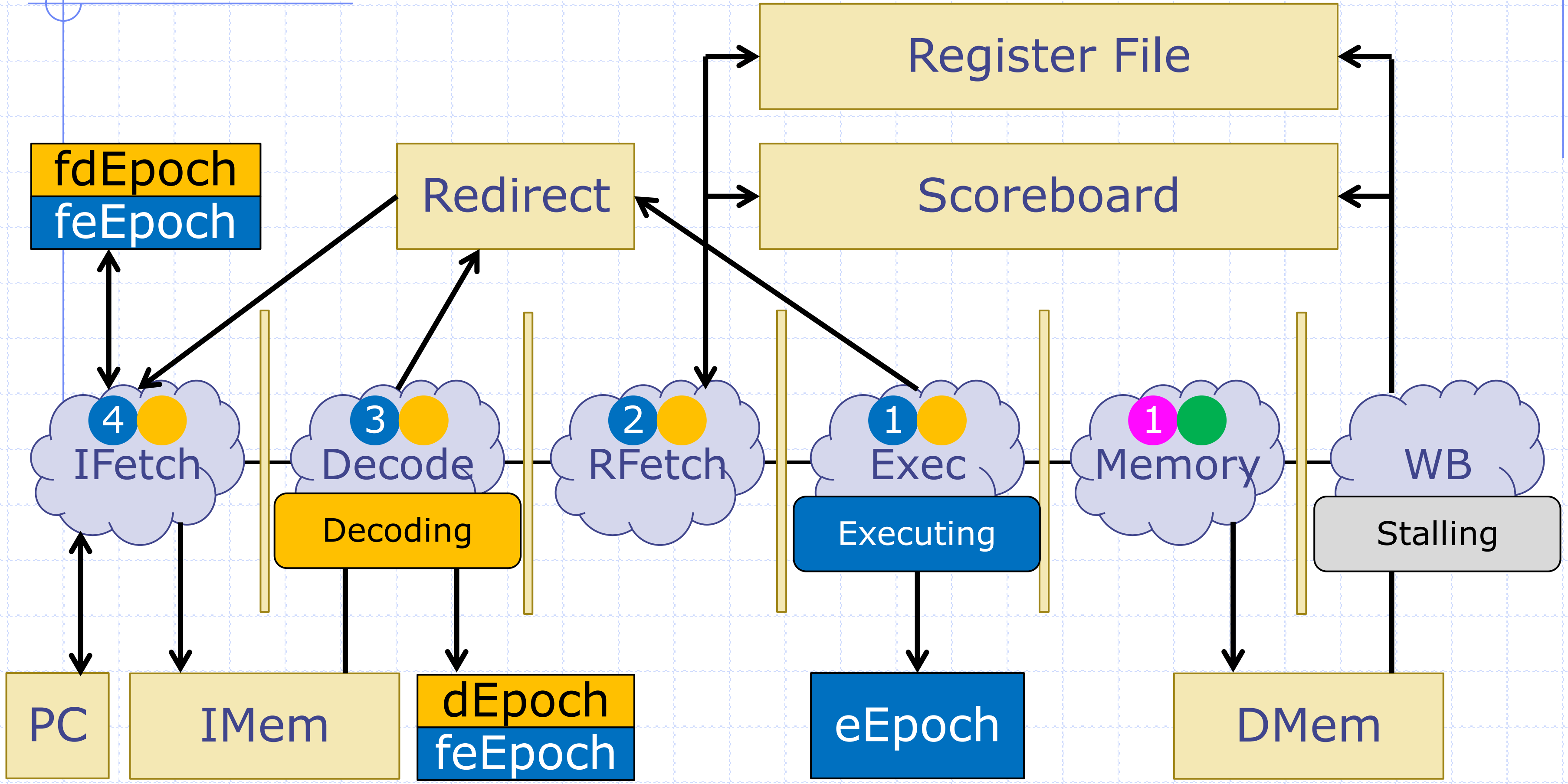
Redirection in Decode then Execute



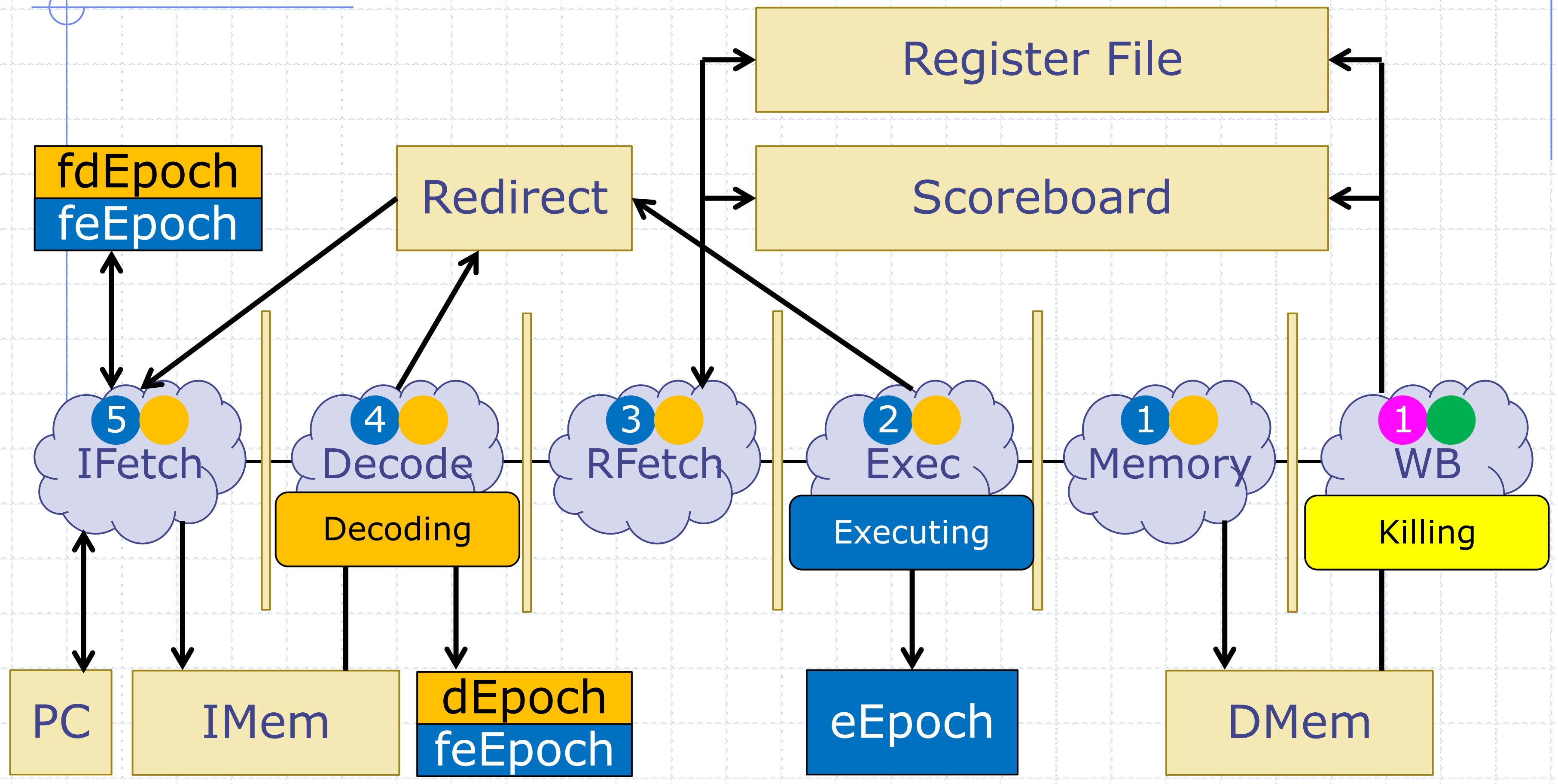
Redirection in Decode then Execute



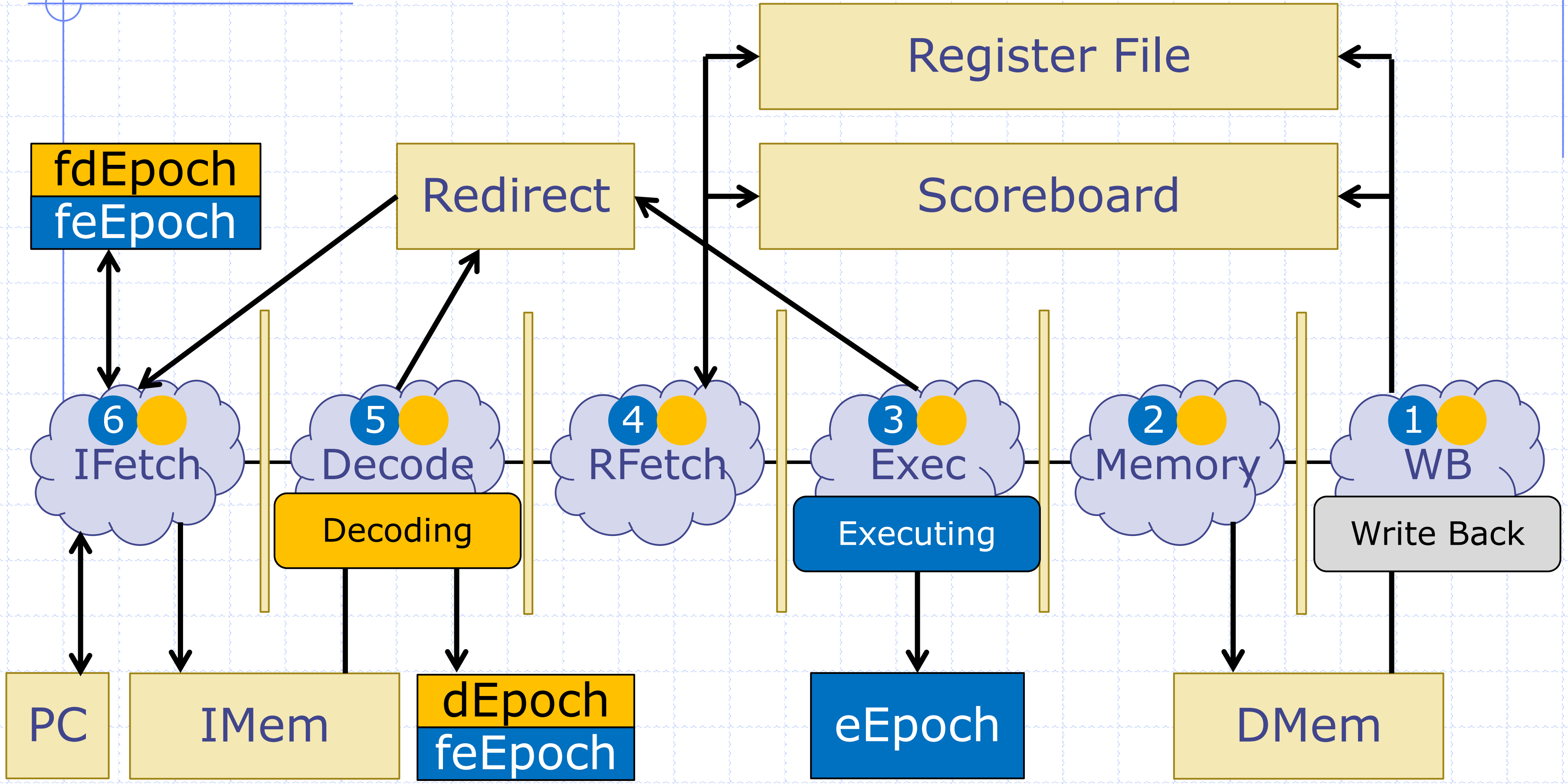
Redirection in Decode then Execute



Redirection in Decode then Execute



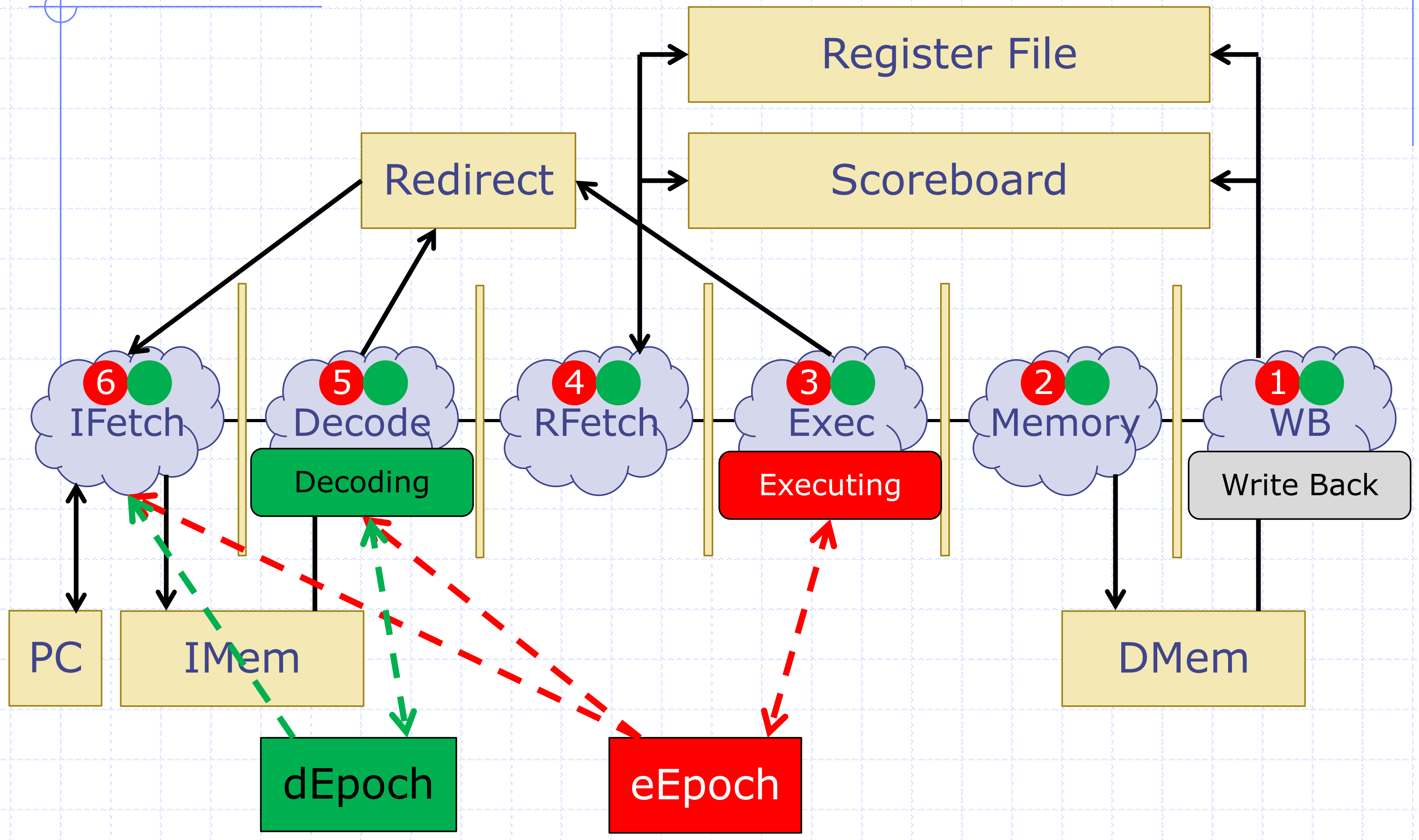
Redirection in Decode then Execute



Global Epoch States

- What if there were no estimates at epochs and everyone looked at the same epoch state
  - How would this work?

