Constructive Computer Architecture

Tutorial 4: RISCV modules

Thomas Bourgeat
6.175 TA

Normal 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

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}

Searchable FIFO

To build a scoreboard

Searchable FIFO
Interface

```
interface SFifo#(numeric type n, type dt, type st);
    method Bool notFull;
    method Action enq(dt x);
    method Bool notEmpty;
    method dt first;
    method Action deq;
    method Action clear;
    Bool search(st x);
endinterface
```

Scoreboard implementation
using searchable Fifos

```
function Bool isFound
         (Maybe#(RIndx) dst, Maybe#(RIndx) src);
  return isValid(dst) && isValid(src) &&
              (fromMaybe(?,dst)==fromMaybe(?,src));
endfunction

module mkCFScoreboard(Scoreboard#(size));
  SFifo#(size, Maybe#(RIndx), Maybe#(RIndx))
       f <- mkCFSFifo(isFound);
  method insert  = f.enq;
  method remove  = f.deq;
  method search1 = f.search1;
  method search2 = f.search2;
endmodule
```

Searchable FIFO
Internal States

Standard FIFO states:

```
Reg#(Bit#(TLog#(n))) enqP <- mkReg(0);
Reg#(Bit#(TLog#(n))) deqP <- mkReg(0);
Reg#(Bool) full <- mkReg(False);
Reg#(Bool) empty <- mkReg(Empty);
```

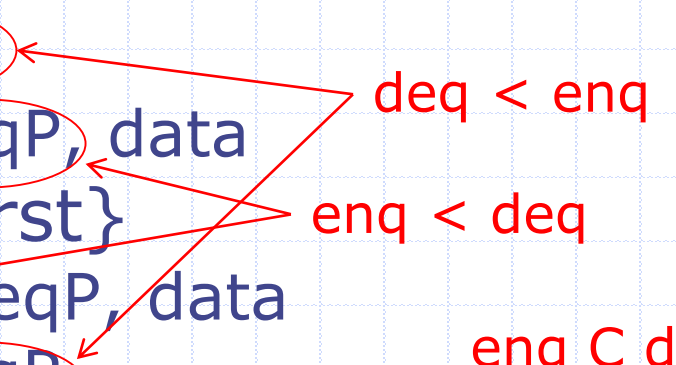
Need any more?

Searchable FIFO
Method Calls

- {notFull, enq}
  - R: full, enqP, deqP
  - W: full, empty, enqP, data
- {notEmpty, deq, first}
  - R: empty, enqP, deqP, data
  - W: full, empty, deqP
- search
  - R: (empty or full), enqP, deqP, data
- clear
  - W: empty, full, enq, deqP

Searchable FIFO
Potential Conflicts

- {notFull, enq}
  - R: full, enqP, deqP
  - W: full, empty, enqP, data
- {notEmpty, deq, first}
  - R: empty, enqP, deqP, data
  - W: full, empty, deqP
- search
  - R: (empty or full), enqP, deqP, data
- clear
  - W: empty, full, enq, deqP



enq C deq
Same as FIFO

Search is read-only -> it can always come first
Clear is write-only -> it can always come last

Searchable FIFO
Implementation 1

- Implementation:
  - mkCFFifo with a search method
- Schedule:
  - search < {notFull, enq, notEmpty, deq, first} < clear
  - {notFull, enq} CF {notEmpty, deq, first}

Searchable FIFO
Implementation 1

```
module mkSFifo1(SFifo#(n, t, t)) provisos(Eq#(t));
  // mkCFFifo implementation

  method Bool search(t x);
    Bool found = False;
    for(Integer i = 0; i < valueOf(n); i = i+1) begin
      Bool validEntry = full[0] ||
            (enqP[0]>deqP[0] && i>=deqP[0] && i<enqP[0]) ||
            (enqP[0]<deqP[0] && (i>=deqP[0] || i<enqP[0]));
      if(validEntry && (data[i] == x)) found = True;
    end
    return found;
  endmethod
endmodule
```

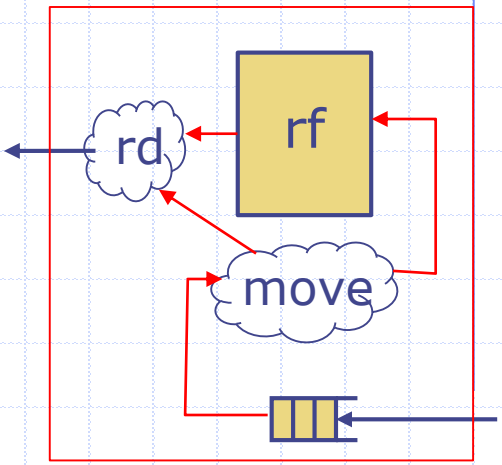
Searchable FIFO
Custom Search Function

```
module mkSFifo1(function Bool isFound( dt x, st y ),
SFifo#(n, dt, st) ifc);
  // mkCFFifo implementation
  method Bool search(st x);
    Bool found = False;
    for(Integer i = 0; i < valueOf(n); i = i+1) begin
      Bool validEntry = full[0] ||
            (enqP[0]>deqP[0] && i>=deqP[0] && i<enqP[0]) ||
            (enqP[0]<deqP[0] && (i>=deqP[0] || i<enqP[0]);
      if(validEntry && isFound(data[i], x)) found = True;
    end
    return found;
  endmethod
endmodule
```

Scoreboard

- When using a SFifo for a scoreboard, the following functions are used together:
  - {search, notFull, enq}
  - {notEmpty, deq}
- Are enq and deq still commutative like in the CFFifo case?
  - No! Search has to be able to be done with enq, and search is not commutative with deq

