Constructive Computer Architecture

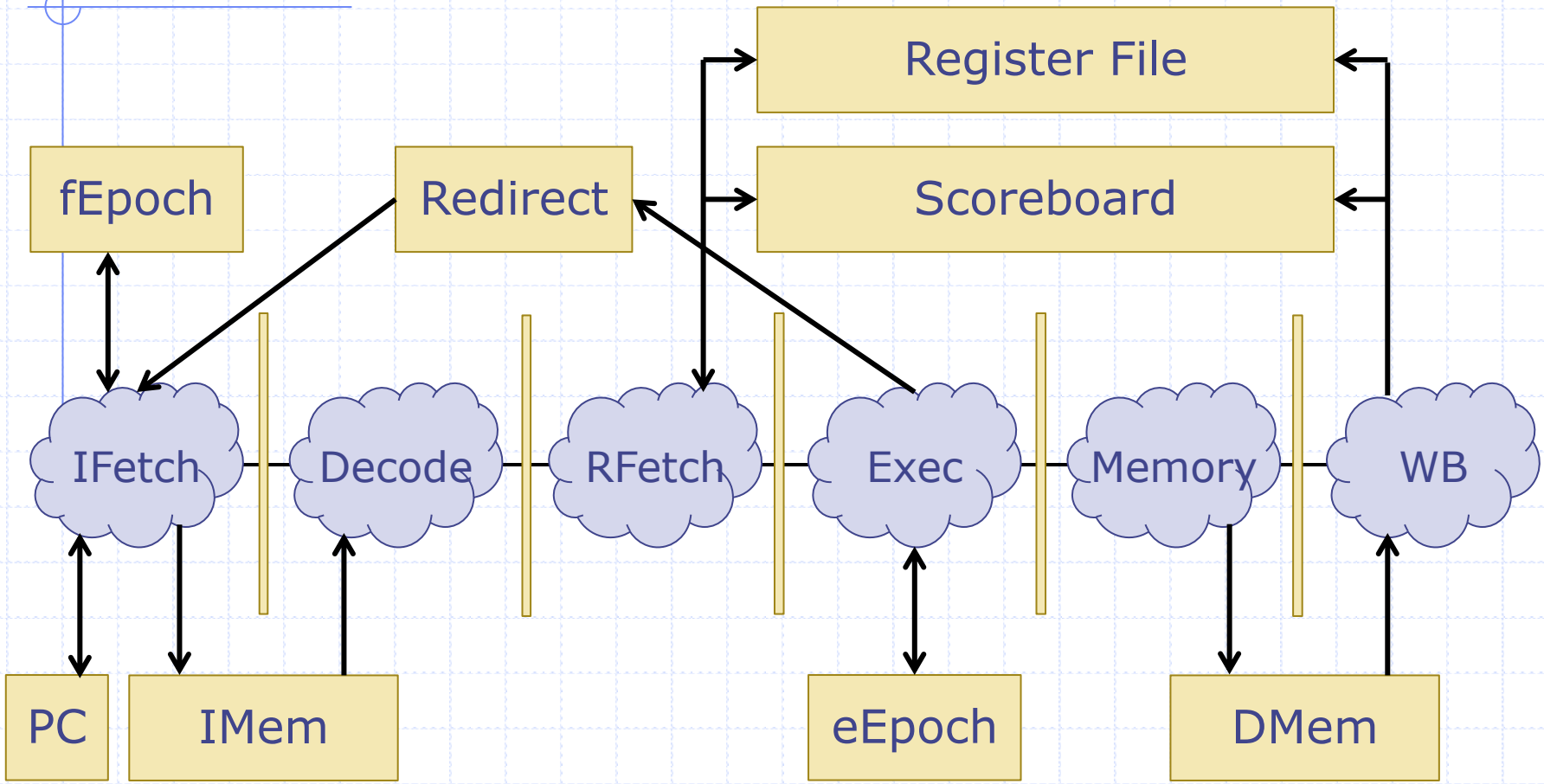
Tutorial 6: Five Details of SMIPS Implementations

Andy Wright
6.S195 TA

Introduction

- Lab 6 involves creating a 6 stage pipelined SMIPS processor from a 2 stage pipeline
  - This requires a lot of attention to architectural details of the processor, especially at the points of interaction between the stages.
- This tutorial will cover some details of the SMIPS architecture that will be useful for the current and future labs

6 stage SMIPS pipeline



5 Details

- Processor State
- Poisoning Instructions
- ASAP Prediction Correction
- Pipeline Feedback
- Removing Pipeline Stages

5 Details

- **Processor State**
- Poisoning Instructions
- ASAP Prediction Correction
- Pipeline Feedback
- Removing Pipeline Stages

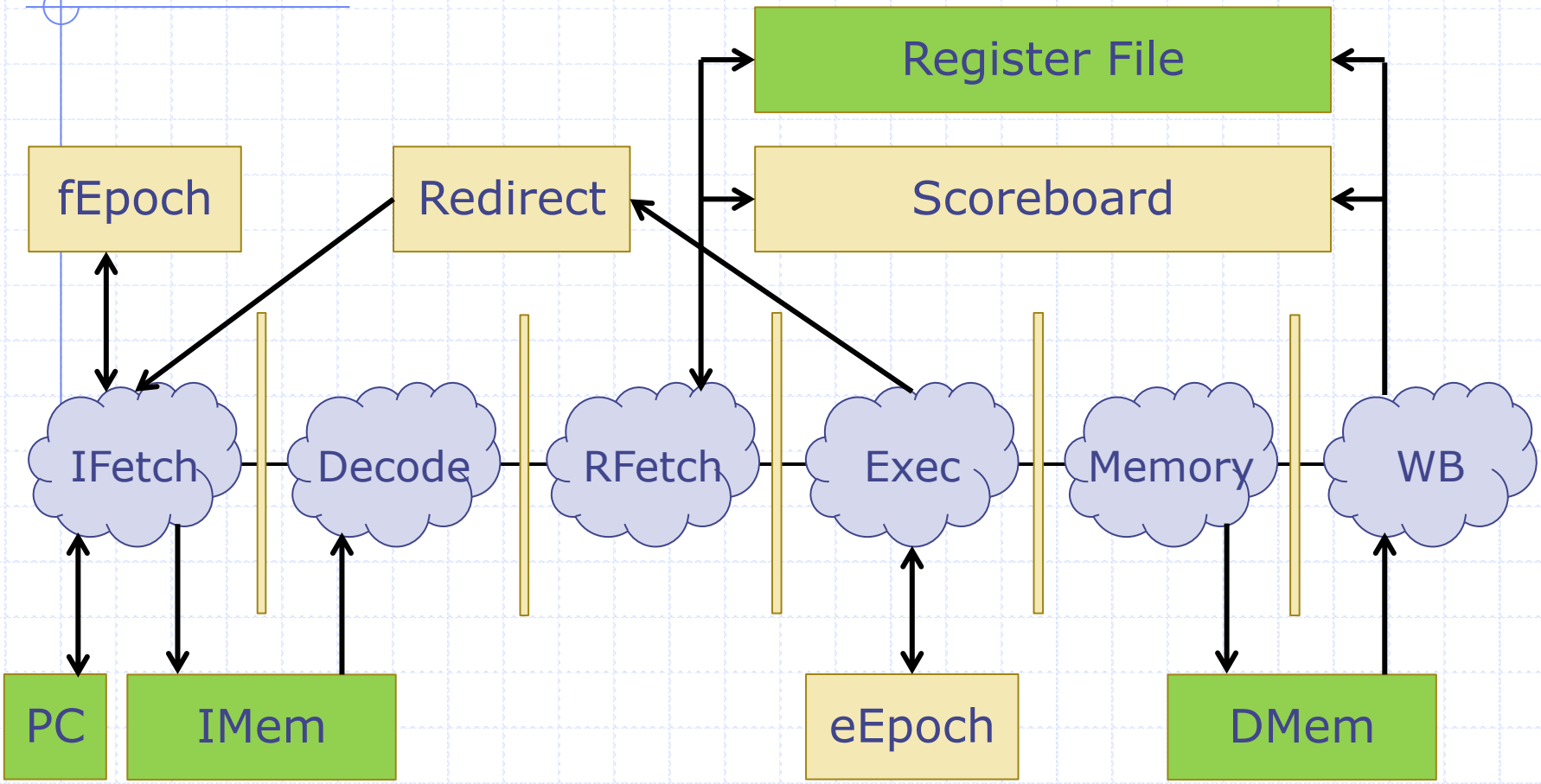
Processor State

- The processor state is (PC, RFile, Mem)
- Instructions can be seen as functions of a processor state that return the new processor state
  - addi(PC, RFile, Mem) = (PC+4, RFile', Mem)
    - RFile' is RFile updated with the result of the addi instruction
- The instruction memory can be seen as a function of PC that returns Instructions
  - Imem: (PC) ->
    ((PC, Rfile, Mem) -> (PC, RFile, Mem))

Processor State

- If your SMIPS processor from lab is not working:
  - Was an instruction executed on the wrong processor state?
    - RAW hazards
    - Not using the right PC in the execute stage
  - Was the wrong instruction executed?
    - A wrong path instruction from branch misprediction updated the processor state

Processor State



Green blocks make up the processor state. All other state elements make sure the right processor state is used to compute instructions, and to make sure the right instructions are executed.

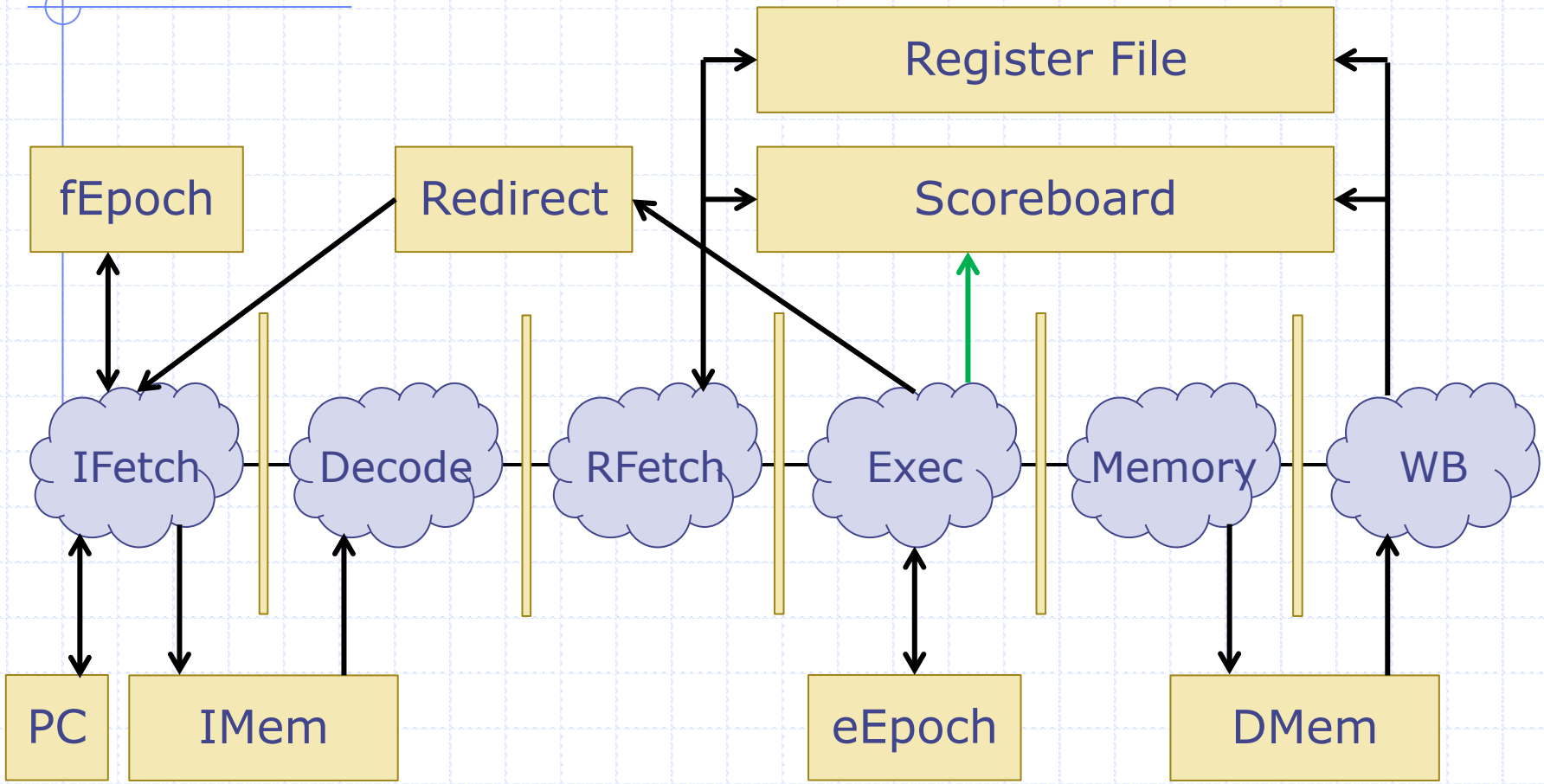
5 Details

- Processor State
- **Poisoning Instructions**
- ASAP Prediction Correction
- Pipeline Feedback
- Removing Pipeline Stages

Poisoning Instructions

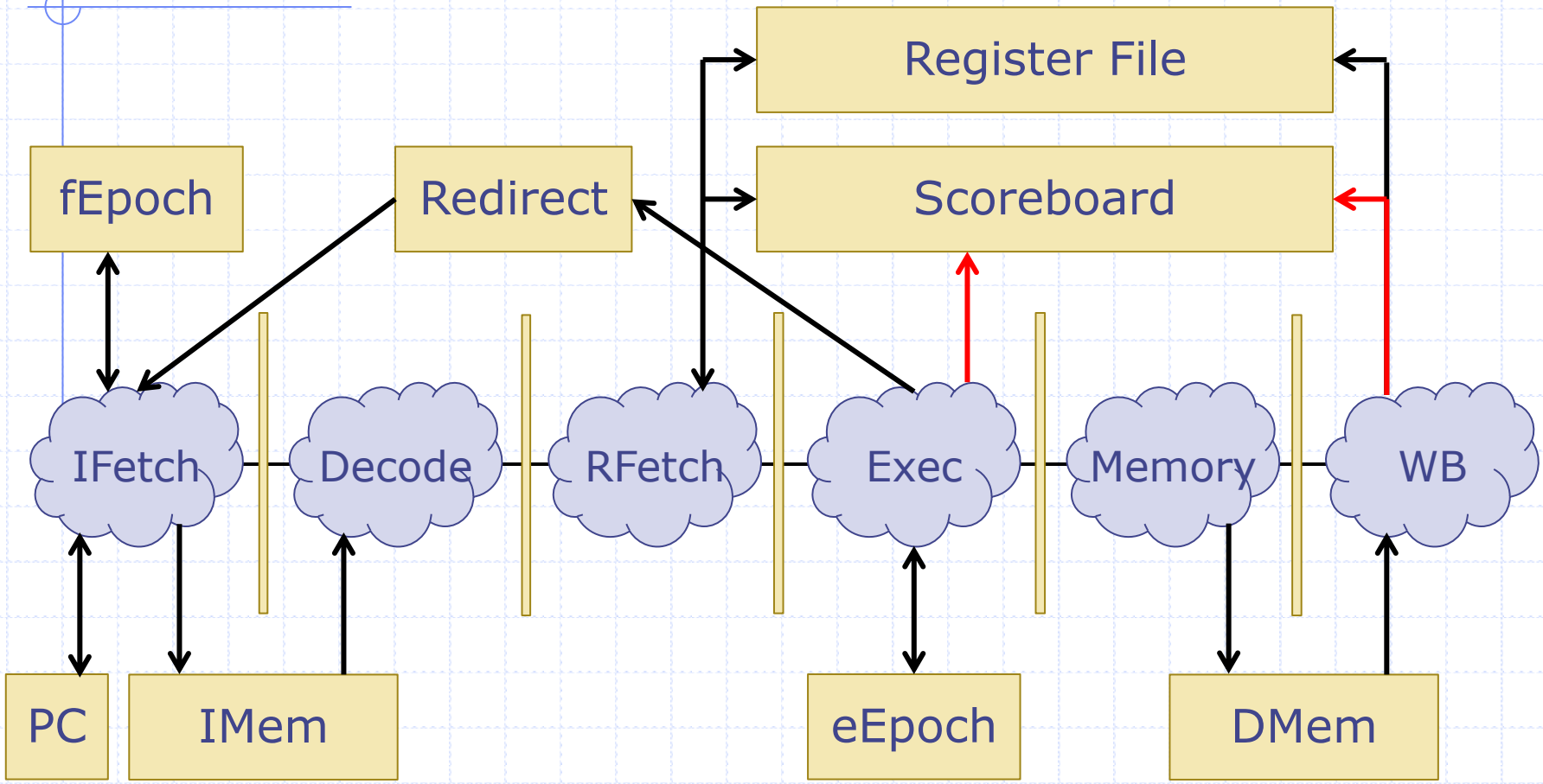
- Why poison? It's a way to mark that an instruction should be killed at a later stage.
  - This mark could be as simple as using an invalid value in a maybe data type
- Instructions are poisoned when epochs don't match
- Why not kill in place?

Kill-In-Place Pipeline



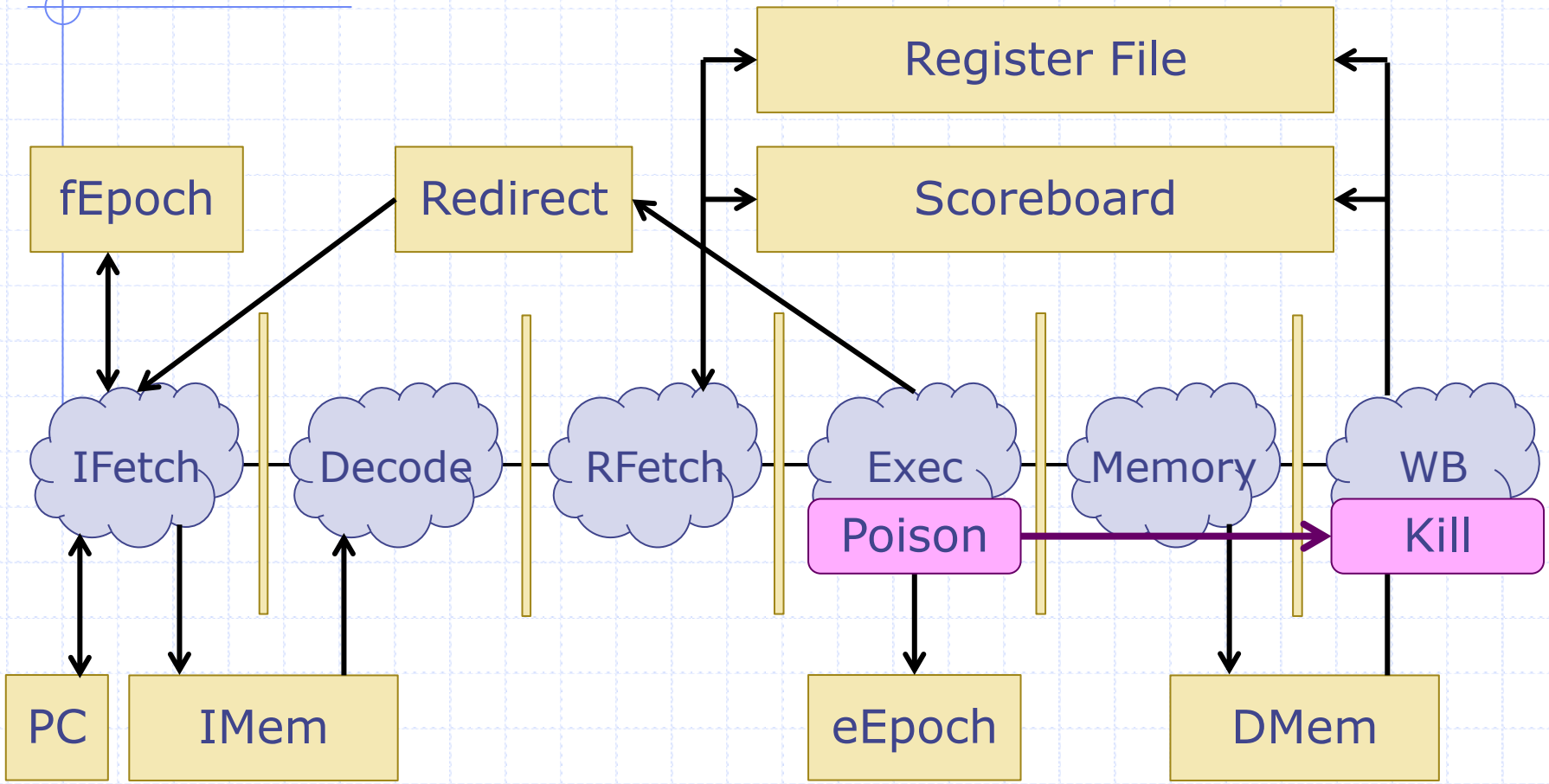
Scoreboard entries need to be removed when instructions are killed.

Kill-In-Place Pipeline



Both Exec and WB try to call sb.remove(). This will cause Exec to conflict with WB. Also, the scoreboard implementation doesn't allow out-of-order removal.

Poisoning Pipeline



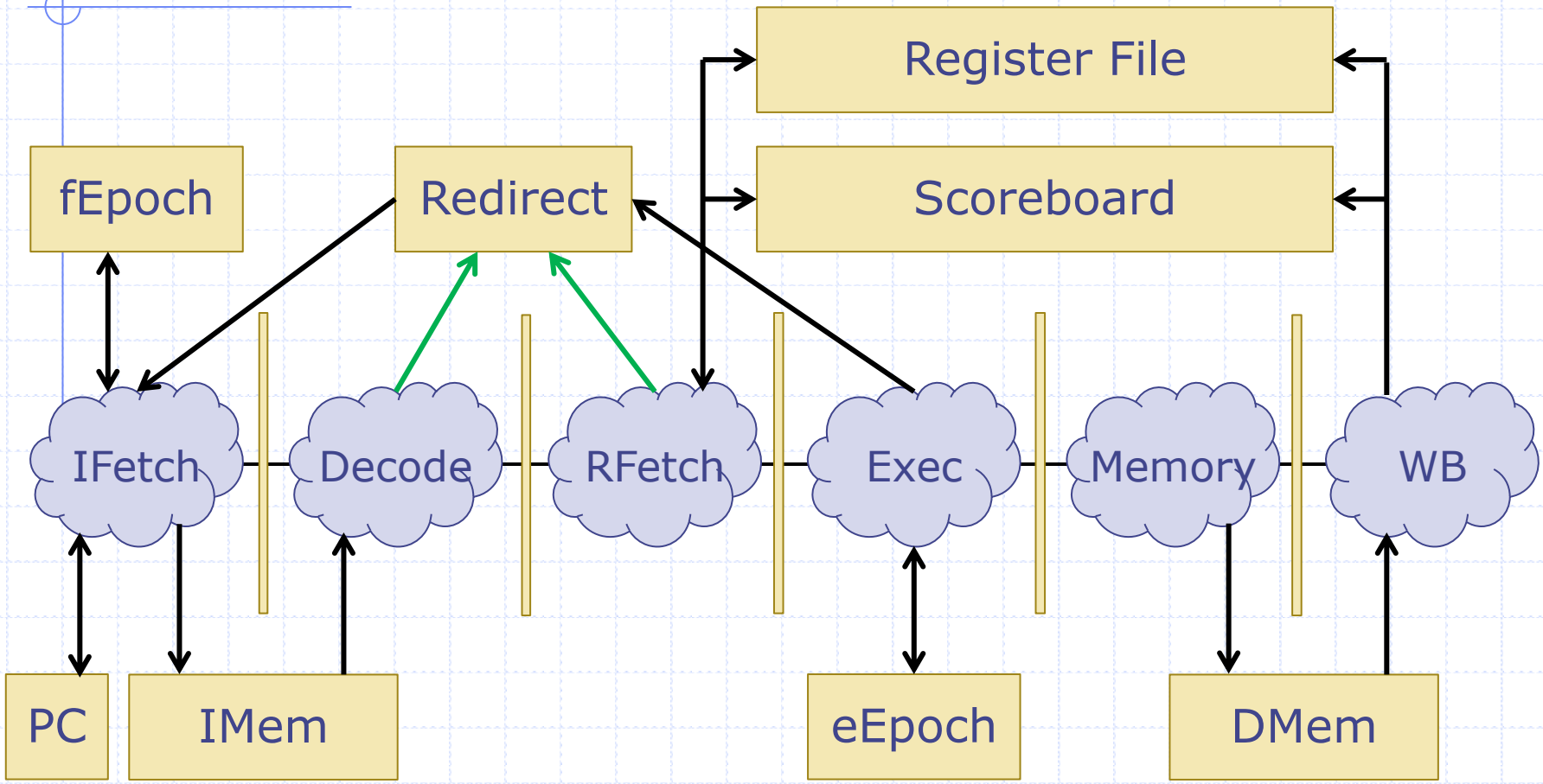
5 Details

- Processor State
- Poisoning Instructions
- **ASAP Prediction Correction**
- Pipeline Feedback
- Removing Pipeline Stages

ASAP Prediction Correction

- Different instructions that affect the program flow can be resolved at different times
  - Absolute Jumps – Decode
  - Register Jumps – RFetch
  - Branches – Exec
- You can save cycles on each misprediction by correcting the PC once you have computed what the next PC should have been.