Global Epoch States

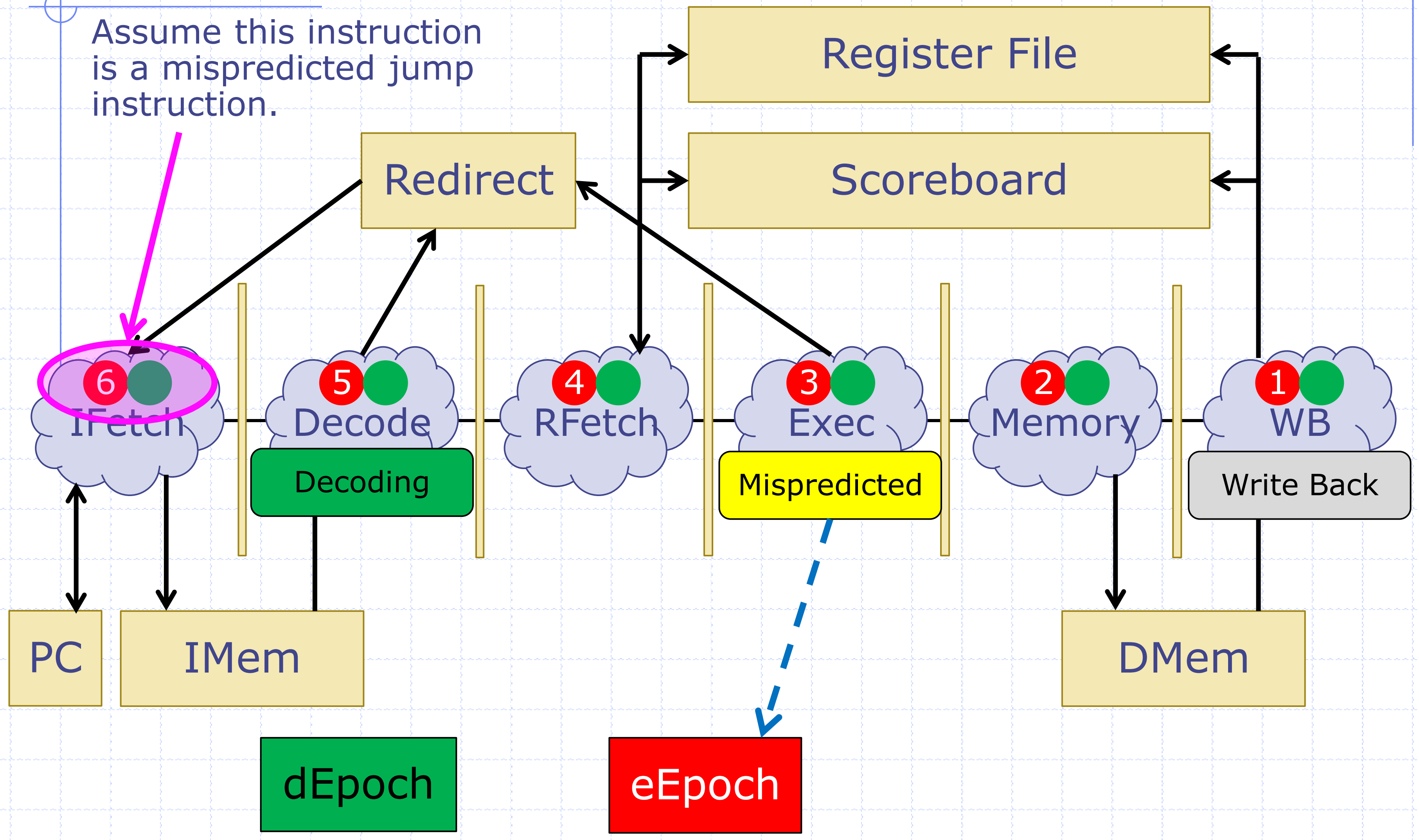


Global Epoch States

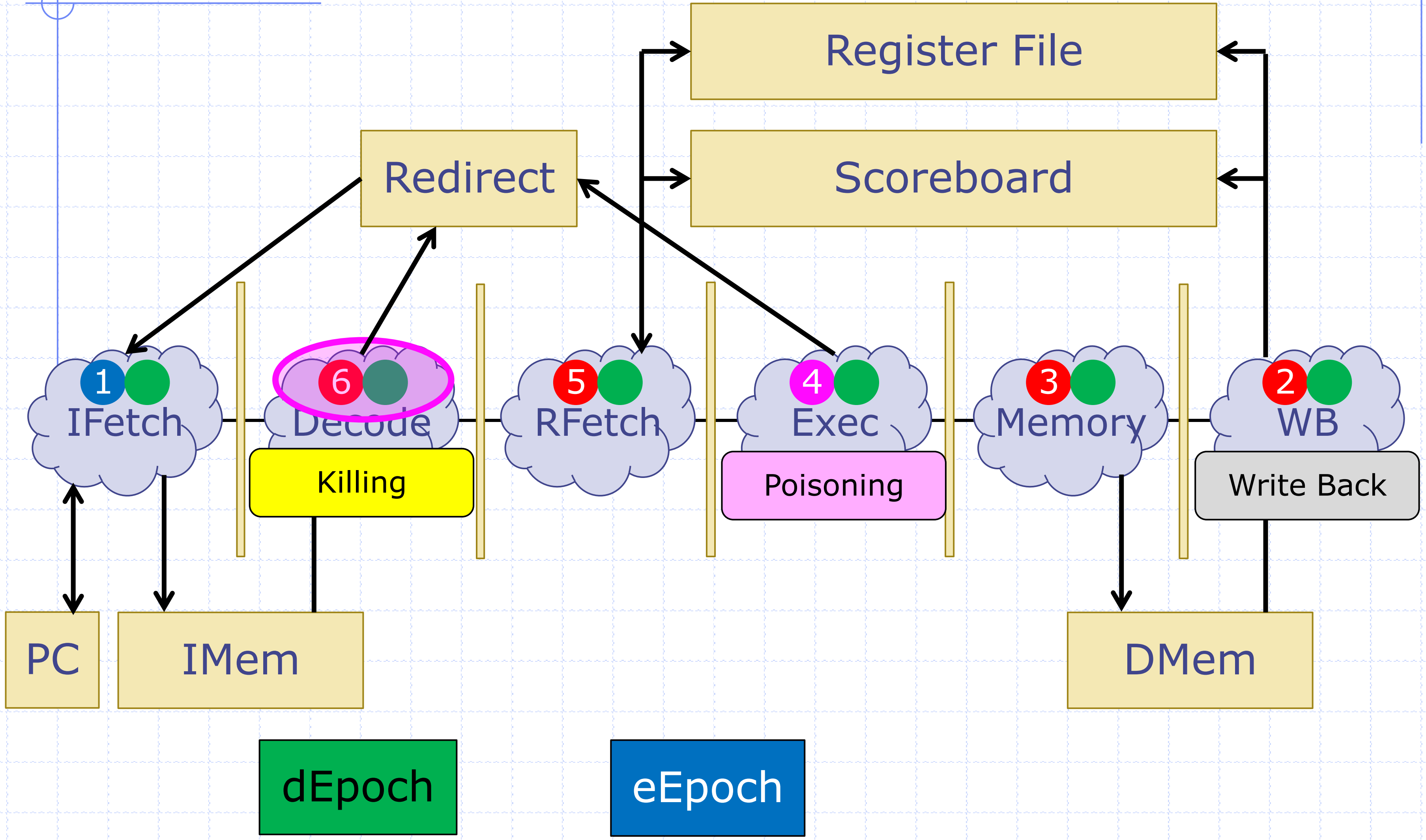
- What if execute sees a misprediction, then decode sees one in the next cycle?
    - The decode instruction will be a wrong path instruction, so it will not redirect the PC

Global Epoch States

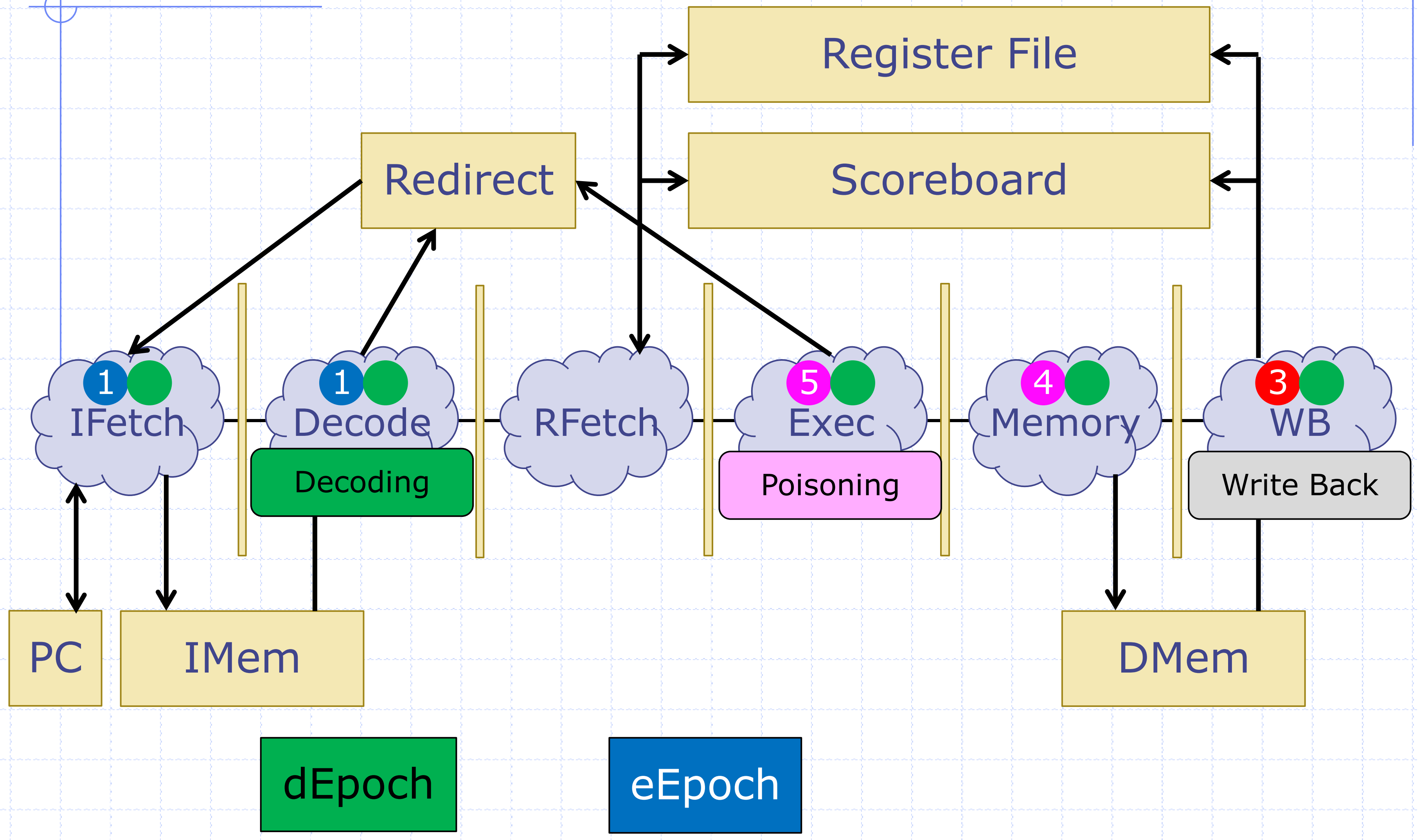
Assume this instruction is a mispredicted jump instruction.