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
```

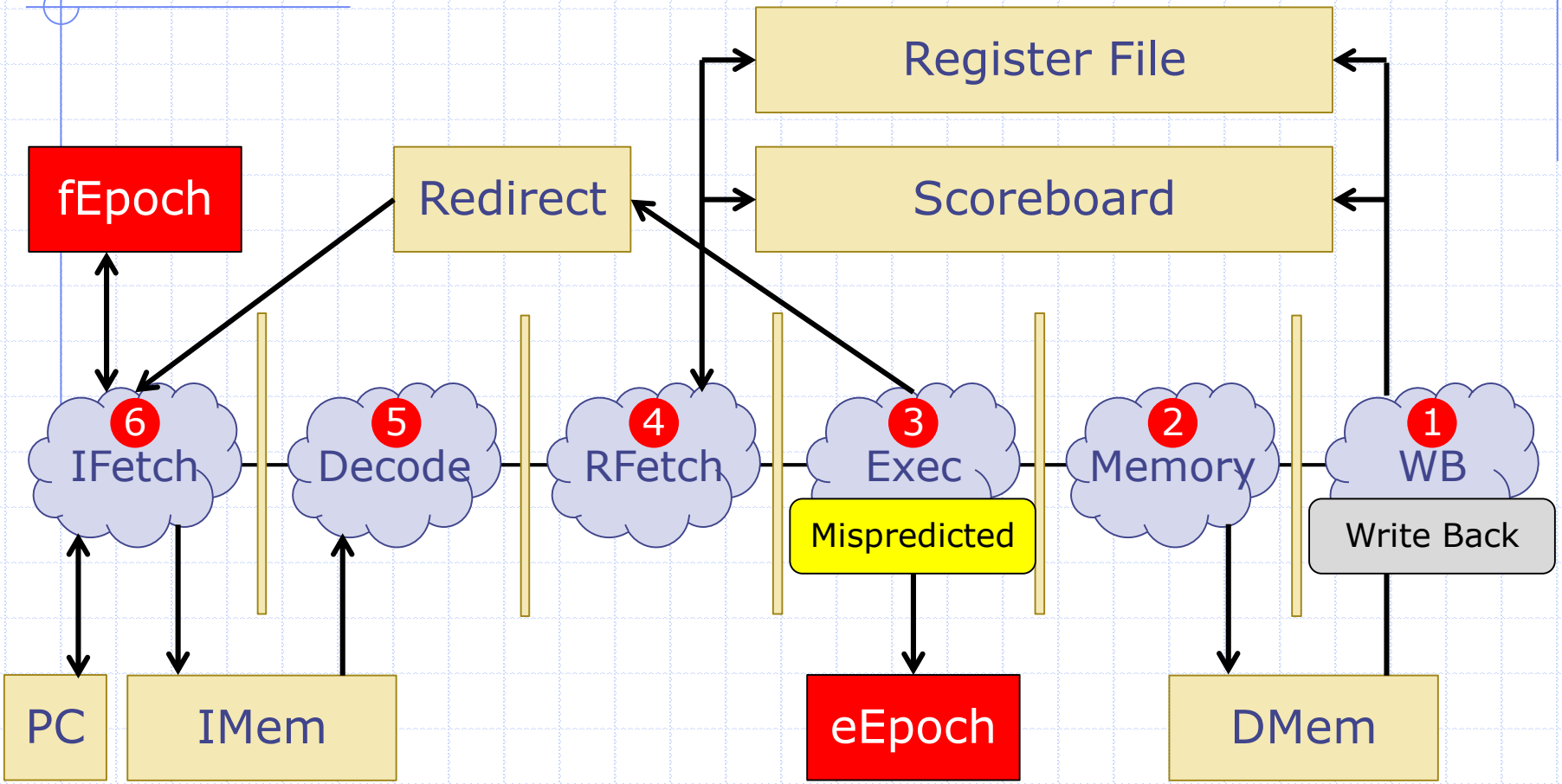
wr < {rd1, rd2}

Epoch Tutorial

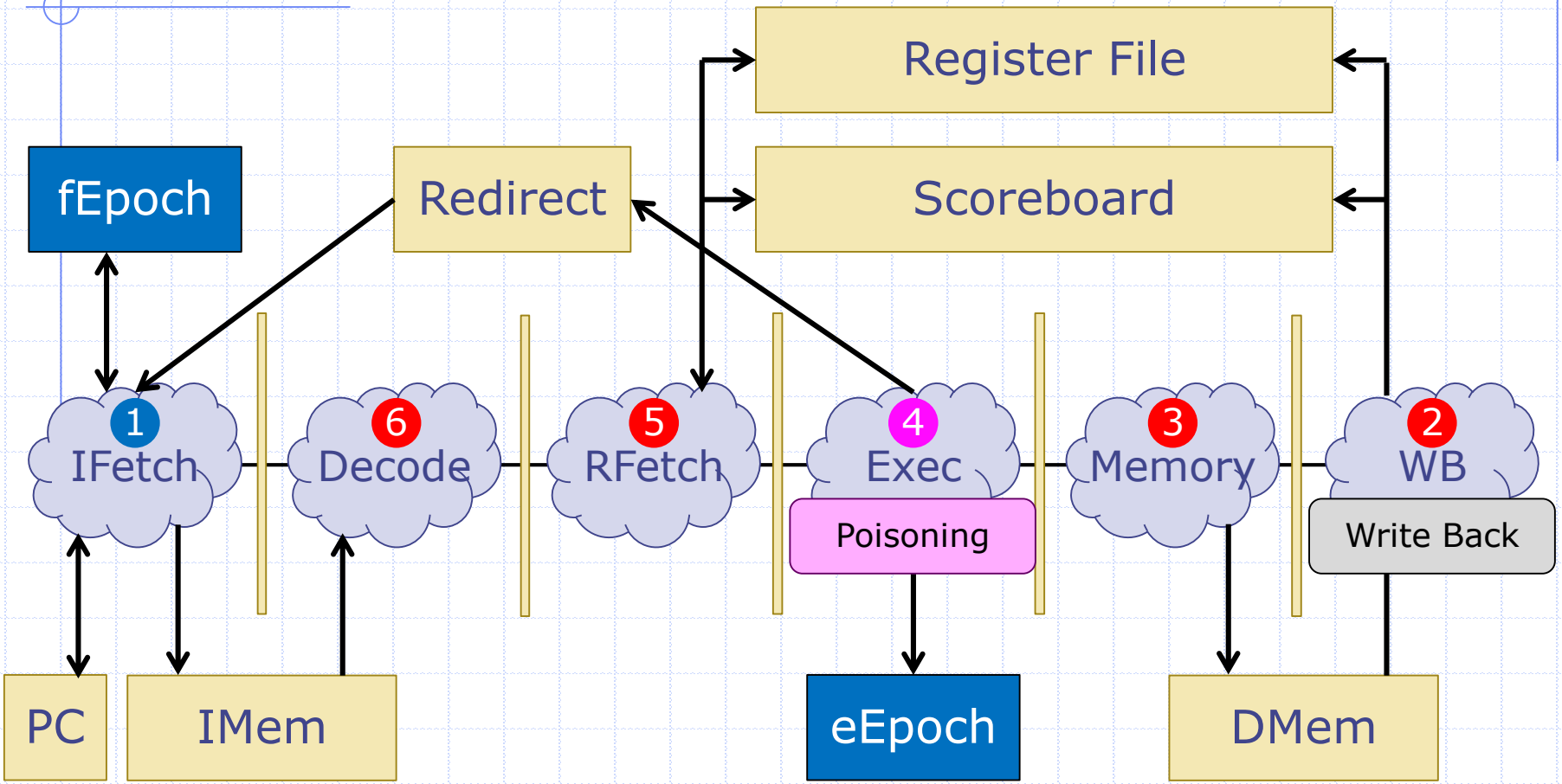
Handling Multiple Epochs

- 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.

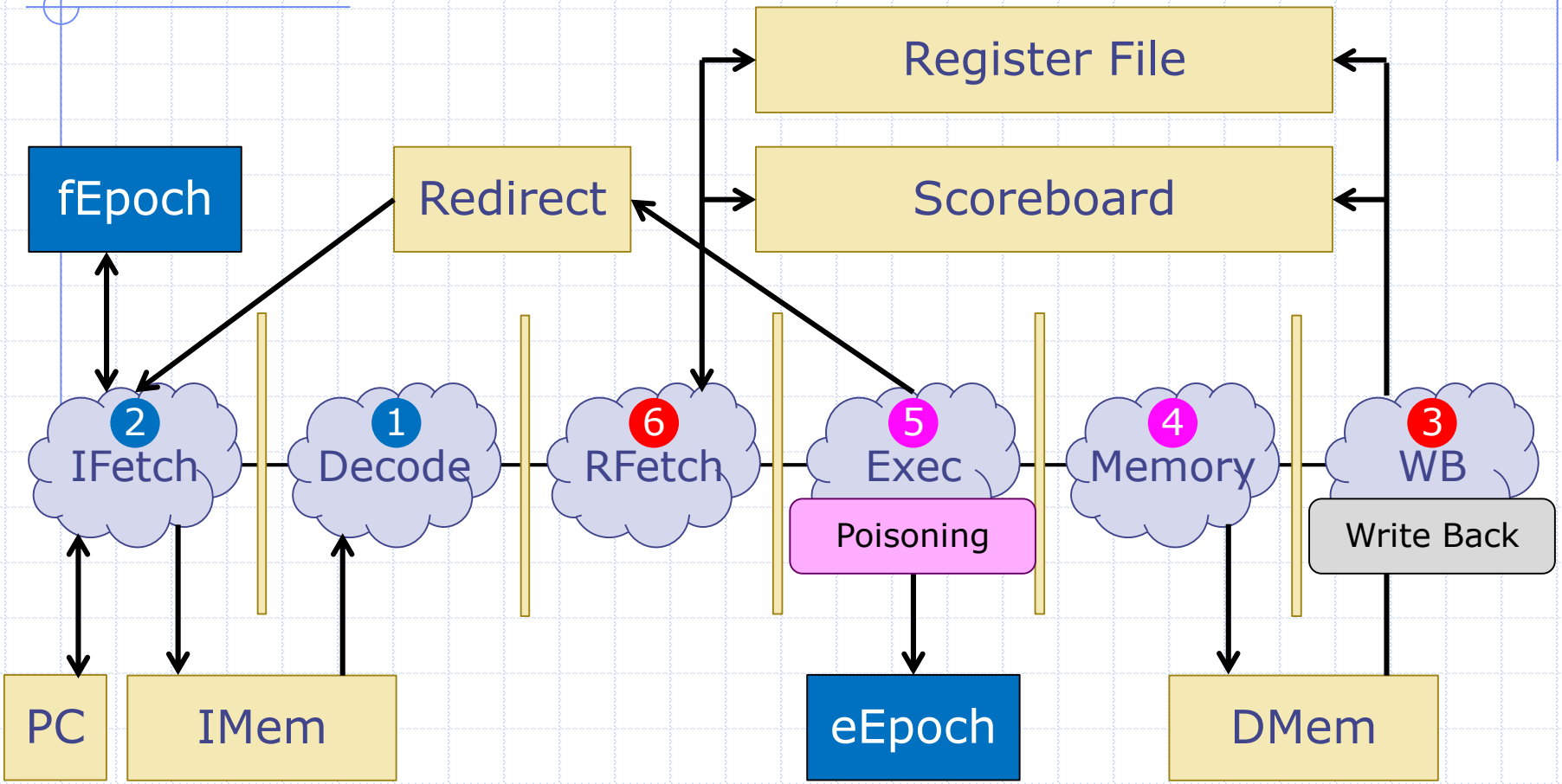
Correcting PC in Execute



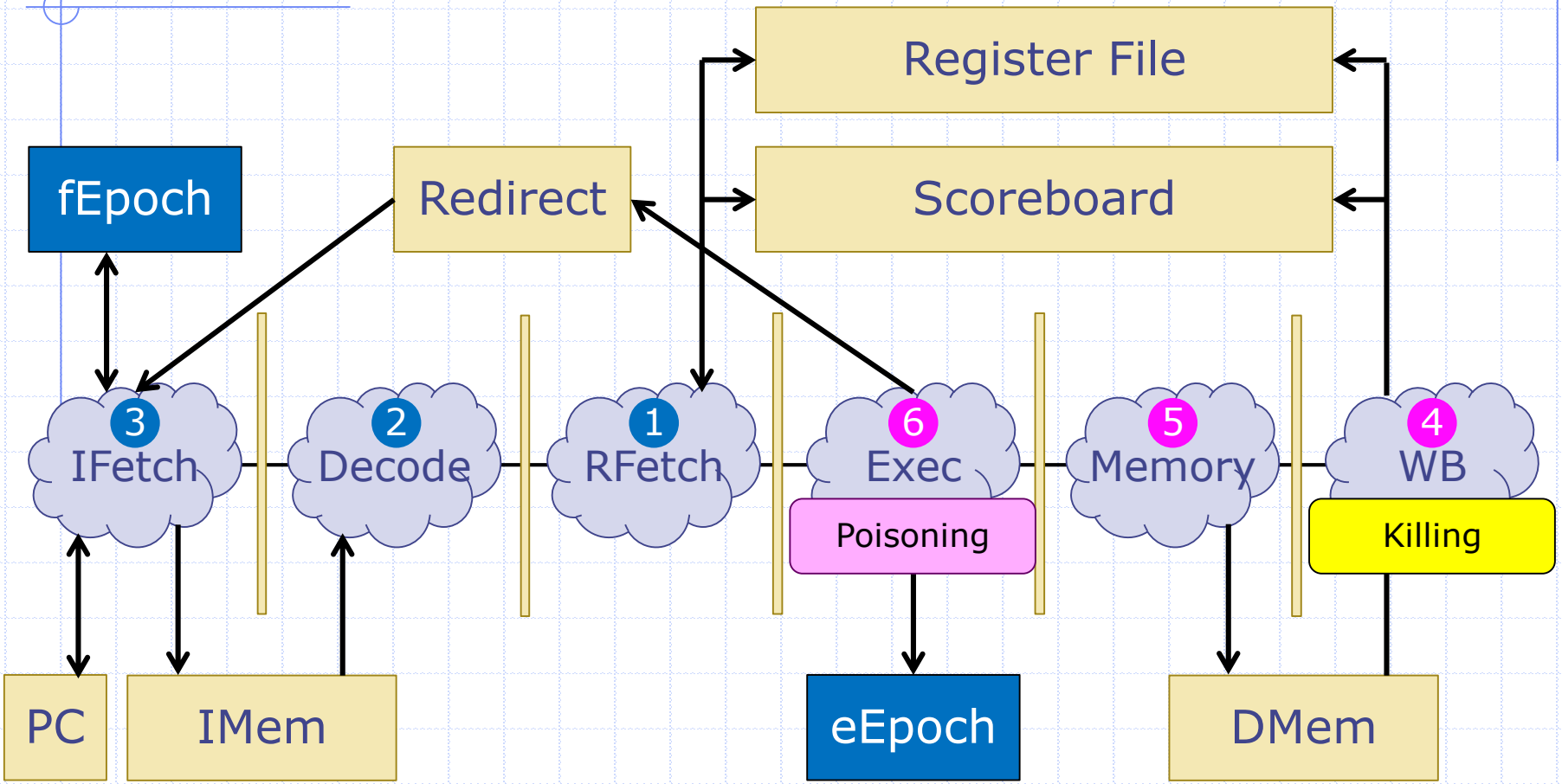
Correcting PC in Execute