ASAP Prediction Correction

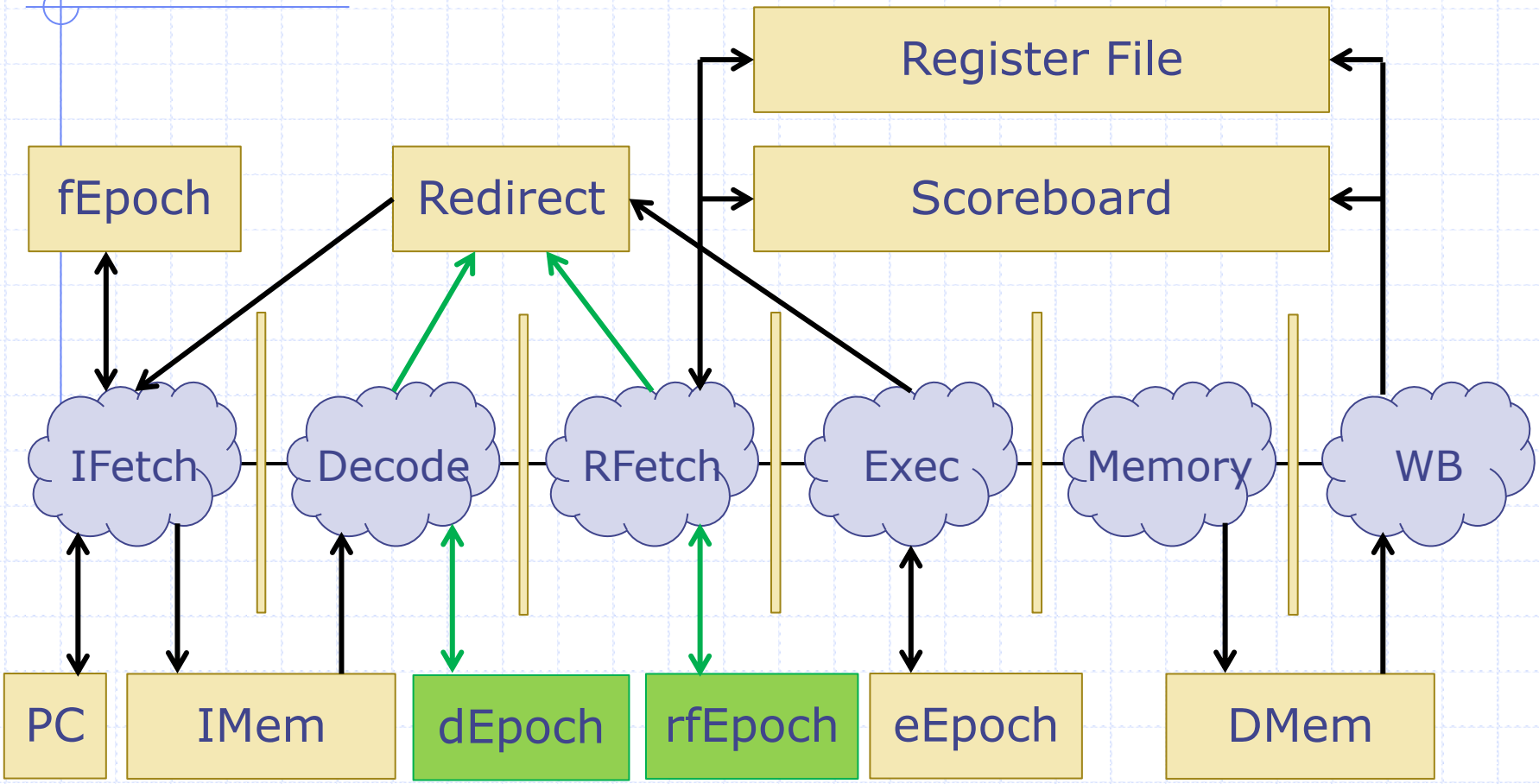


This is the general idea, we want Decode, RFetch, and Exec to be able to correct the PC

ASAP Prediction Correction

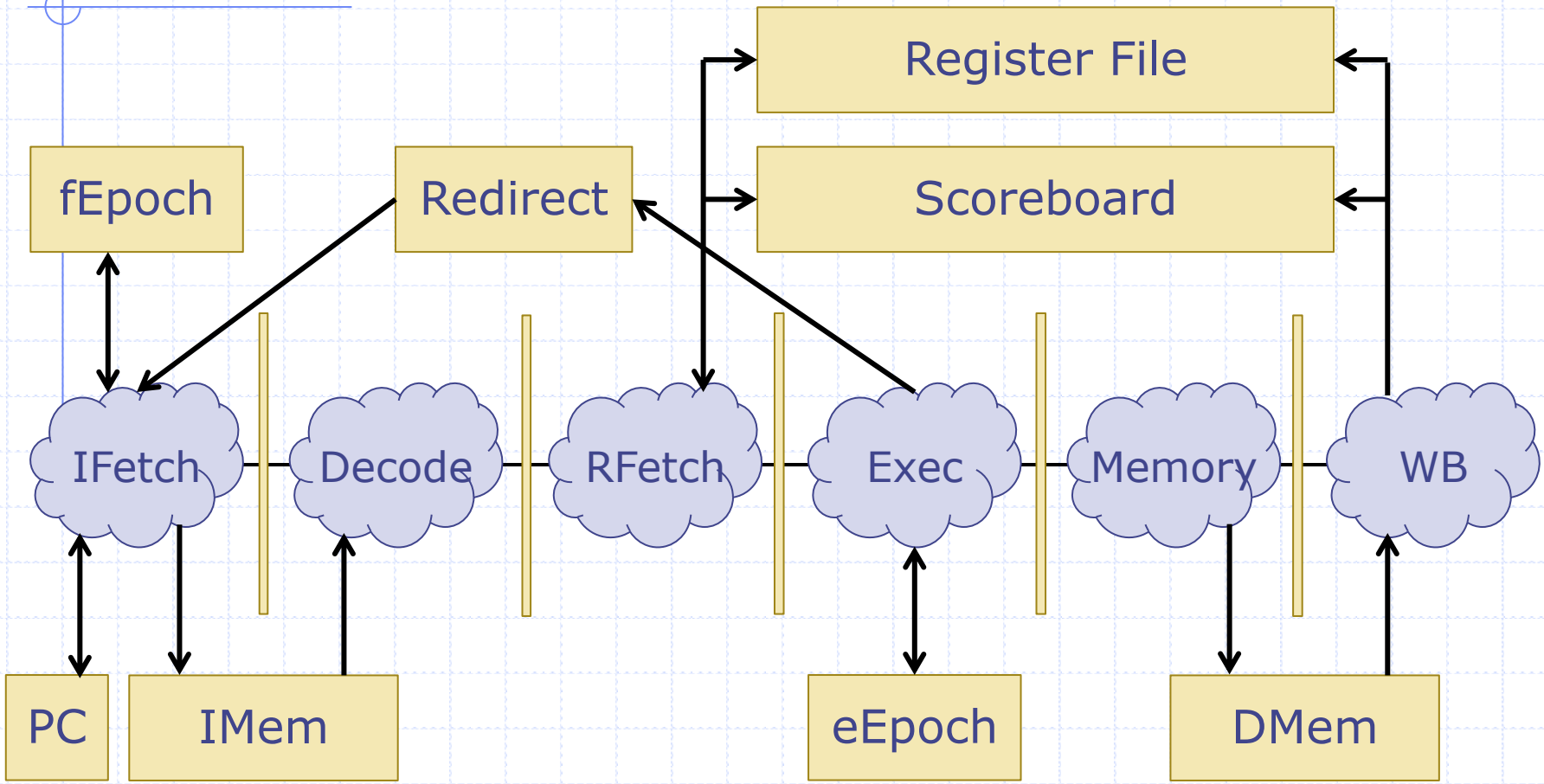
- How is this actually done?
- How do you keep from allowing wrong path instructions to update the PC?
- How do you keep track of everything?
- How?
  - More Epochs!

ASAP Prediction Correction



Decode, RFetch, and Exec each have their own epoch.

