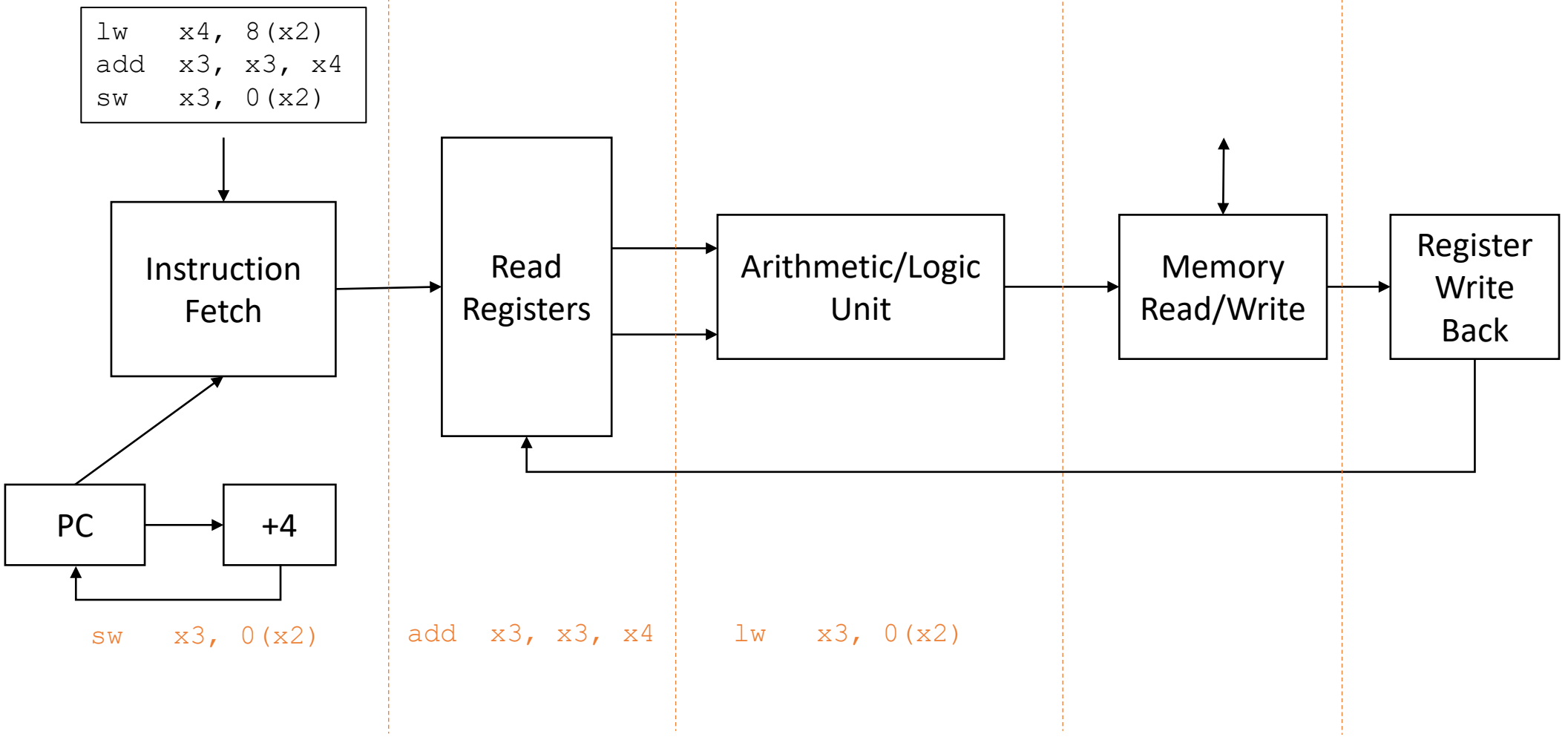
CSE 451: Operating Systems Spring 2020

Module 0
(Instruction Level) Parallelism (cont.)