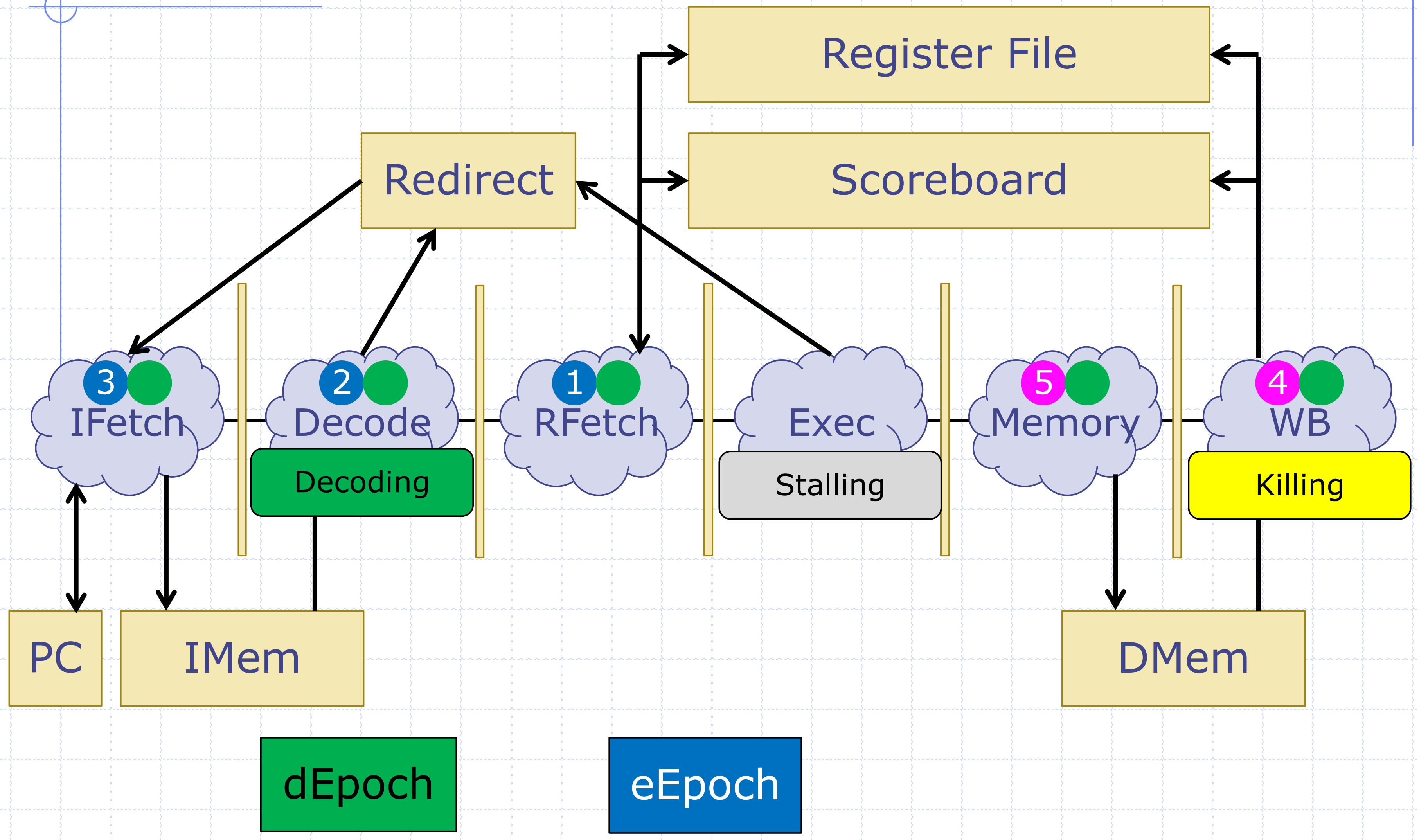
Global Epoch States



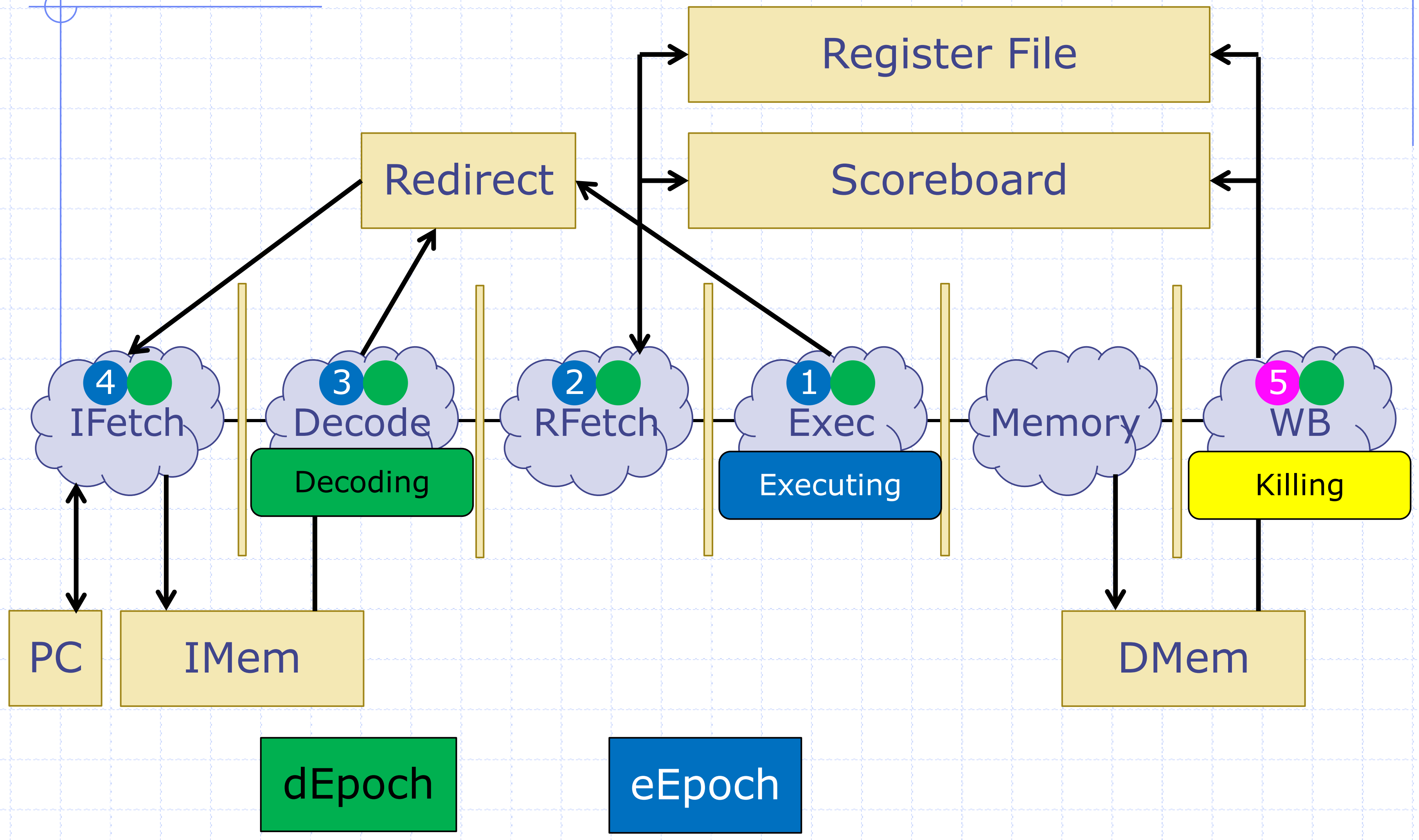
Global Epoch States



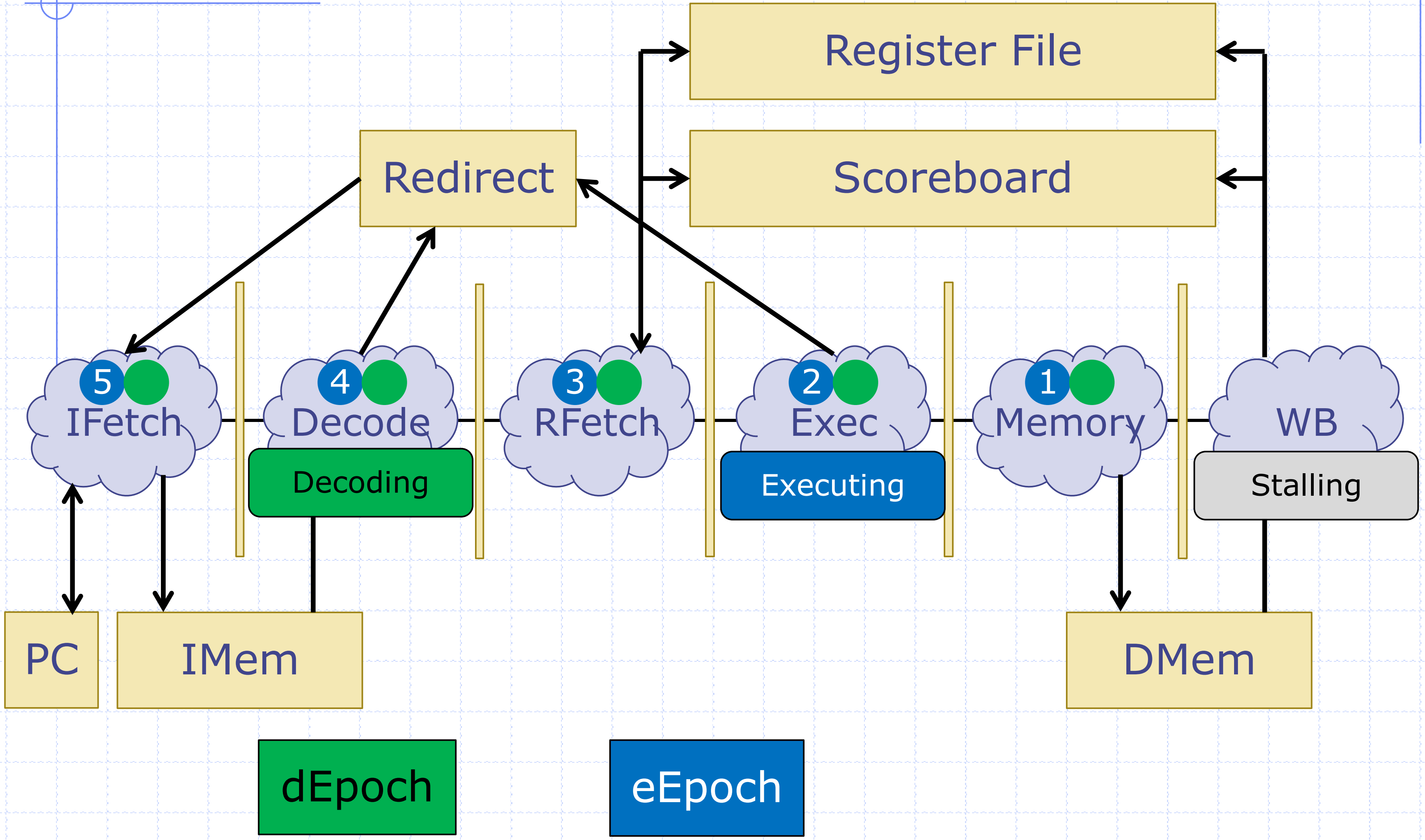
Global Epoch States



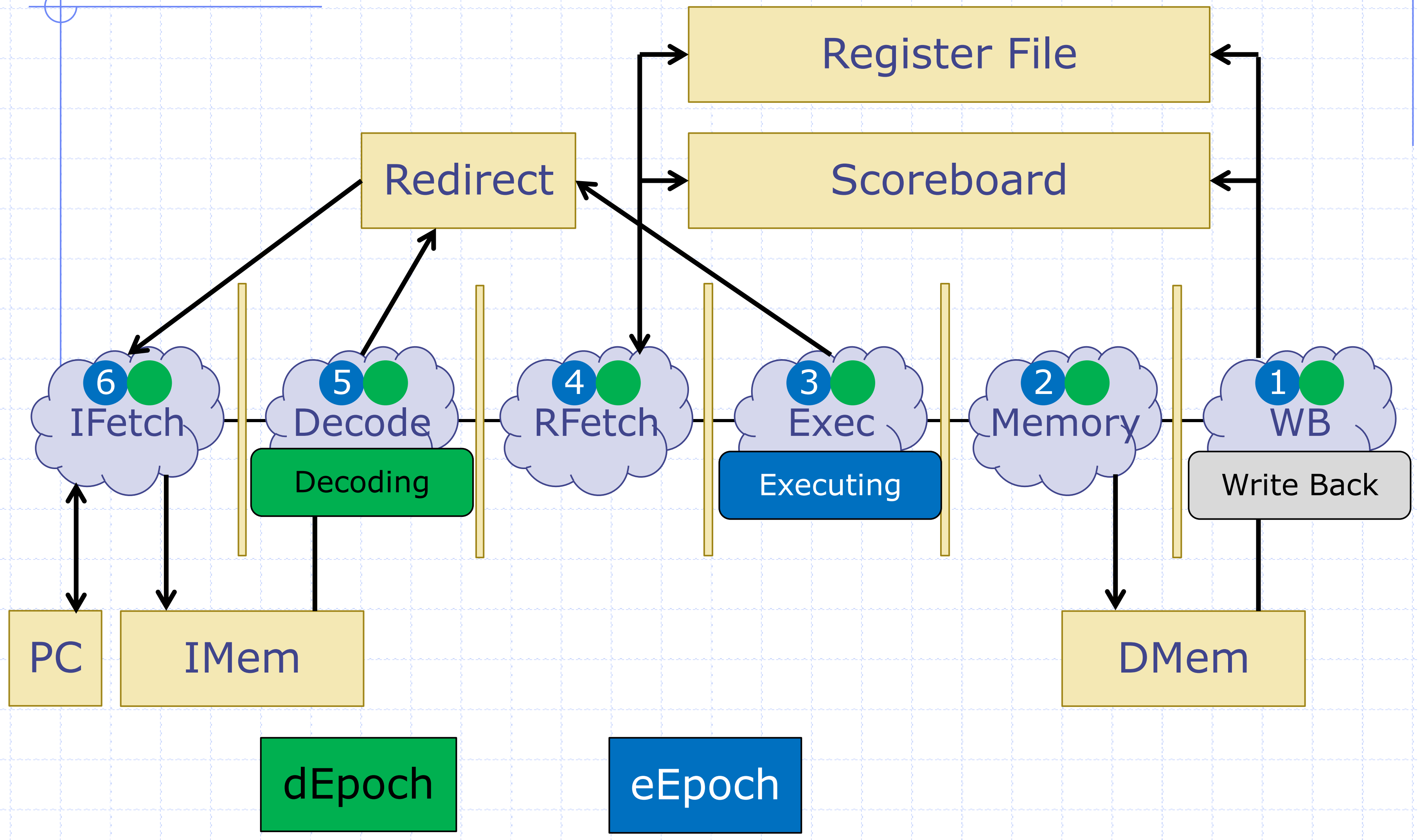
Global Epoch States



Global Epoch States



Global Epoch States