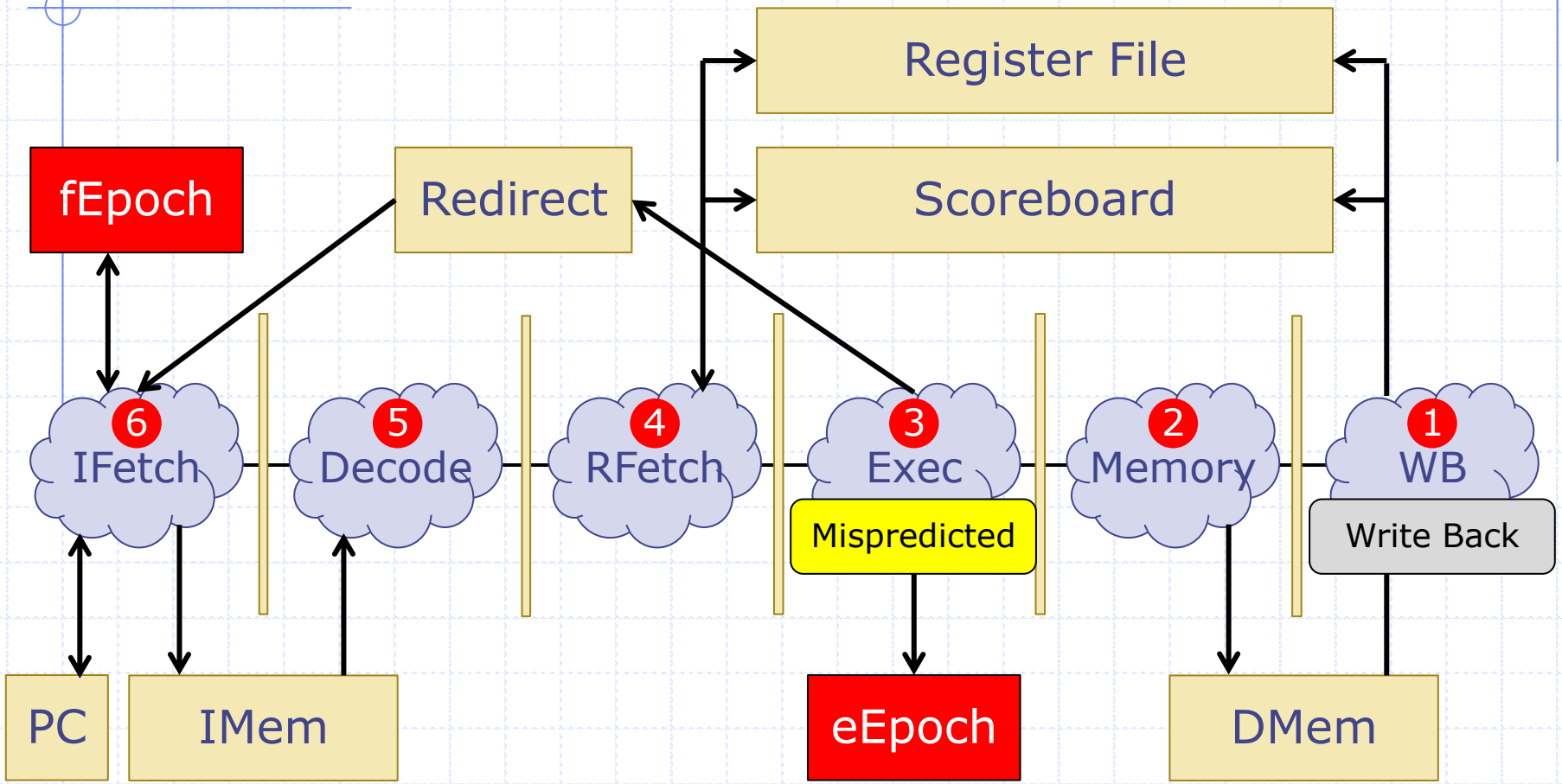
ASAP Prediction Correction



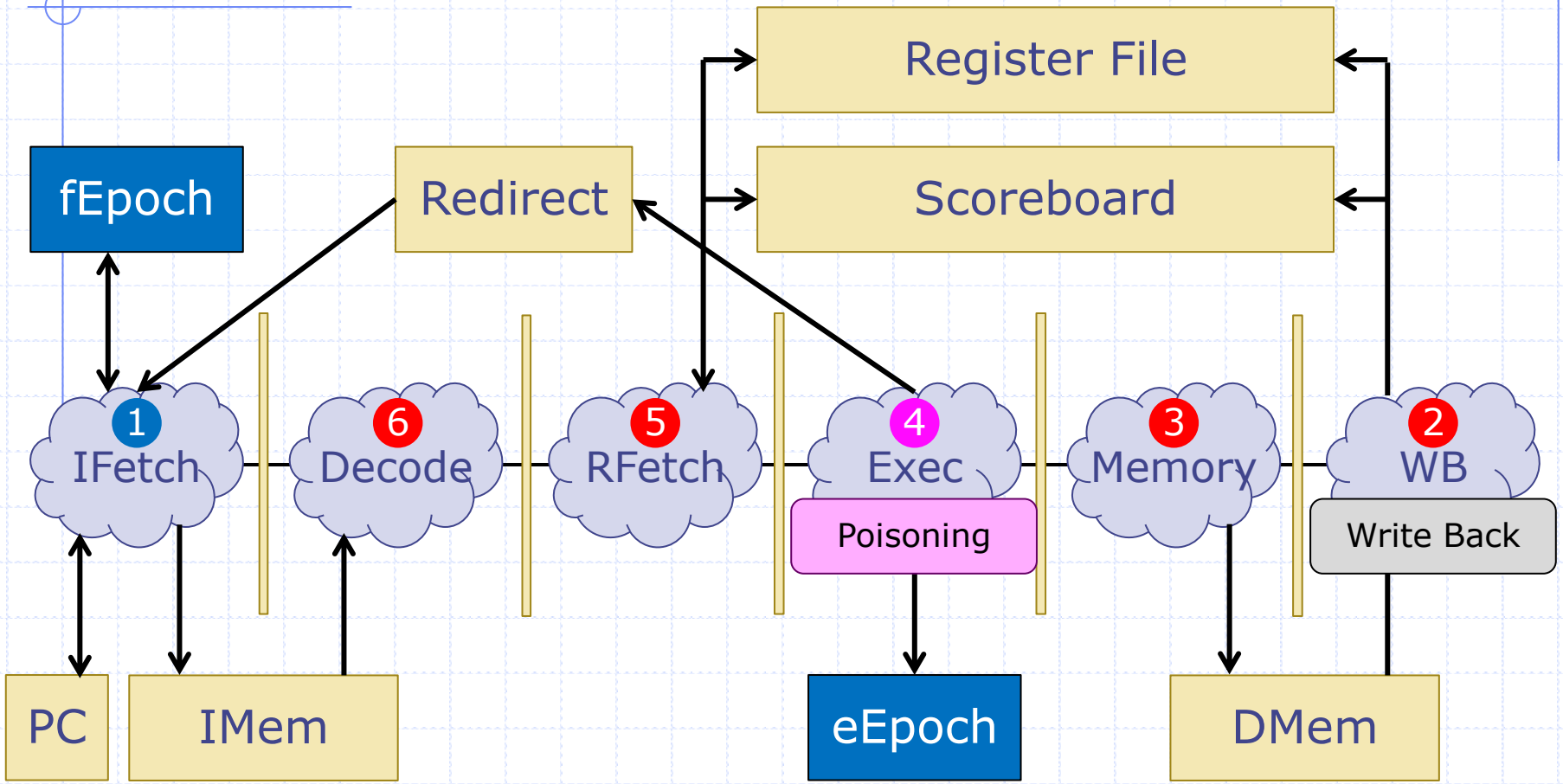
Each epoch acts just like eEpoch does

Correcting PC in Execute

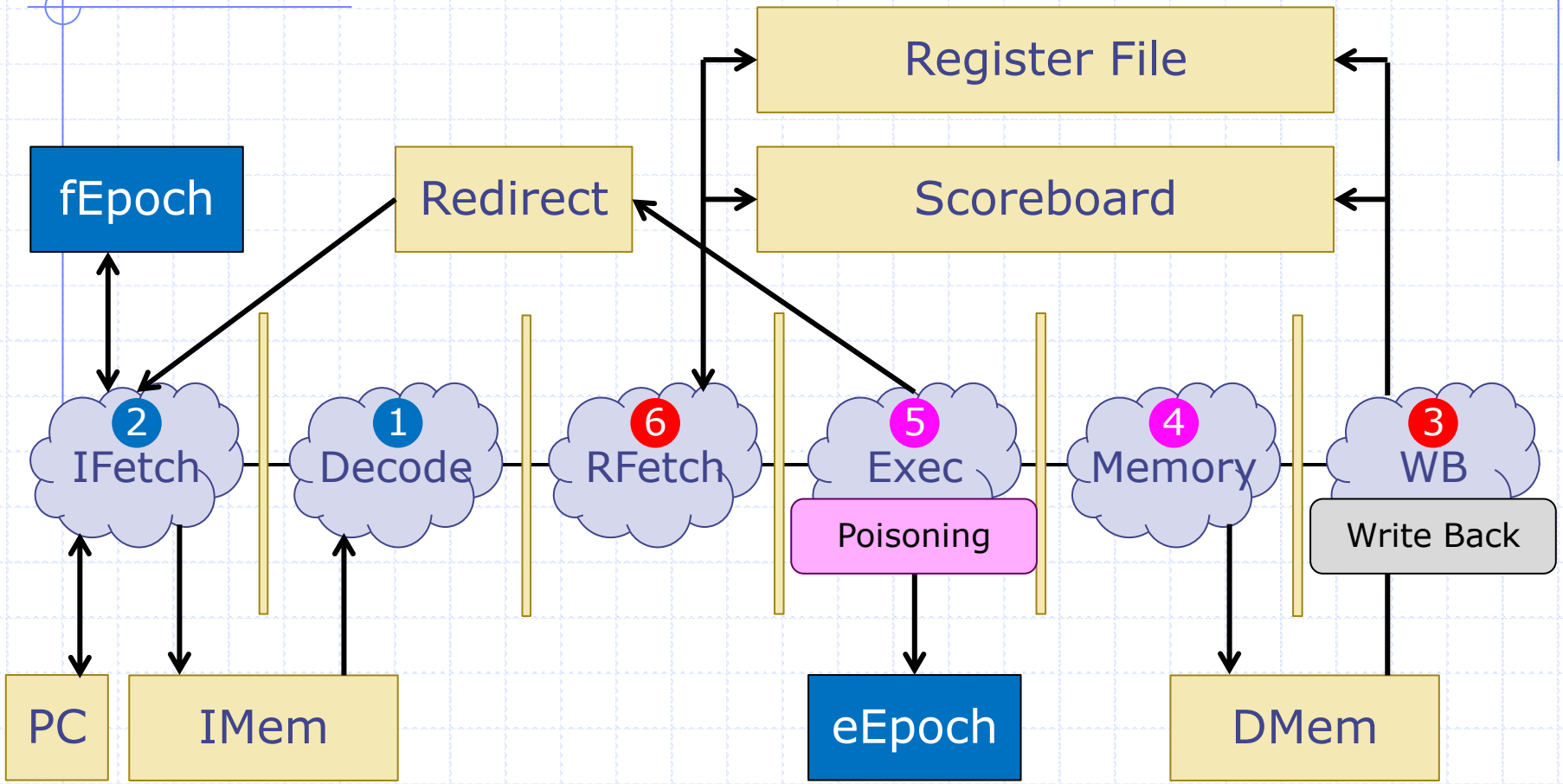
Correcting PC in Execute



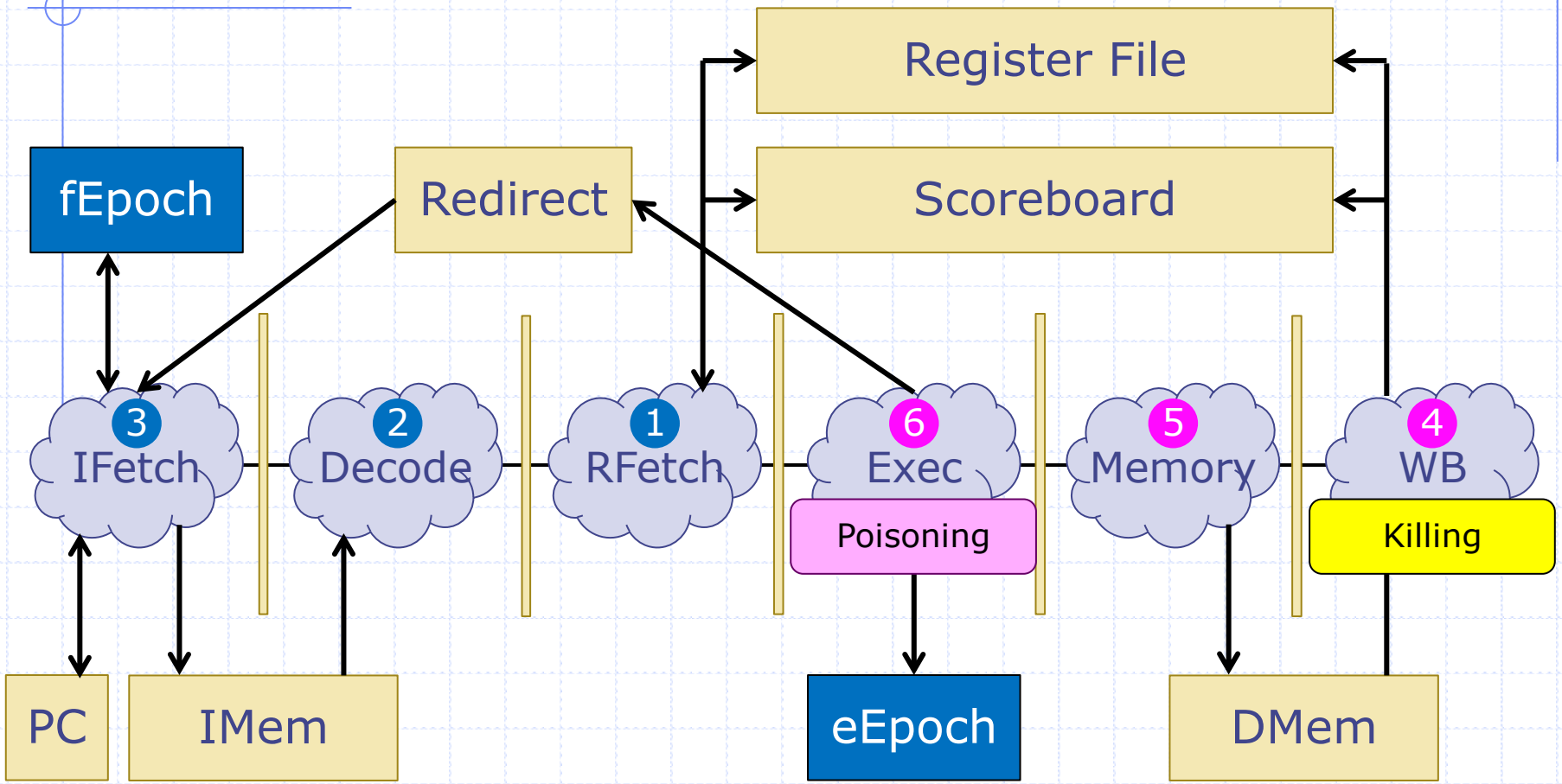
Correcting PC in Execute



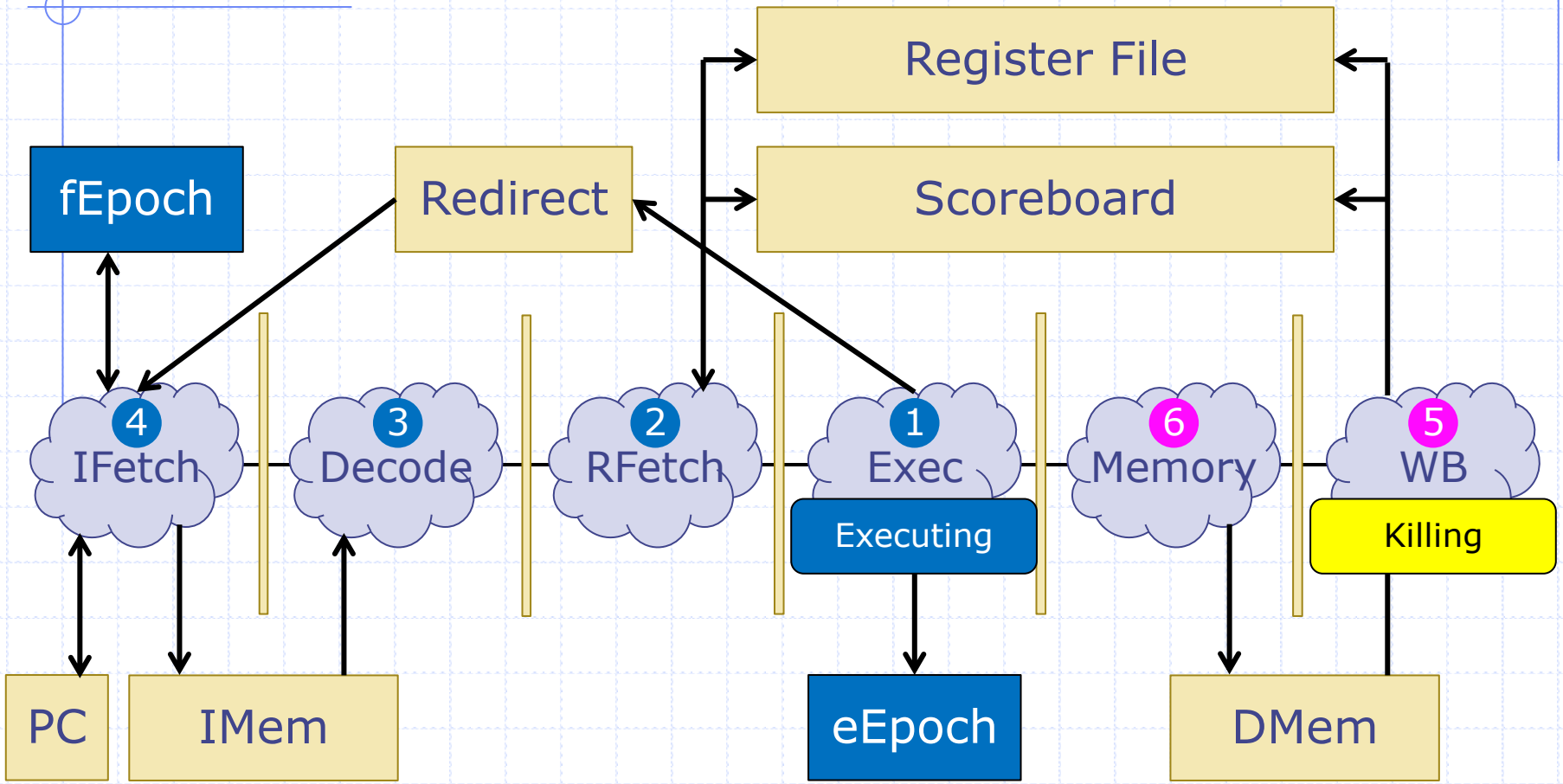
Correcting PC in Execute



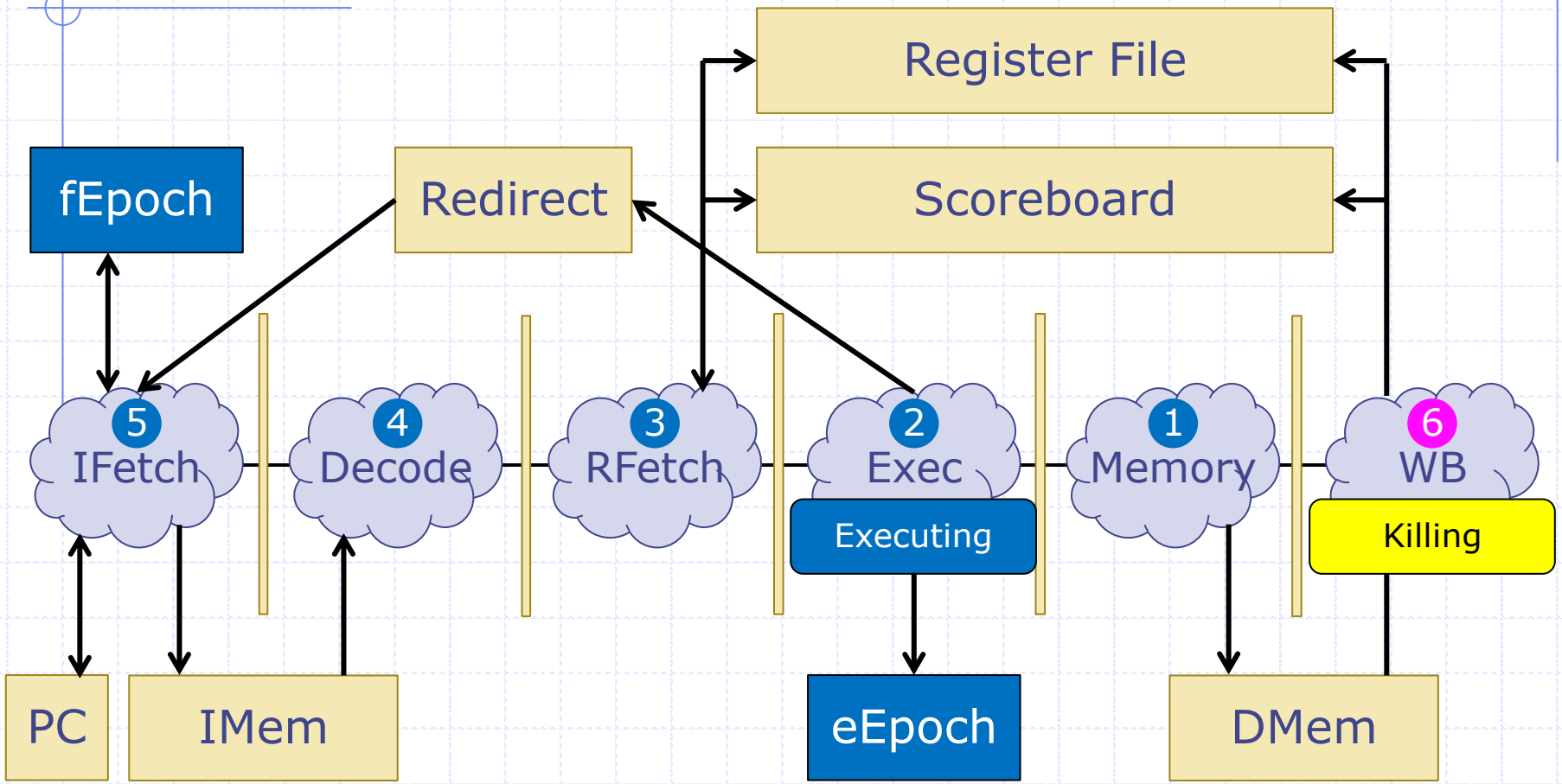
Correcting PC in Execute



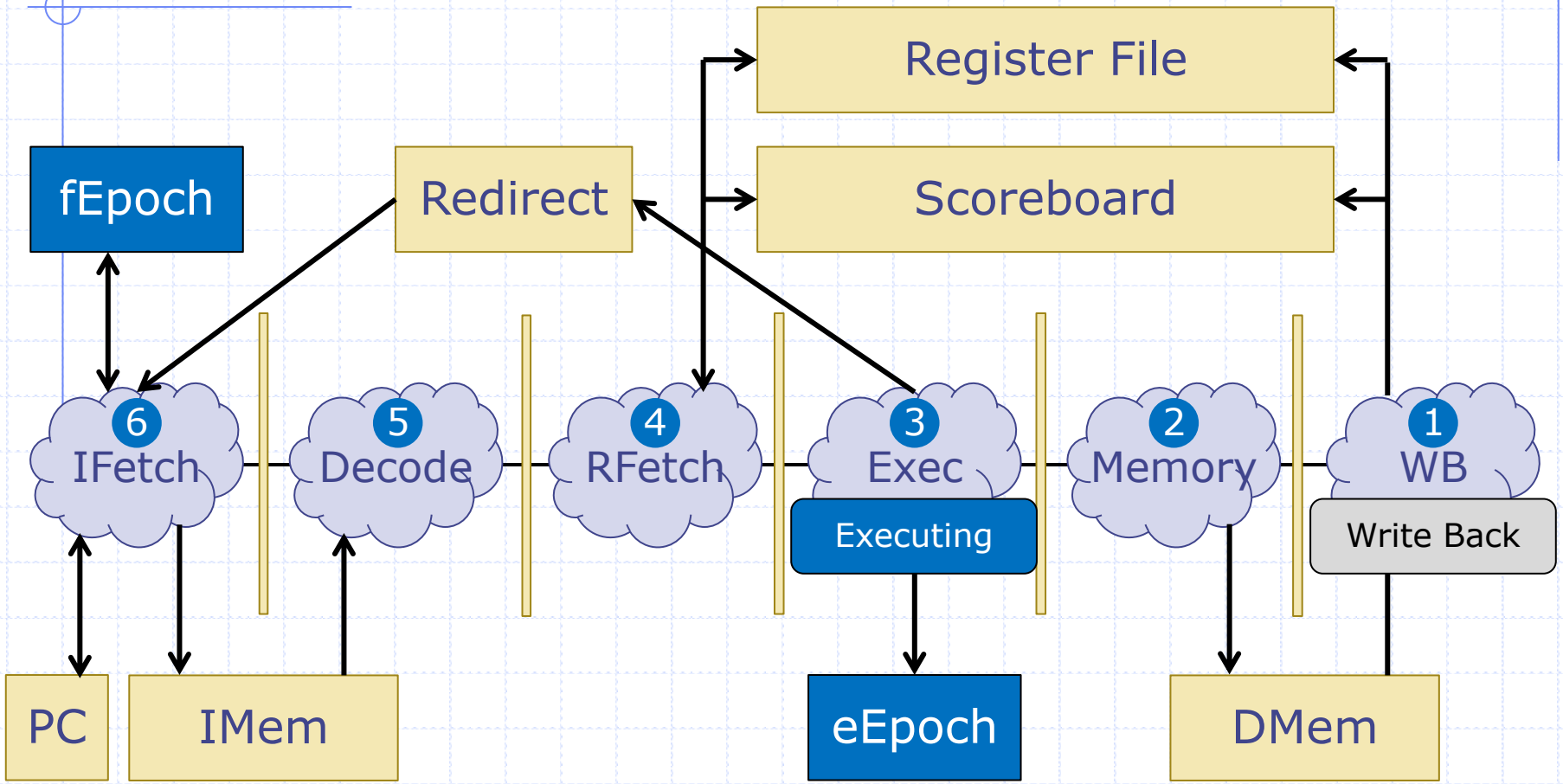
Correcting PC in Execute



Correcting PC in Execute

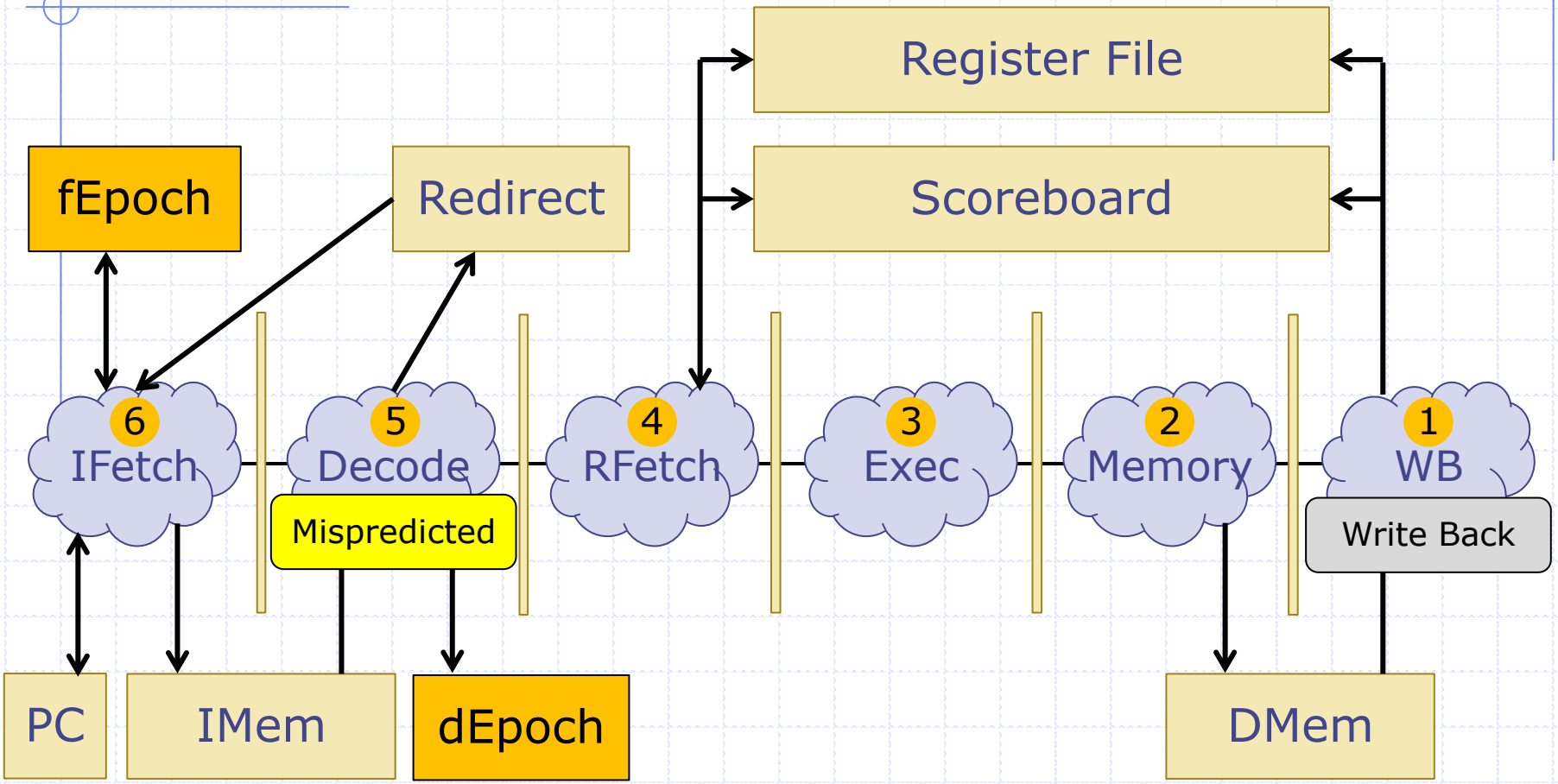


Correcting PC in Execute

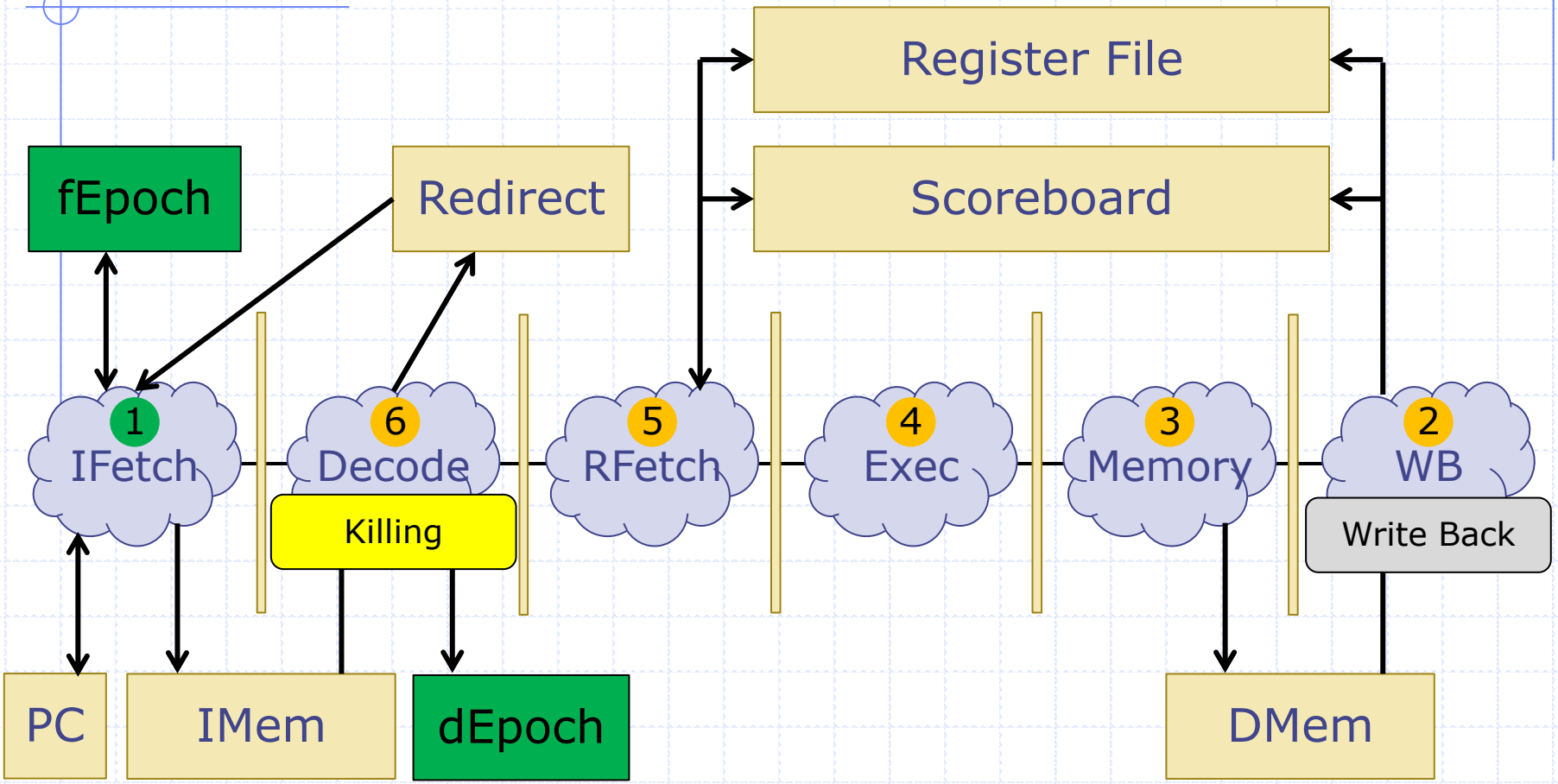


Correcting PC in Decode

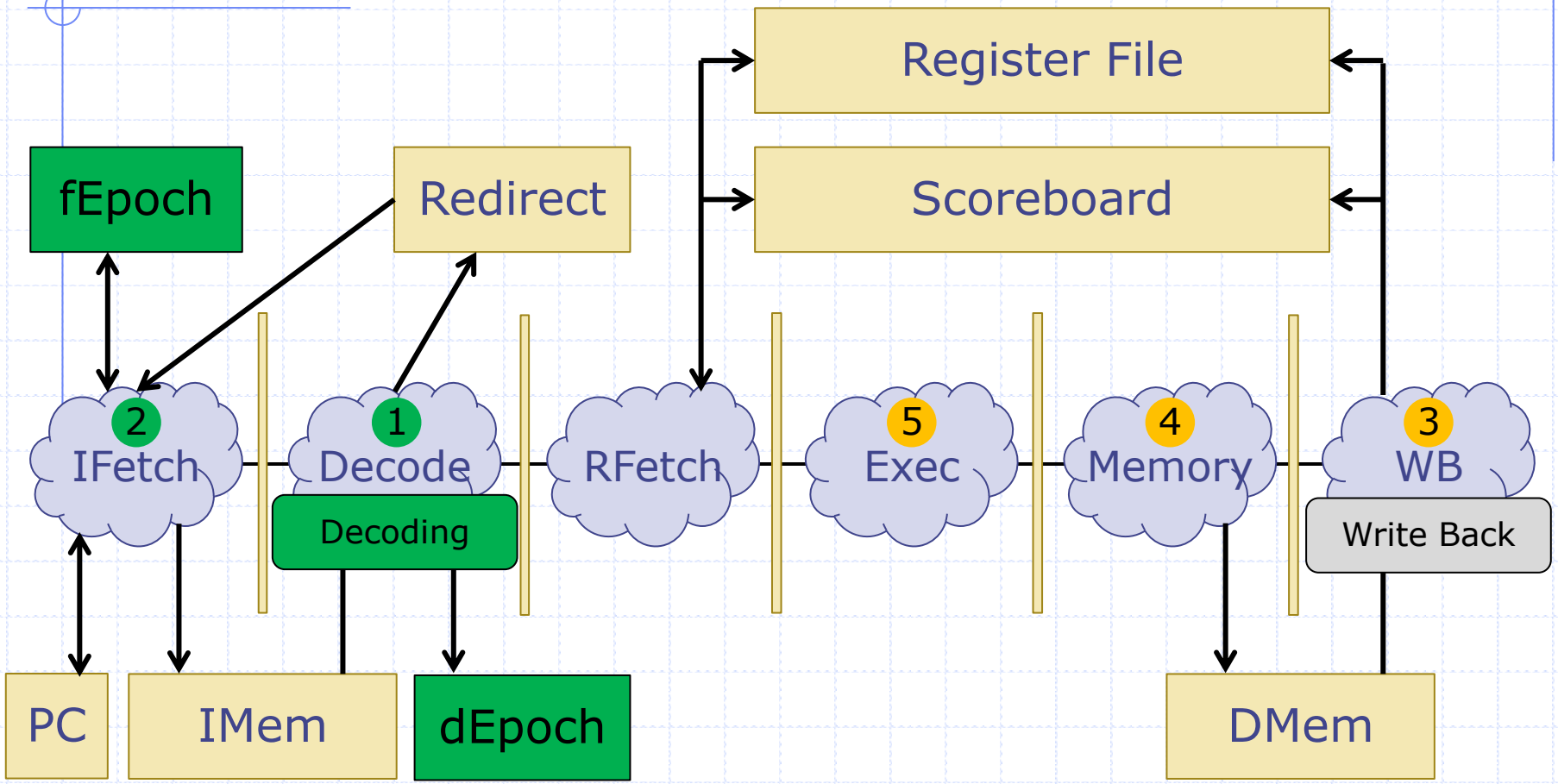
Correcting PC in Decode



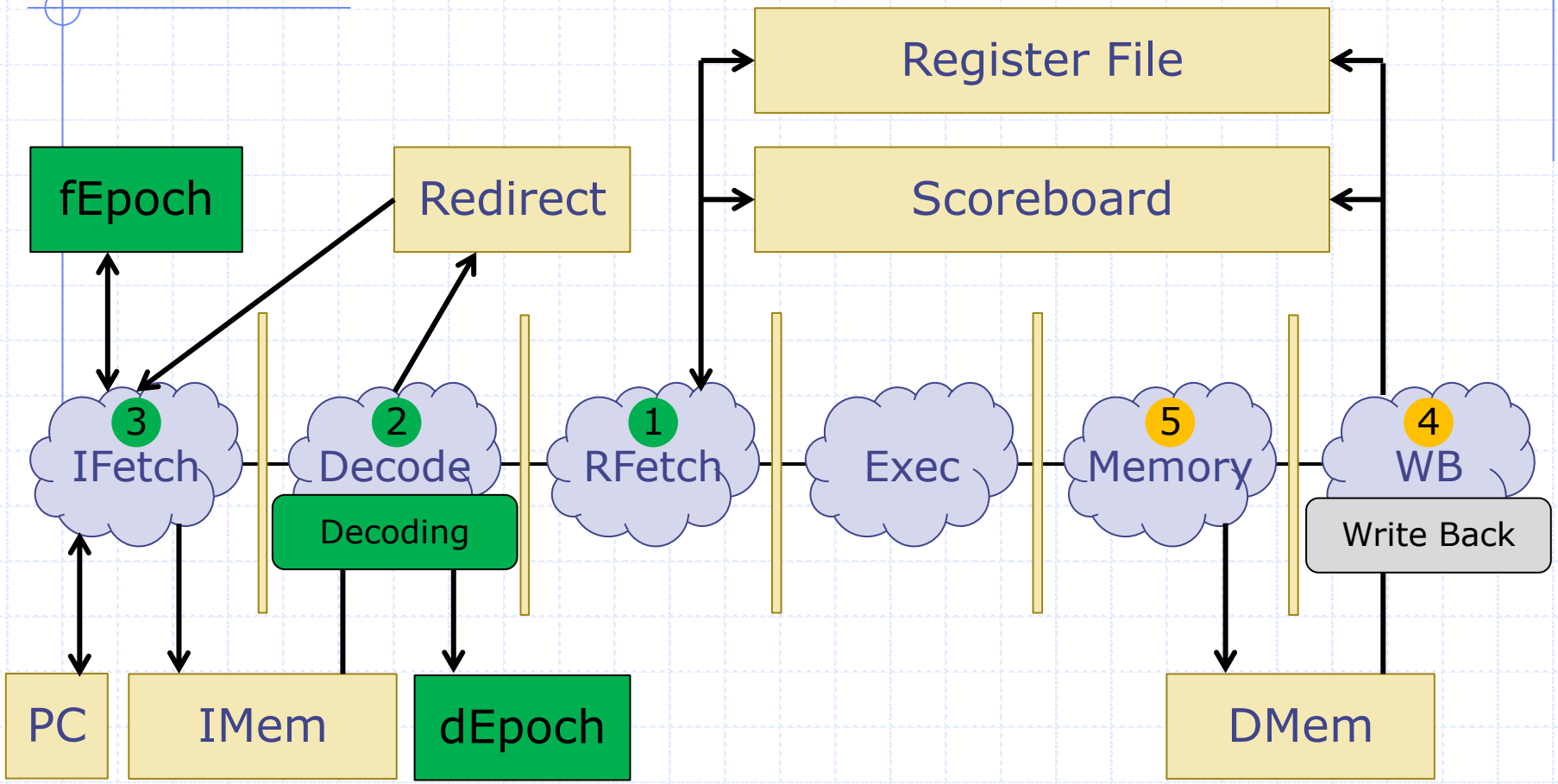
Correcting PC in Decode



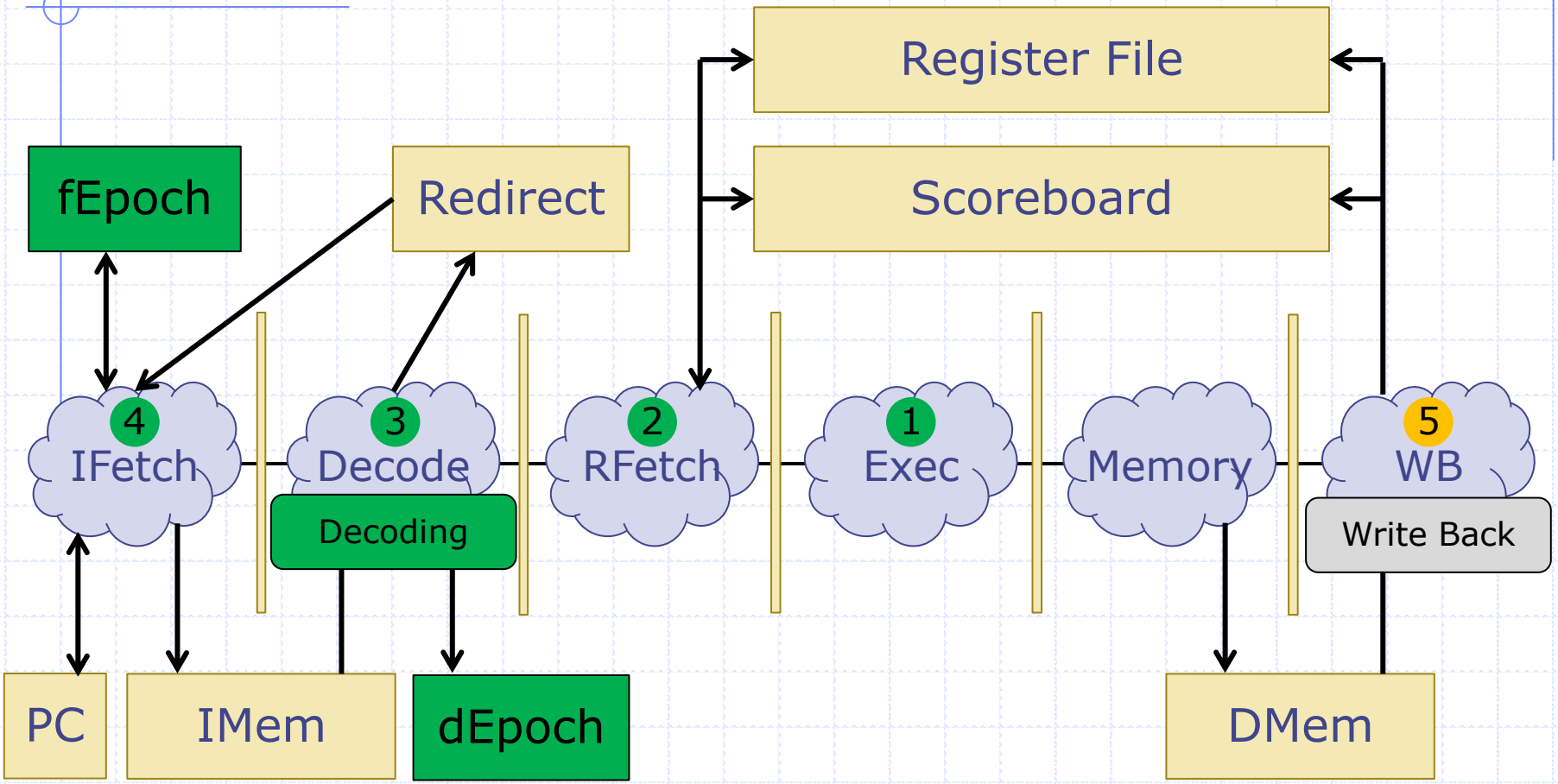
Correcting PC in Decode



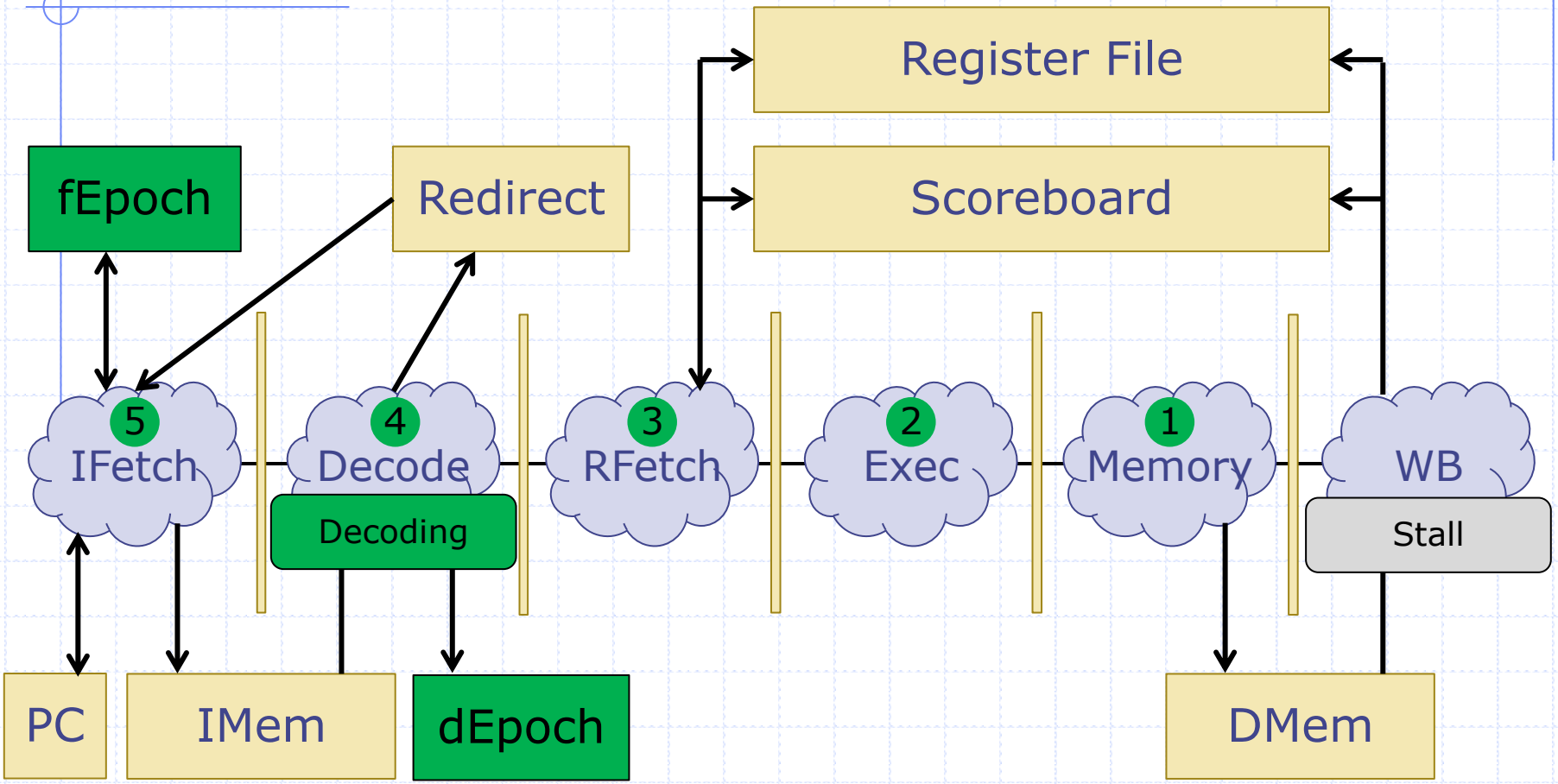