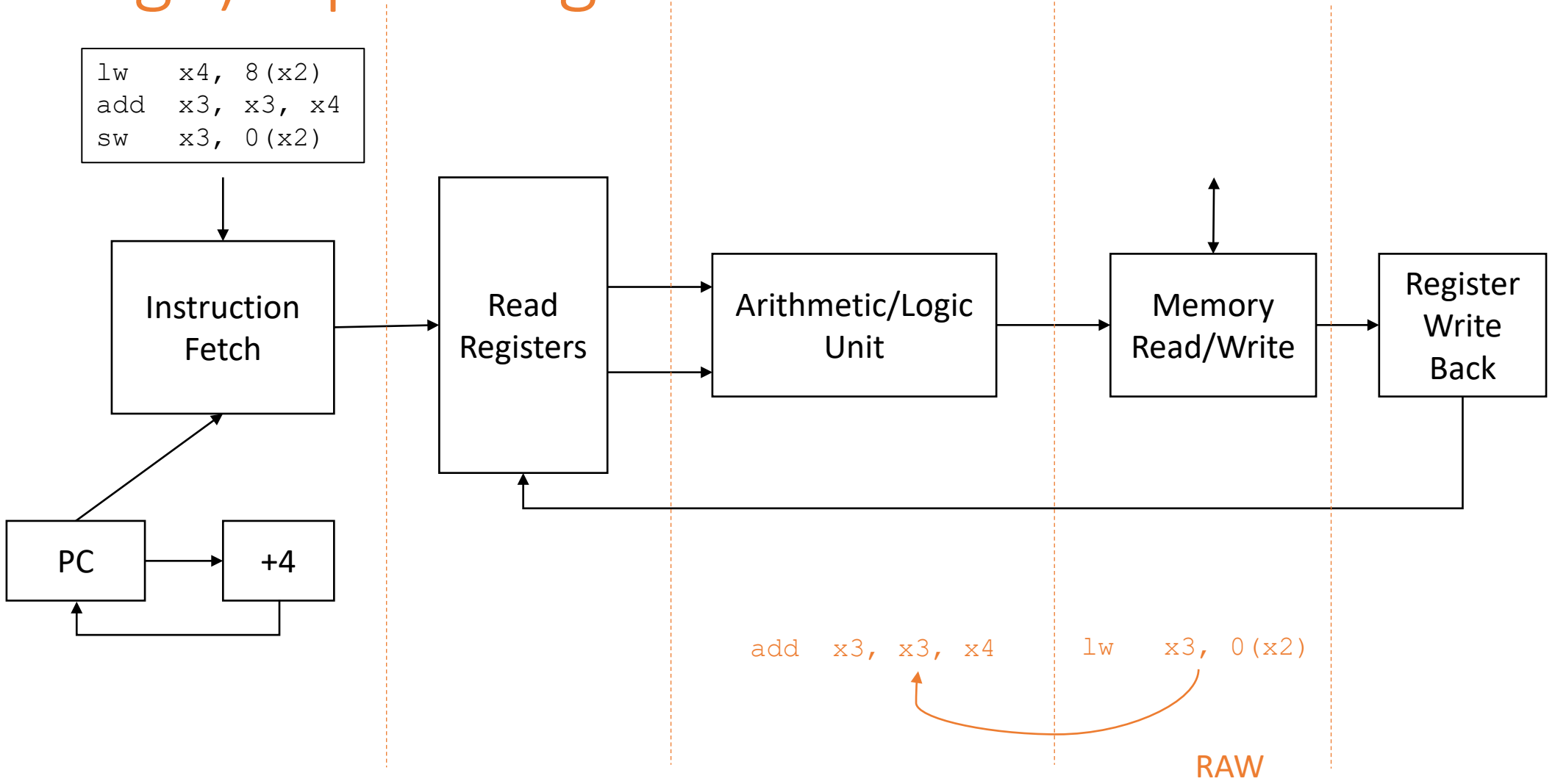
Today: Overview

- Control hazards
  - Speculation
- False dependences
  - Renaming
- Superscalars
  - More general parallelism