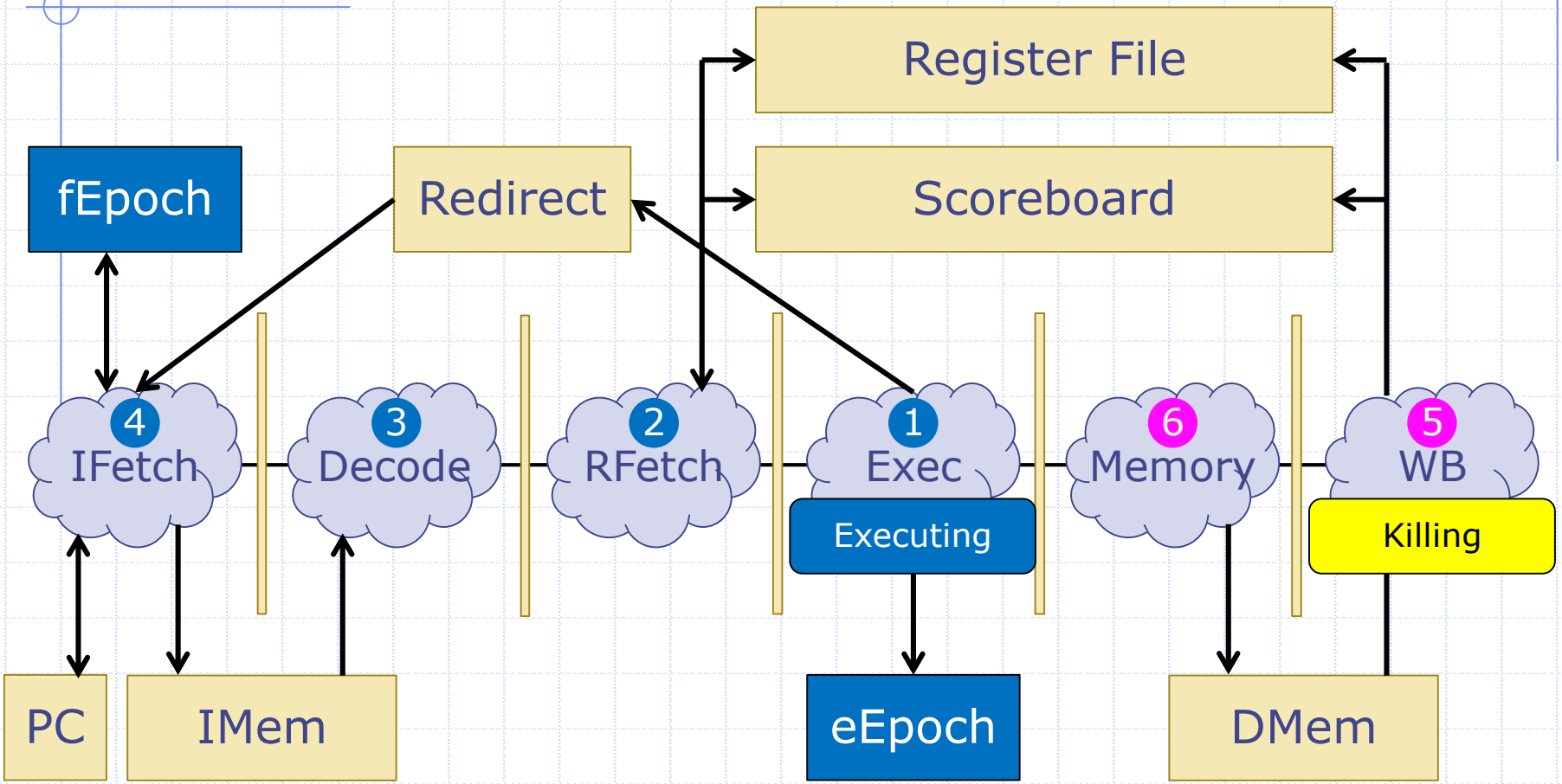
Correcting PC in Execute



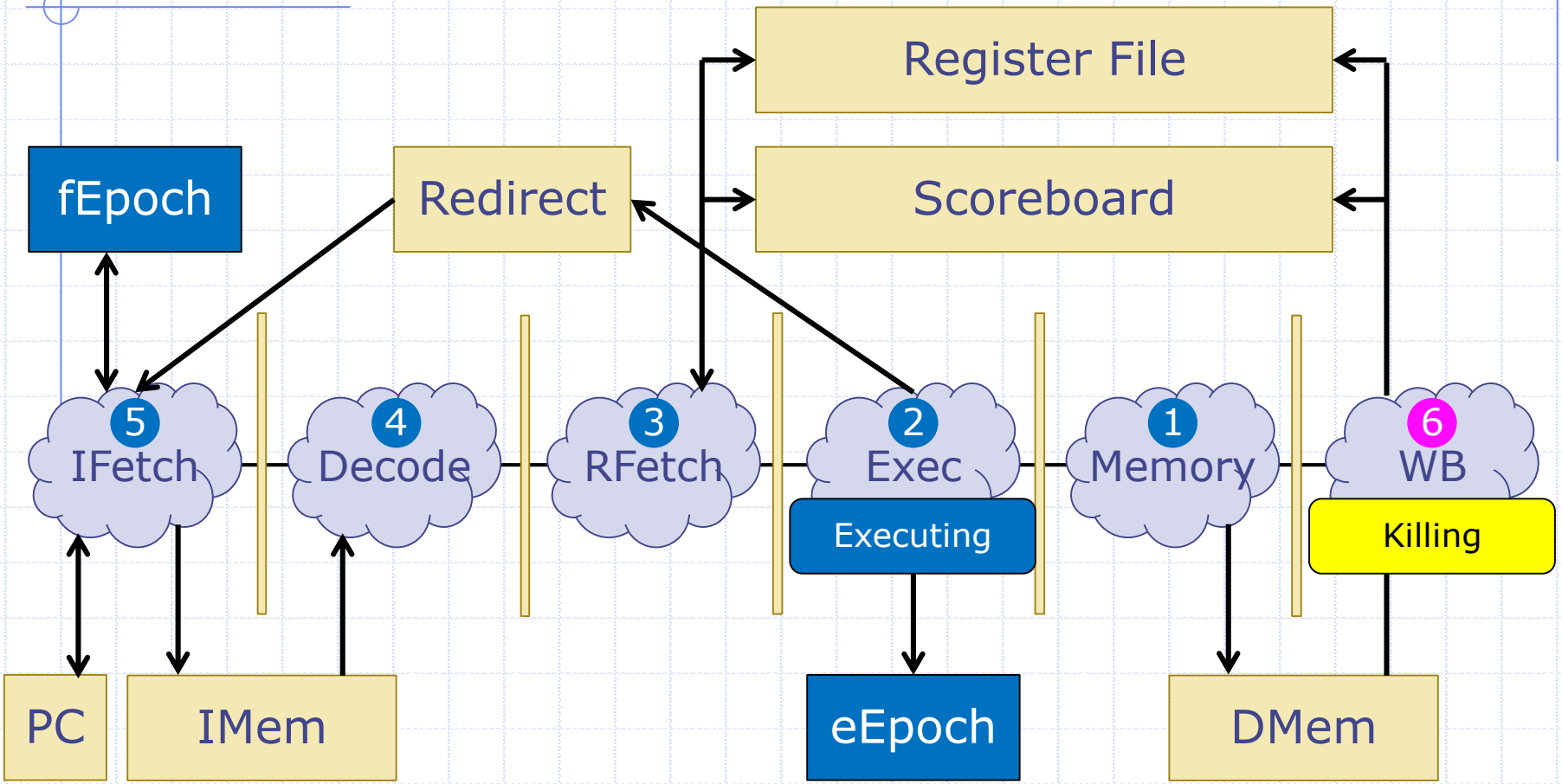
Correcting PC in Execute



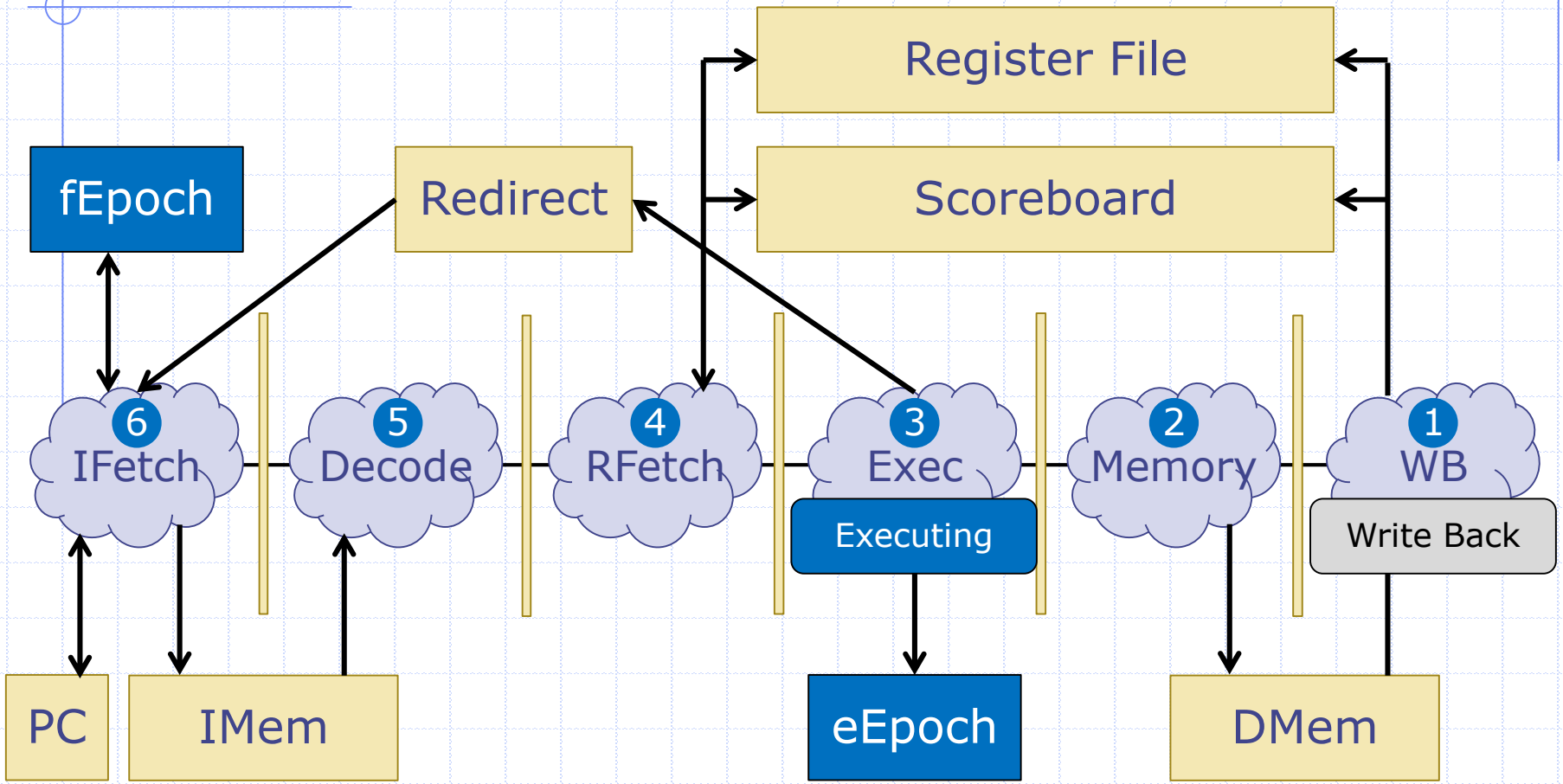
Correcting PC in Execute



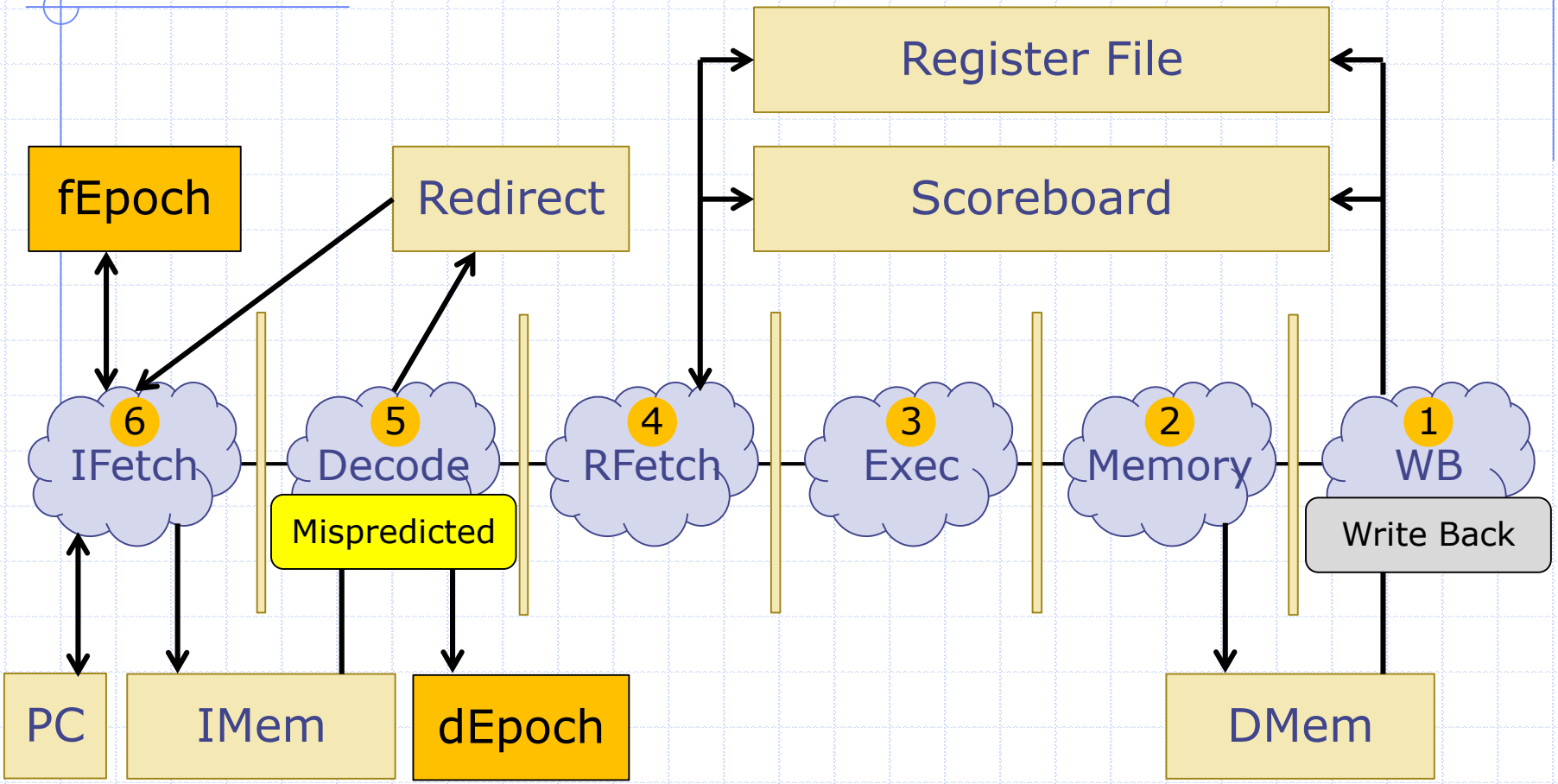
Correcting PC in Execute



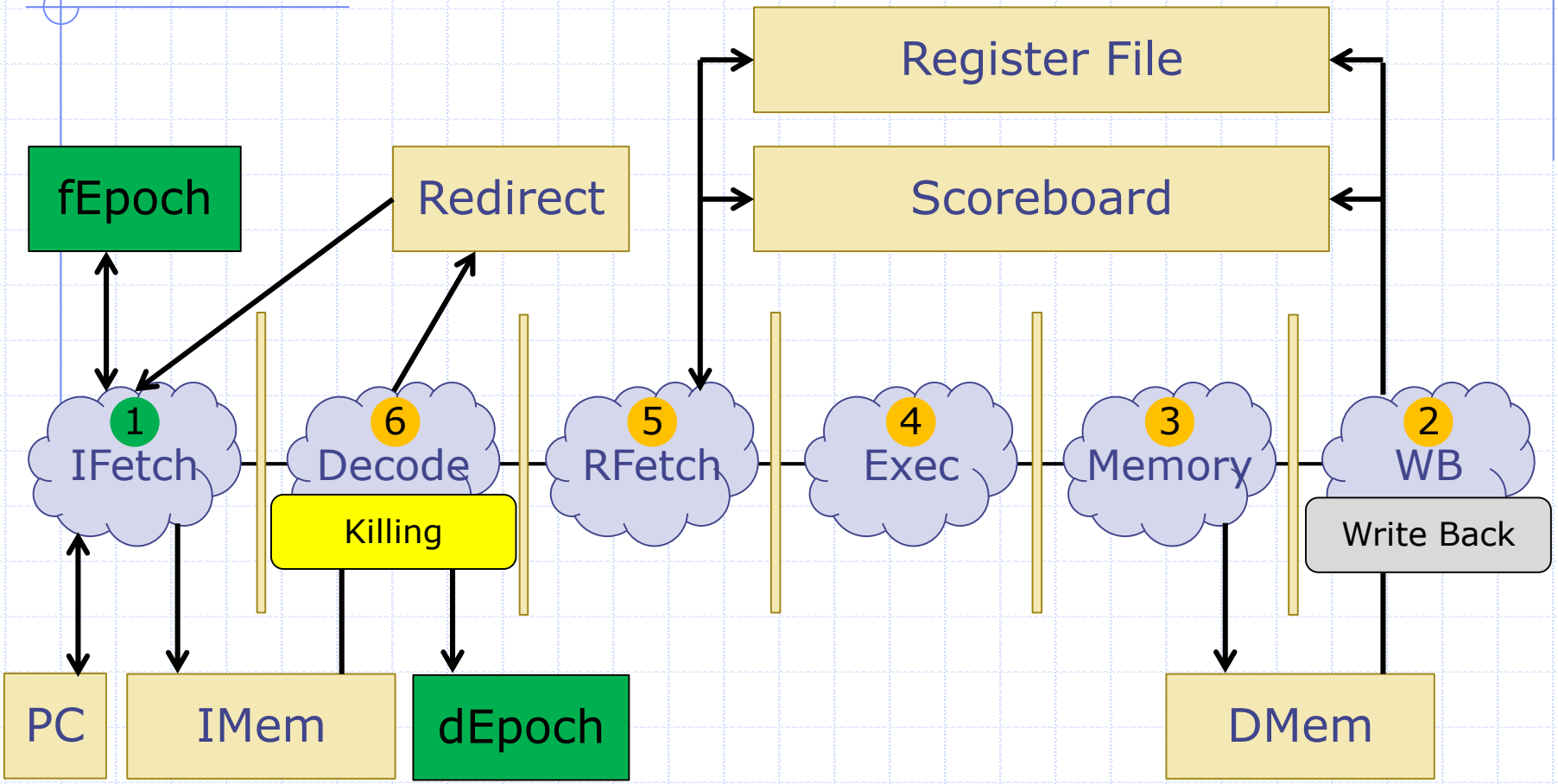
Correcting PC in Execute



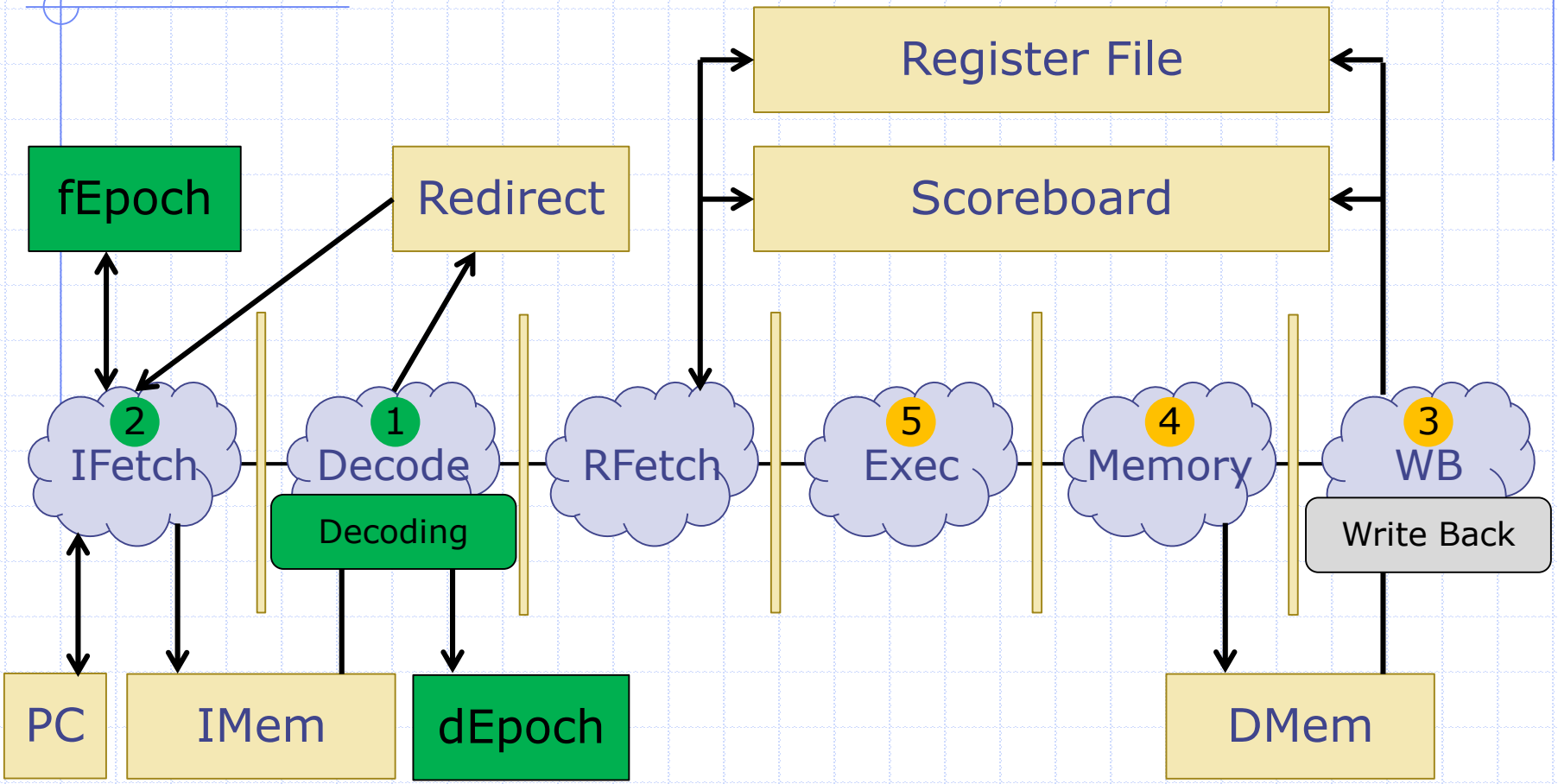