Correcting PC in Decode



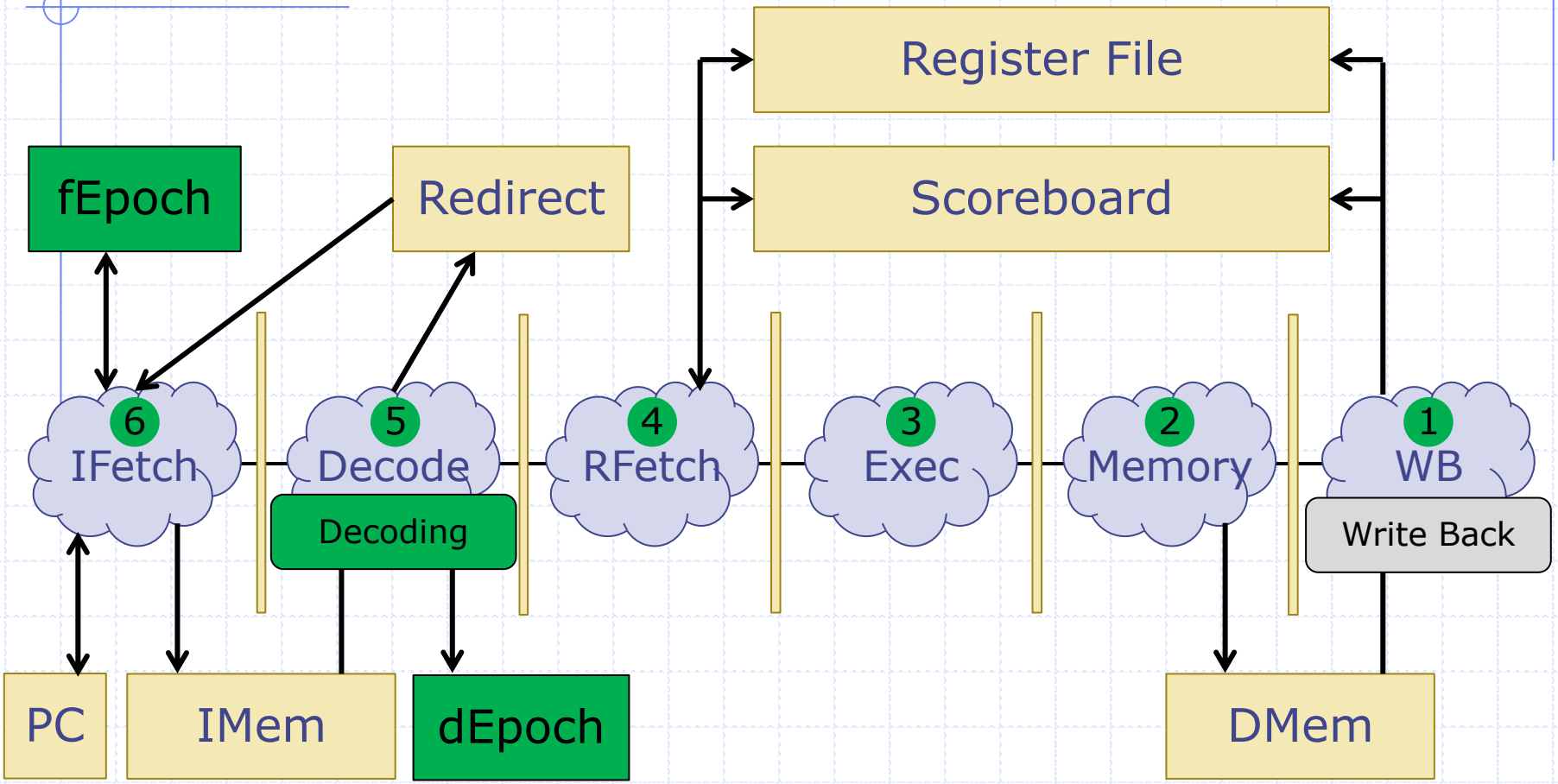
Correcting PC in Decode



Correcting PC in Decode

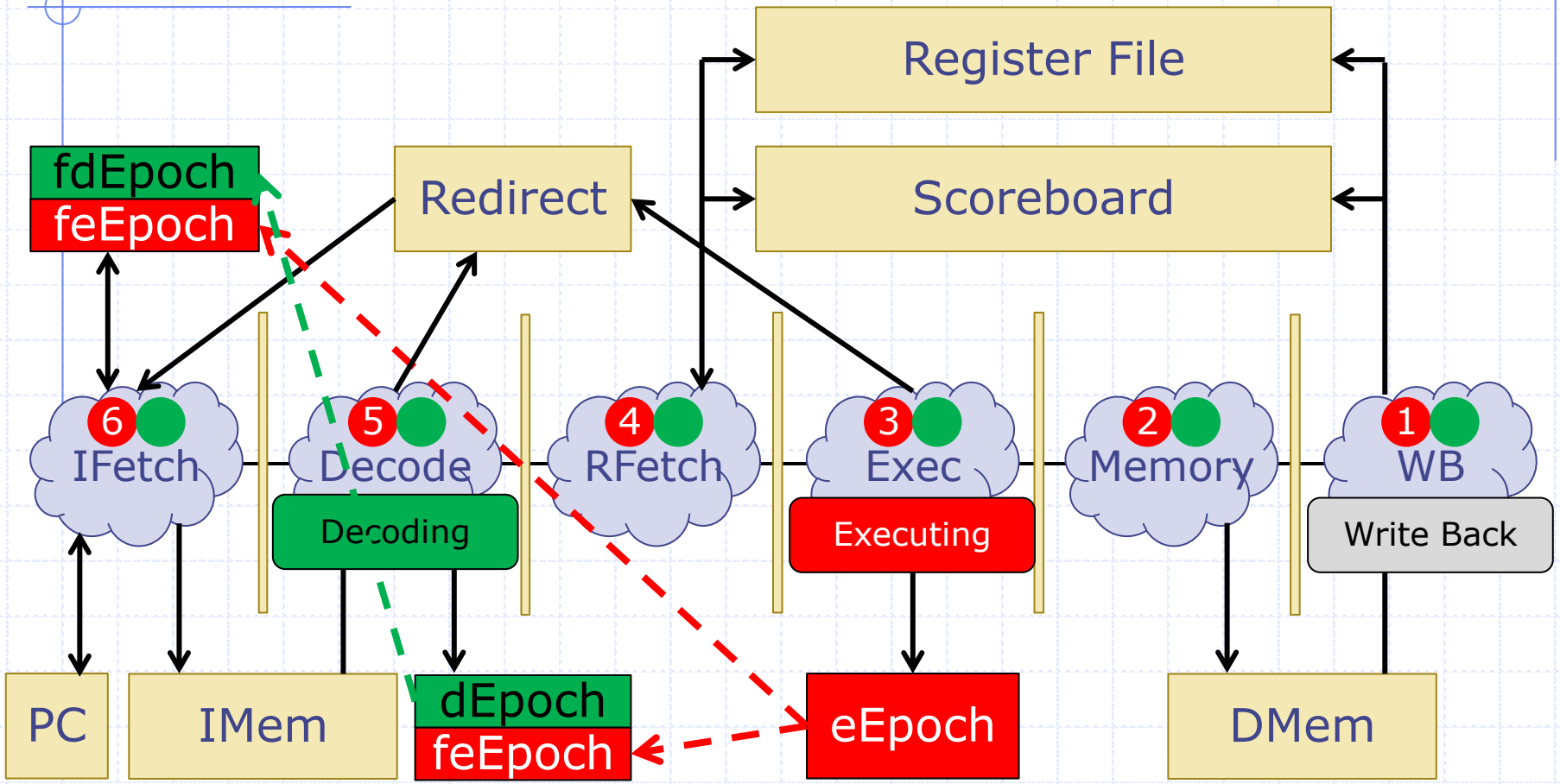


Correcting PC in Decode



Correcting PC in Decode and Execute

Correcting PC in Decode and Execute



Fetch has local estimates of eEpoch and dEpoch

Decode has a local estimate of eEpoch

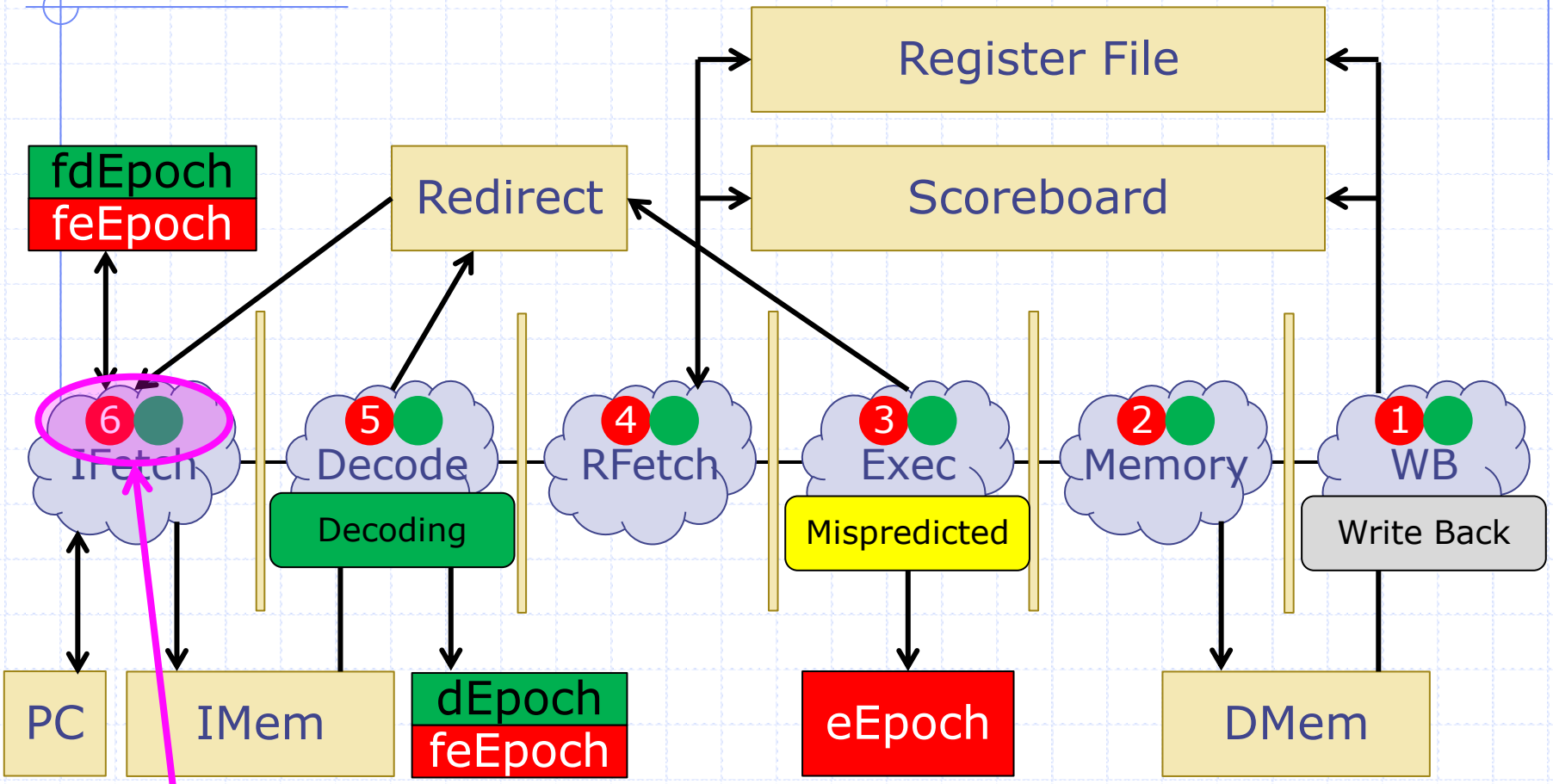
Correcting PC in Decode and Execute

- 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.

Correcting PC in Decode and Execute

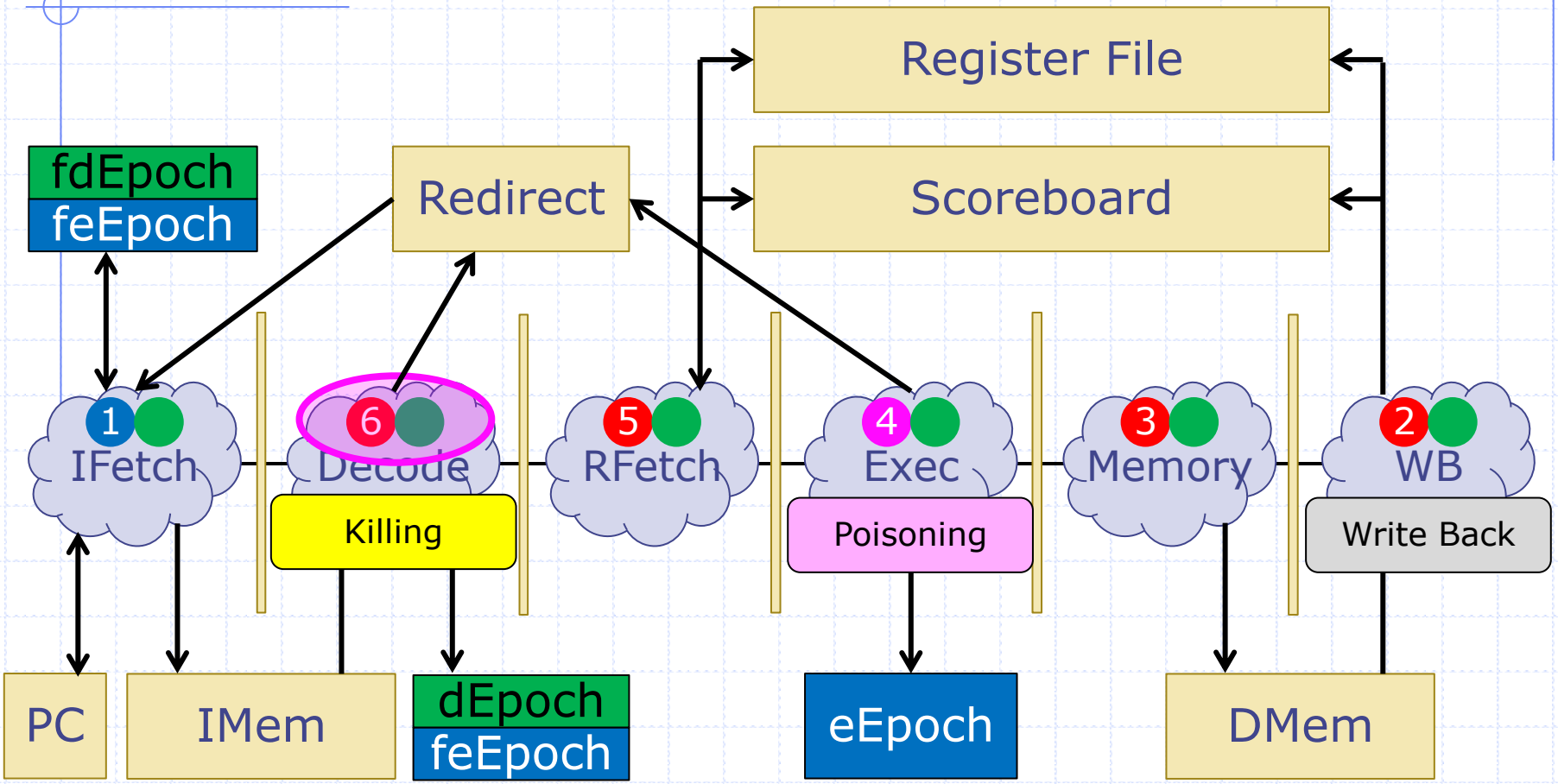
- What if execute sees a misprediction, then decode sees one in the next cycle?
  - The decode instruction will be a wrong path instruction, so it should not try to redirect the PC

Correcting PC in Decode and Execute



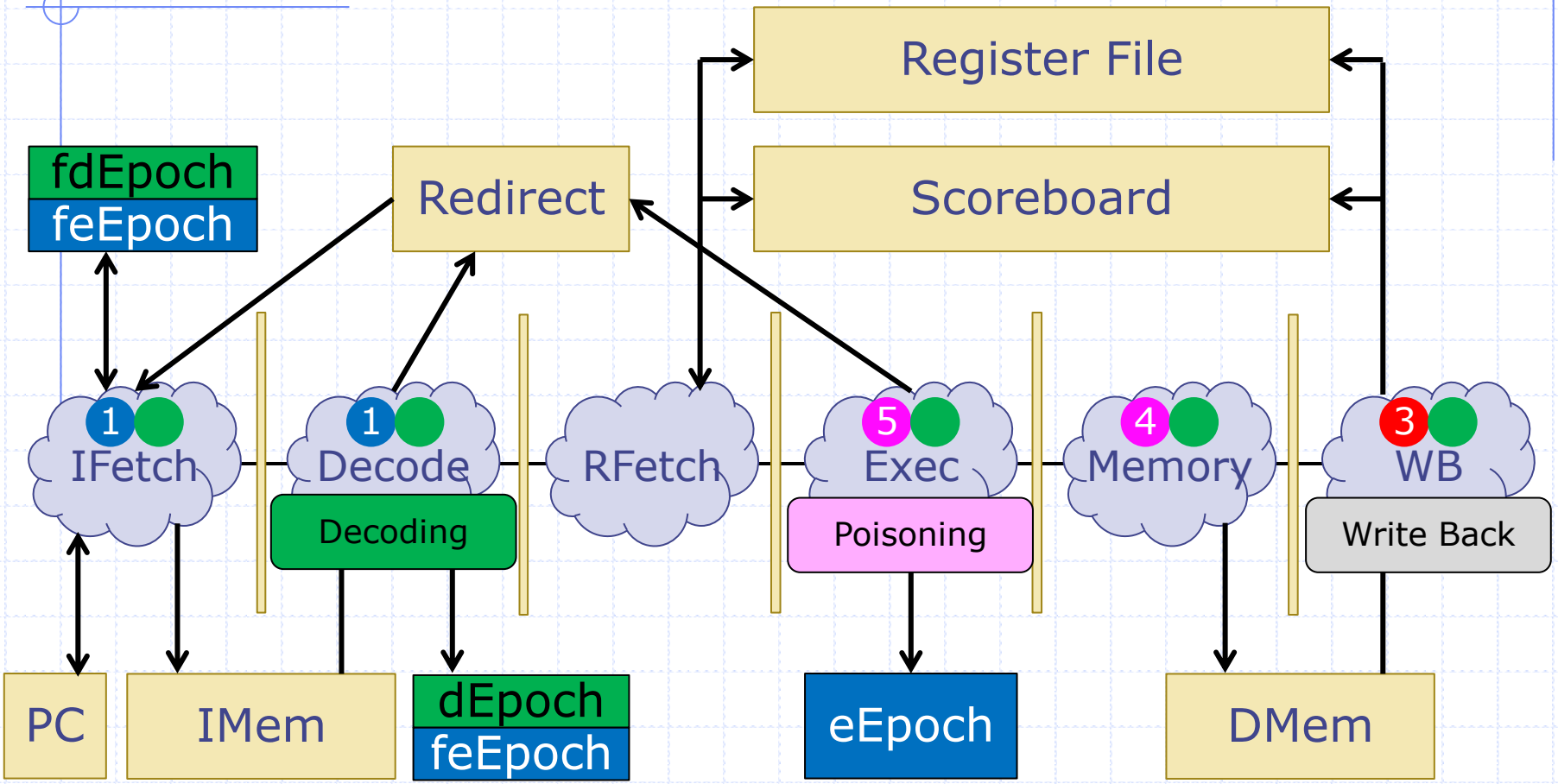
Assume this instruction is a mispredicted jump instruction. It will be in the decode stage next cycle