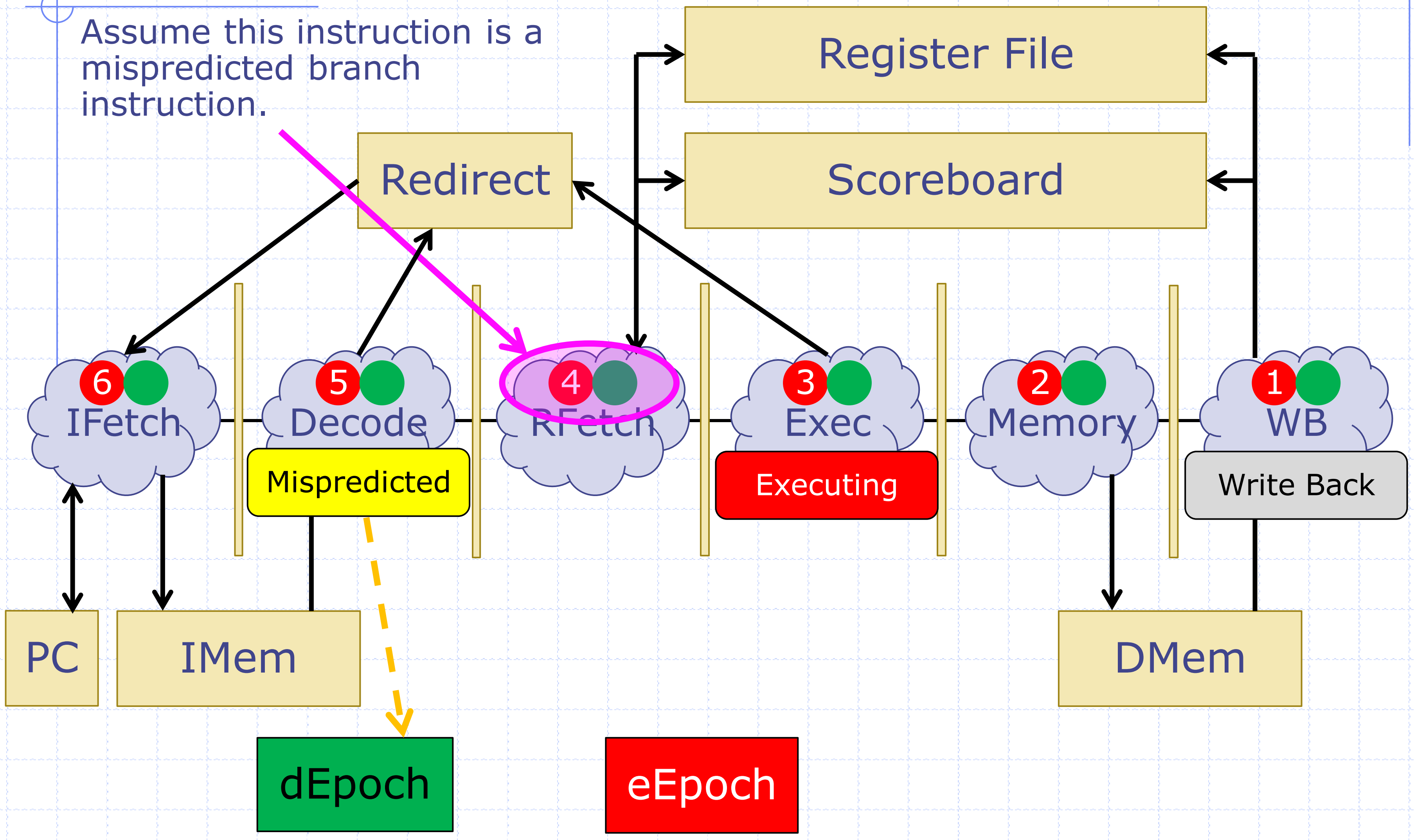


Global Epoch States

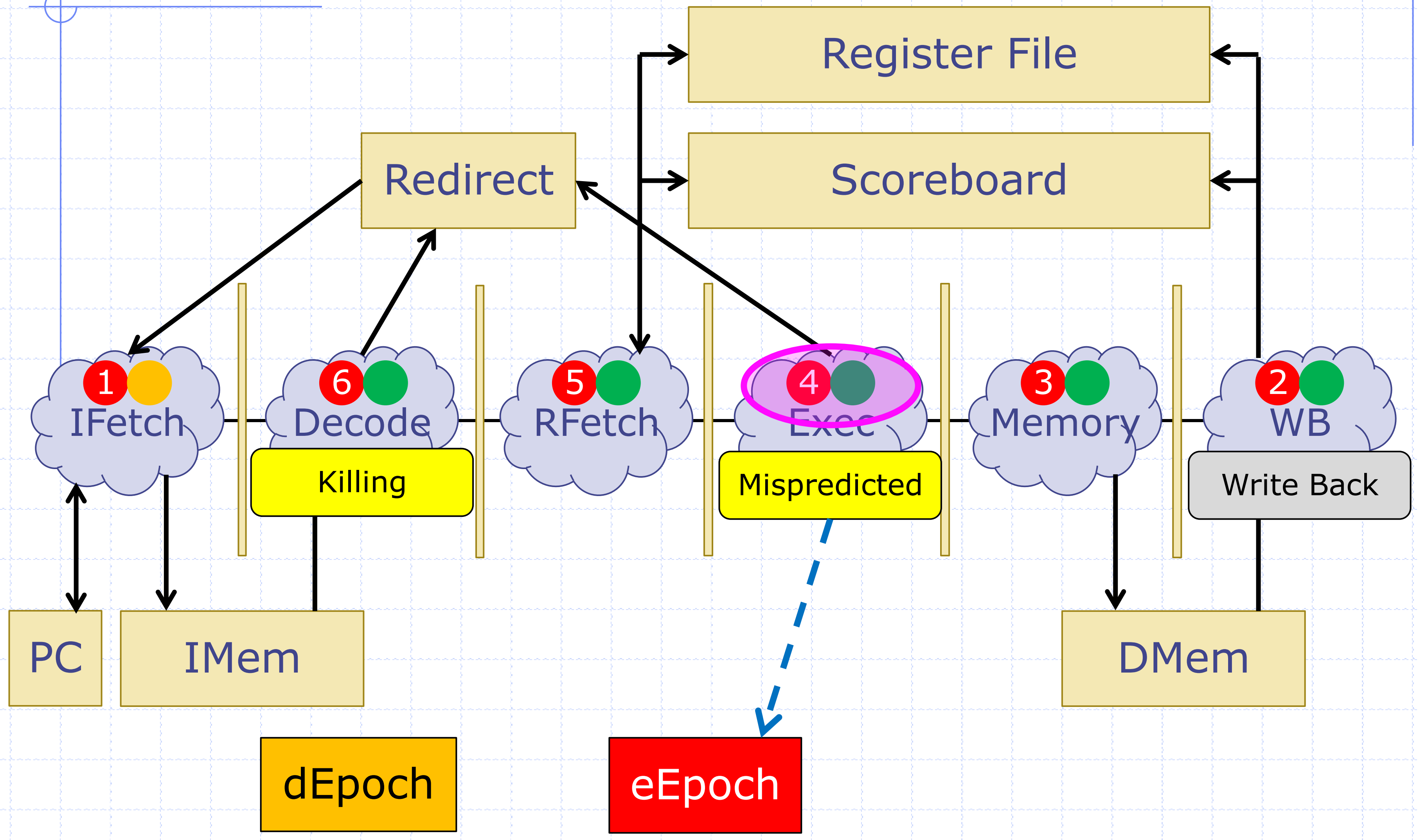
- What if decode sees a misprediction, then execute sees one in the next cycle?
  - The decode instruction will be a wrong path instruction, but it won’t be known to be wrong path until later