Correcting PC in Decode



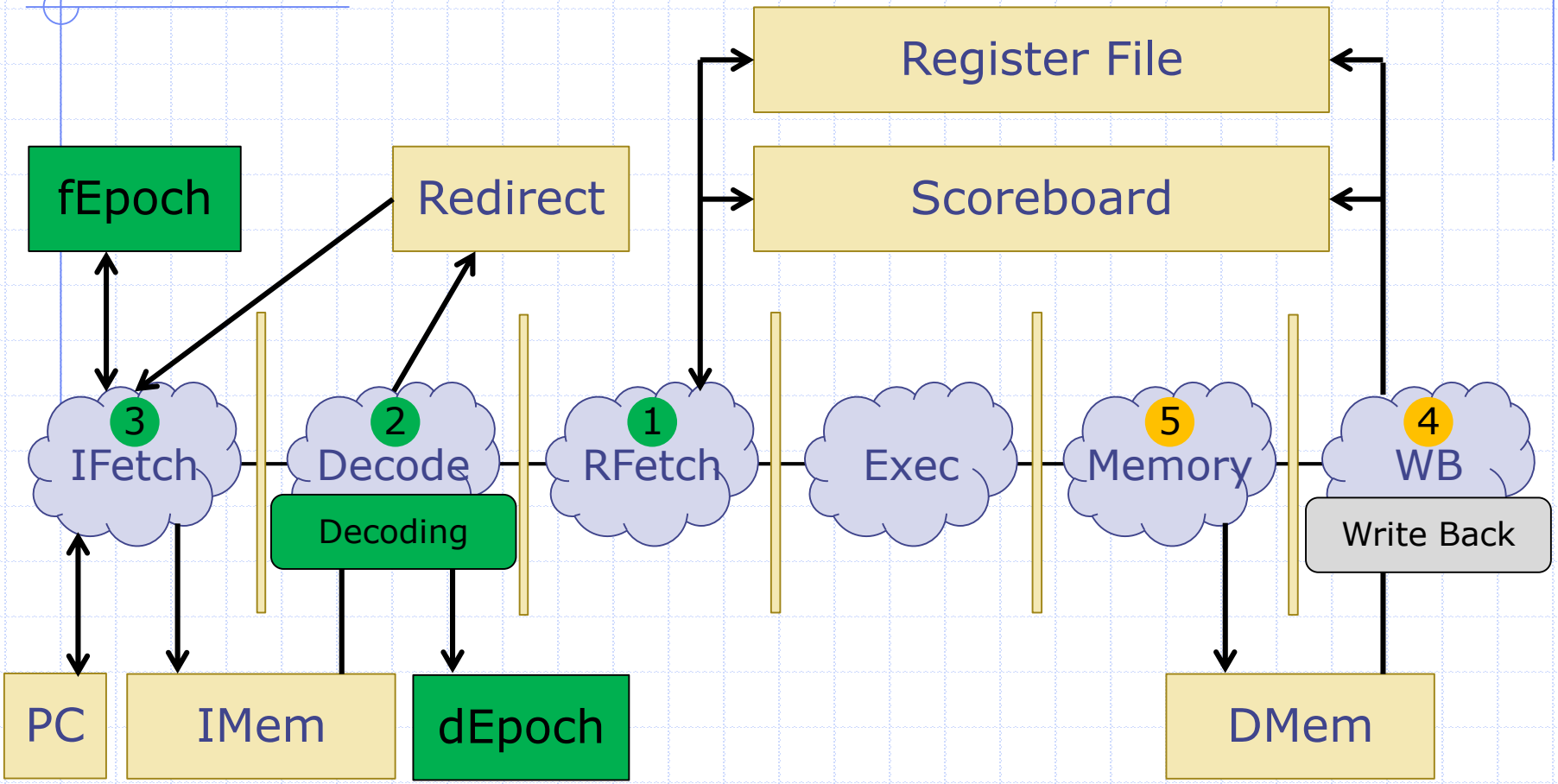
Correcting PC in Decode



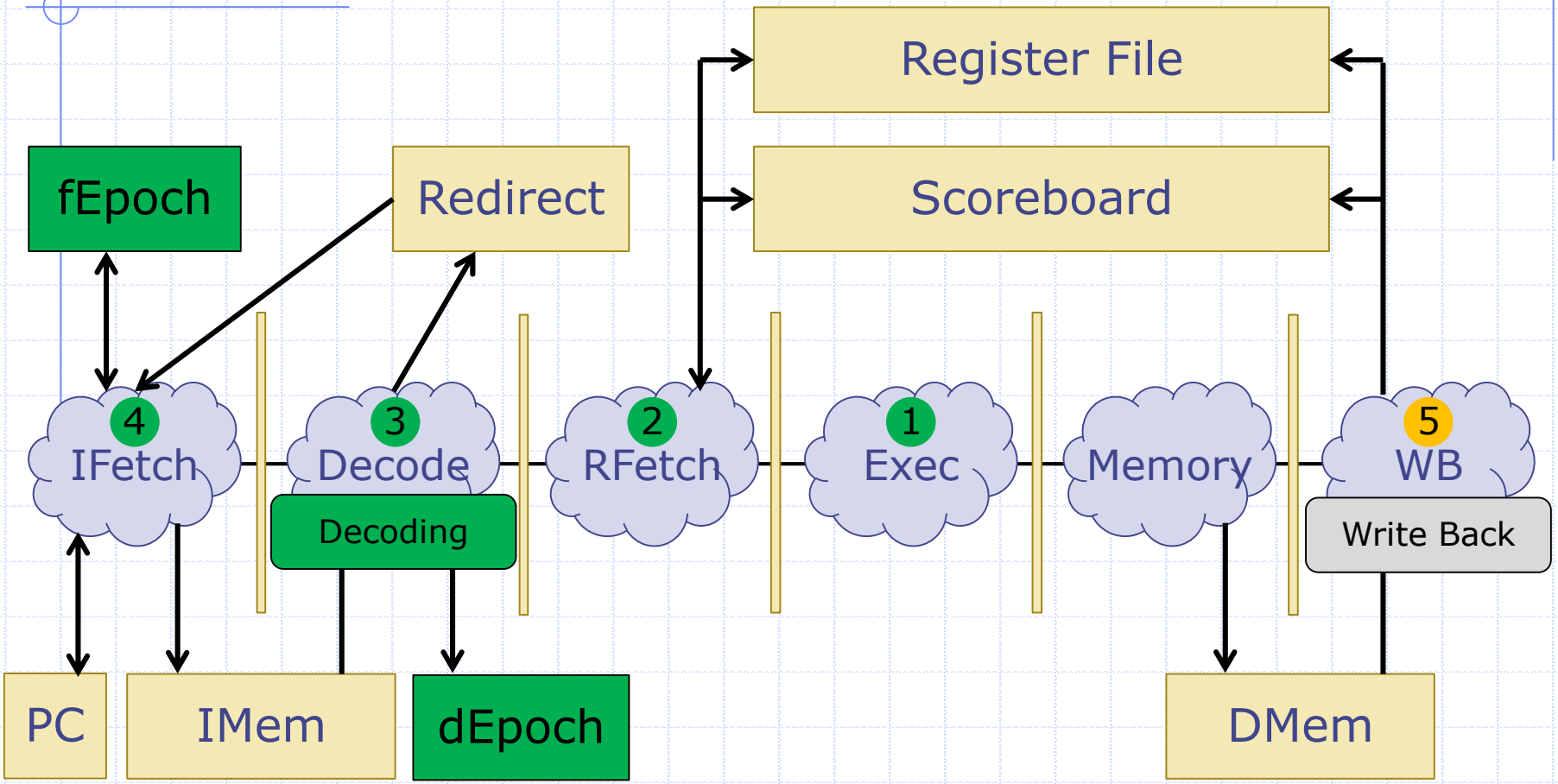
Correcting PC in Decode



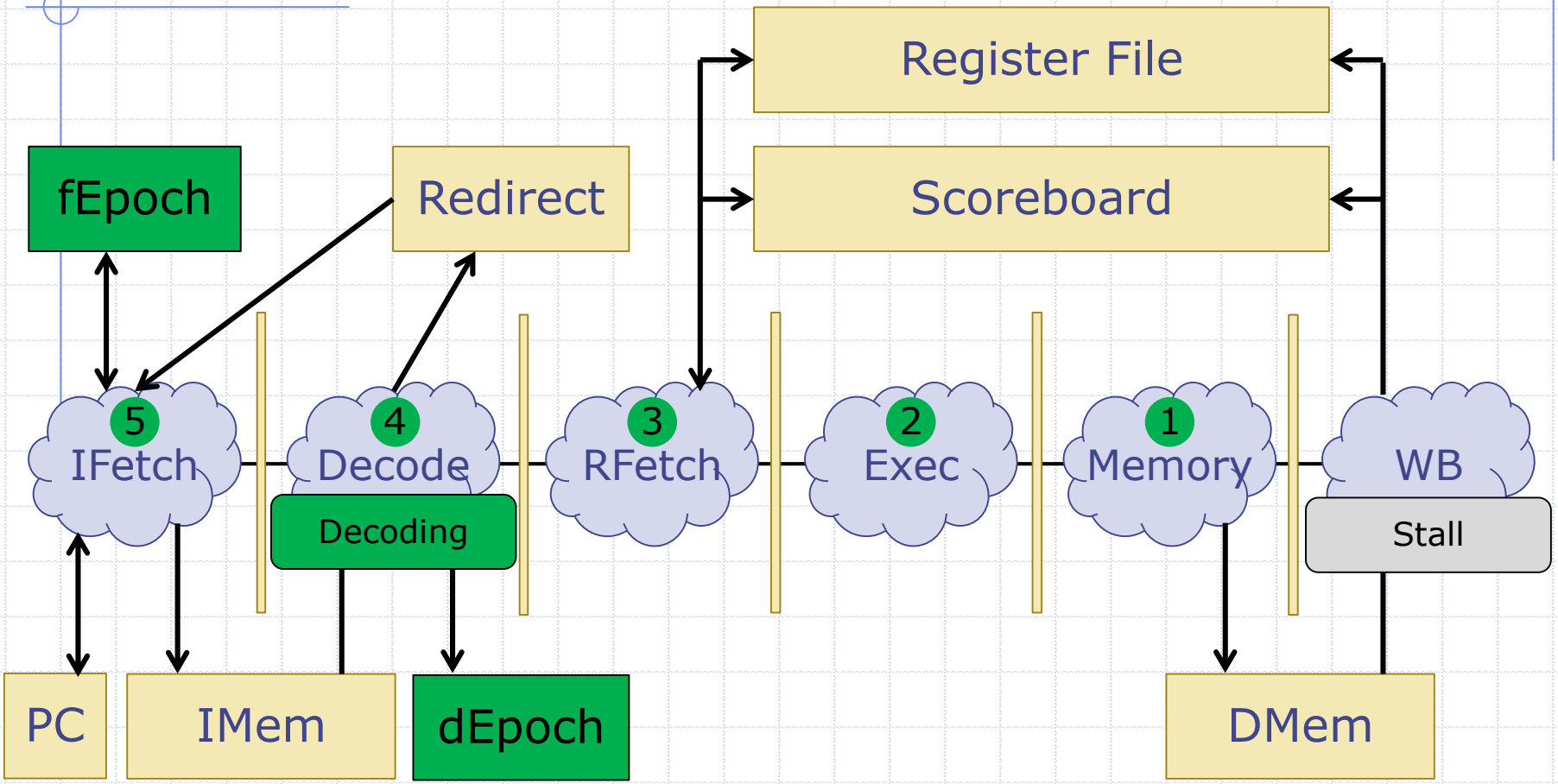
Correcting PC in Decode



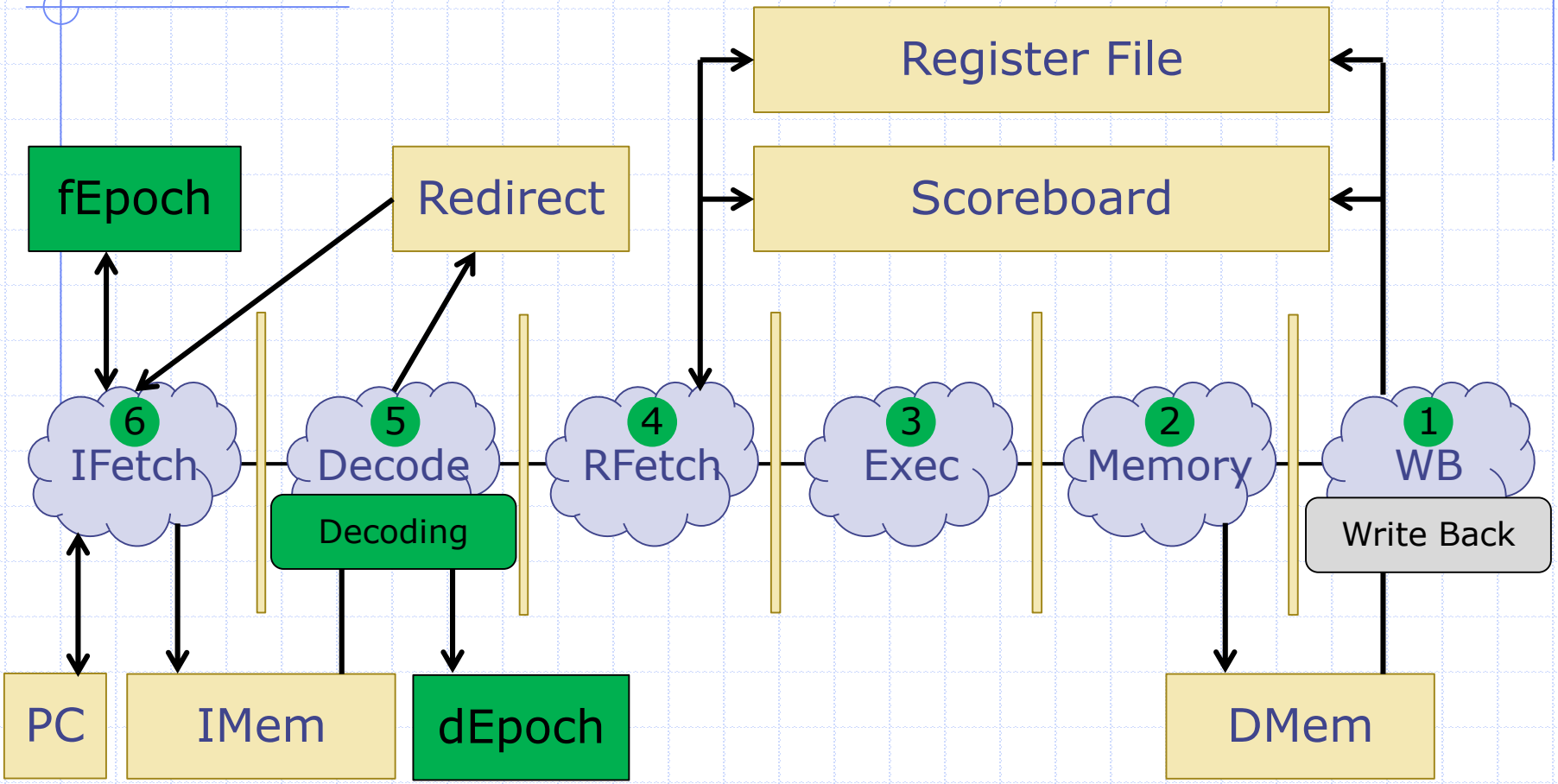
Correcting PC in Decode



Correcting PC in Decode



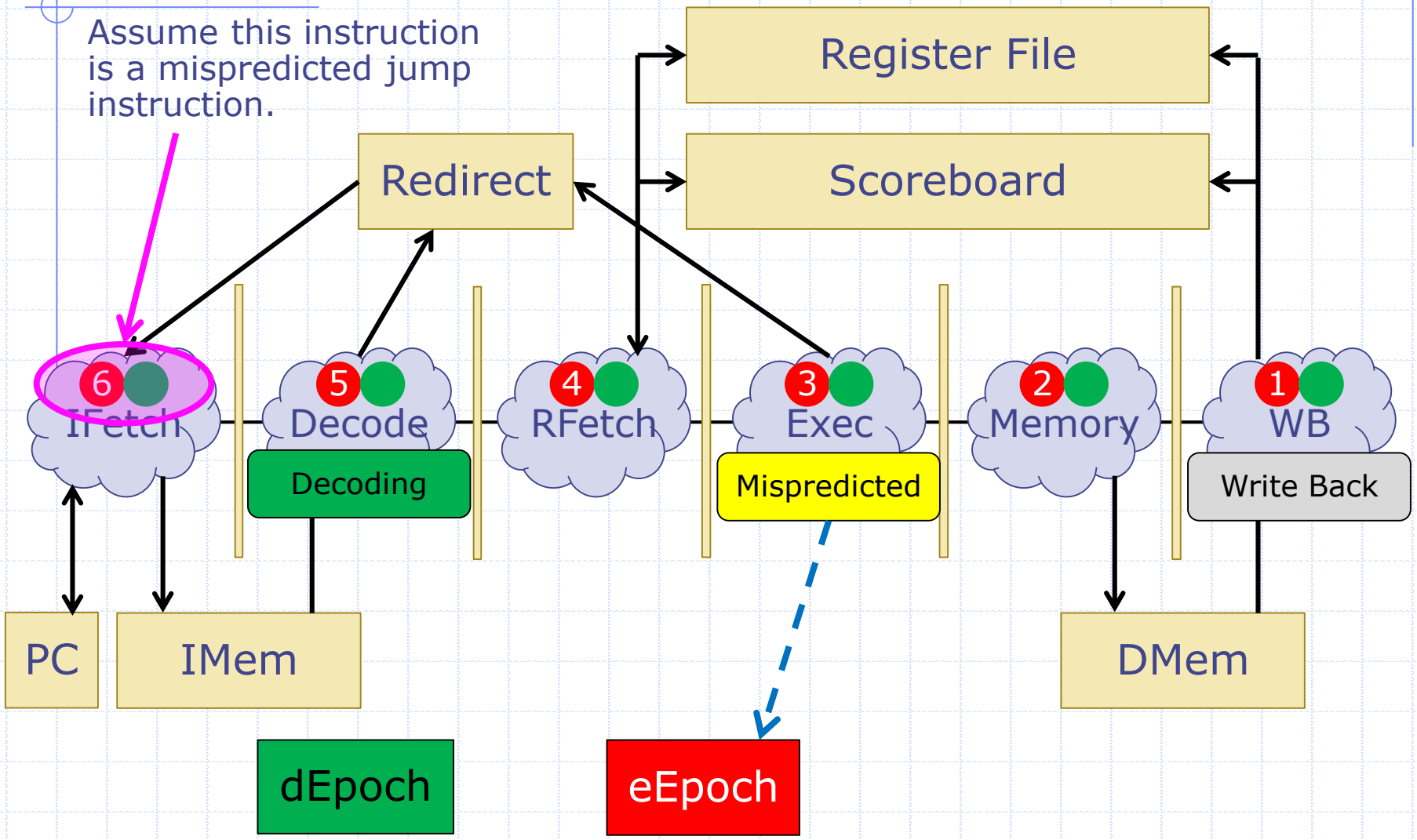
Correcting PC in Decode