Correcting PC in Decode and Execute

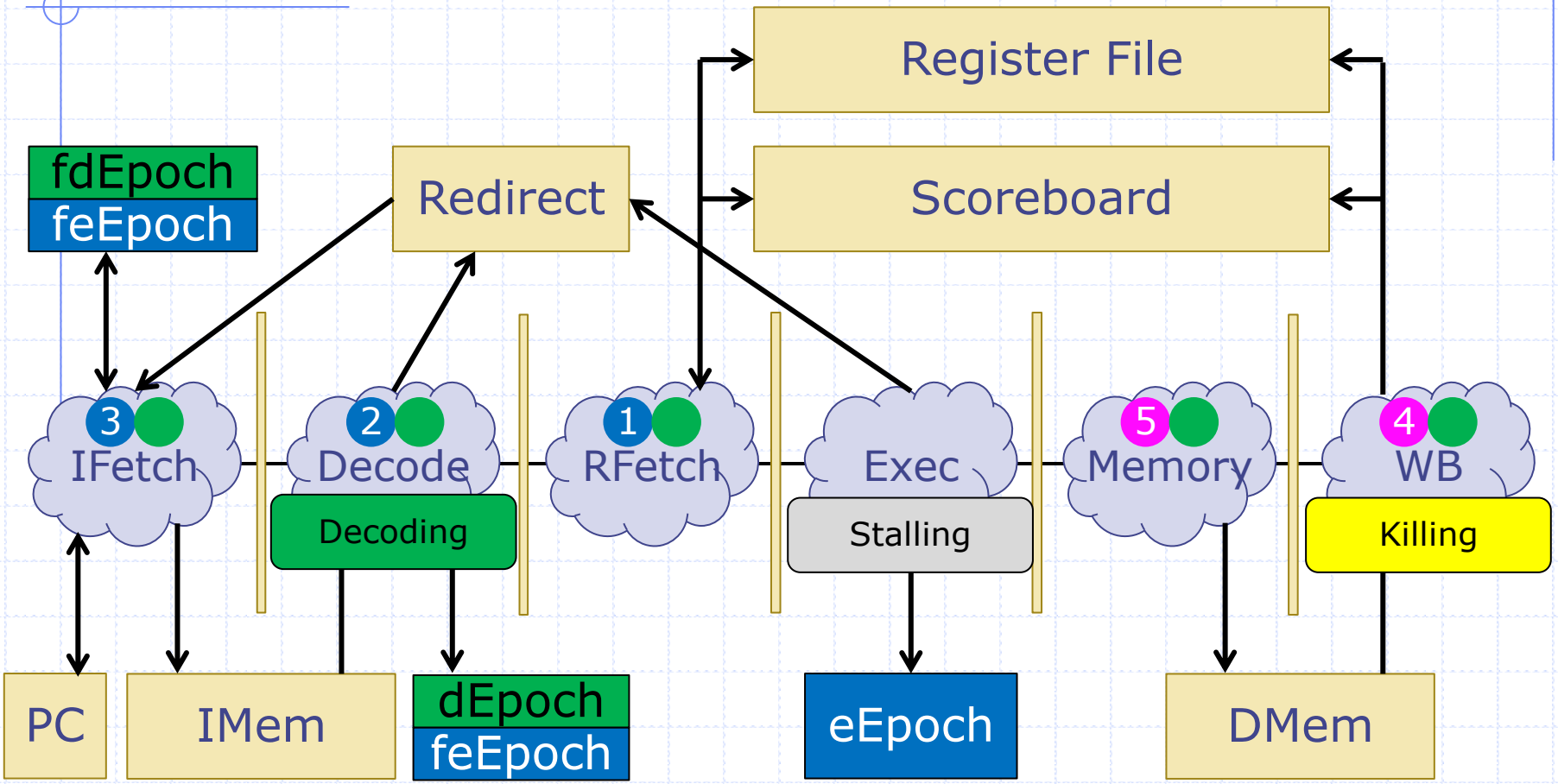


The decode stage knows eEpoch, and recognizes this misprediction is a wrong path instruction. The decode stage kills this instruction.

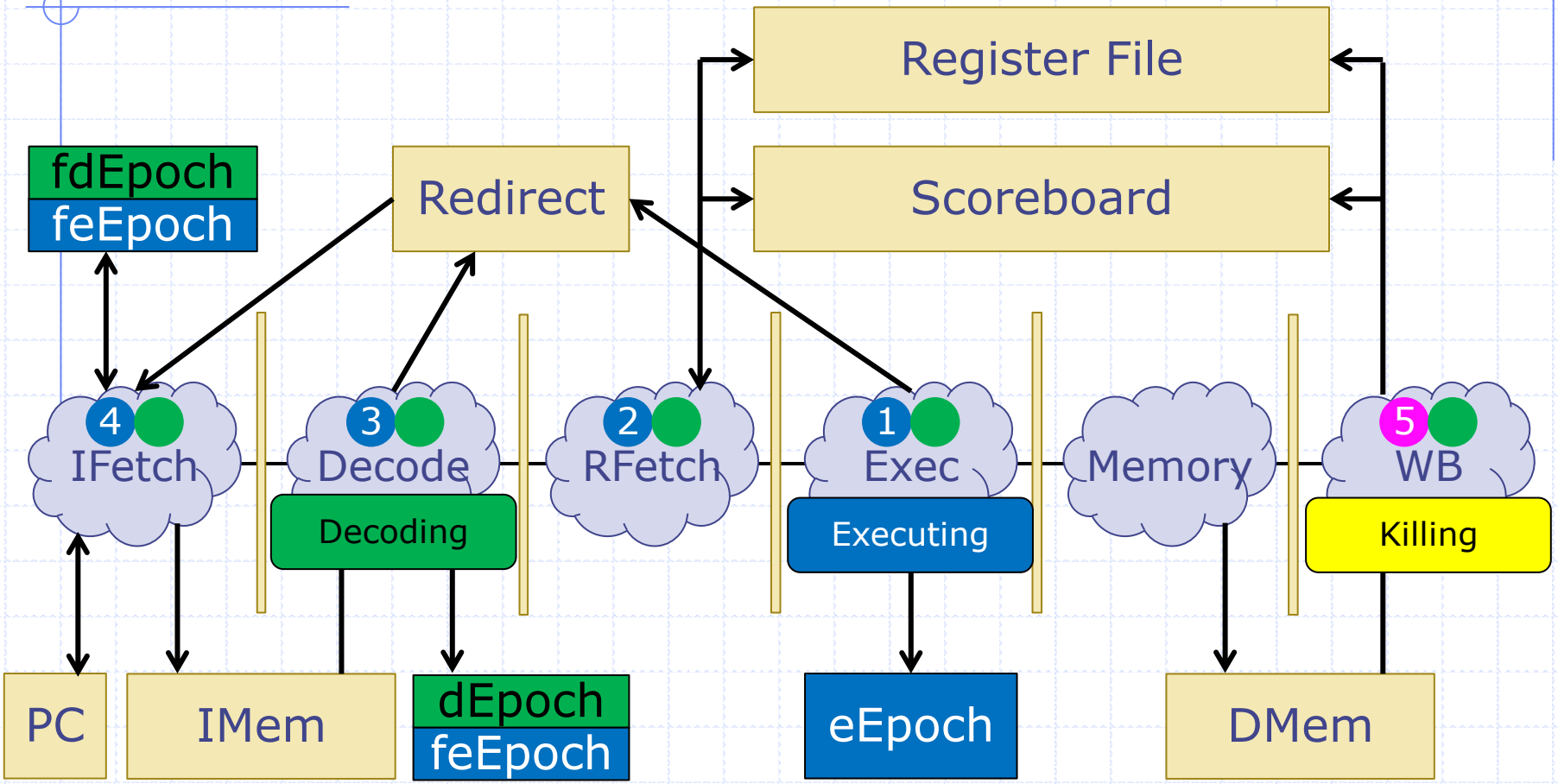
Correcting PC in Decode and Execute



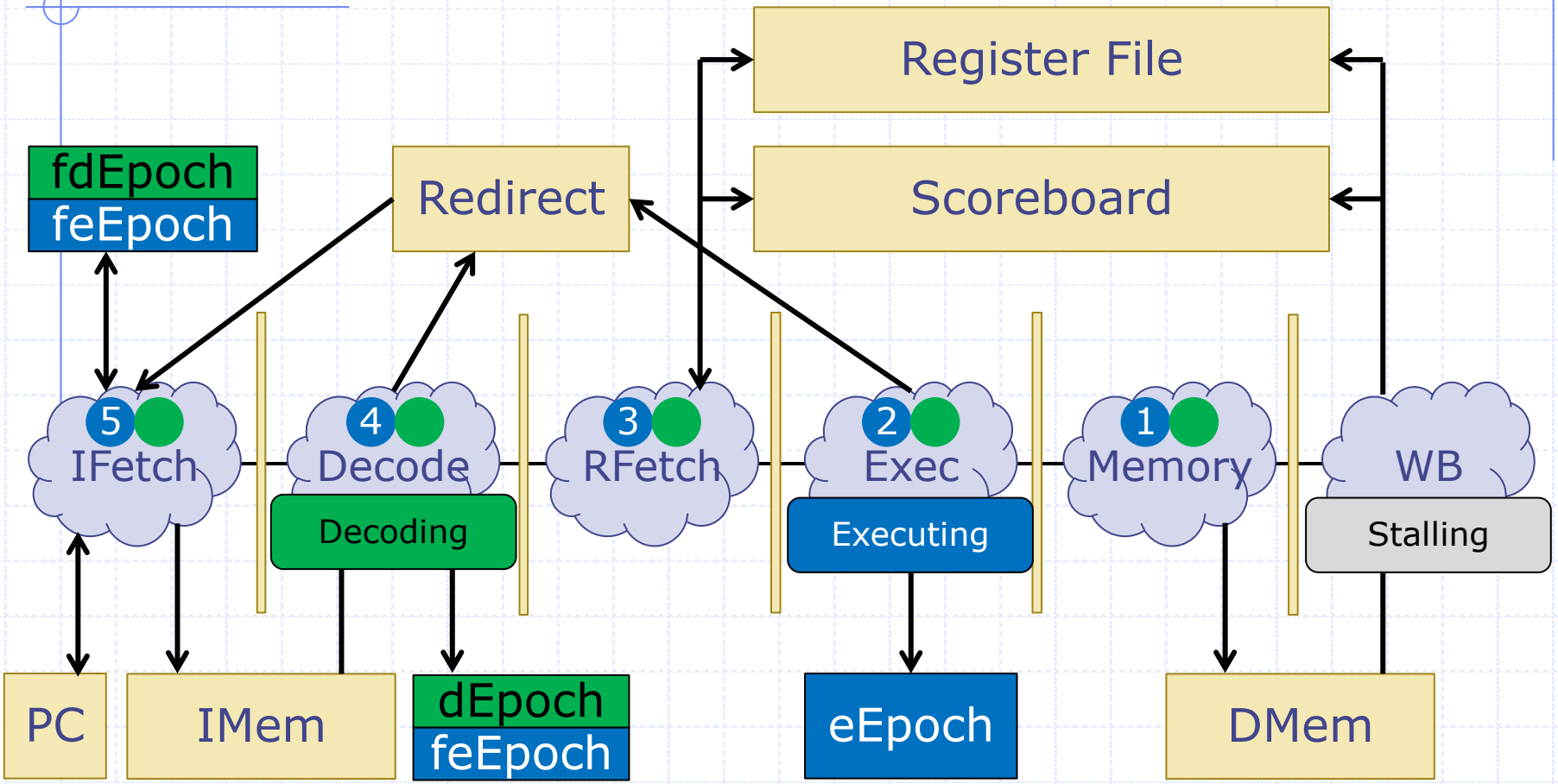
Correcting PC in Decode and Execute



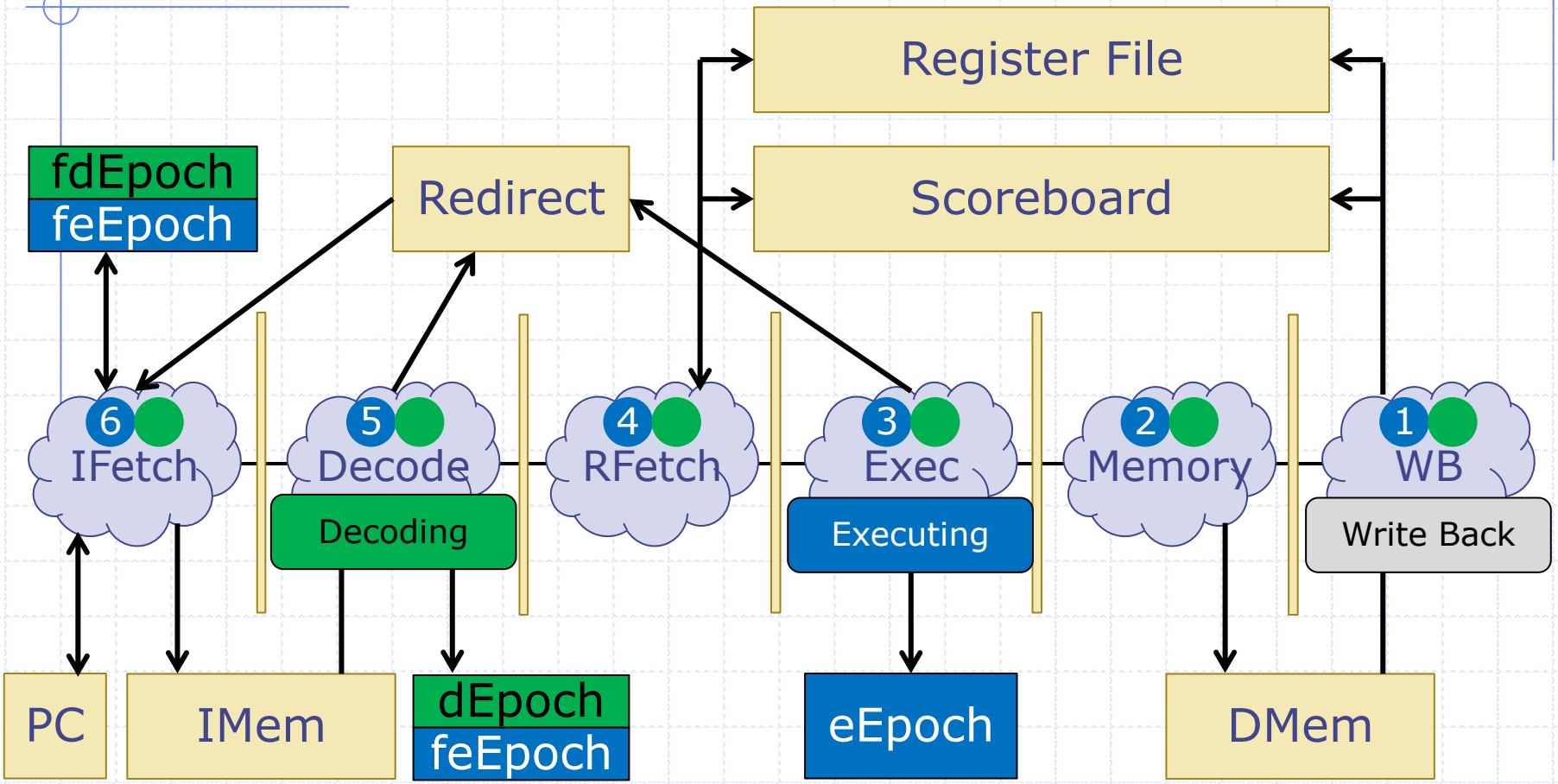
Correcting PC in Decode and Execute



Correcting PC in Decode and Execute



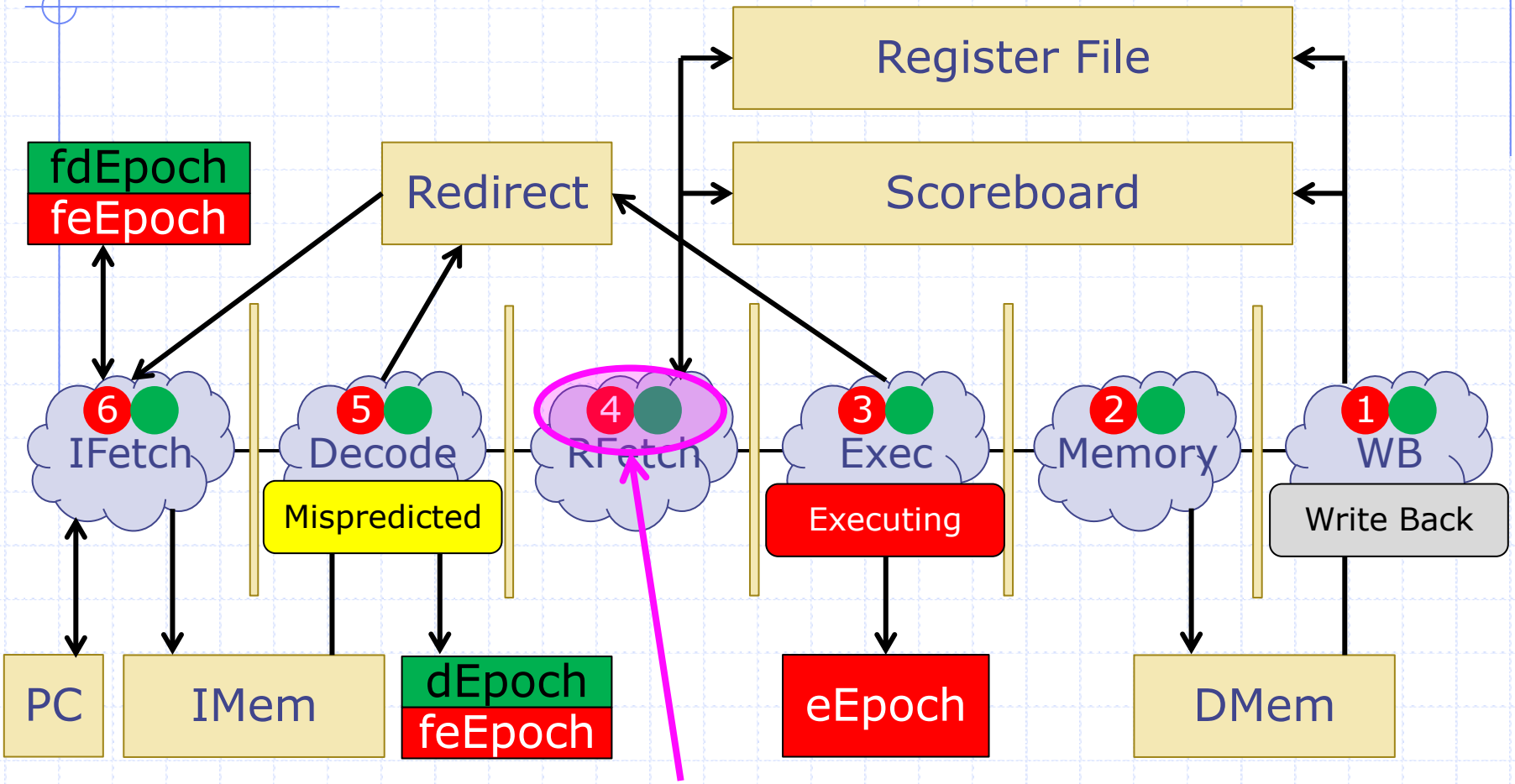
Correcting PC in Decode and Execute



Correcting PC in Decode and Execute

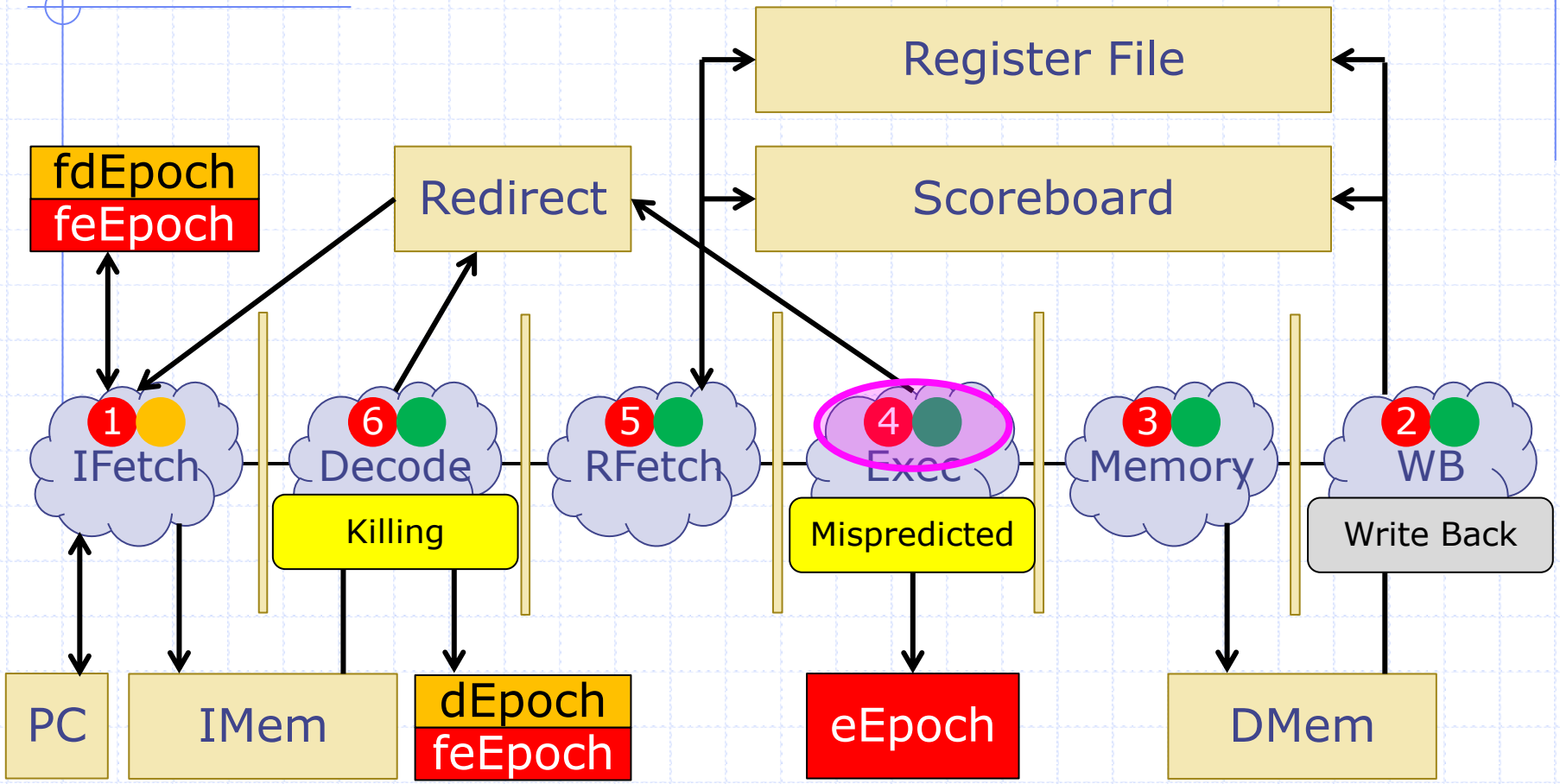
- What if decode sees a misprediction, then execute sees one in the next cycle?
  - The decode instruction will be a wrong path instruction, but it won't be known to be wrong path until later