Global Epoch States

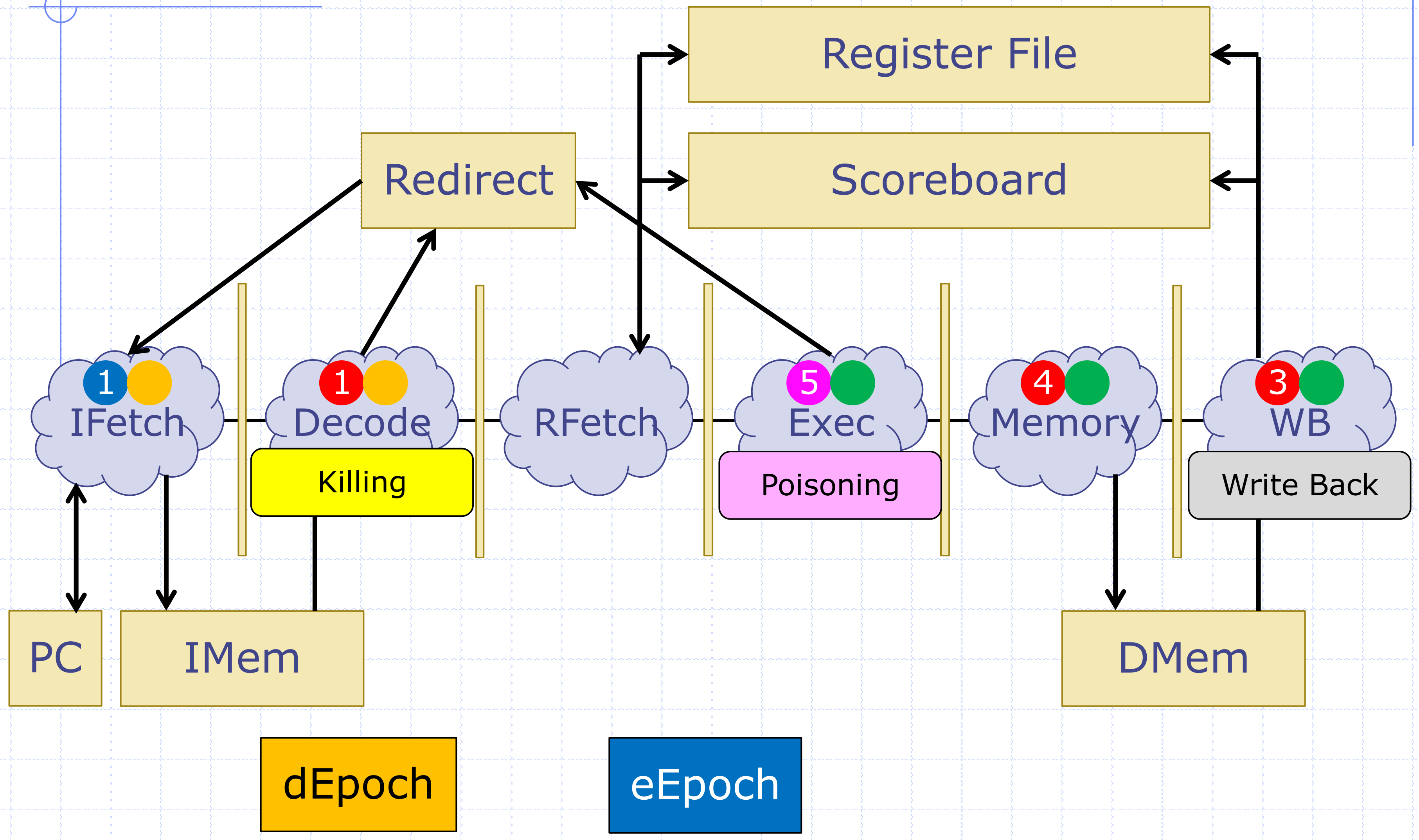
Assume this instruction is a mispredicted branch instruction.