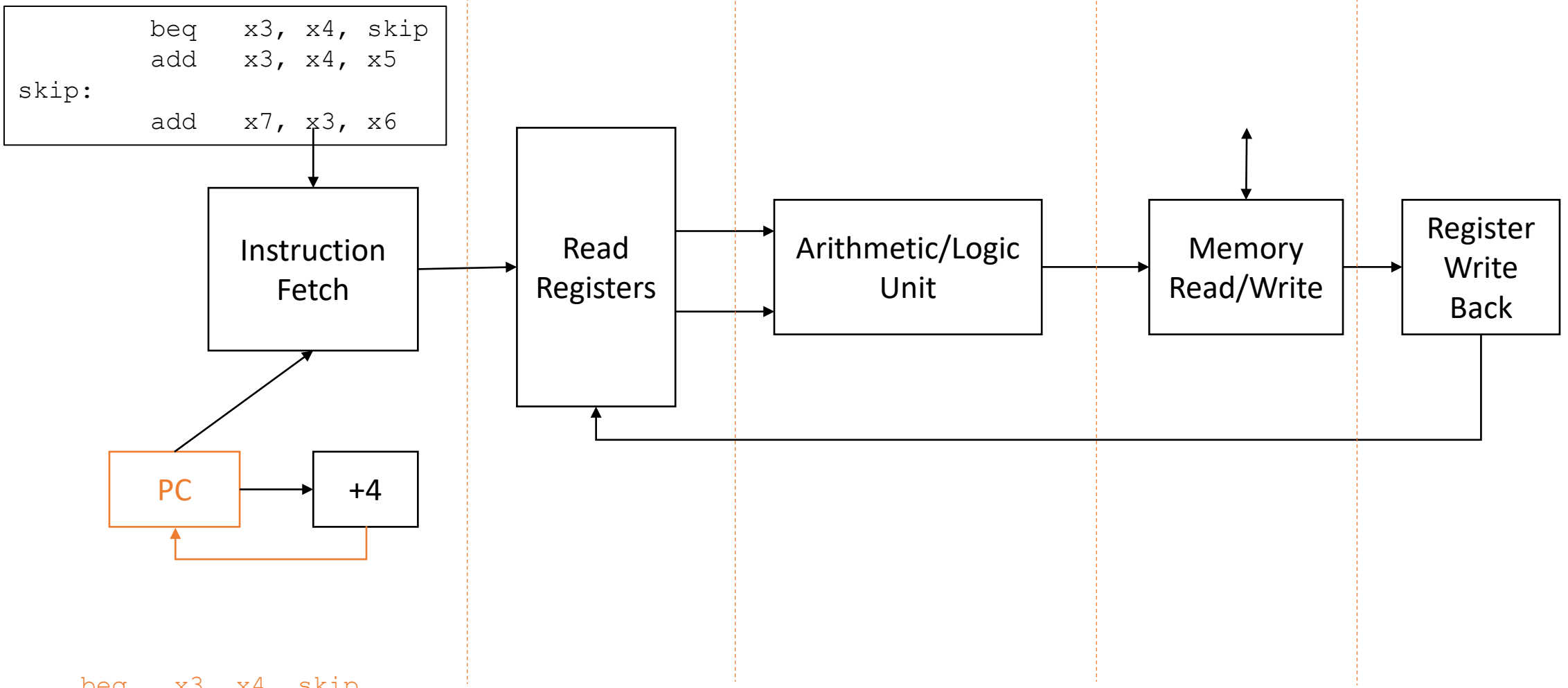
(5-stage) Pipelining: Data Hazard



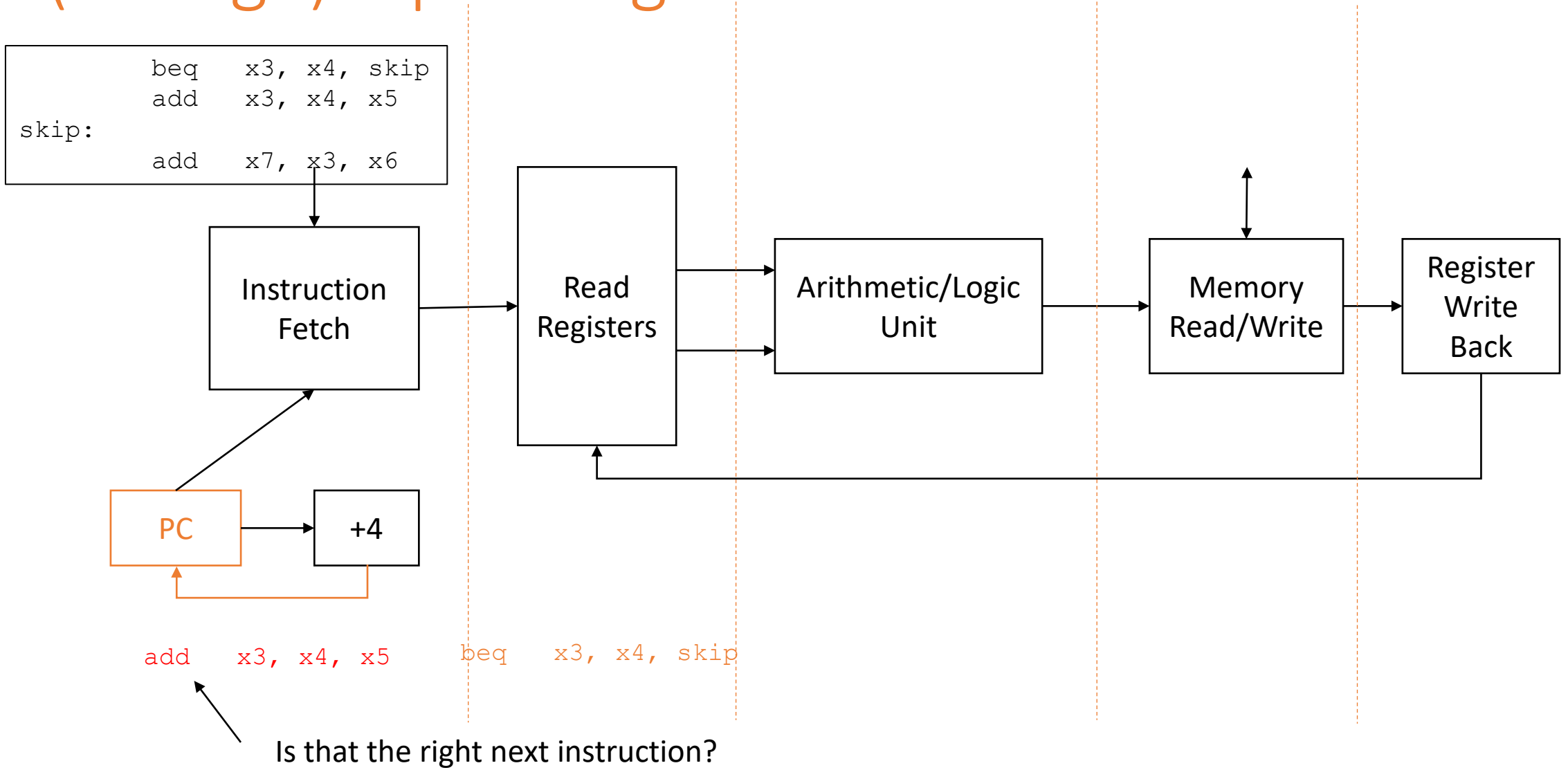
(5-stage) Pipelining: Data Hazard



(5-stage) Pipelining: Control Hazards



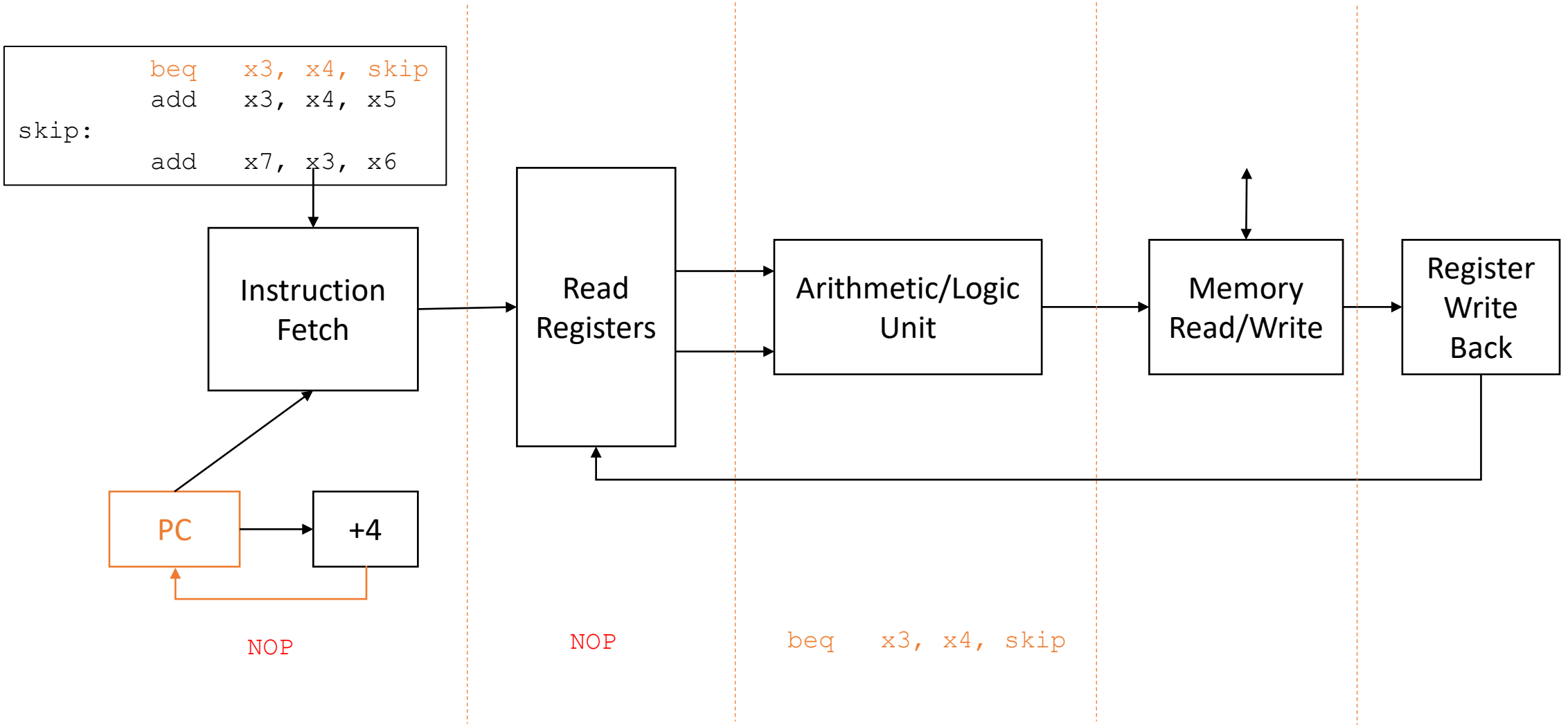
(5-stage) Pipelining: Control Hazards



Dealing with Control Hazards

- We can recognize that we have a branch in the cycle during which it is fetched, but...
- We won’t know whether it’s taken or not until the third cycle of its execution, and...
- We won’t have computed the target address until at least the second cycle of its execution
- What instruction should be fetched immediate after we have fetched the branch?
  - We don’t know!