Correcting PC in Decode and Execute



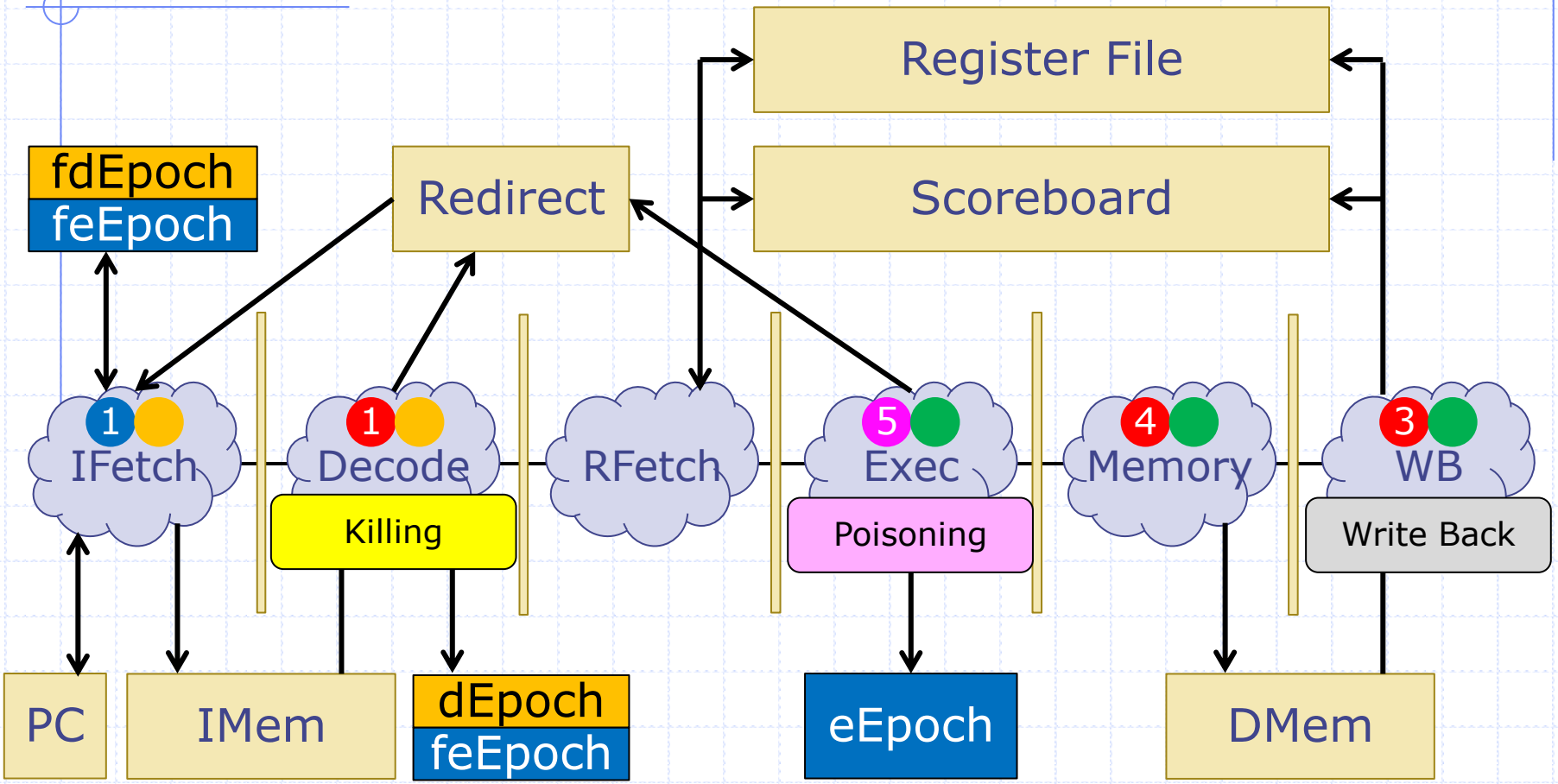
Assume this instruction is a mispredicted branch instruction. It will be in the execute stage next cycle

Correcting PC in Decode and Execute



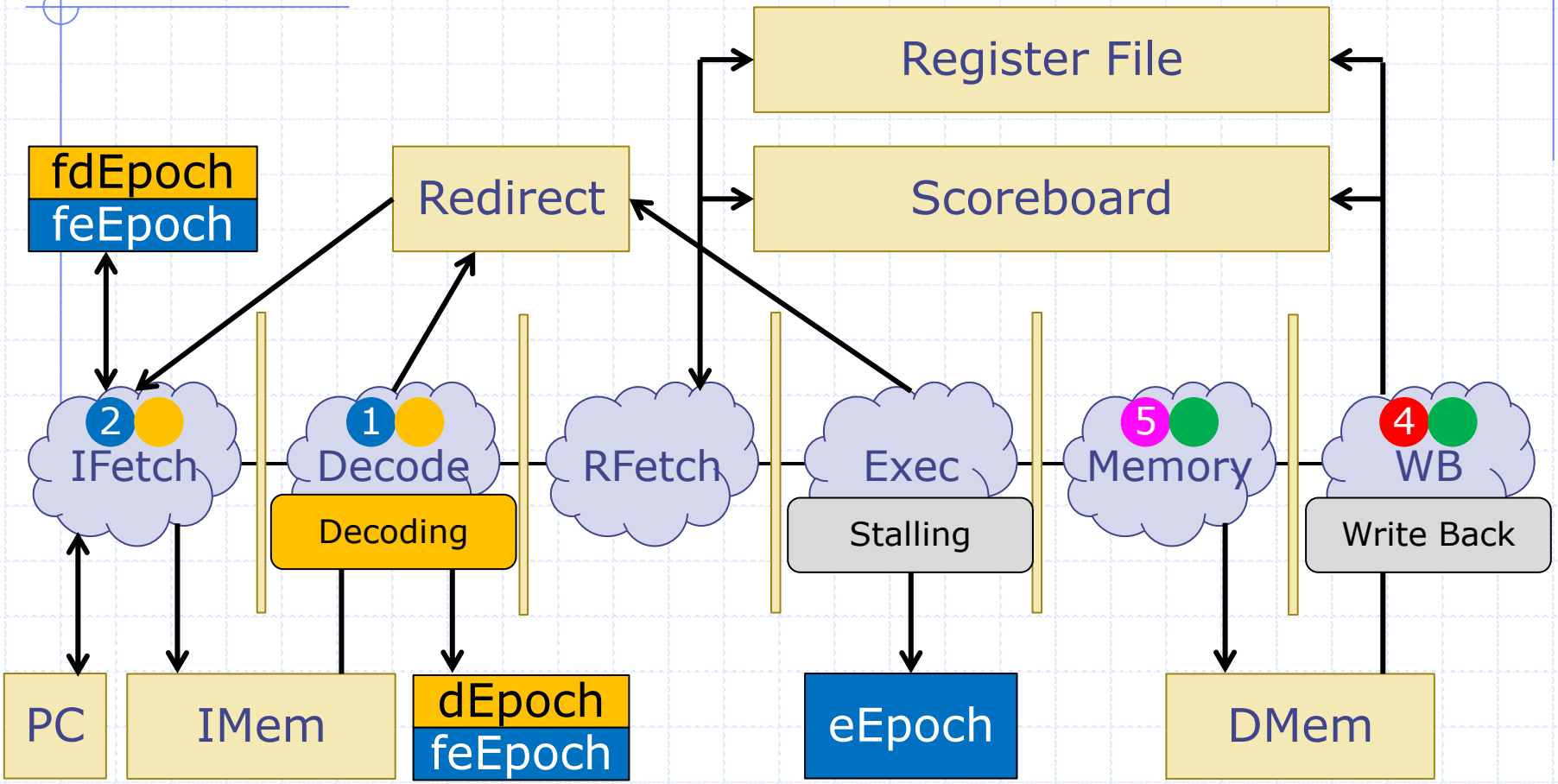
The PC was just "corrected" to a different wrong path instruction

Correcting PC in Decode and Execute



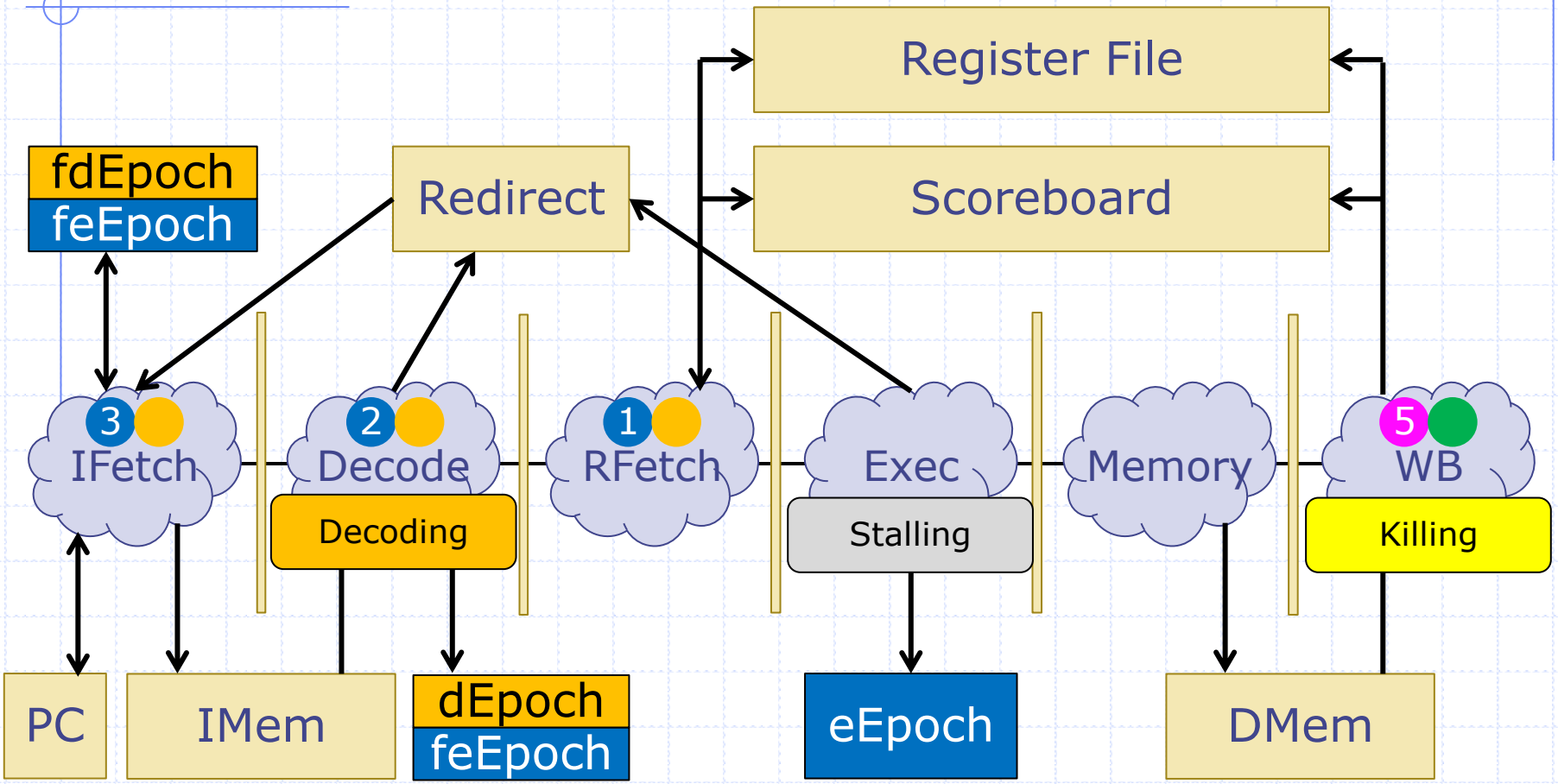
The PC was just corrected to a correct path instruction

Correcting PC in Decode and Execute



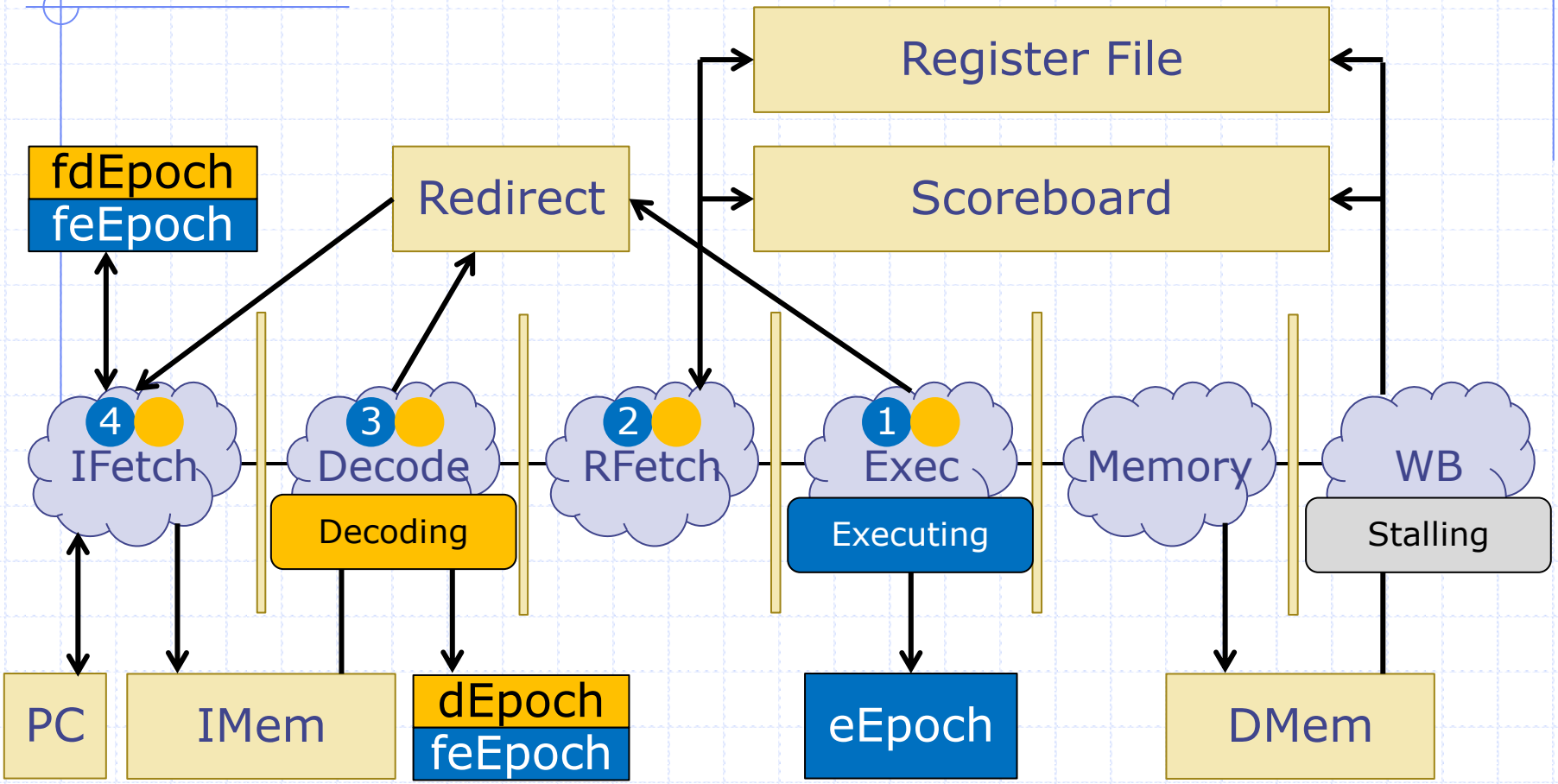
The PC was just corrected to a correct path instruction

Correcting PC in Decode and Execute



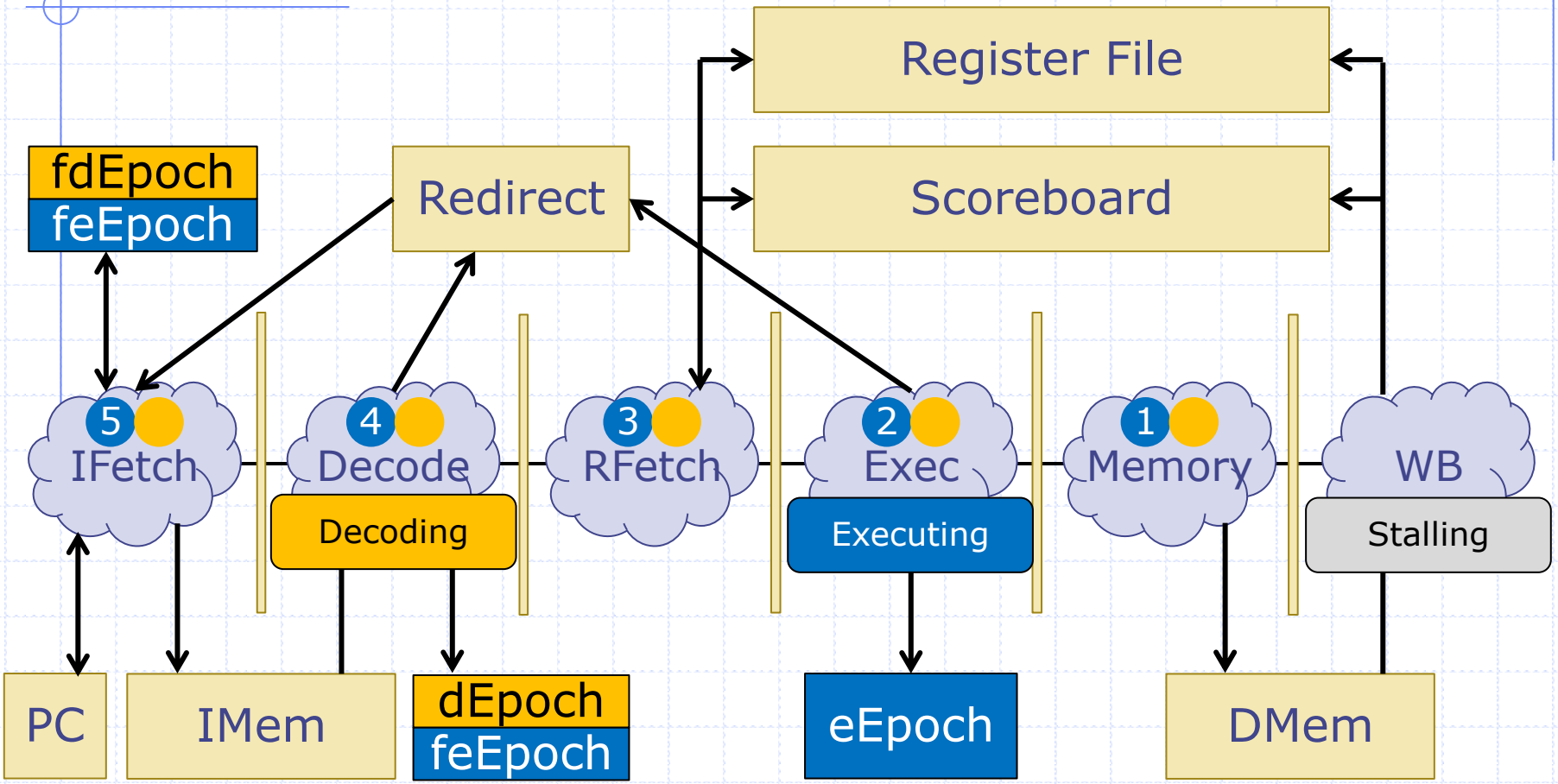
The PC was just corrected to a correct path instruction

Correcting PC in Decode and Execute



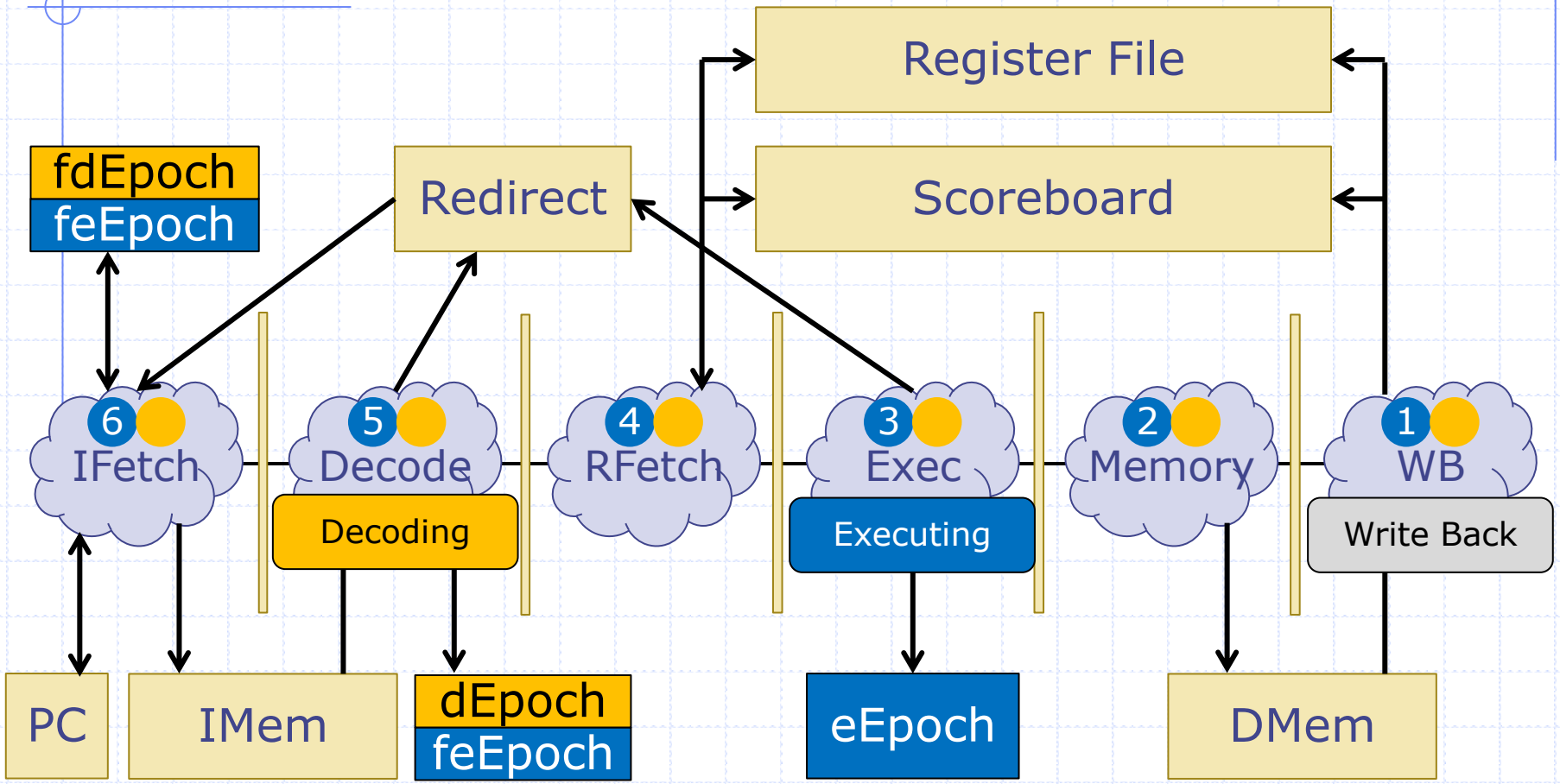
The PC was just corrected to a correct path instruction

Correcting PC in Decode and Execute



The PC was just corrected to a correct path instruction

Correcting PC in Decode and Execute



The PC was just corrected to a correct path instruction

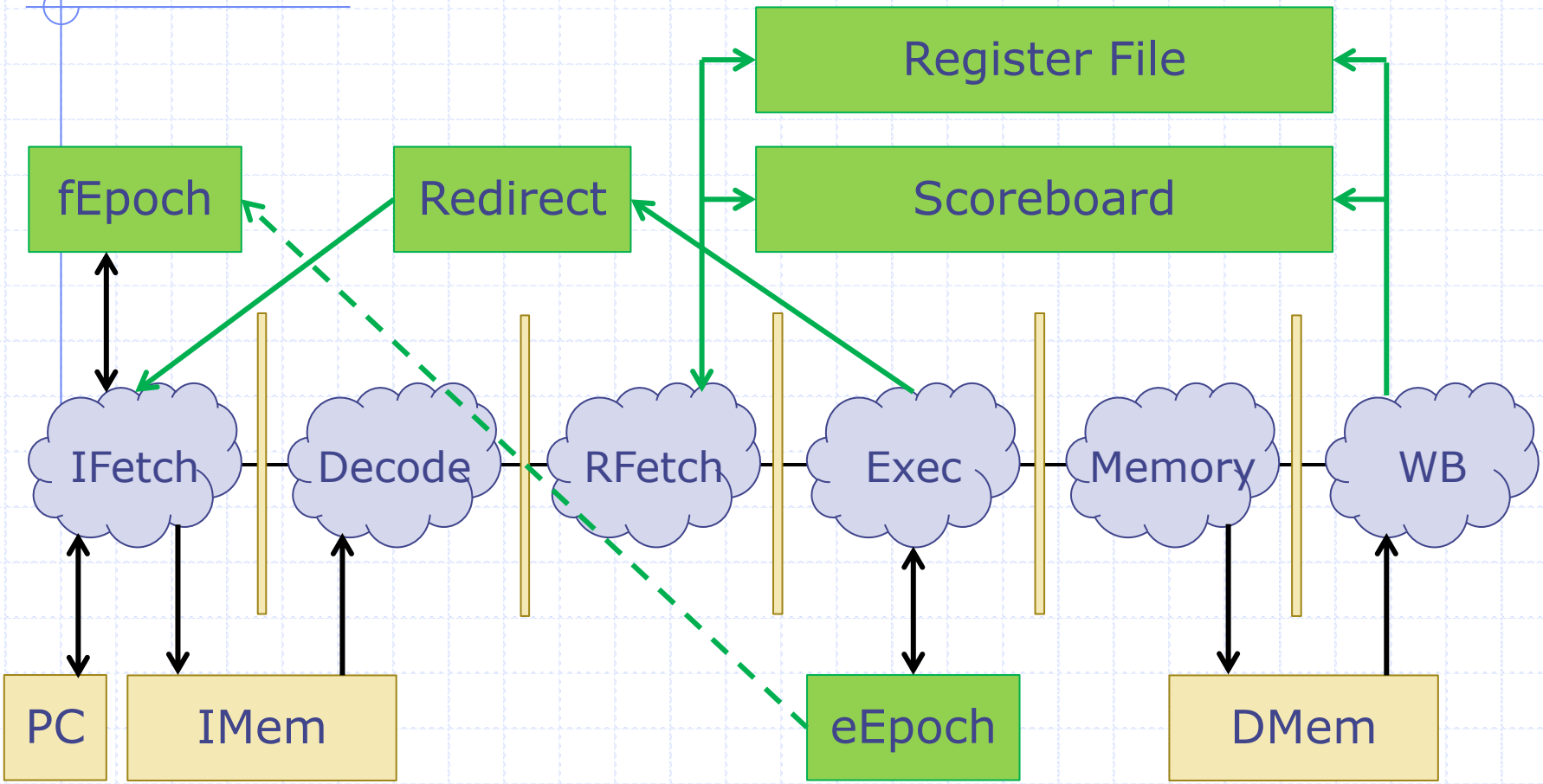
5 Details

- Processor State
- Poisoning Instructions
- ASAP Prediction Correction
- **Pipeline Feedback**
- Removing Pipeline Stages