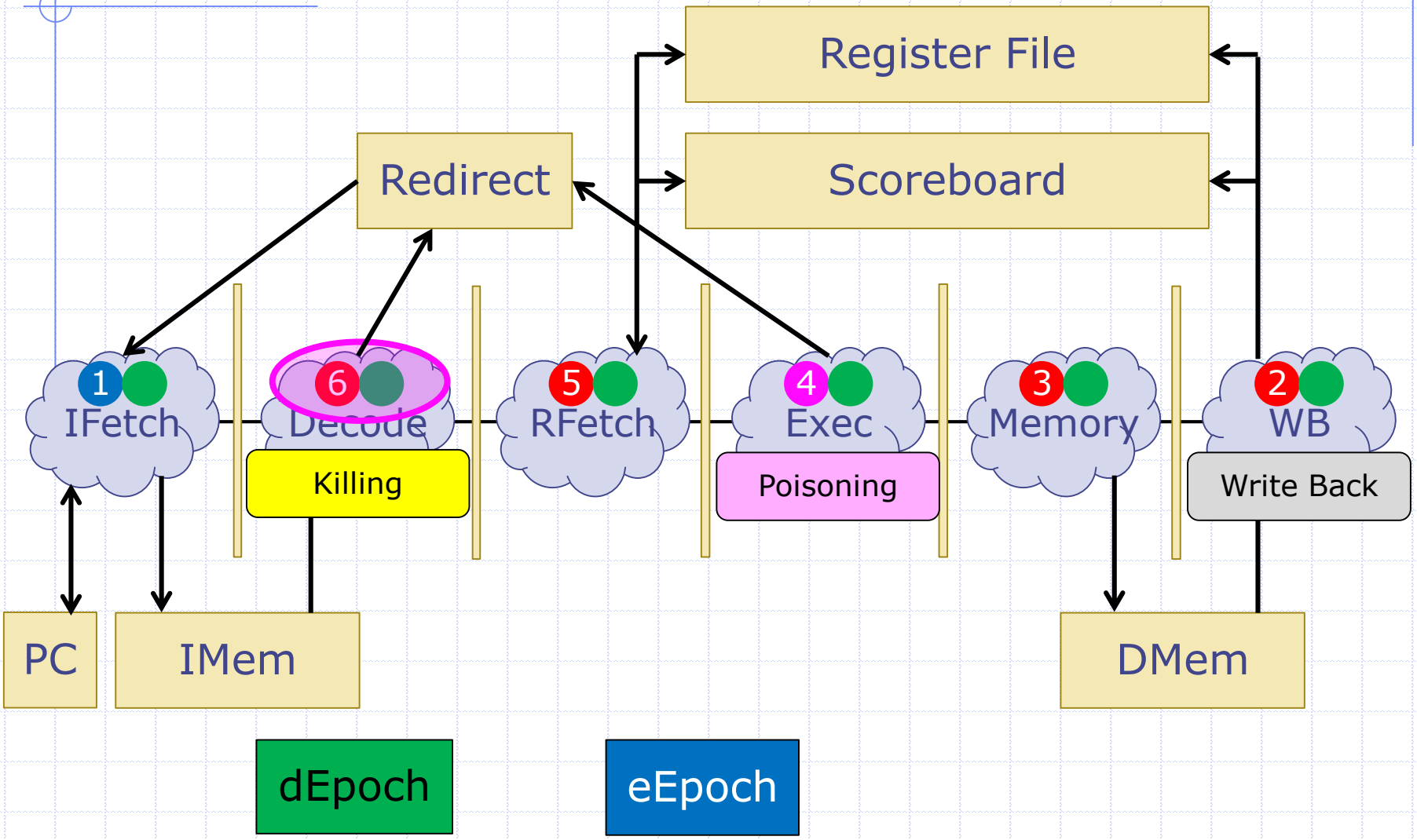
Global Epoch States

- 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

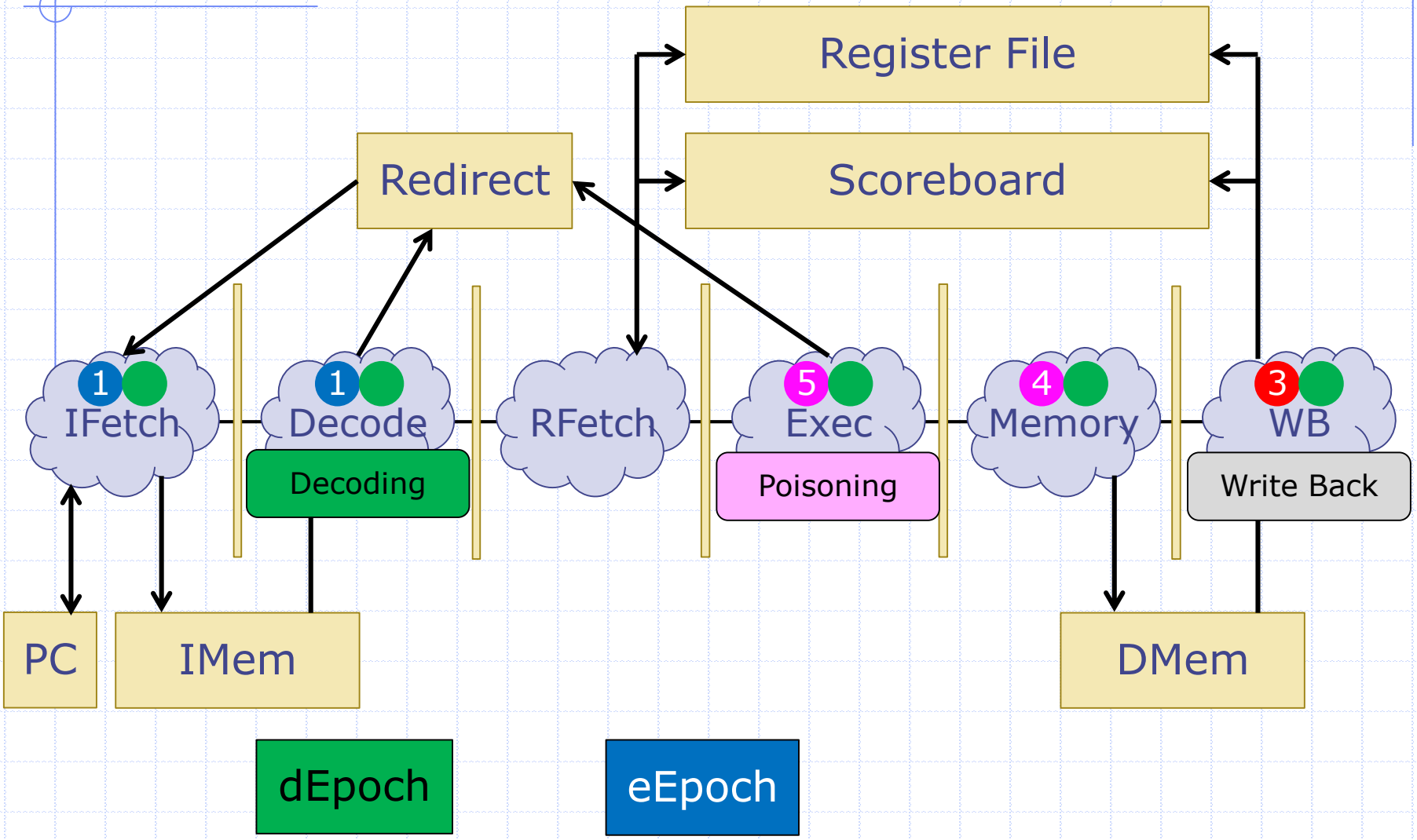
Global Epoch States



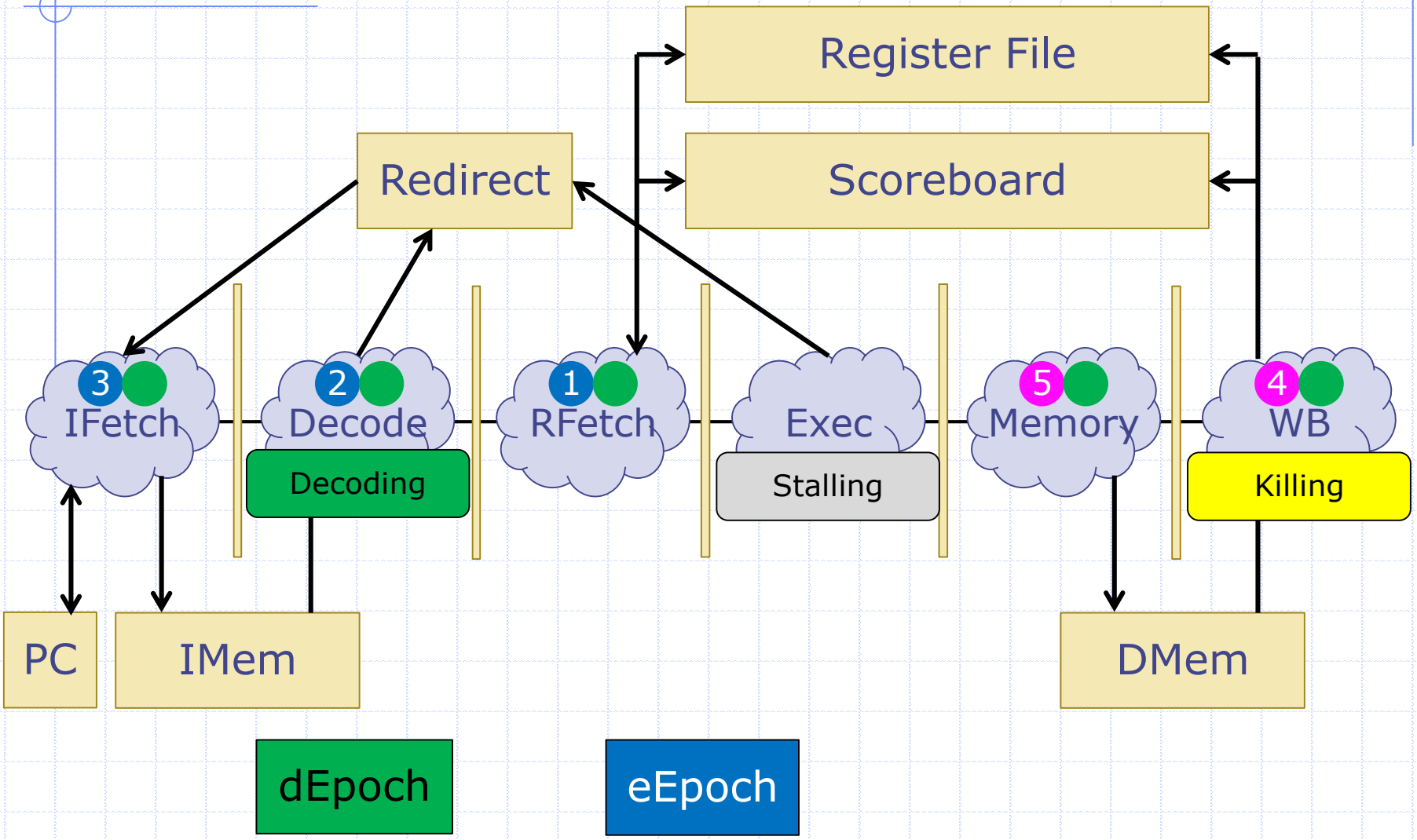
Global Epoch States



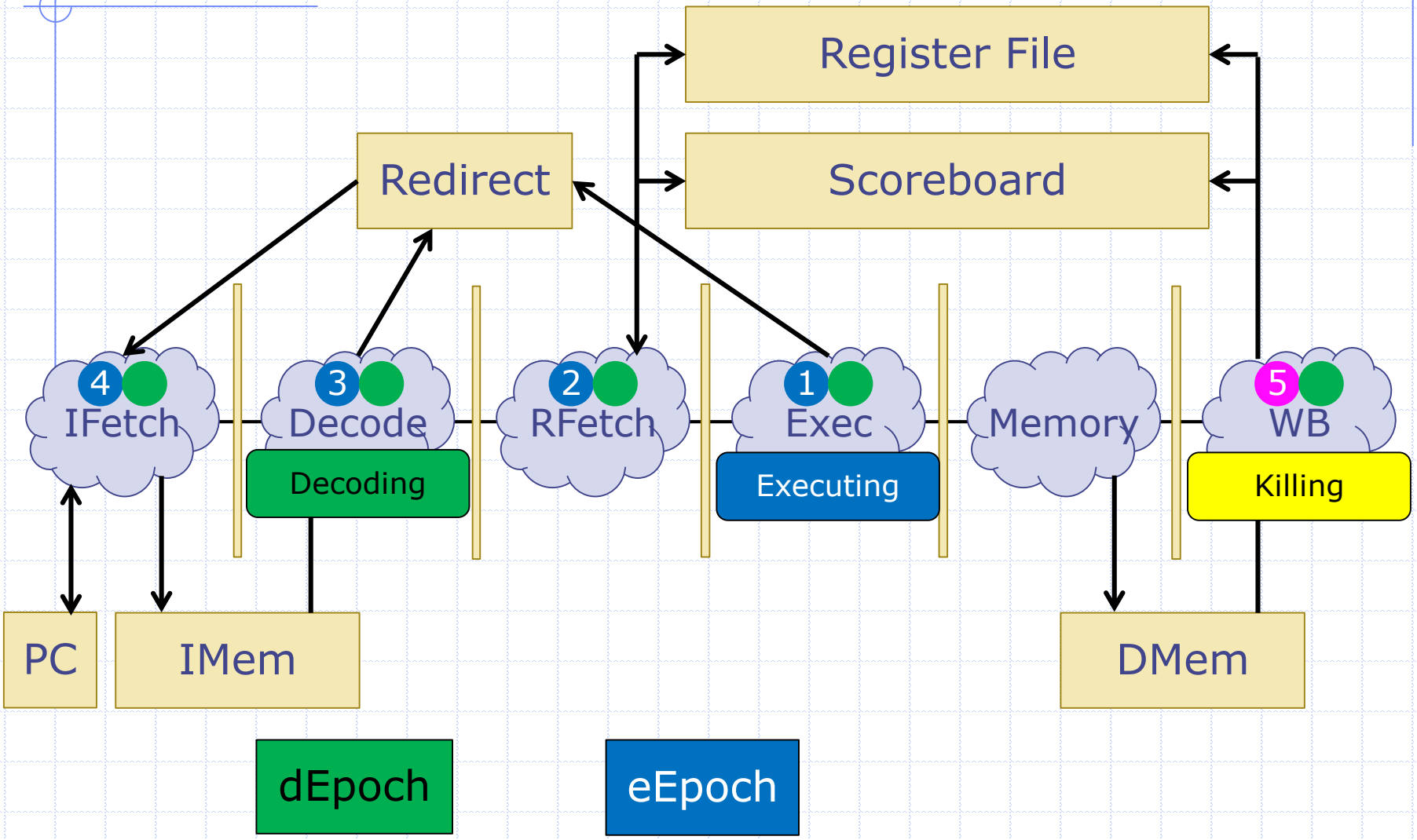
Global Epoch States



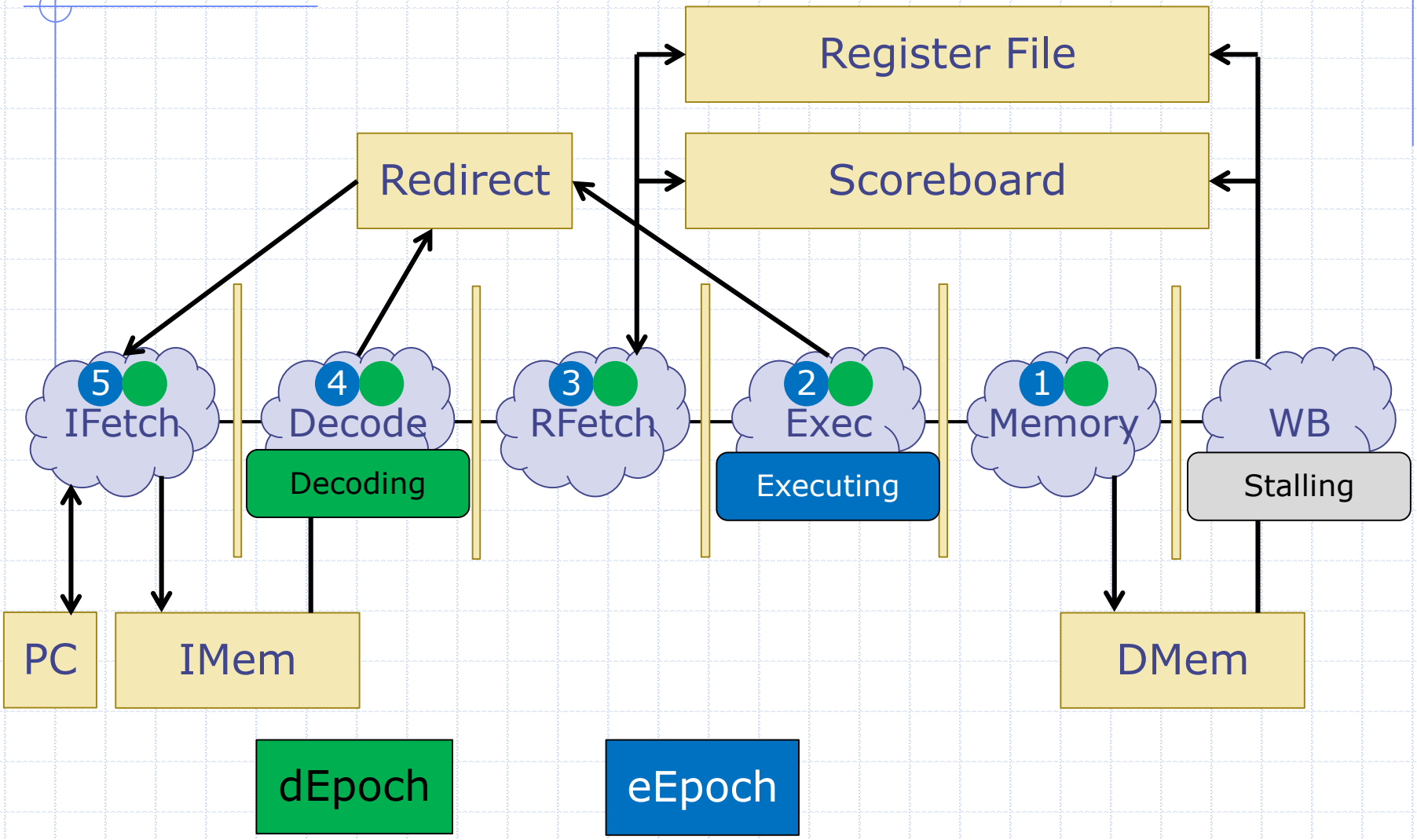
Global Epoch States



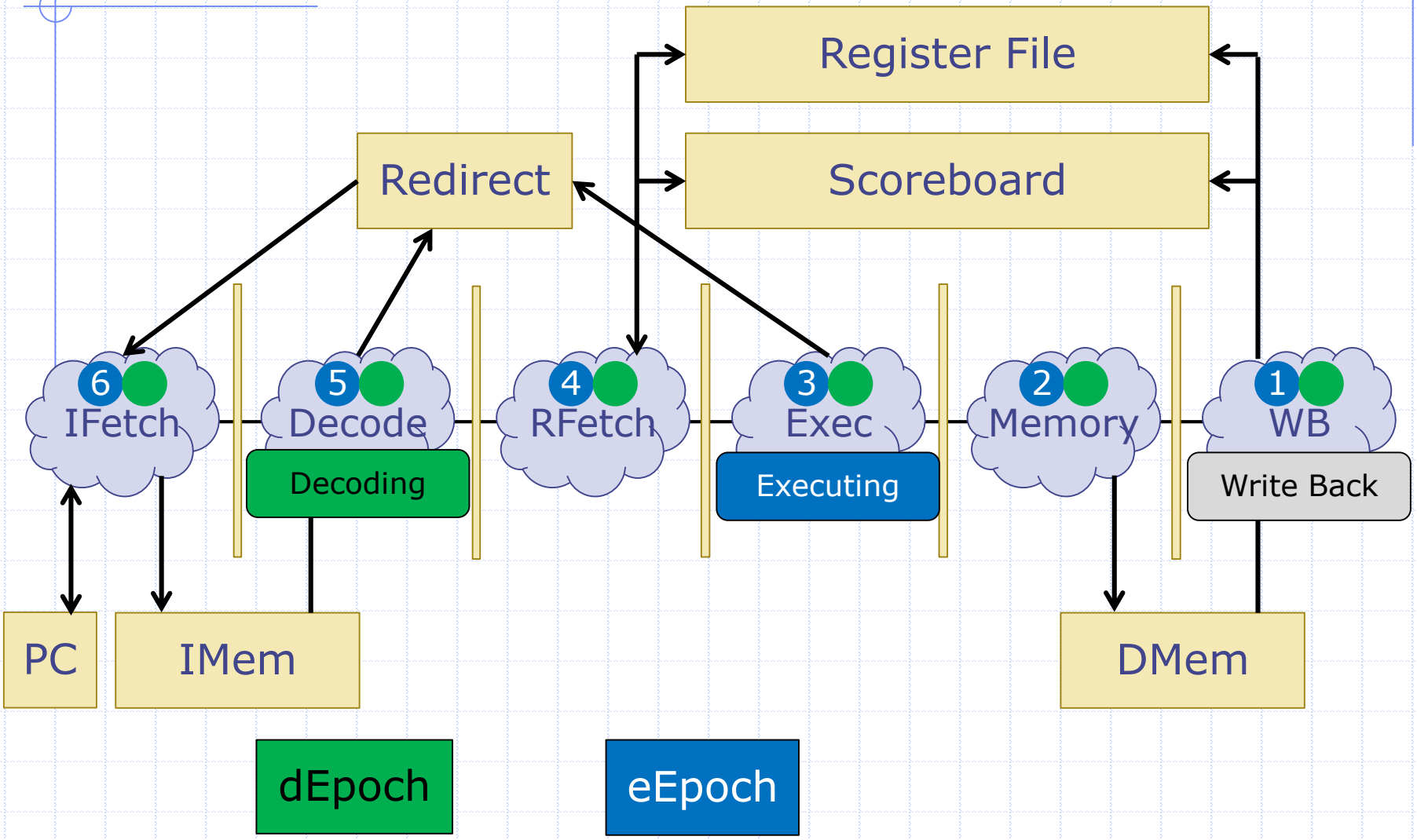