Option 0: Pessimistic / Bubbles

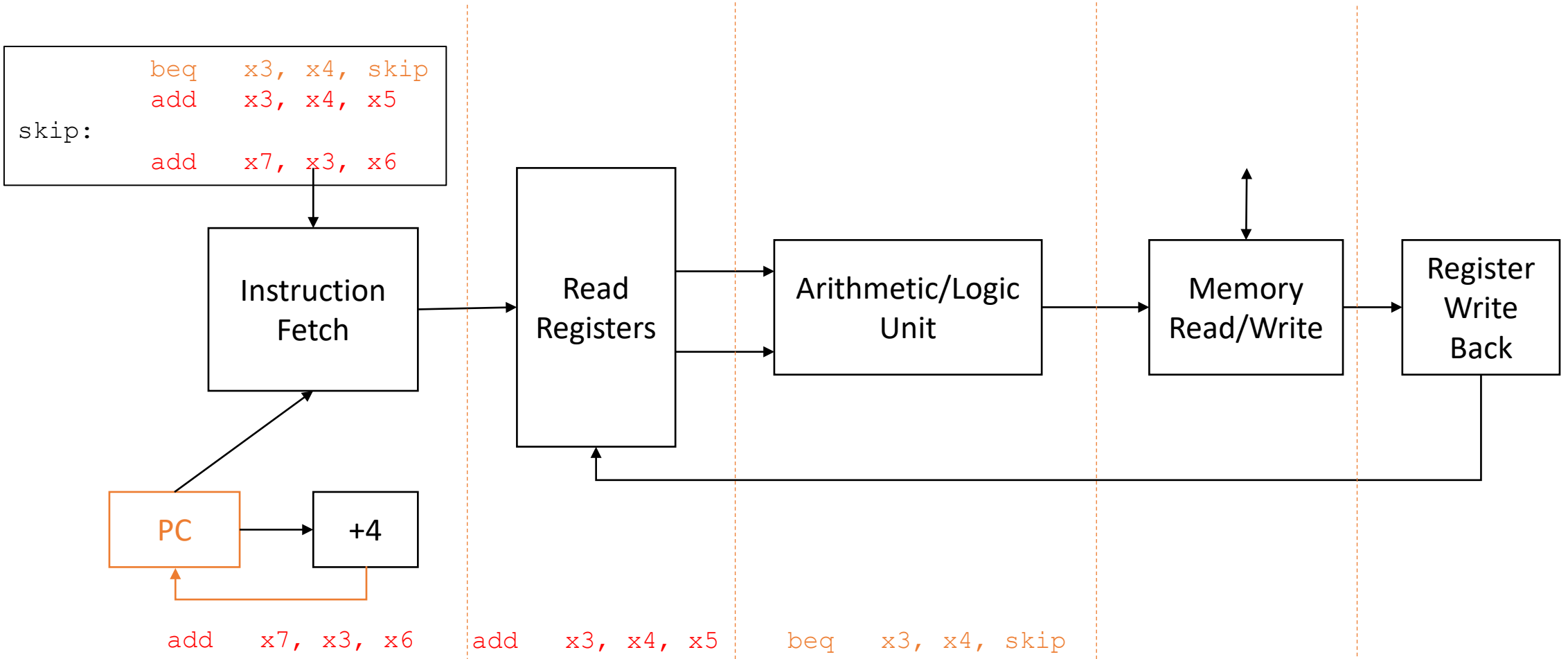


Con: Executing a branch always costs 3 instruction issue slots

Option 0+: Expose “branch delay slots” in ISA

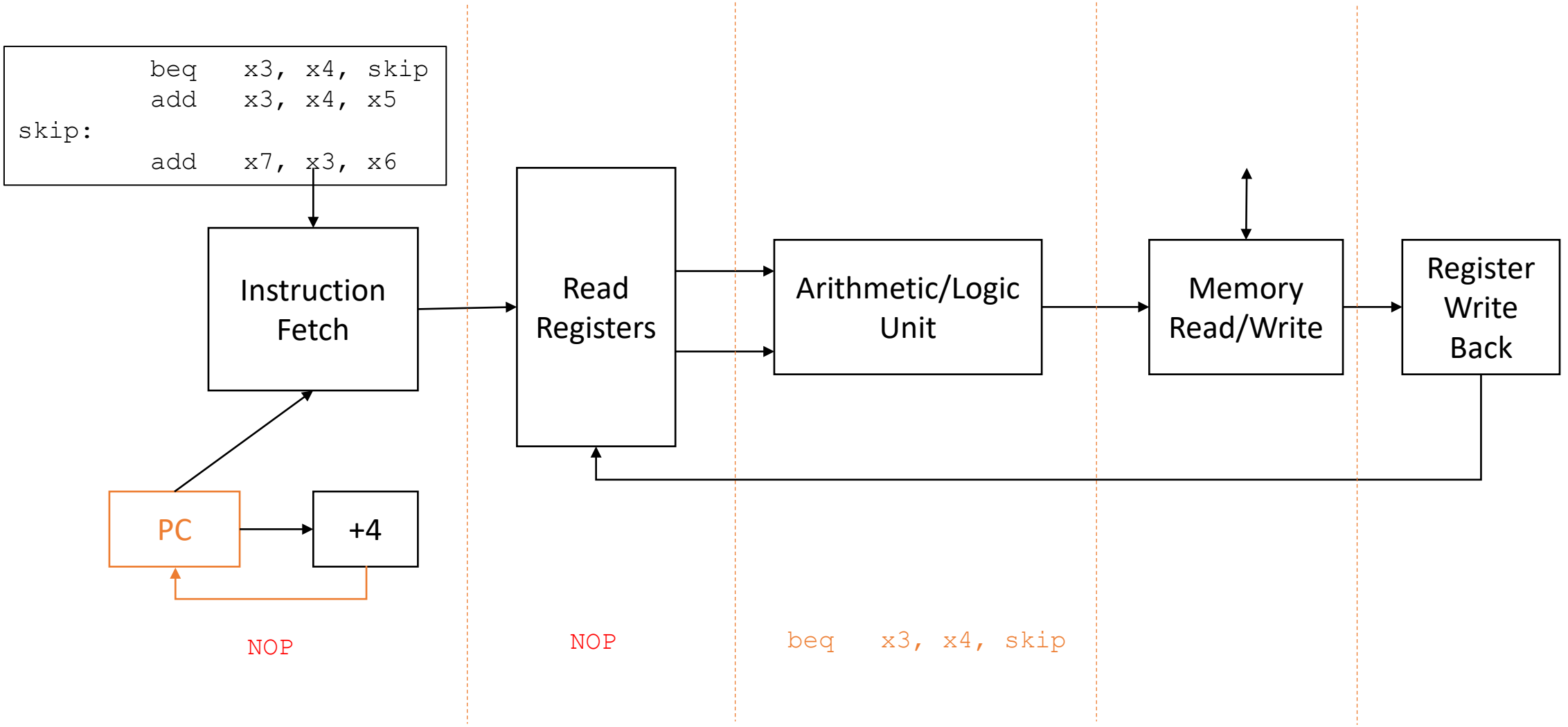
- This is a variant of the general scheme “make it the next layer up’s problem” (also used in operating systems)
- Compiler understands that the next two instructions after a branch are always executed
  - tries to find instructions that “naturally” would come before branch and move the move them to after branch
  - <some instruction>
beq ...
becomes
beq ...
<some instruction>
  - When is it legal for the compiler to do that?
  - If the compiler can’t find any instructions to fill the slots, it puts NOPs there