Pipeline Feedback

- You can forward changes to the processor state to later instructions in the pipeline
  - Forwarding PC state
    - Redirect fifo
    - Epoch updates
  - Forwarding register file state
    - Register file data
    - Scoreboard entries
  - Forwarding memory state
    - Covered later in class (maybe)

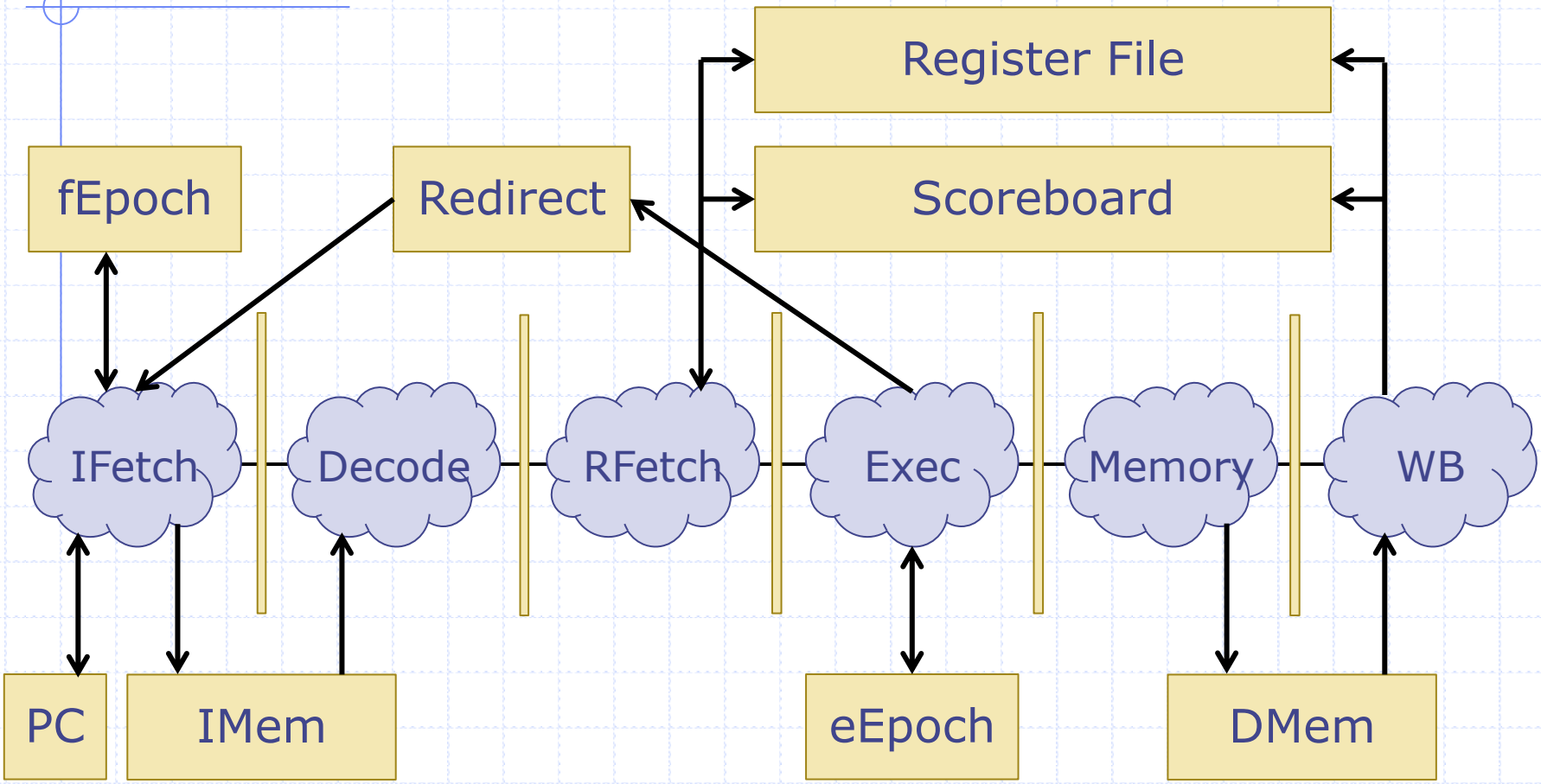
Pipeline Feedback



Pipeline Feedback

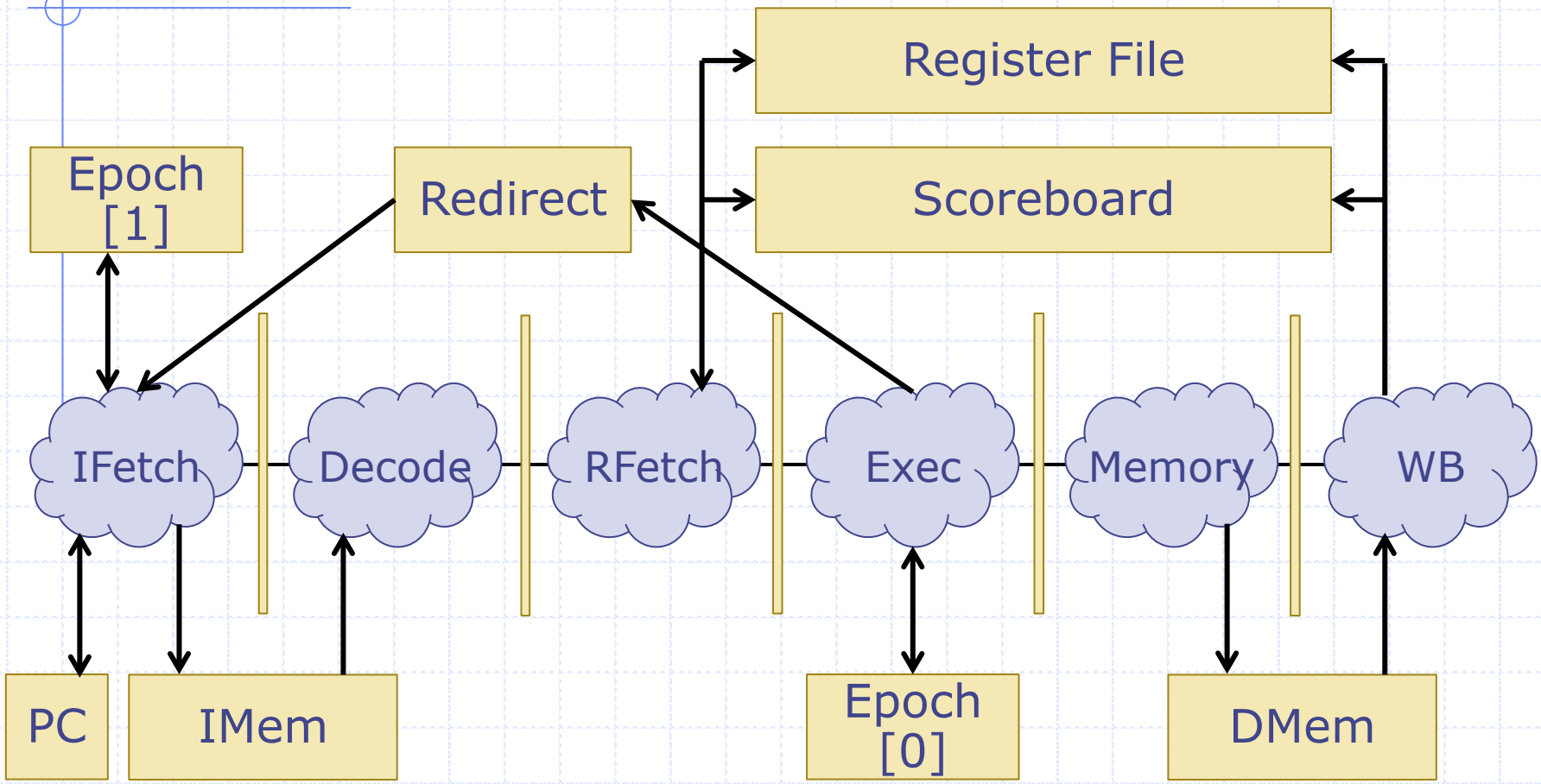
- Better processor performance can be obtained by faster feedback
  - EHRs can be used to make state updates appear to happen in less than a cycle
- How can Epoch and PC feedback be sped up?

fEpoch and PC feedback



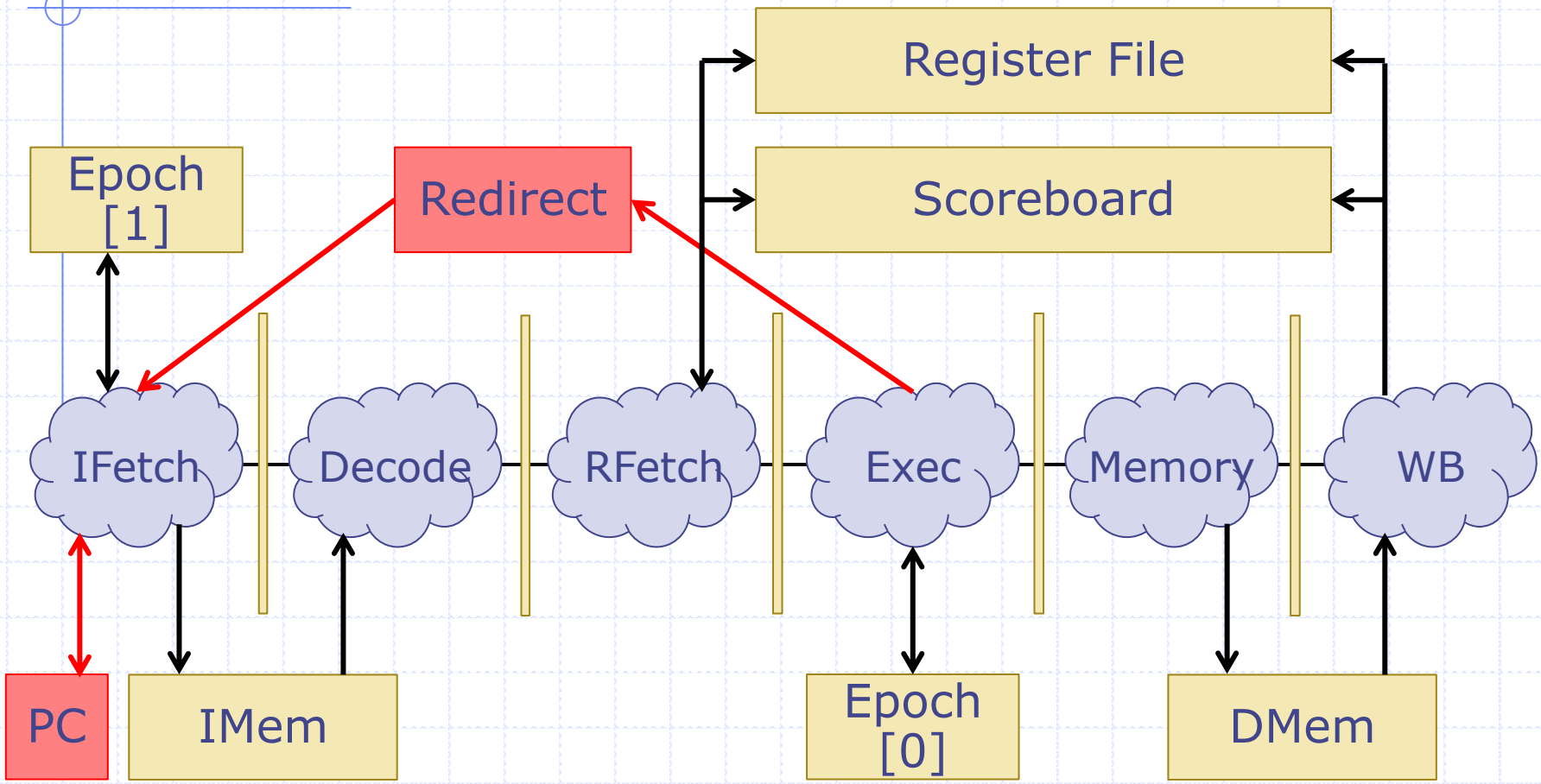
Normally updates to fEpoch and PC epoch have to pass through the redirect fifo. When IFetch sees entries in the redirect fifo, it makes the changes to fEpoch and PC

fEpoch and PC feedback



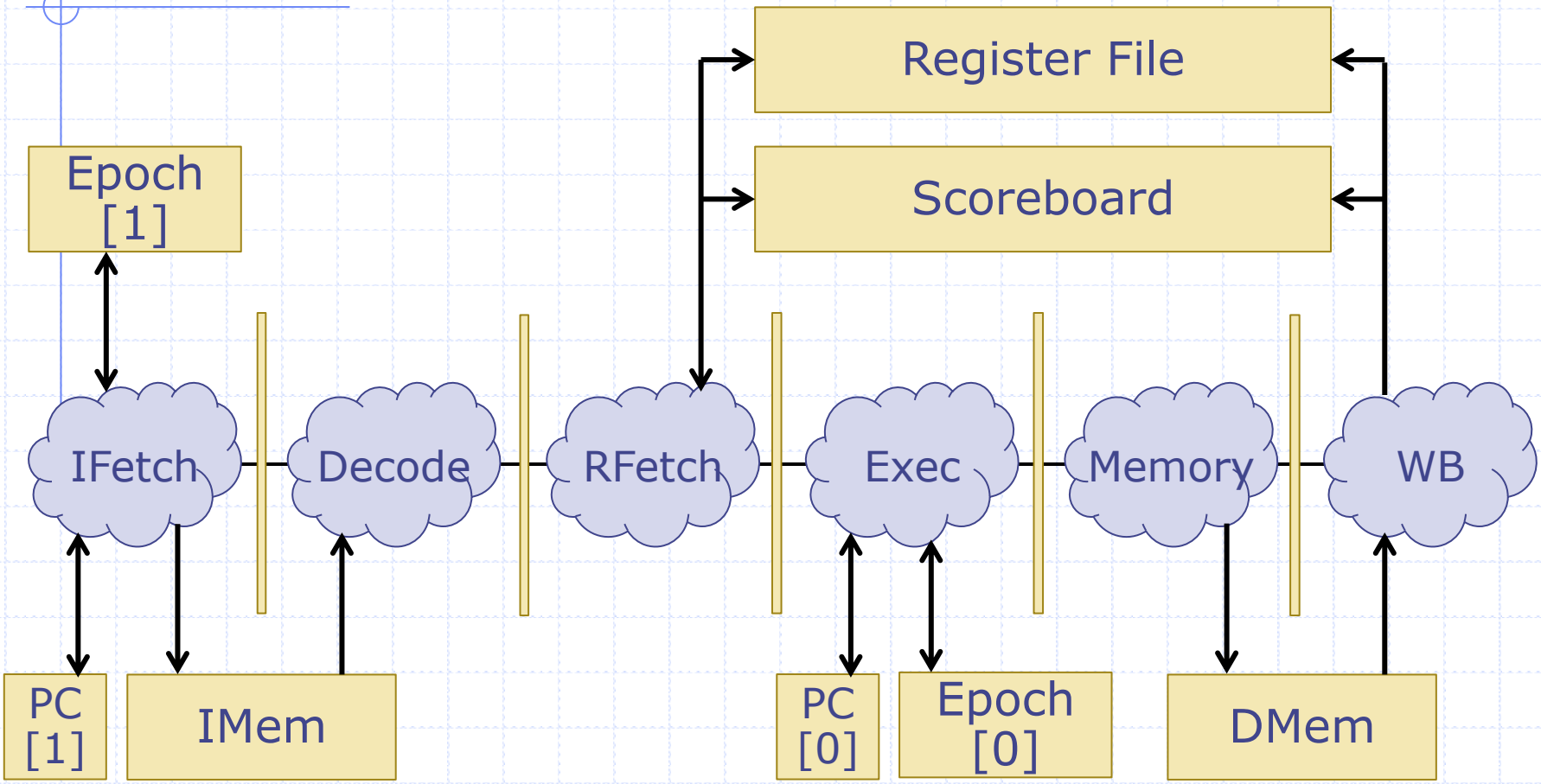
Changes to the Epoch can now be seen by IFetch in the same cycle that execute made the changes

fEpoch and PC feedback



The PC is still coming through the redirect fifo, so the epoch and the pc will get out of synch!

fEpoch and PC feedback

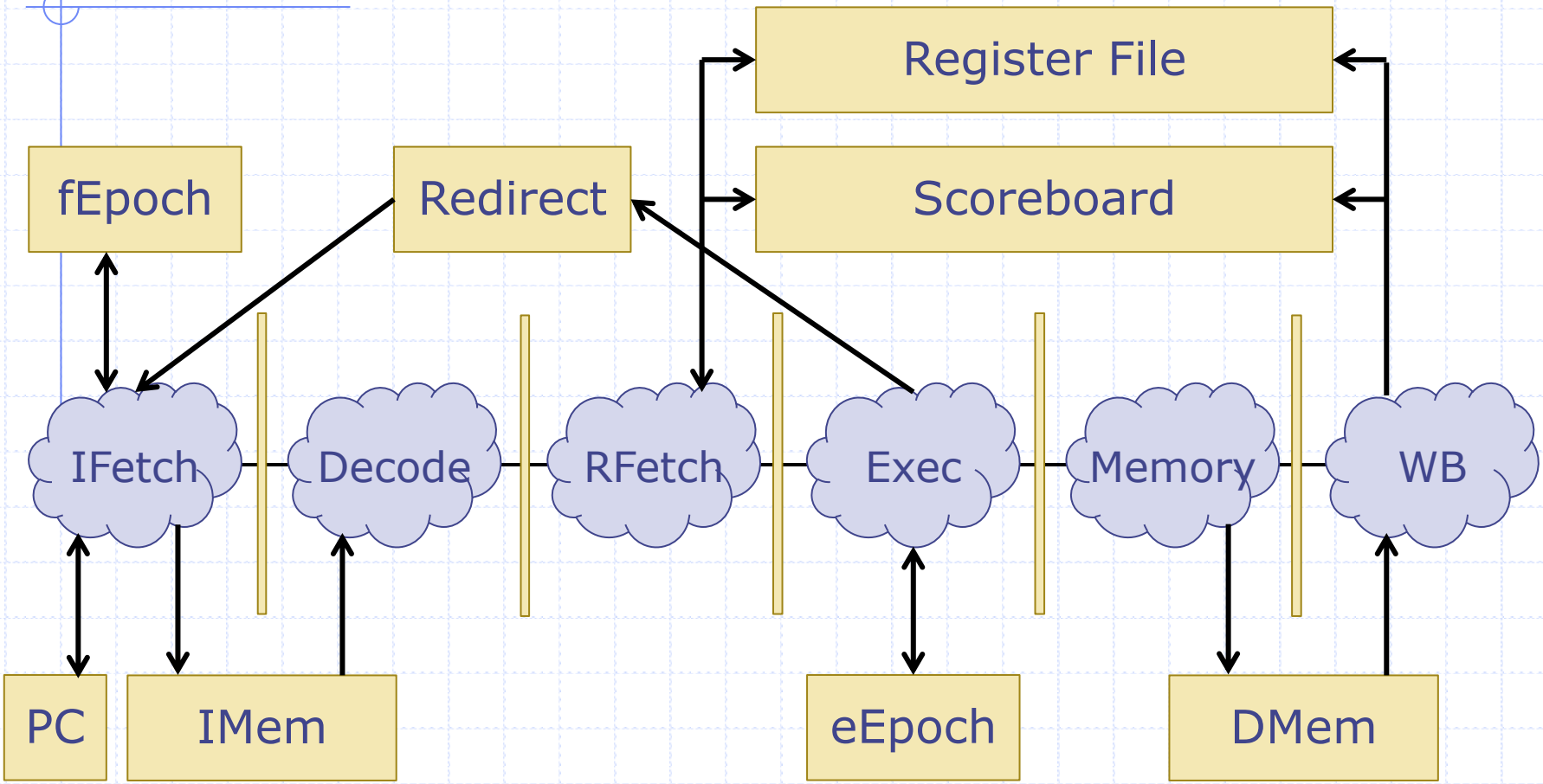


Make the PC an EHR too! Whenever Execute sees a misprediction, IFetch reads the correct next instruction *in the same cycle*!

Pipeline Feedback

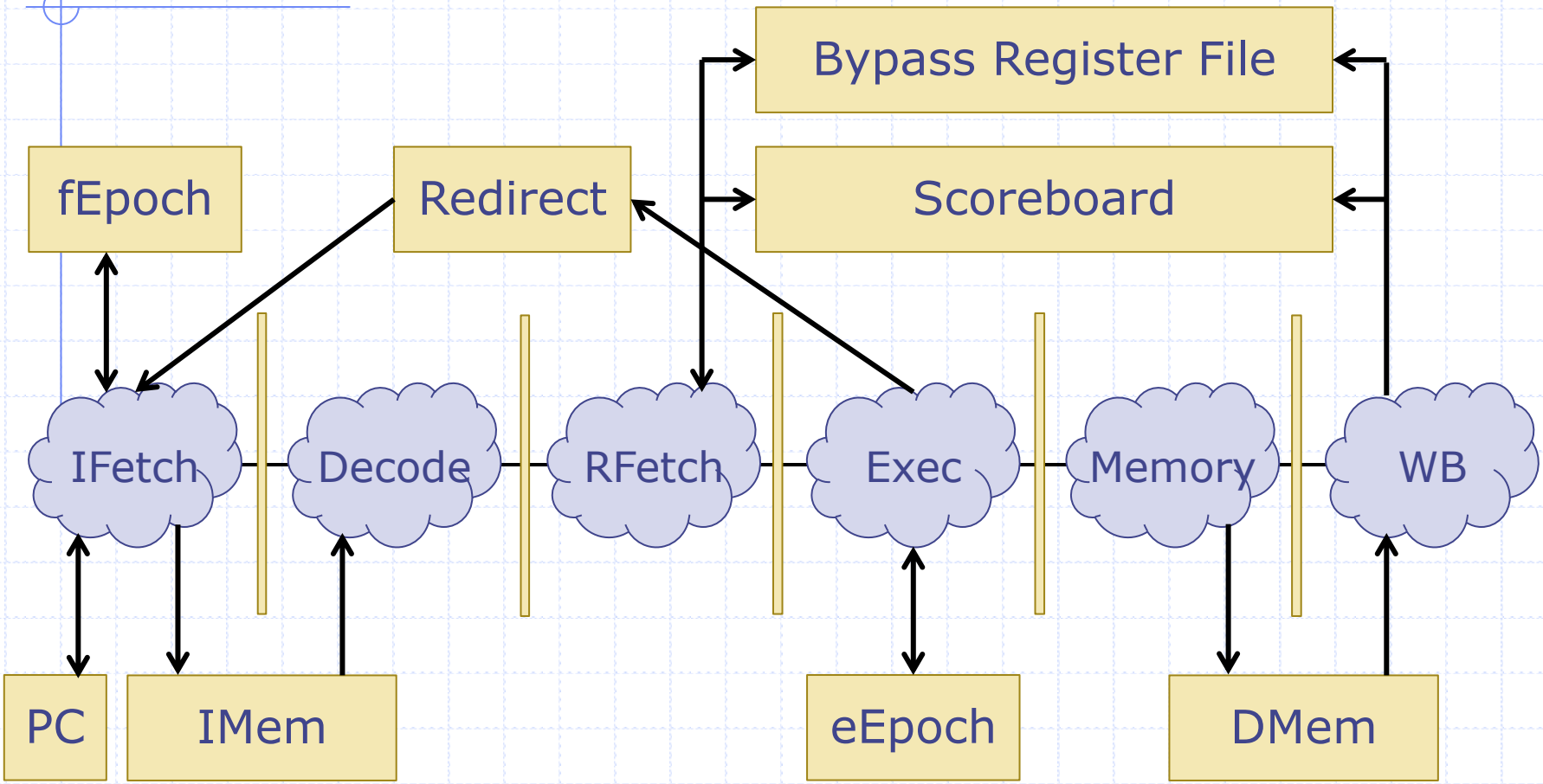
- How can Register File and Scoreboard feedback be sped up?