Global Epoch States



Global Epoch States



Global Epoch States

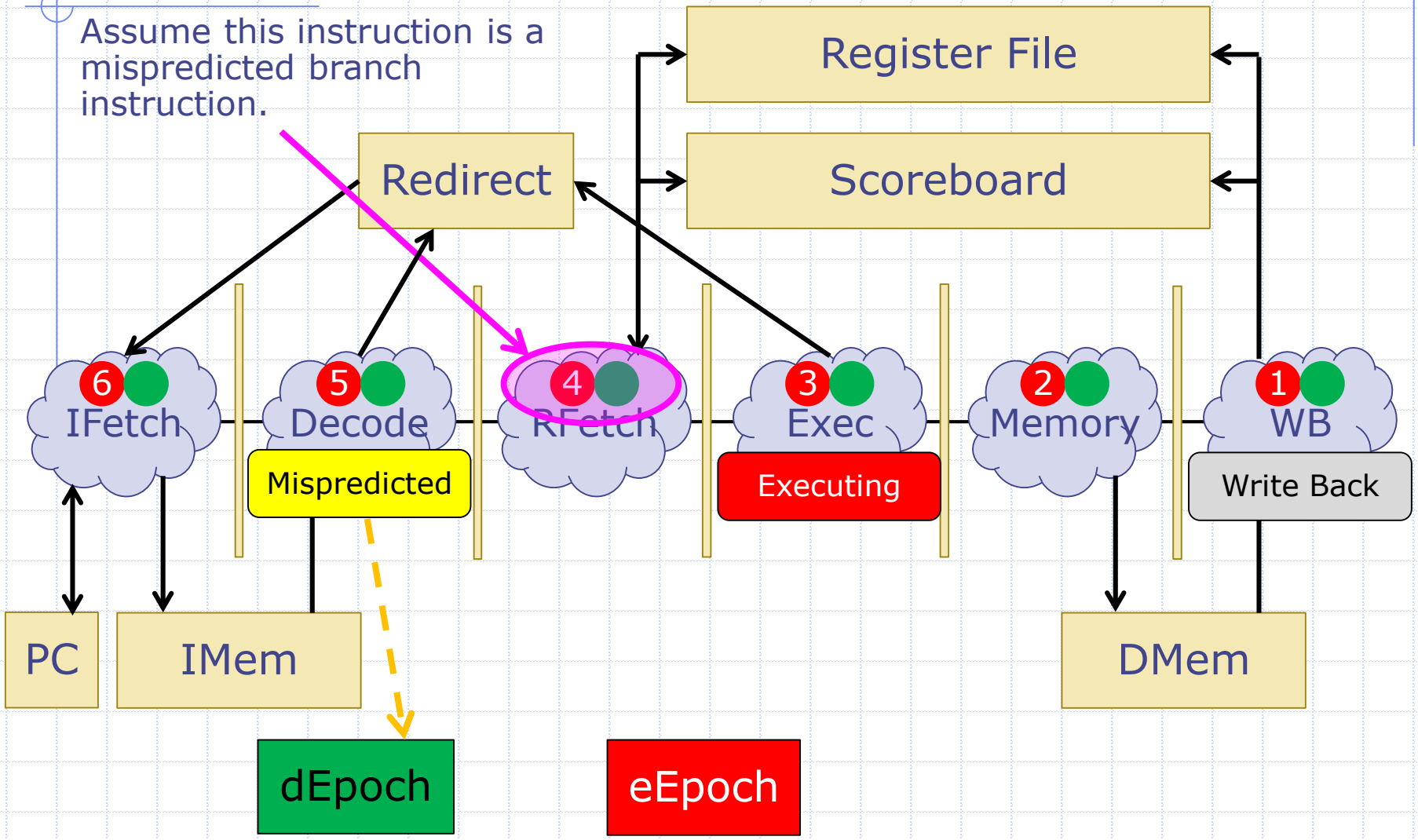


Global Epoch States

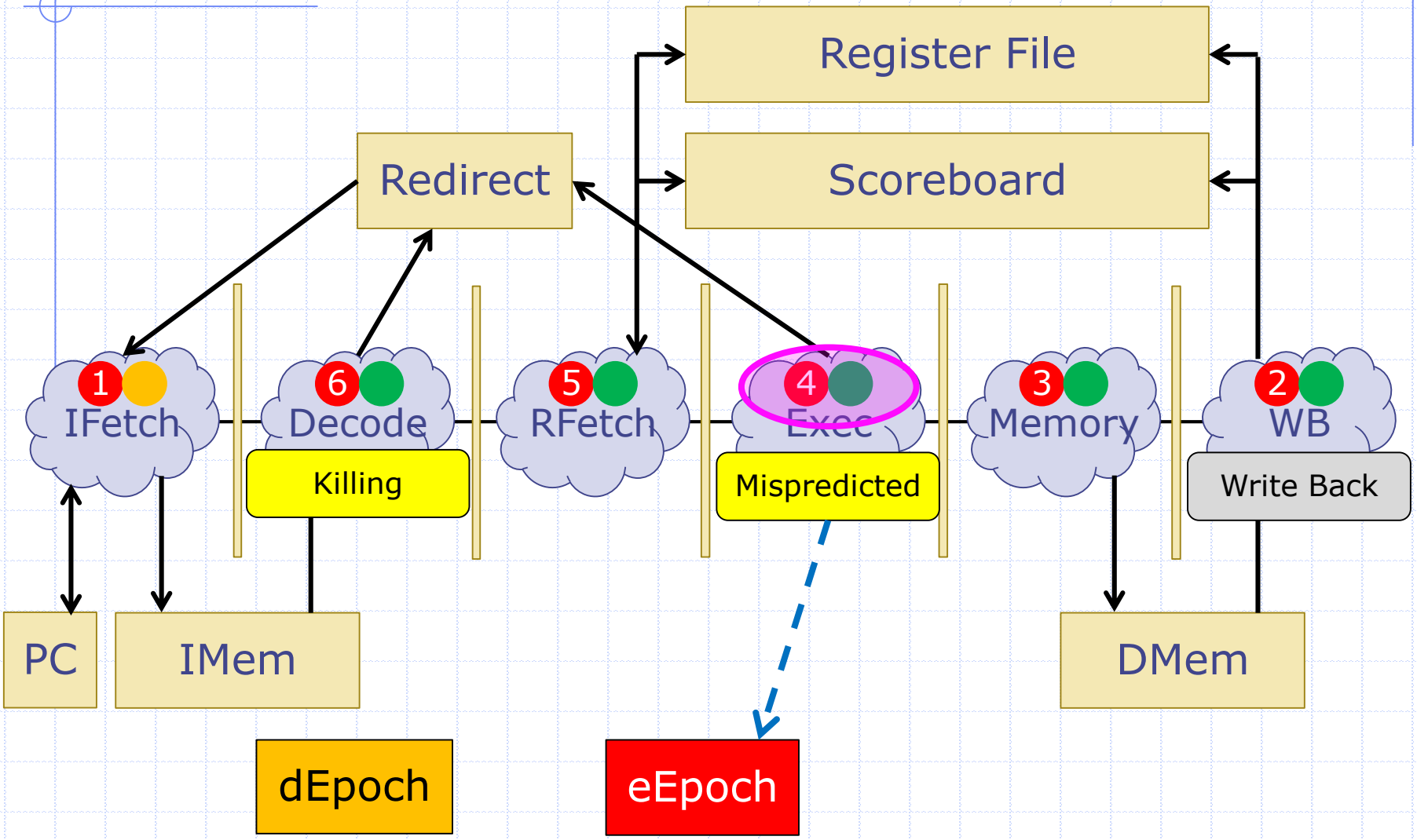
- What if decode sees a misprediction, then execute sees one in the next cycle?
  - The decode instruction will be a wrong path instruction, but it won't be known to be wrong path until later

Global Epoch States