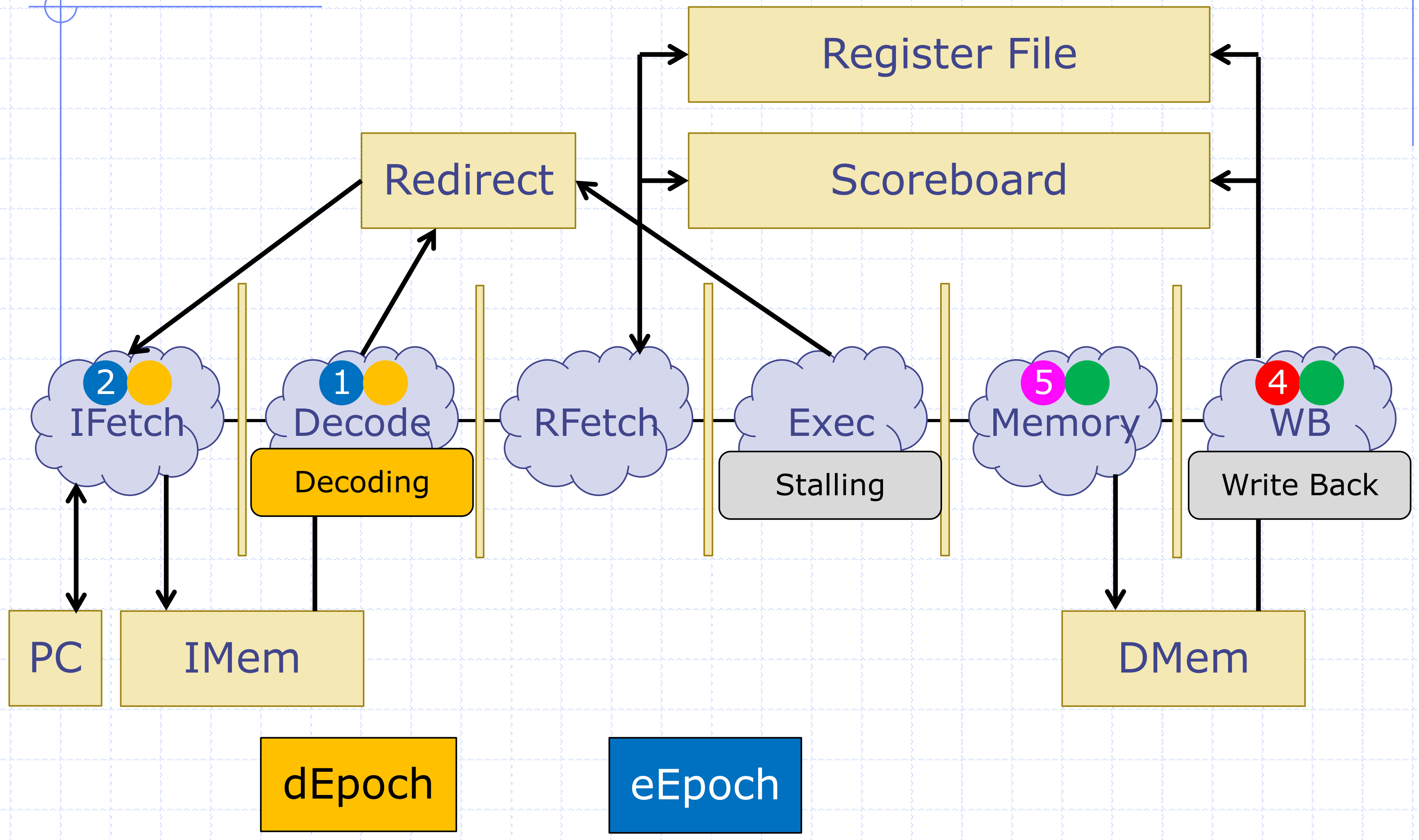
Global Epoch States



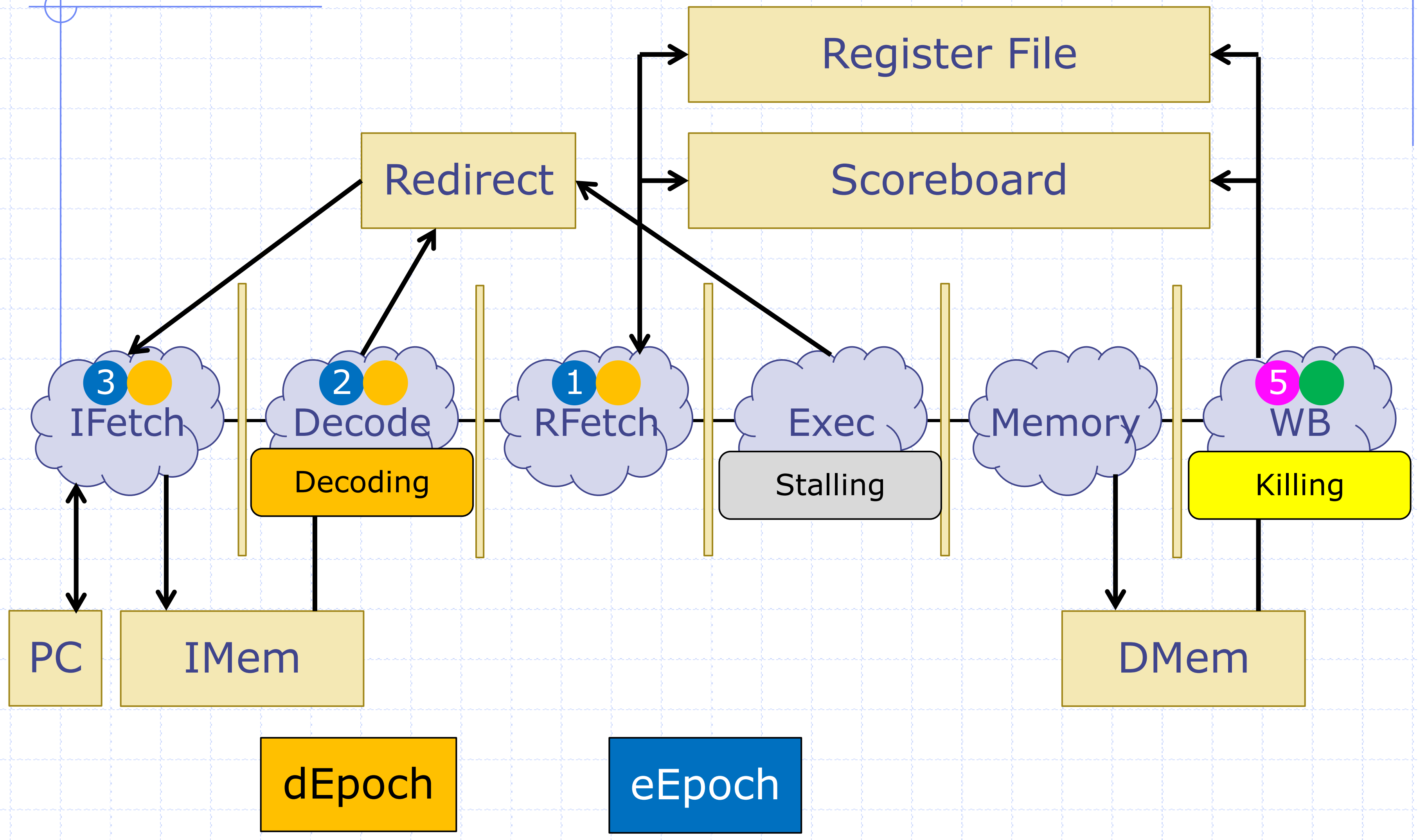
Global Epoch States



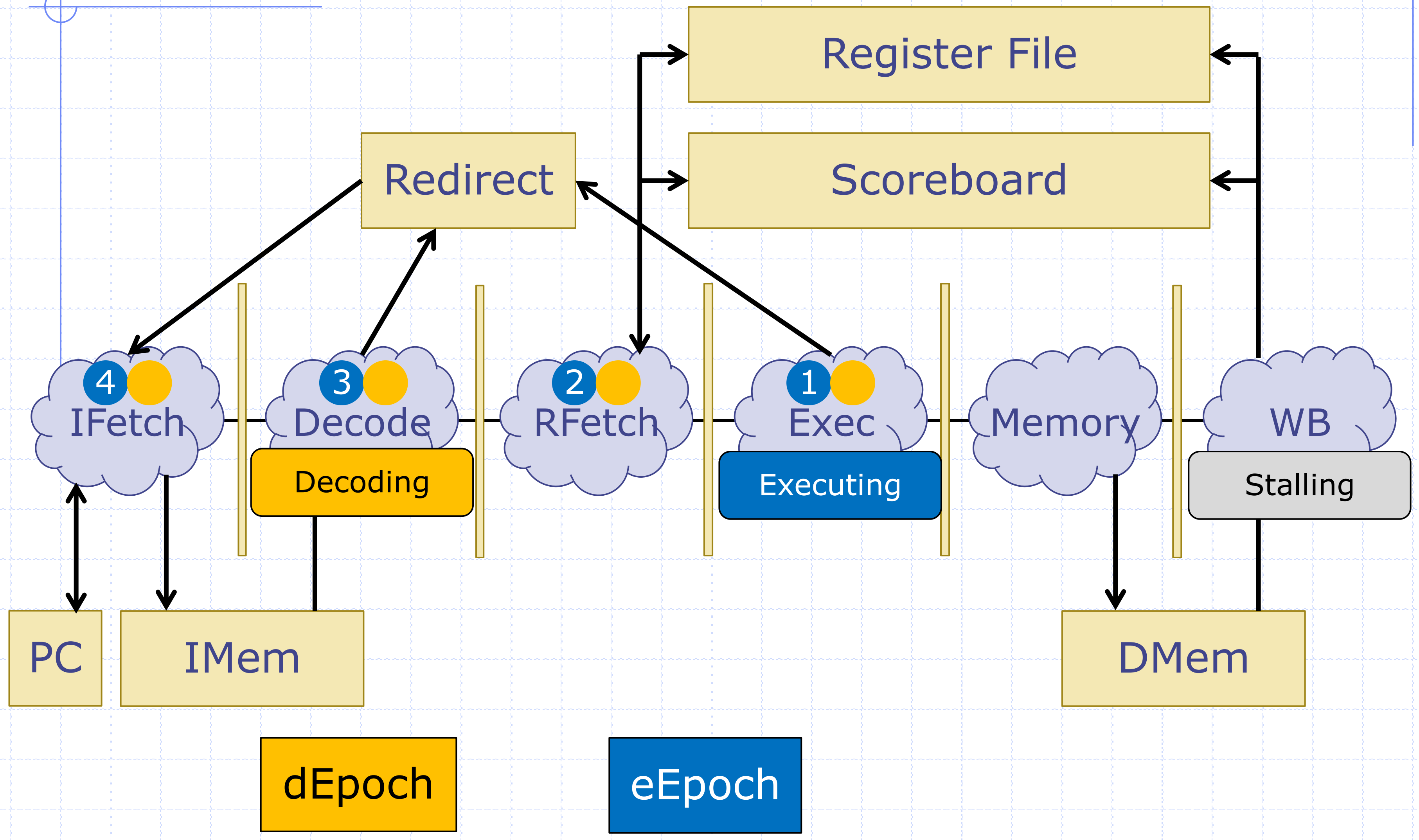
Global Epoch States



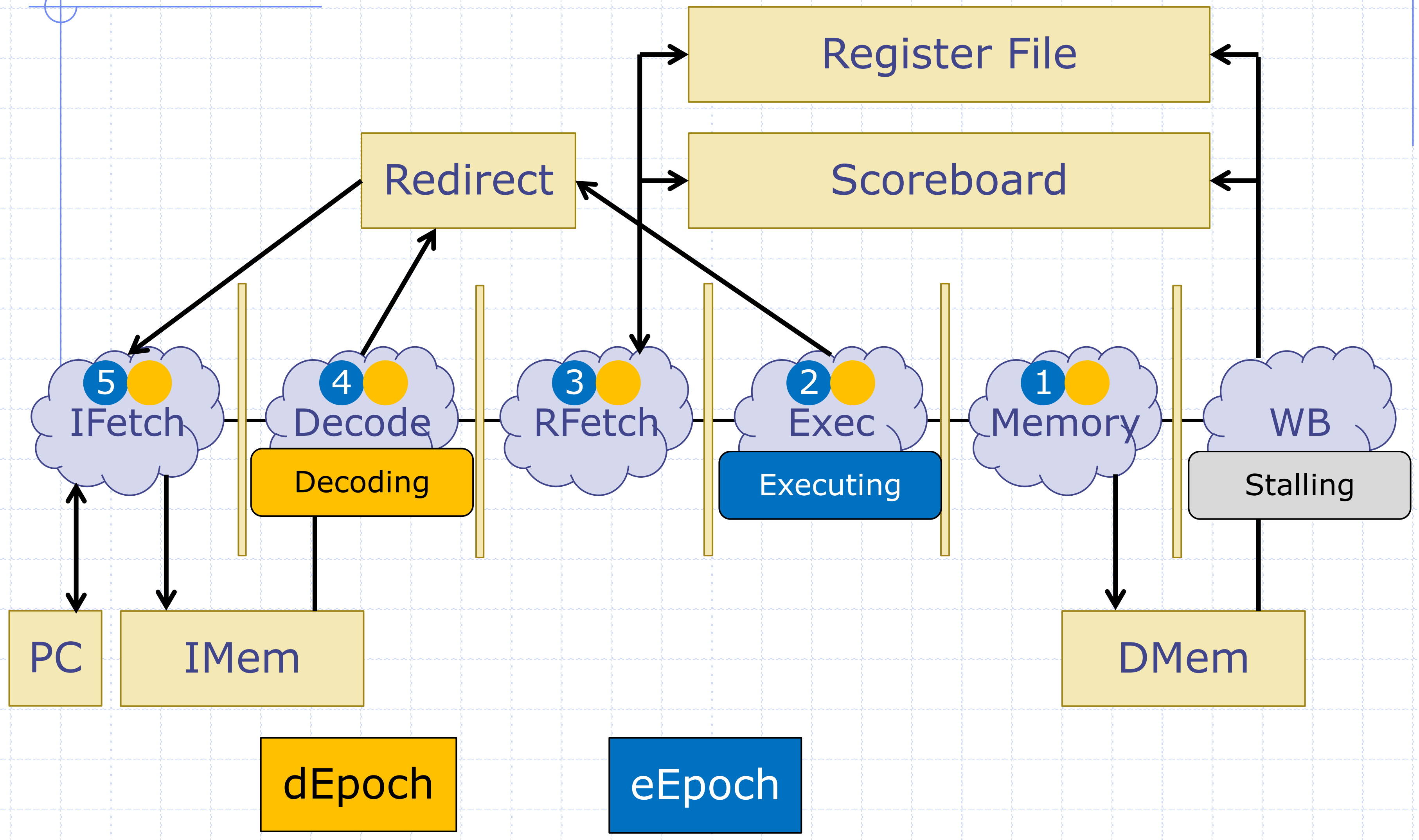
Global Epoch States



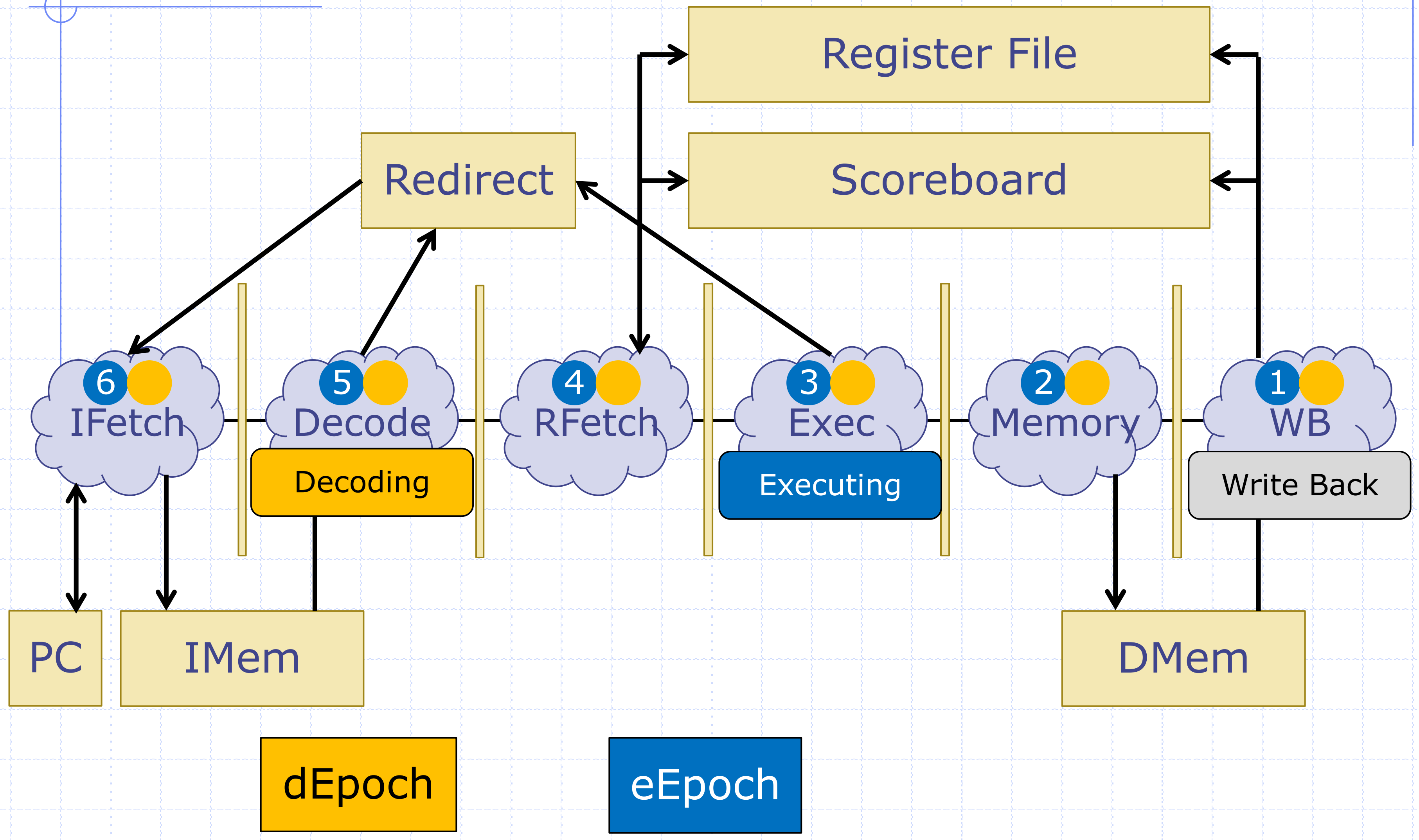
Global Epoch States



Global Epoch States



Global Epoch States

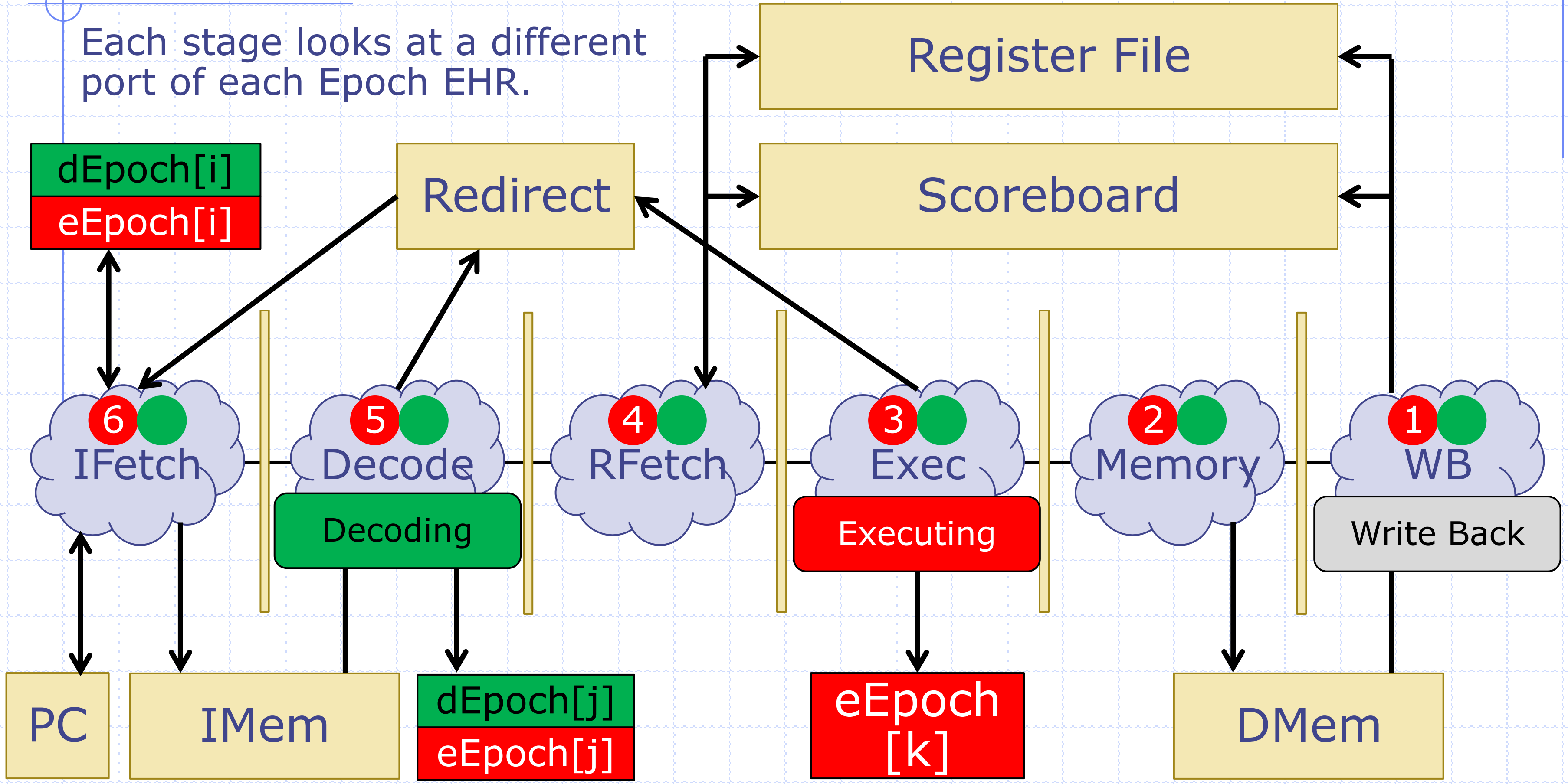


Implementing Global Epoch States

- How do you implement this?
  - There are multiple ways to do this, but the easiest way is to use EHRs

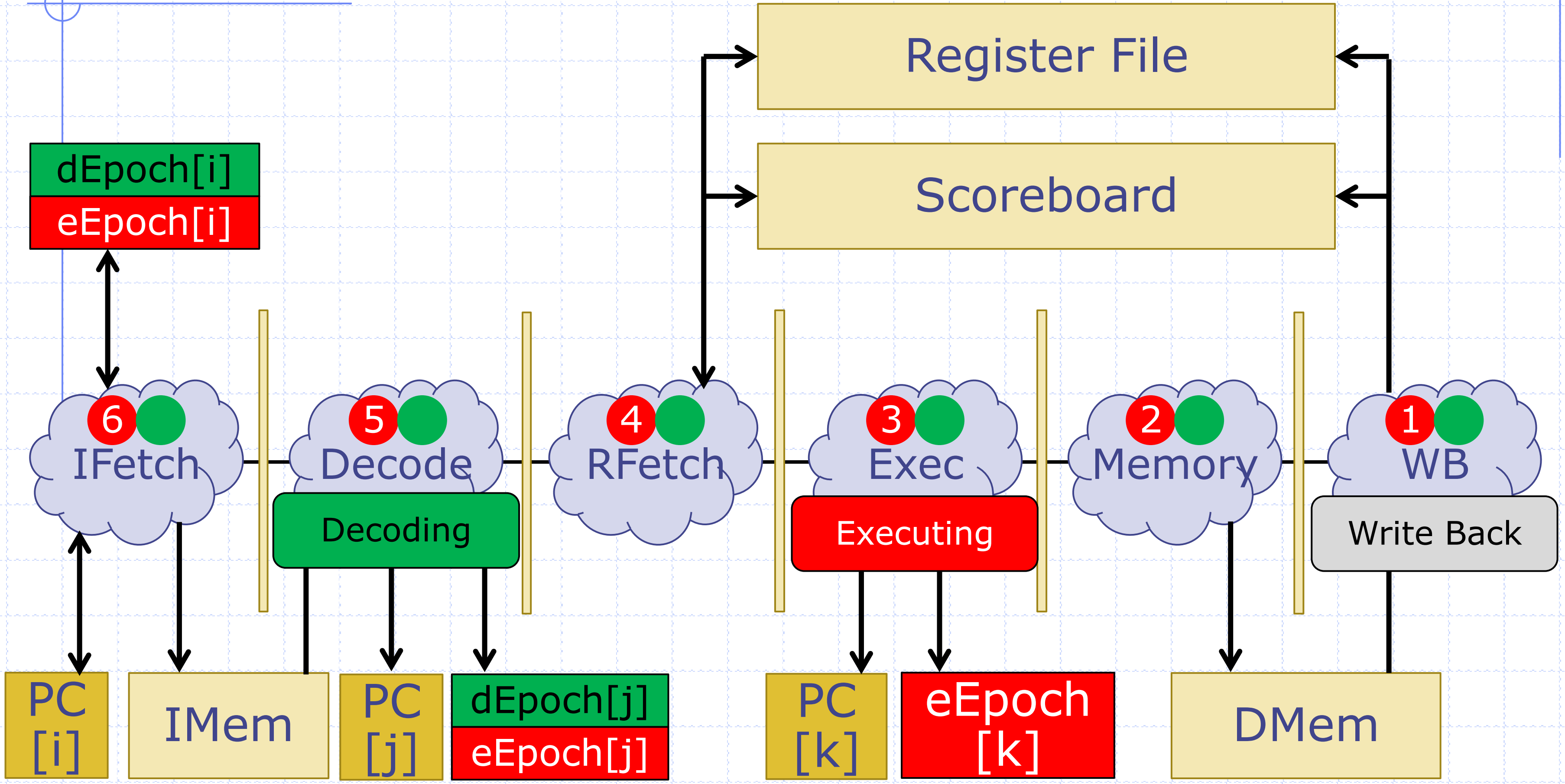
Implementing Global Epoch States with EHRs

Each stage looks at a different port of each Epoch EHR.



There's still a problem! PC redirection and epoch update needs to be atomic!