RFile and SB feedback



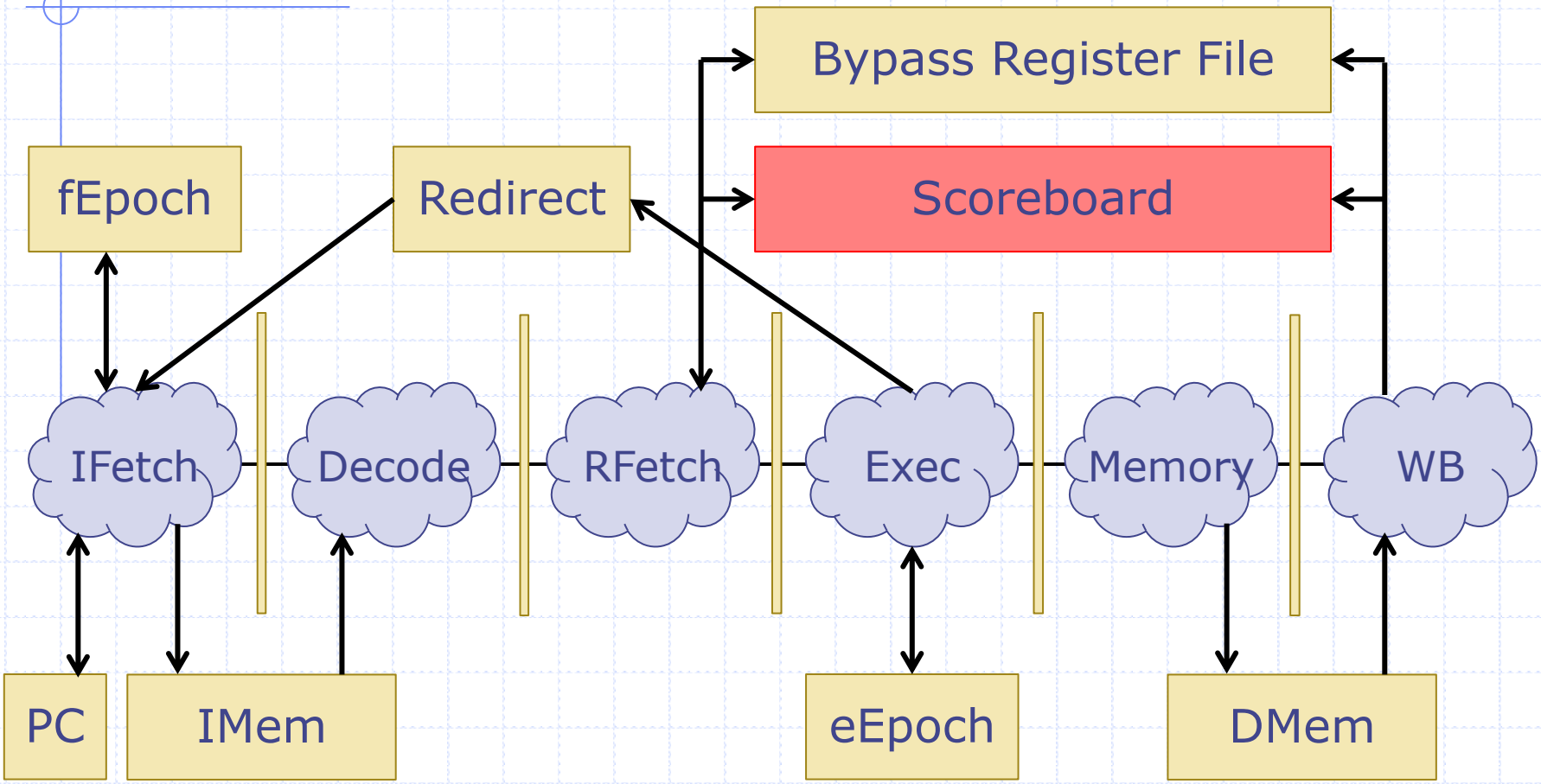
Normally updates (writes) to the register file and updates (removes) to the scoreboard are seen in the next cycle.

RFile and SB feedback



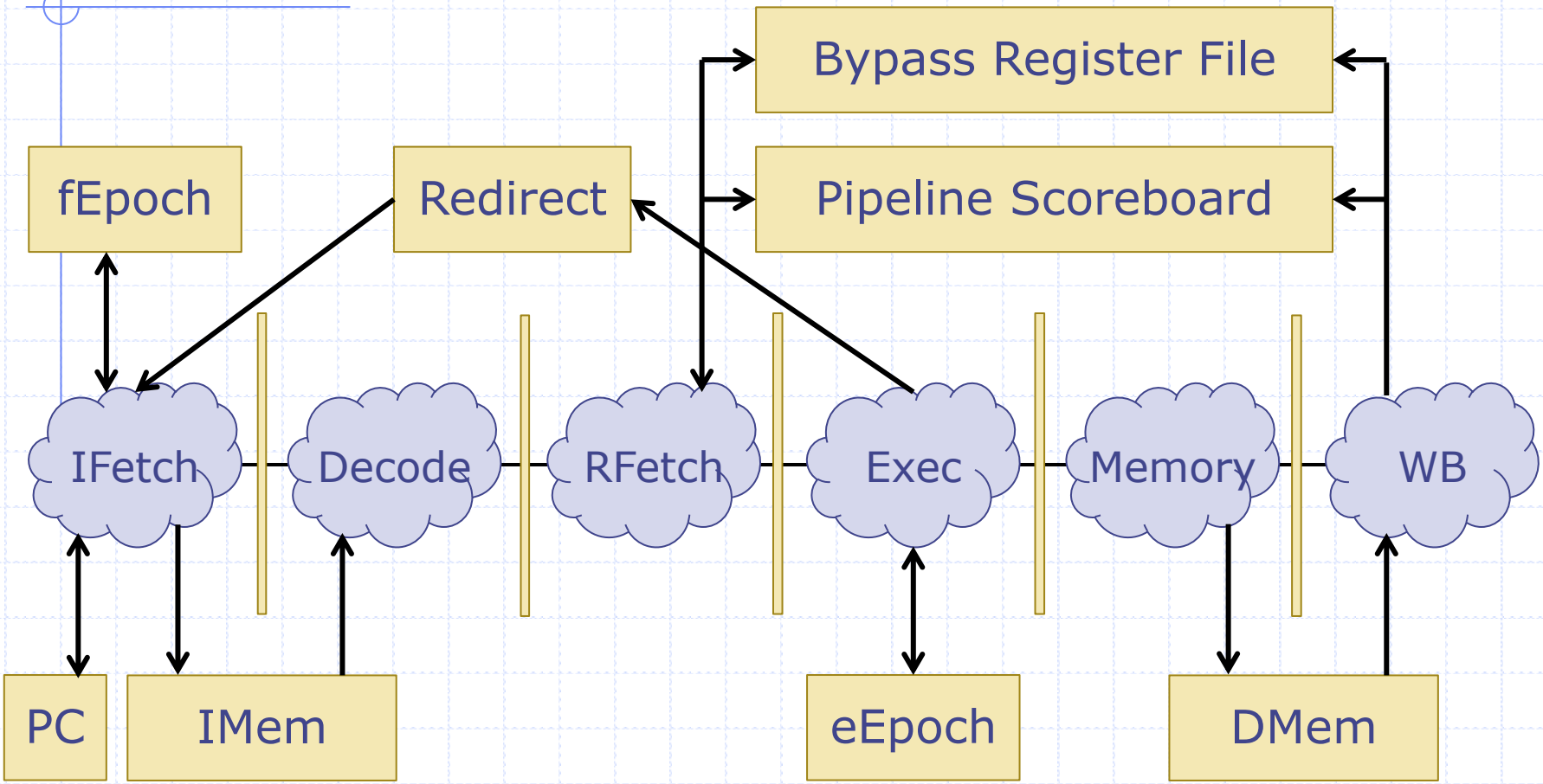
A bypass register file will allow the result from a write to be read by RFetch in the same cycle.

RFile and SB feedback



In this case, the scoreboard is still stalling as much as before

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)

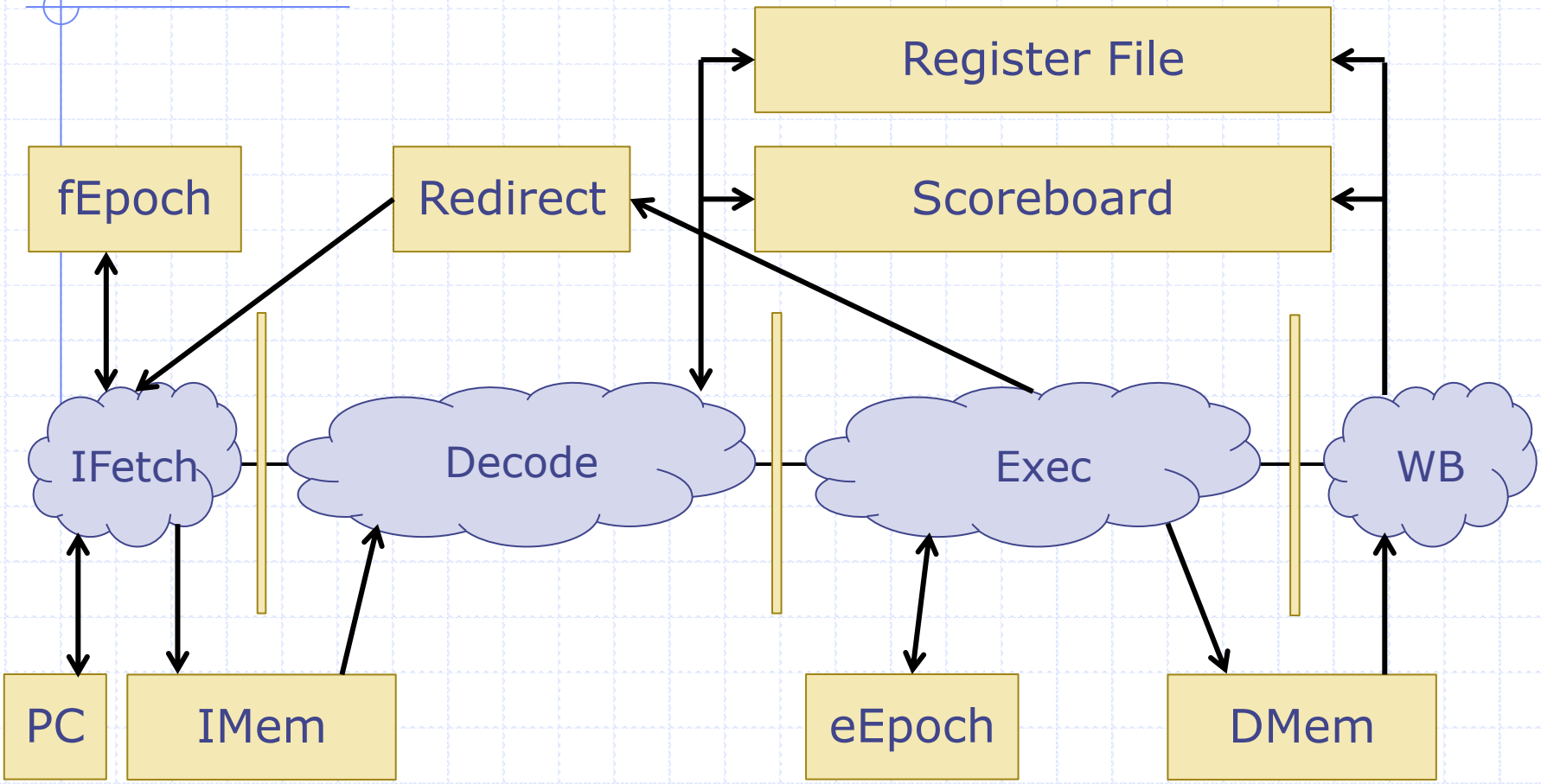
5 Details

- Processor State
- Poisoning Instructions
- ASAP Prediction Correction
- Pipeline Feedback
- **Removing Pipeline Stages**

Removing Pipeline Stages

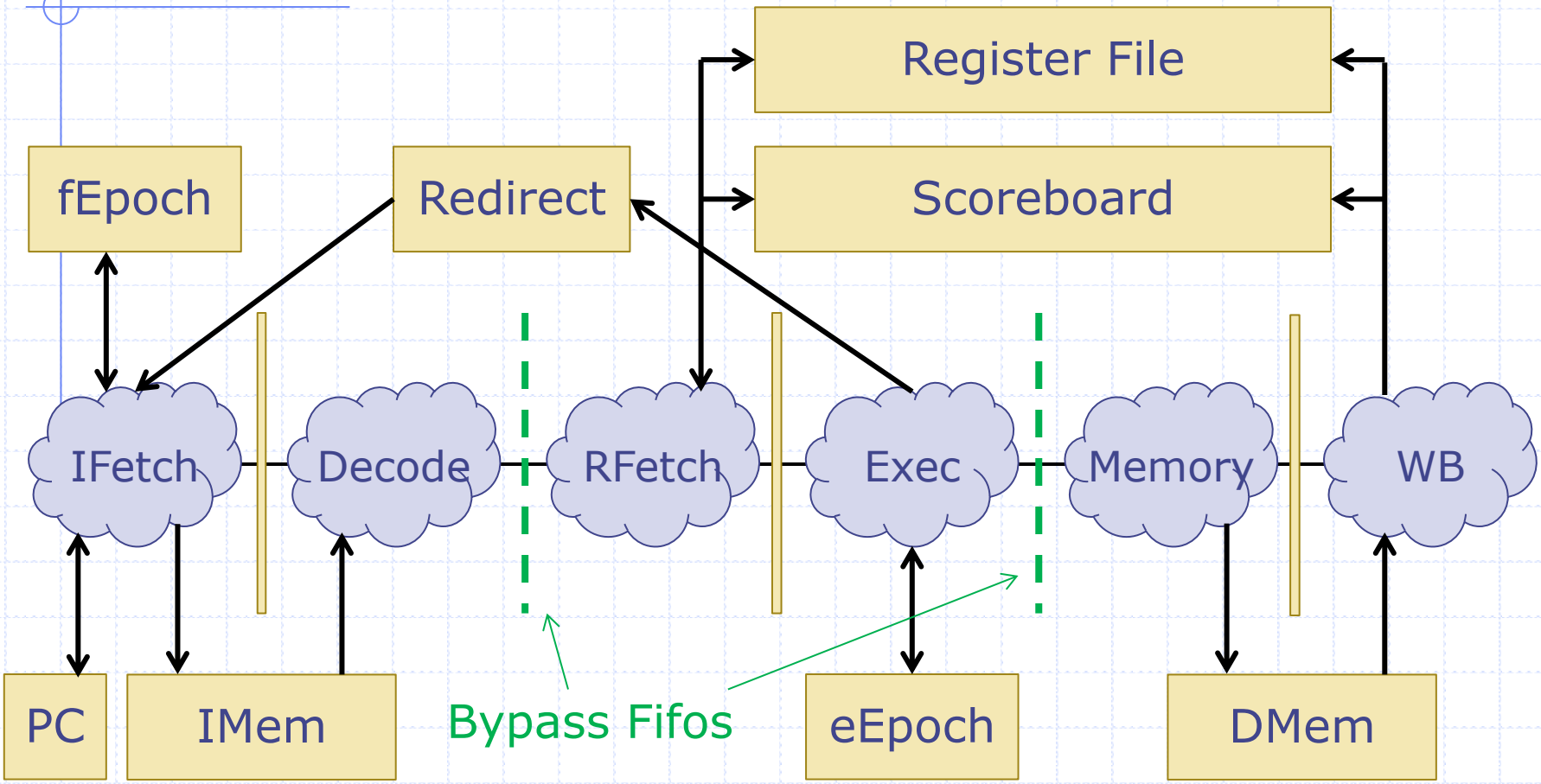
- You will create a 6 stage SMIPS pipeline in Lab 6
  - 6 is a lot of stages and may not be necessary
  - How much work would it be to turn it into a shorter pipeline? Say 4 stages?
  - How much work would it be to shift work from one pipeline stage to another?

Removing Pipeline Stages



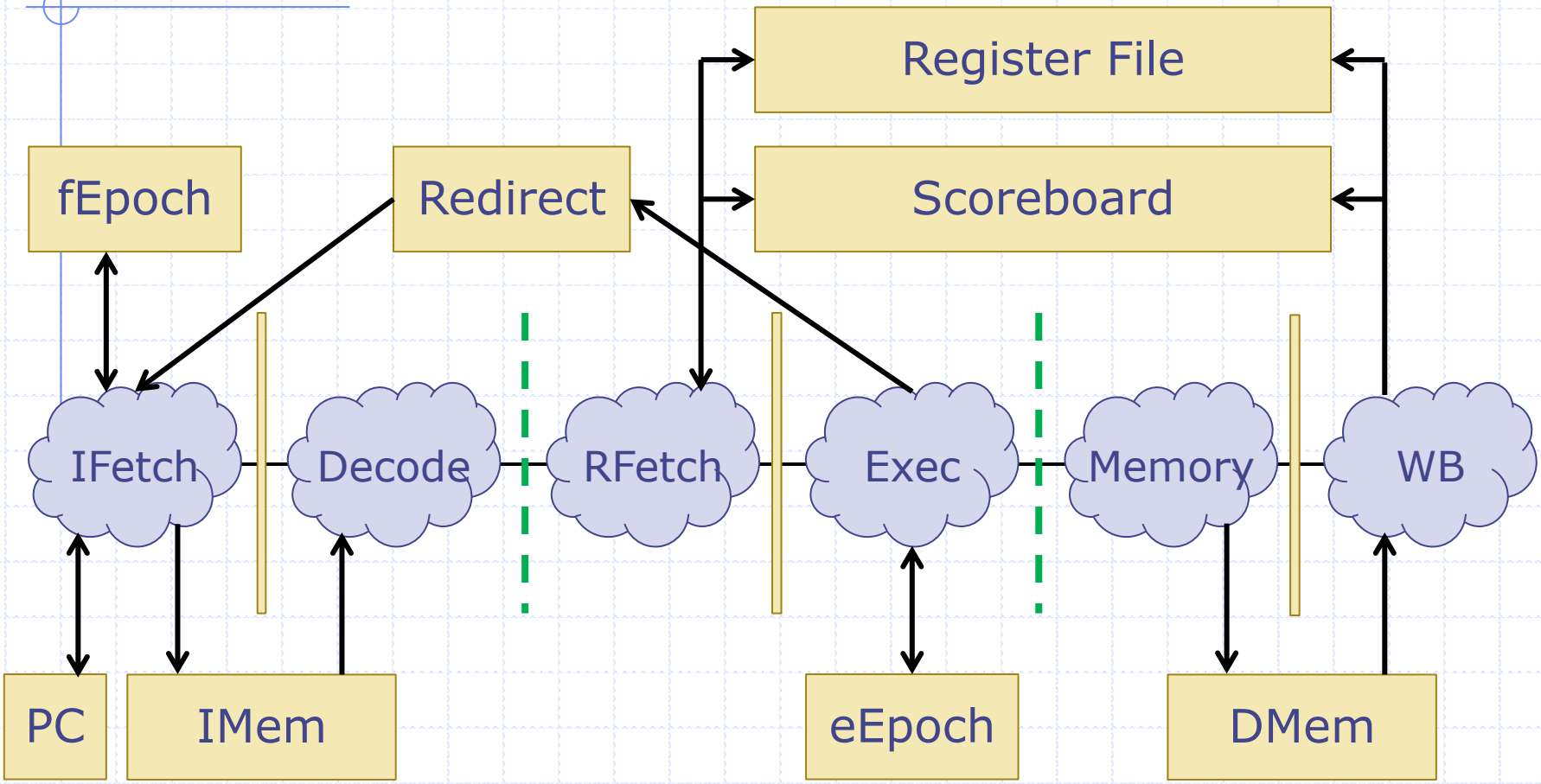
Say you want a 4 stage pipeline where Decode does RFetch and Exec does Memory. How do you make this machine with as little work as possible?

Removing Pipeline Stages



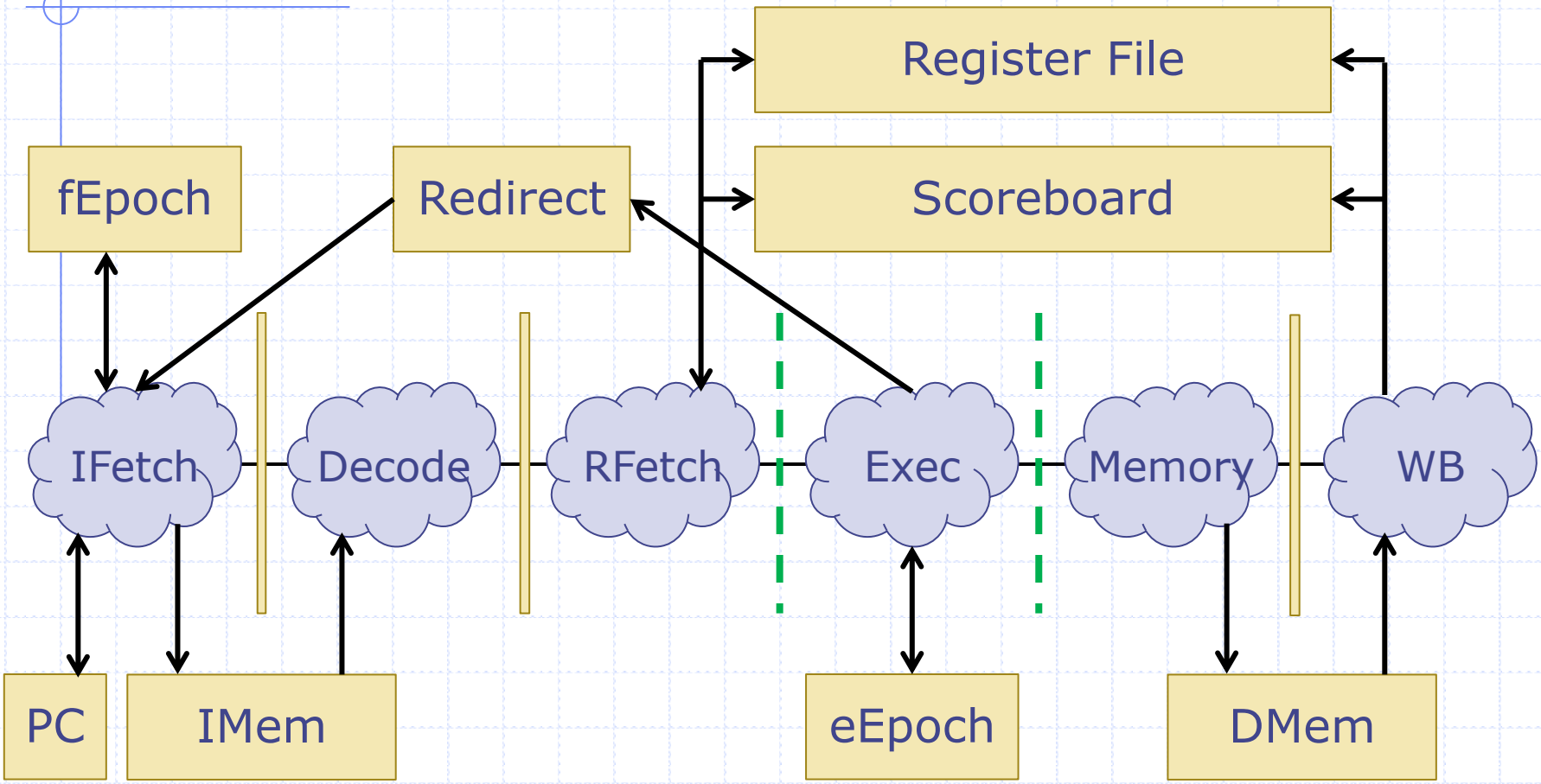
Replace CFFifos between stages with bypass fifos. The bypass fifos will act as wires when possible.

Removing Pipeline Stages



Now you want to move RFetch to the Exec stage to reduce the amount of time stalled for RAW hazards. How much work is this?

Removing Pipeline Stages



Just change the types of fifos between Decode and RFetch and between RFetch and Exec.

Removing Pipeline Stages

- The 6 stage pipeline is very flexible.
- You can try out many different stage configurations in FPGA synthesis to see how your IPS (Instructions Per Seconds) changes.

Conclusion

- Processor State
  - Most processor errors can be related to operating on the wrong processor state
- Poisoning Instructions
  - Needed for lab 6
- ASAP Prediction Correction
  - This will be covered in more depth in lab 7.
- Pipeline Feedback
  - Can be used to speed up processors
- Removing pipeline stages
  - Can also be used to speed up processors

Questions?