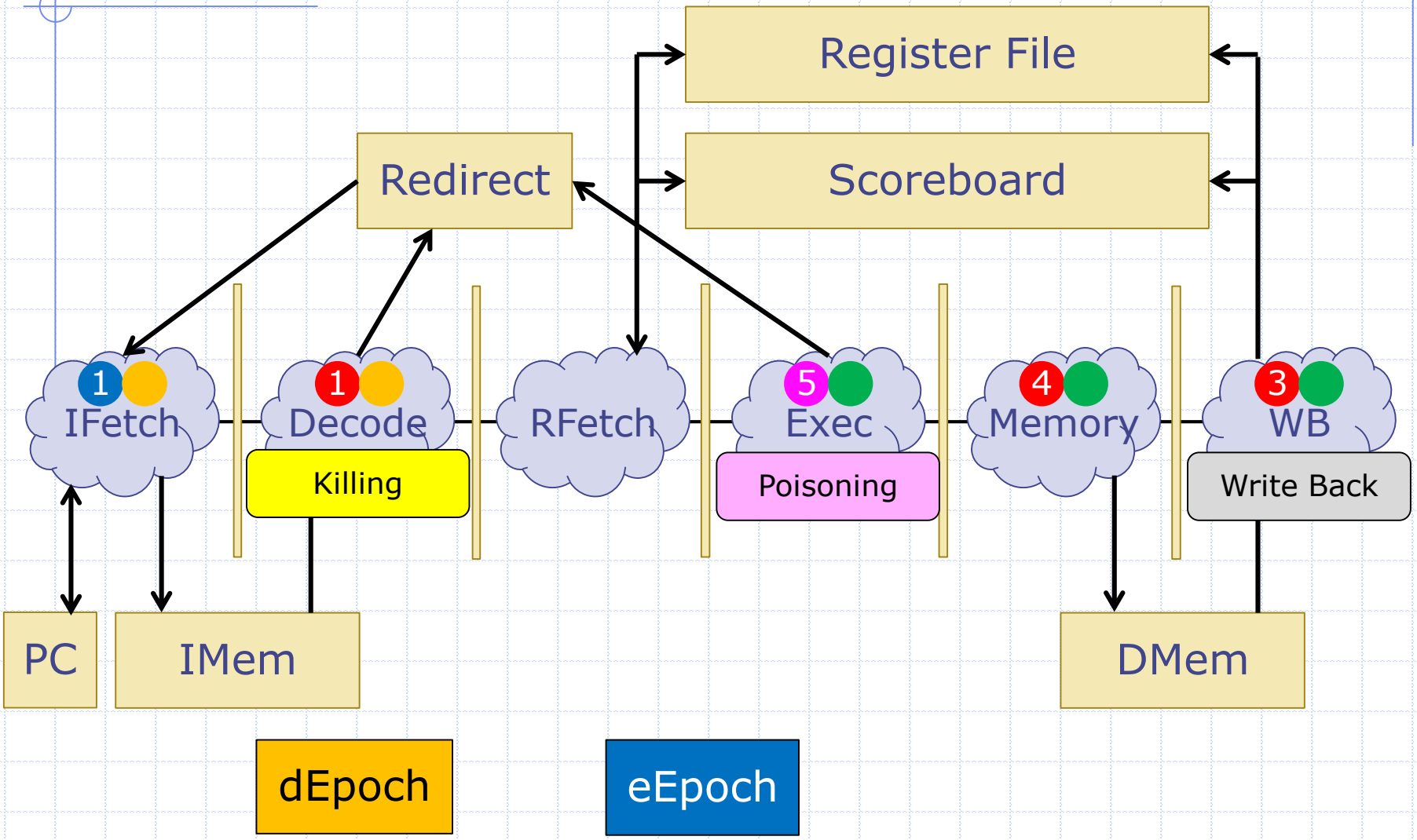
Assume this instruction is a mispredicted branch instruction.



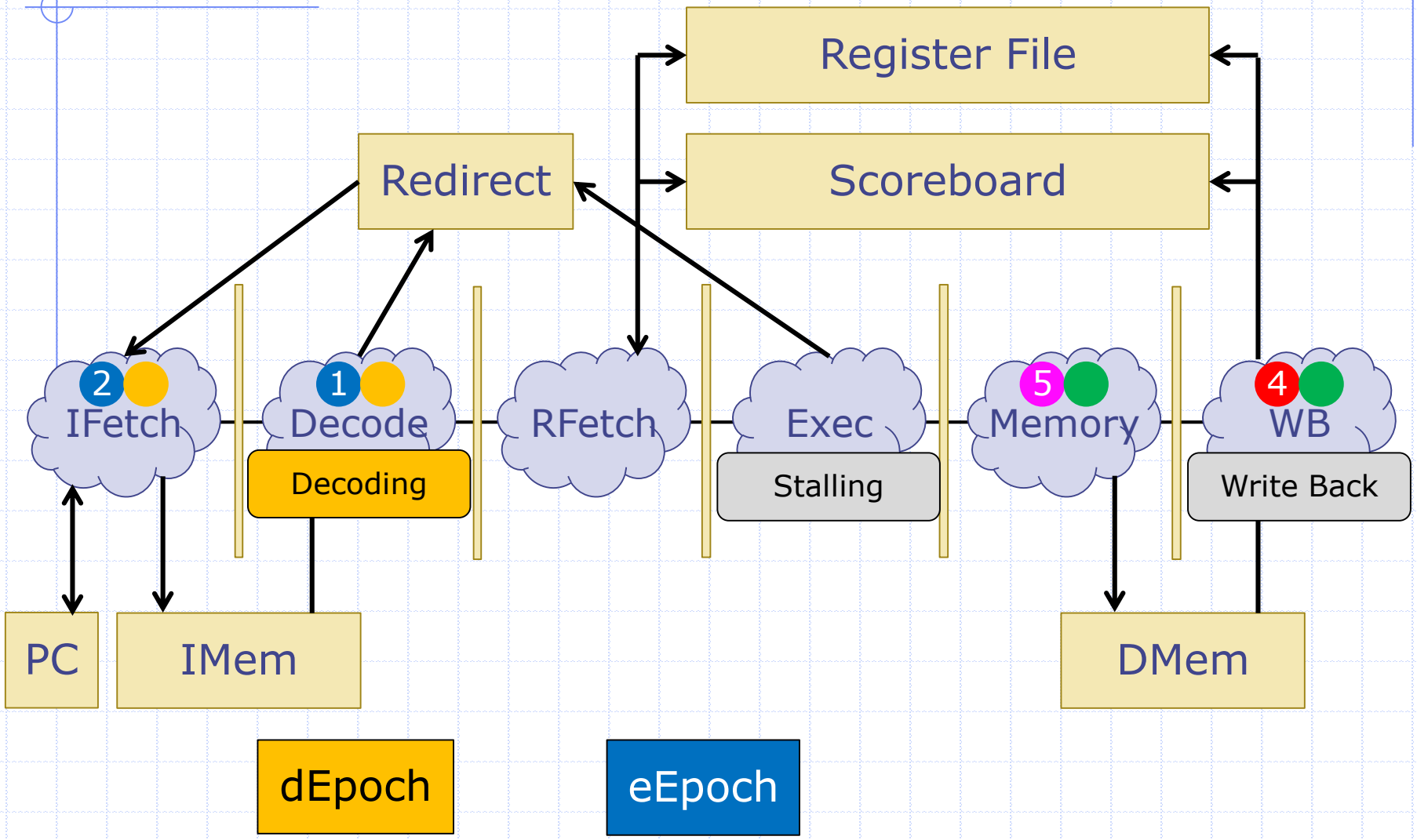
Global Epoch States



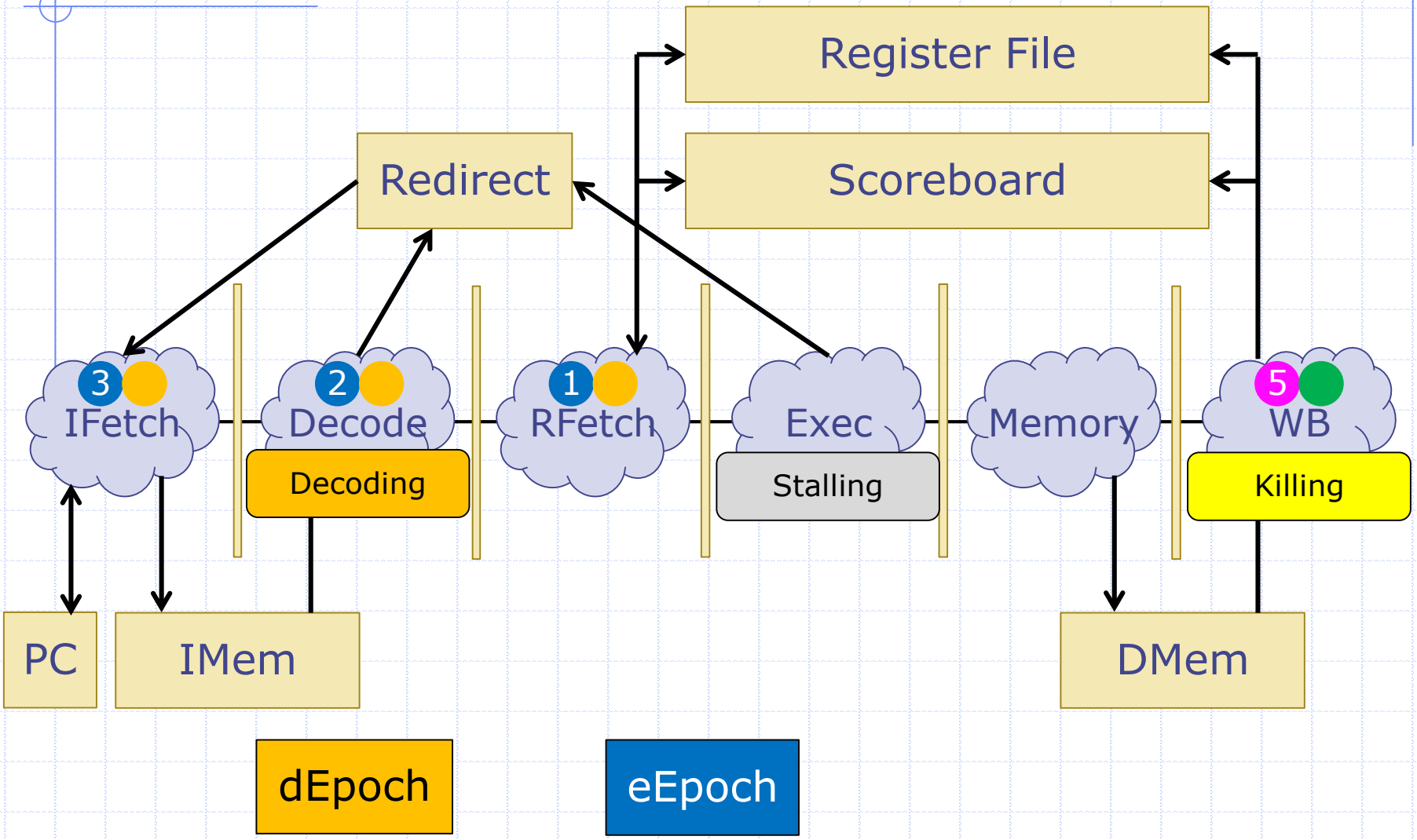
Global Epoch States



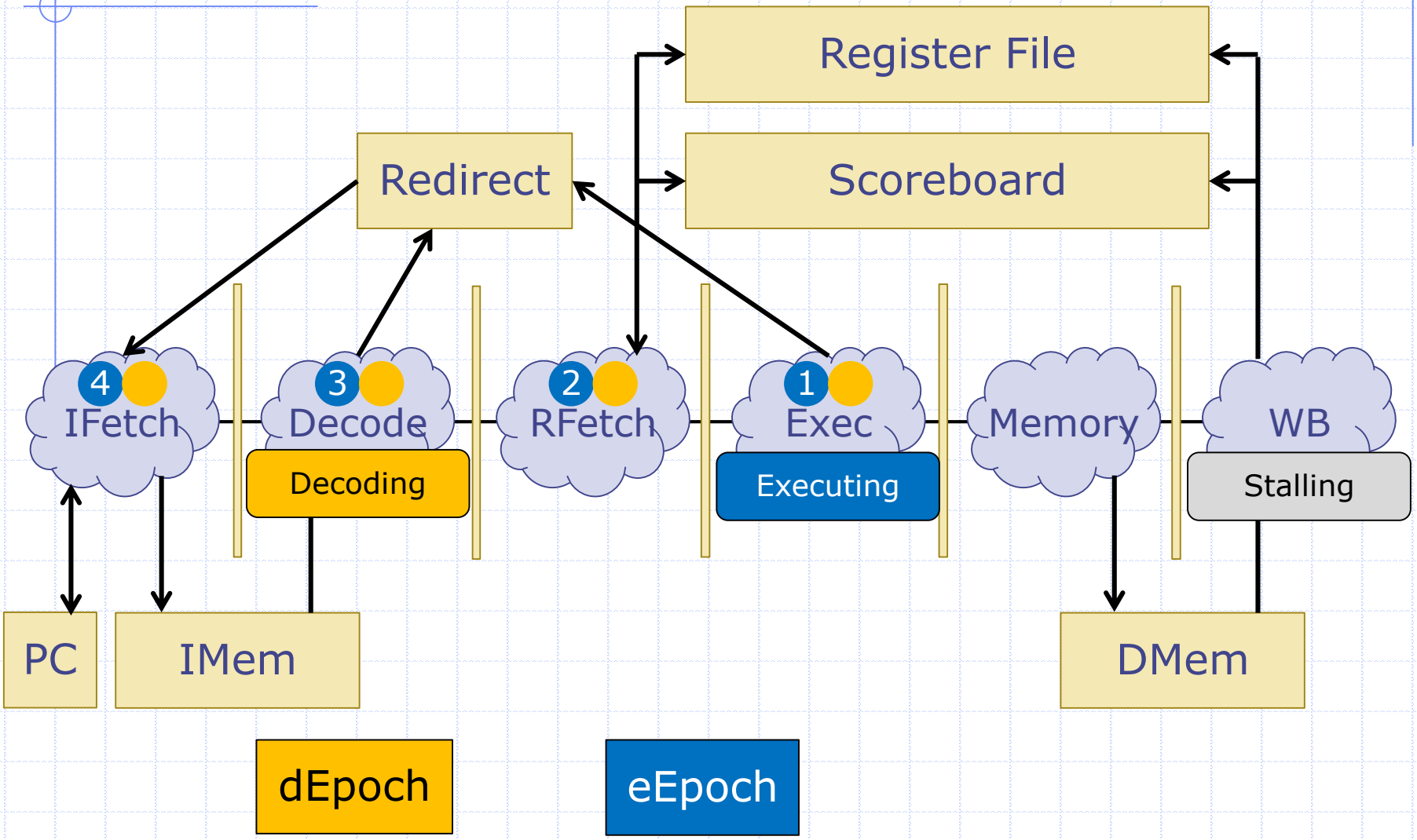
Global Epoch States



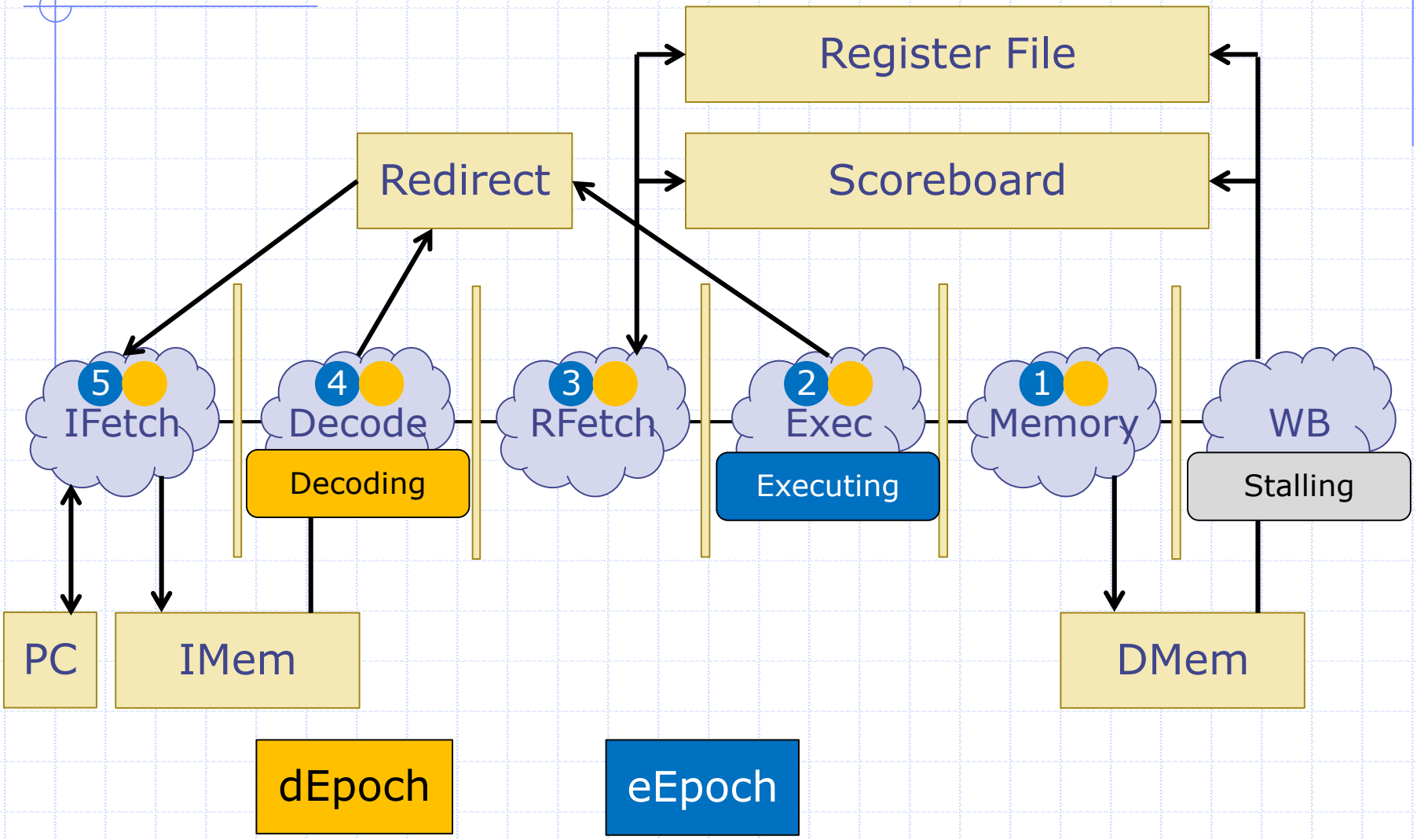
Global Epoch States



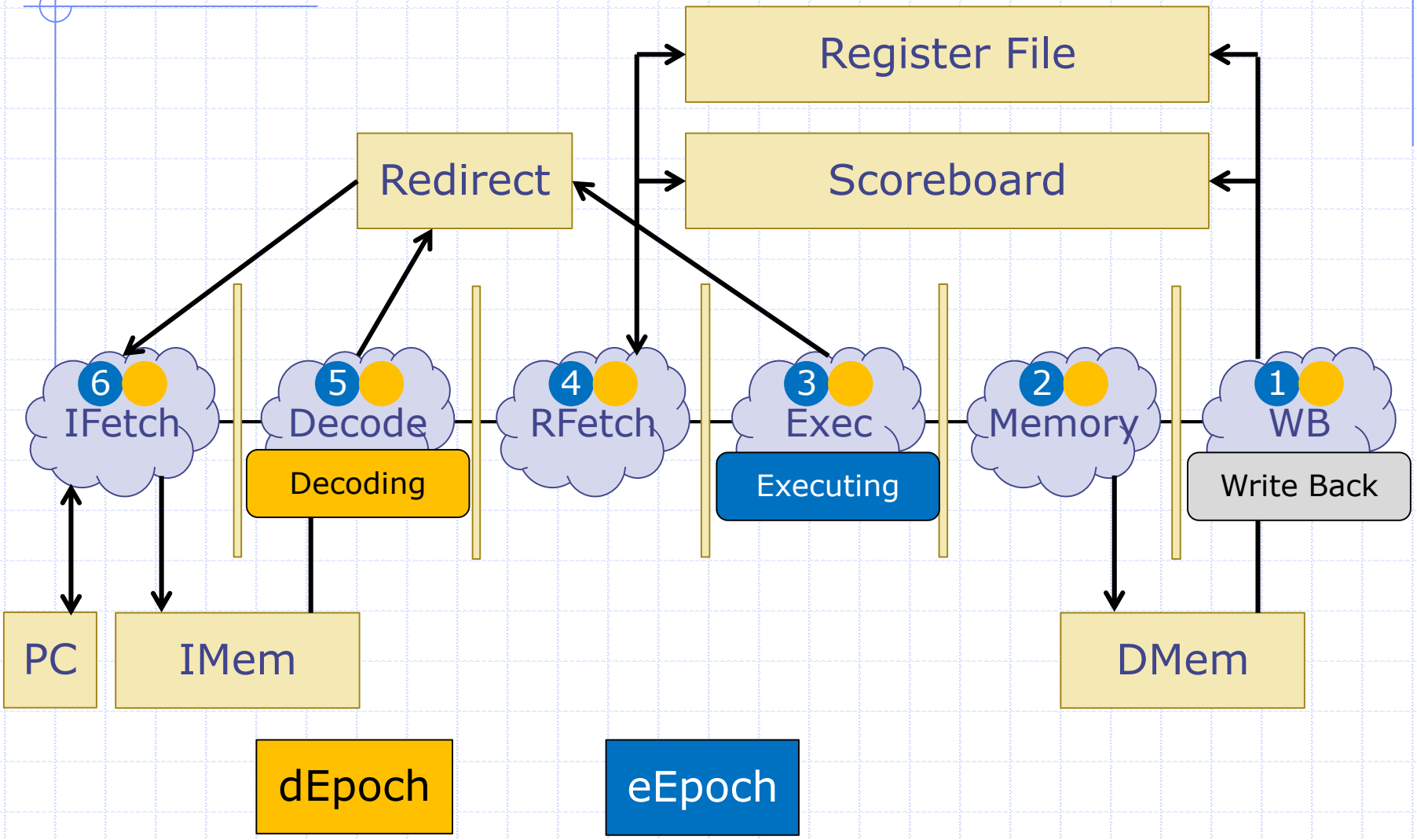