Option 1: Speculate #1: Assume Not Taken



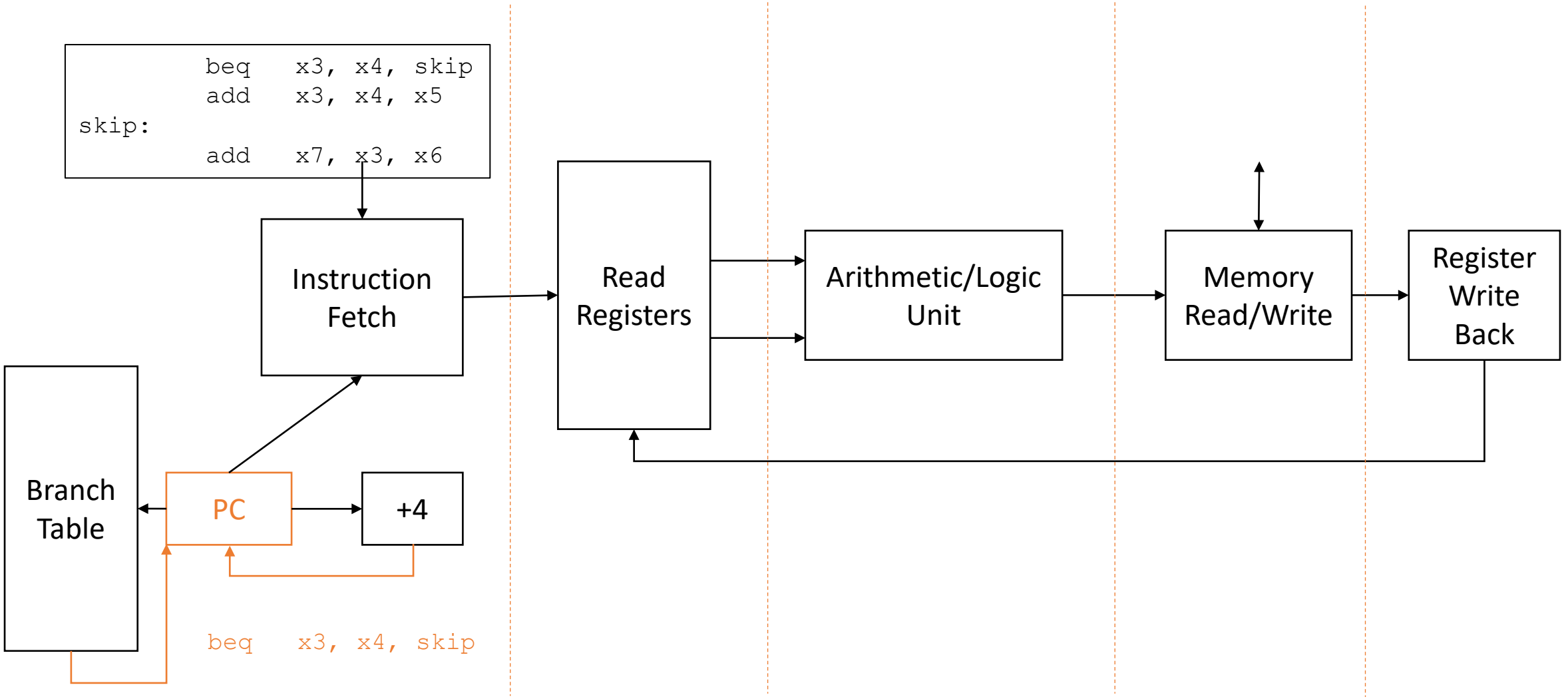
If we find out the branch is taken, we have to purge the two mis-fetched instructions

Speculate #2: Assume Taken



Don’t know the target address soon enough. This is bad...

Speculate #3: Branch Table Prediction



Branch Table Maintenance

PC	Next Inst Address
0x3EF0804	0x3EF0808
0x28808	0x421FC0
...	...
...	...

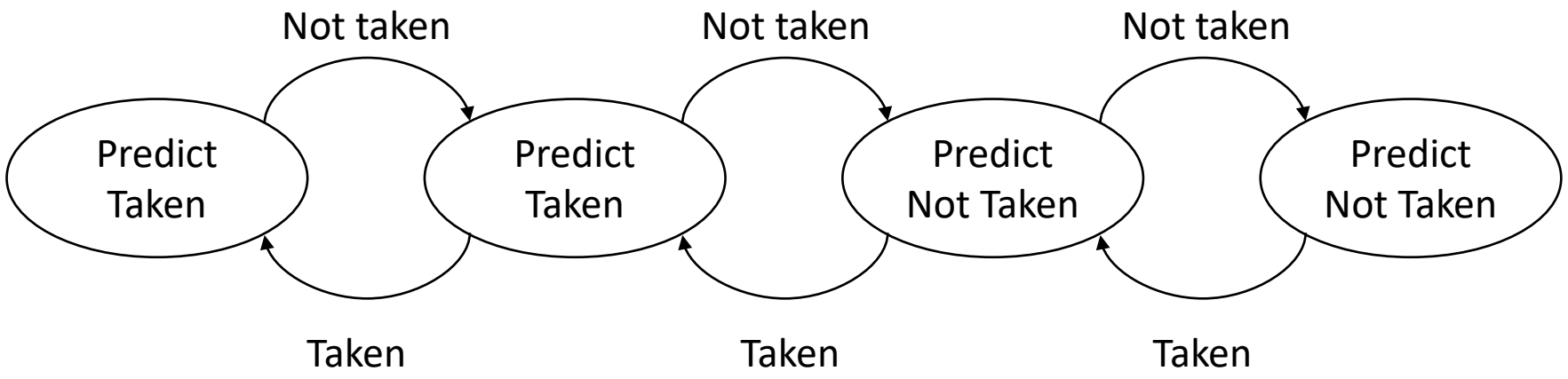
- Table is indexed by value of the PC
- When you fetch a branch, find a row for that branch
  - Enter its PC as the tag, if row doesn't currently exist
  - When you figure out what the actual next instruction is, fill in the next address column of that row
- As you fetch instructions, use the PC's value to look for a matching row in the table
  - If you find one, set the PC to the next inst address field
- That is, predict that the next time the branch is executed it will do the same thing it did the last time
  - "The future looks like the past"