Implementing Global Epoch States with EHRs



Make PC an EHR and have each pipeline stage redirect the PC directly

New Logic for Global Epoch EHRs

- Fetch
  - Sends PC to instruction memory, pass current epoch states along with PC in fetch to decode FIFO
- Decode
  - Kill in place if instruction epochs don't match global epochs
  - If current instruction is valid, but PPC is incorrect, update decode epoch and PC EHR
  - Drop decode epoch from instruction structure sent to next stage
- Execute
  - Poison if instruction execute epoch doesn't match global execute epoch
  - If current instruction is valid, but PPC is incorrect, update execute epoch and PC EHR
  - Drop execute epoch from instruction structure passed to next stage

Why is Fecth so Simple?

- Fetch stage used to have to prioritize between the two redirect FIFOs and drop decode redirection if the execute epochs don’t match
  - Why isn’t this needed anymore?
    - Try reasoning about this on your own

How Does EHR Port Ordering Change Things?

- Originally we had redirect FIFOs from Decode and Execute to Instruction Fetch. What ordering is this?
  - Fetch – 2
  - Decode – 0 or 1
  - Execute – 0 or 1 (not the same as Decode)
- Does the order between Decode and Execute matter?
  - Not much...
- Having Fetch use ports after Decode and Execute increase the length of combinational logic
  - The order between Decode/Execute and Fetch matters most! (both for length of combinational logic and IPC)

Questions?