Global Epoch States



Global Epoch States



Global Epoch States

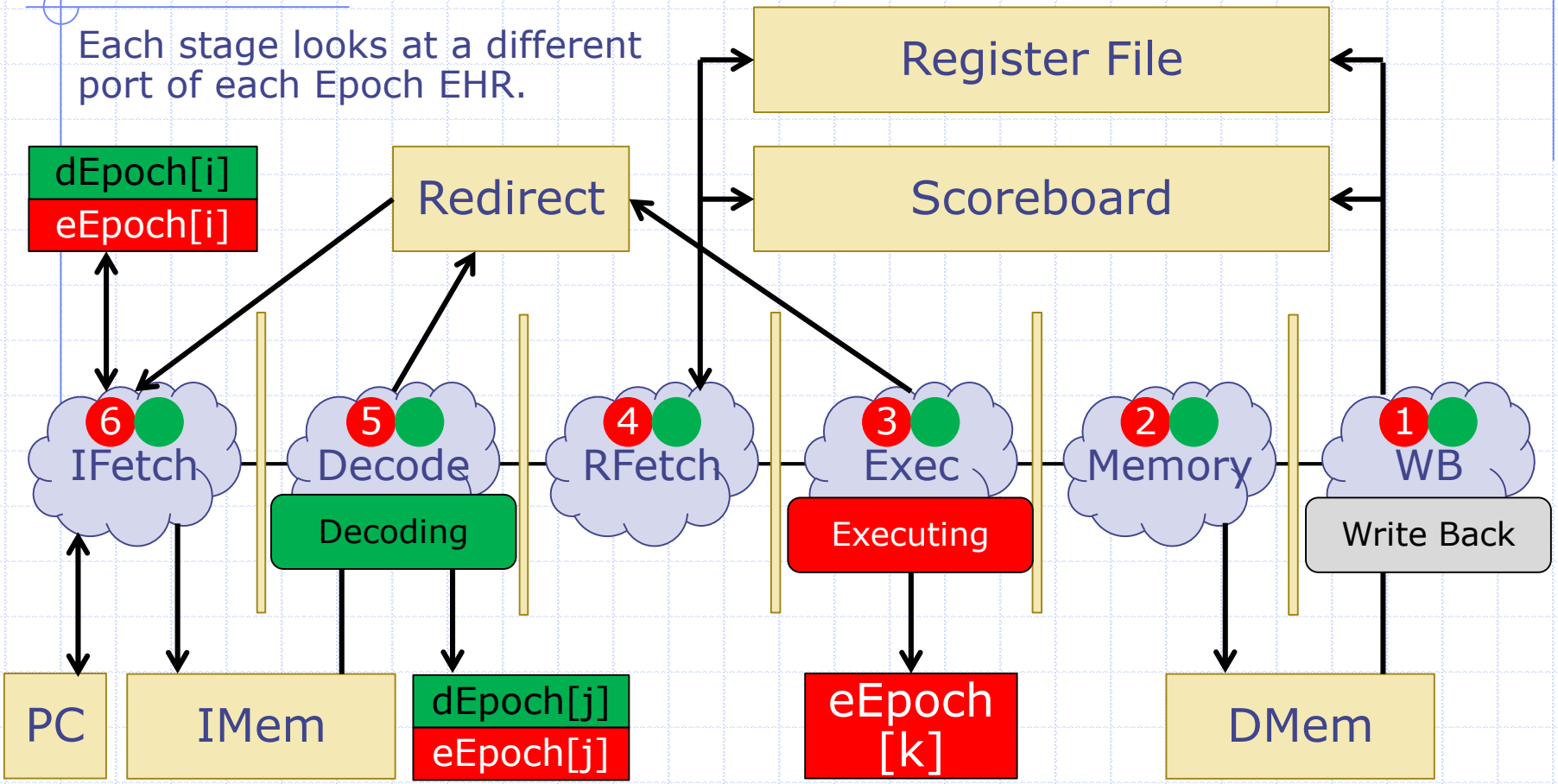


Implementing Global Epoch States

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

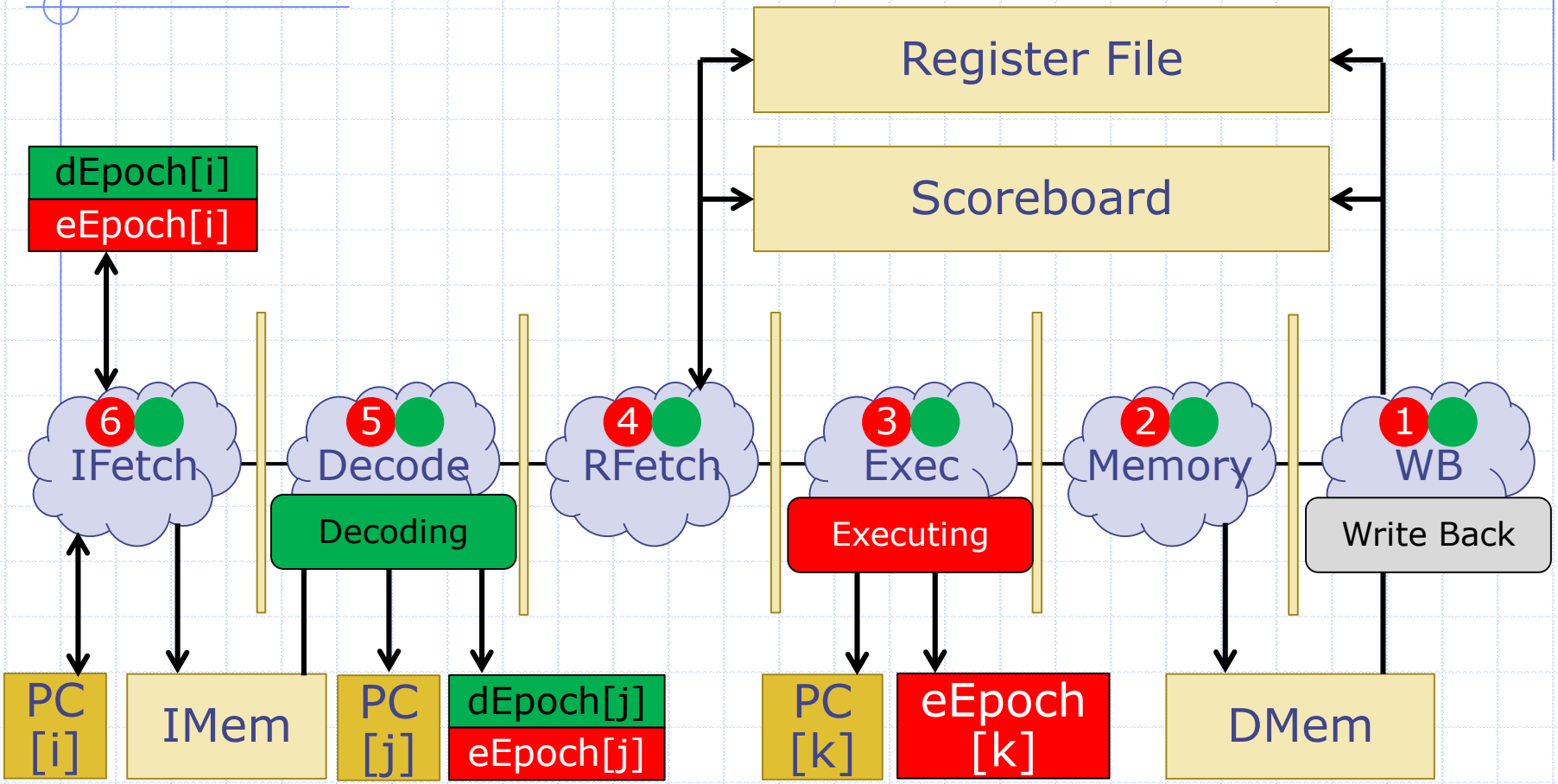
Implementing Global Epoch States with EHRs

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

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?