Branches Due to Loops

- for (i=0; i<N; i++) { ...}
- There's a conditional branch at the bottom of the loop testing if i<N
- The loop is taken each time it's reached, except when i == N
- That last iteration causes the branch table to predict not taken on the first iteration the next time the loop is reached
- So, you get two mis-predictions each time the loop is executed
- Can you do better?
  - Sure. When you figure out the target address for a branch, predict taken if the branch is to lower memory addresses and not-taken if it's to higher addresses
    - The loop branch will always predict taken, which means there's only one mis-prediction per loop execution

Can We Do Even Better?

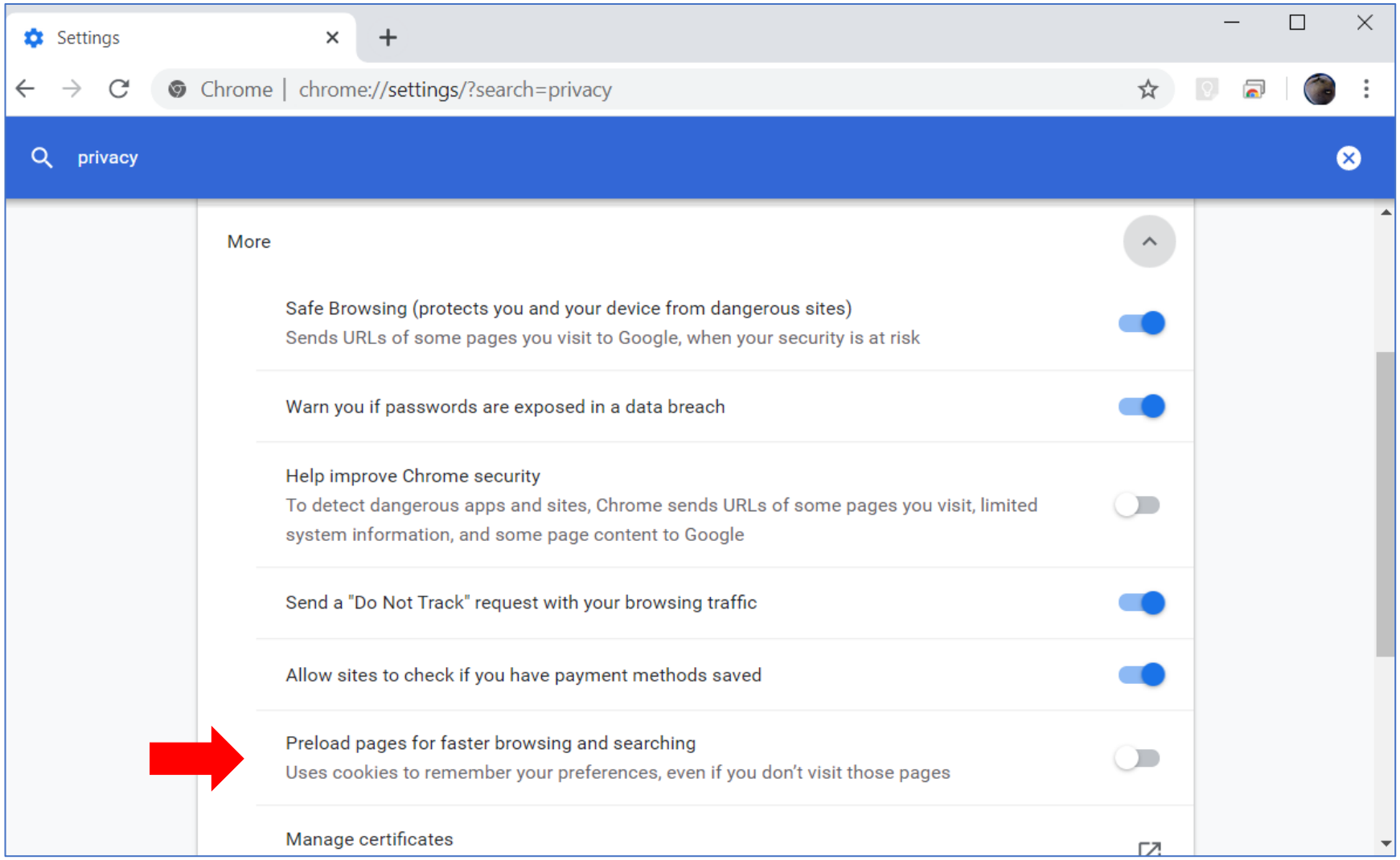
- Maybe
- Branches show up for conditionals as well as loops
  - If...then...else, for instance
- Use a scheme that remembers the past, but is willing to change its mind
  - Add two bits per prediction table row to keep track of state



Control Hazards Summary

- When you're trying to go fast, you can't afford to wait
  - Long latency operations:
    - Decide whether or not a branch will be taken
    - *Retrieve the contents of a web page*

Control Hazards Summary



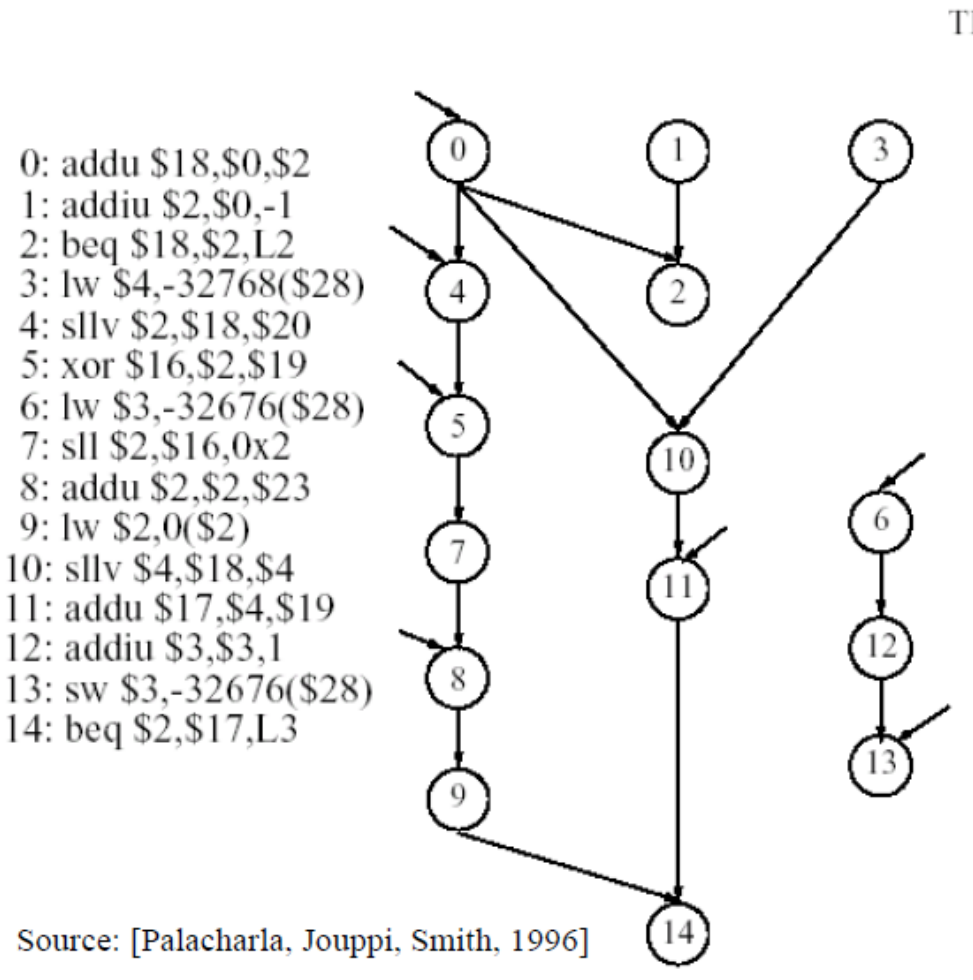
Moving Beyond Pipelines: Superscalars

- Pipelines have some important limitations
  - Only one instruction can be issued per cycle
    - Because of hazards, number of instructions completed per cycle will be less than one
  - A dependence that stalls one instruction necessarily stalls all instructions after it, even if they have no dependences themselves
    - This could be addressed to some extent by the compiler, assuming it understood the implementation of the datapath
  - Even if I add more hardware (e.g., more ALUs), I can’t use them

Superscalars

- Some of the following slides are from https://ece752.ece.wisc.edu/lect05-superscalar-org.pdf

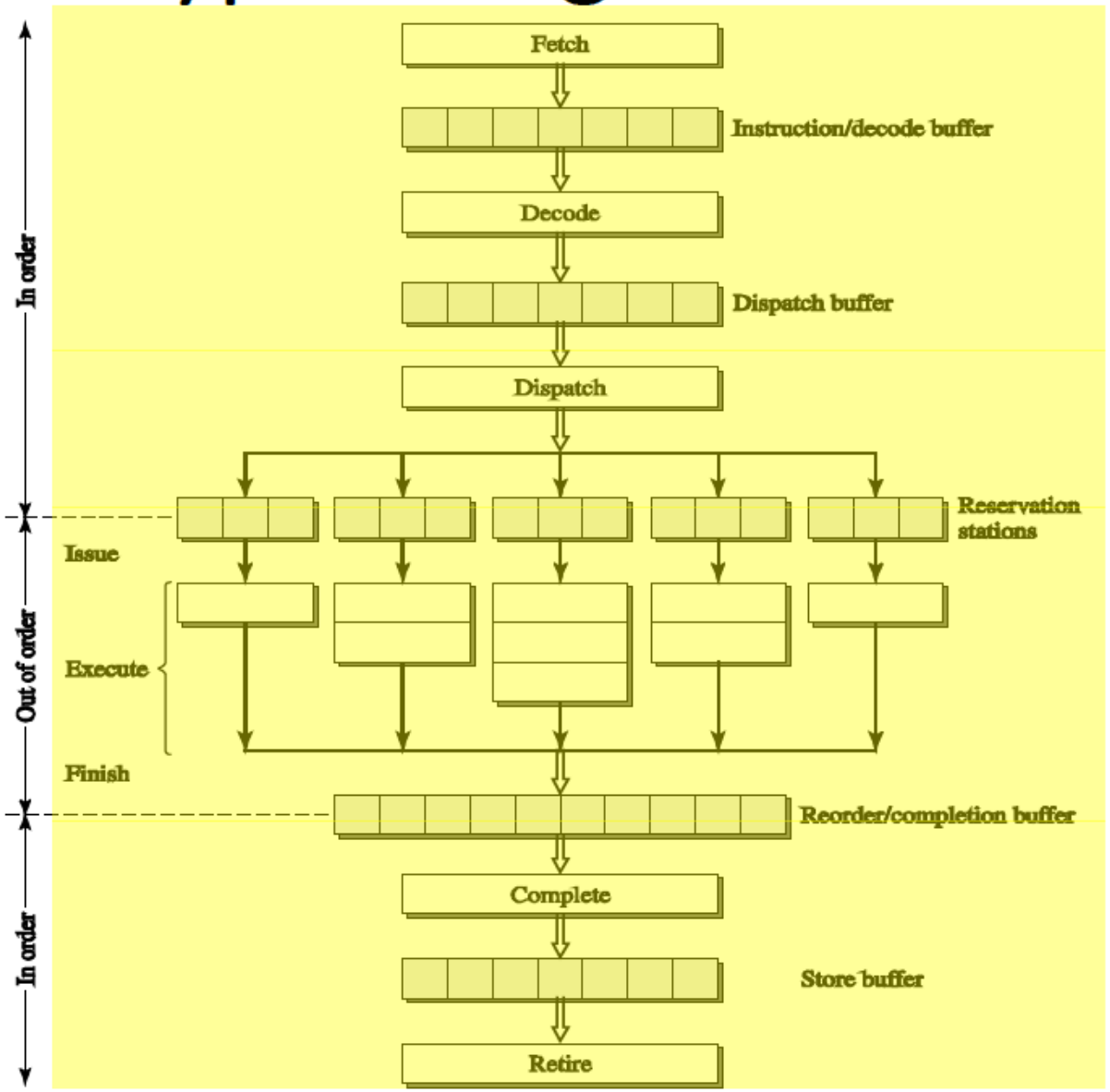
What Does a High-IPC CPU Do?



Source: [Palacharla, Jouppi, Smith, 1996]

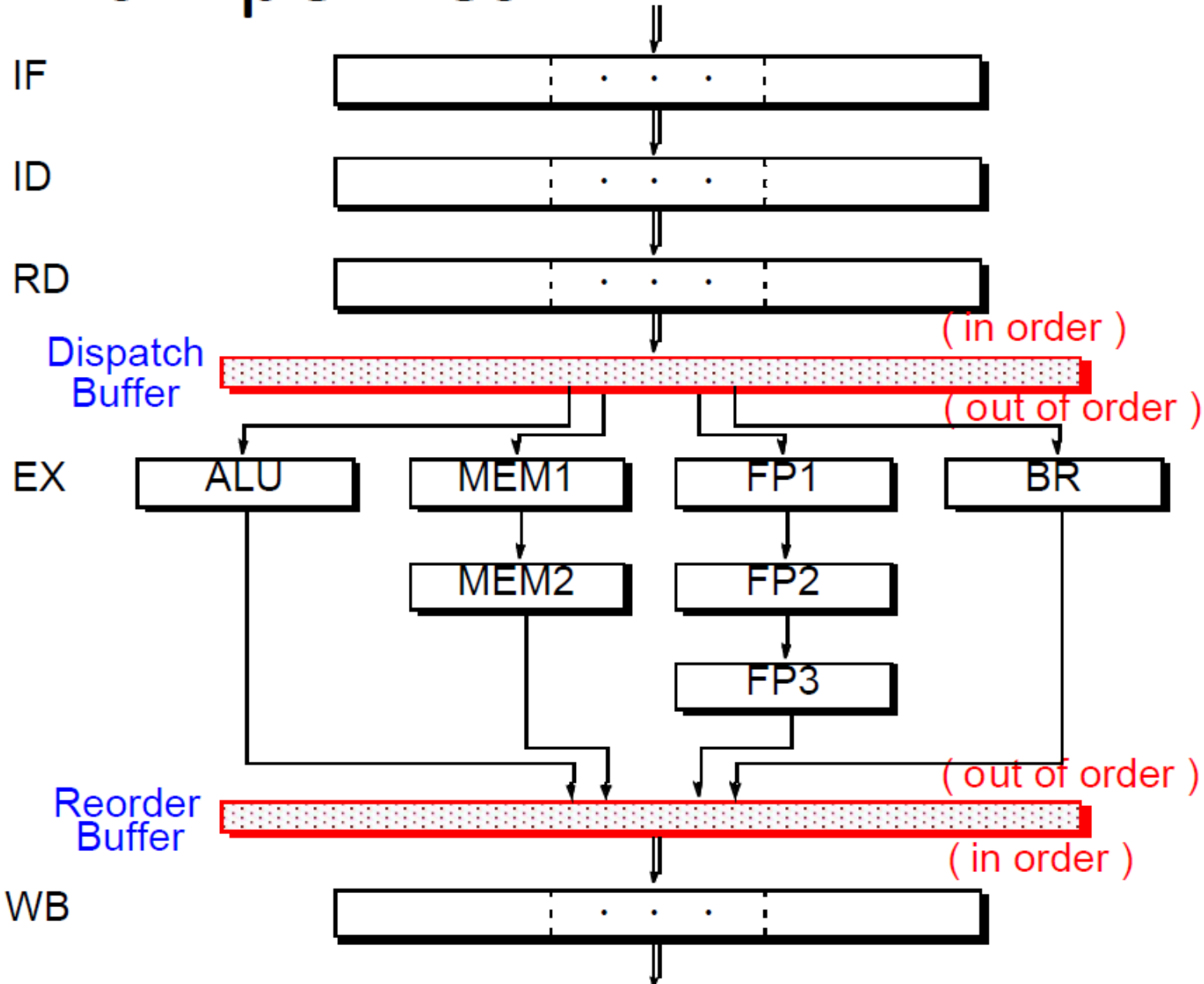
1. Fetch and decode
2. Construct data dependence graph (DDG)
3. Evaluate DDG
4. Commit changes to program state

A Typical High-IPC Processor



1. Fetch and Decode
2. Construct DDG
3. Evaluate DDG
4. Commit results

Dynamic Pipelines



Constructing the Dependence Graph

- The dependences are RAW, WAR, and WAW
- RAW is a “true dependence”
- WAR and WAW are “false dependences”
  - They’re false because they’re conflicts on names, not on values
- WAW:
  - add x3, x2, x1 … add x3, x4, x5 vs add x3, x2, x1 … add x50, x4, x5
- The compiler produces conflicts on names because the ISA has only so many registers (e.g., 16 or 32)
- The hardware implementation can have many more registers
  - The hardware does “register renaming” as it fetches instructions to break false dependences

Register Renaming

- Instead of thinking of this code as naming registers, think of it as naming values

```
   add x3, x2,  x1                add <value3>, x2, x1
...                       vs.
   add x3, x4, x5                 add <value6>, x2, x1
```

- These “value names” identify the true depences (RAW)
- The hardware is free to map the value names to whatever physical registers it wants

Register Renaming

- When you assign a name (one of the hardware registers) to a value, you have to propagate it to future instructions
- Hardware keeps a table telling it which value name (hardware register) currently represents each architectural register name
- add x3, x2, x1 becomes add 51, <x2>, <x1>
  ... ...
  add x2, x3, x4 add 43, 51, <x4>
  add x3, x7, x2 add 48, <x7>, 43

  Hardware maintains a table with a row that says “x3 is 51” from the time it sees the first instruction until the time is sees the name “x3” refer to a different value

Register Renaming Summary

- WAW and WAR dependences can be eliminated by renaming
- That's true for the CPU executing instructions that modify registers
- That's true for software (including the OS) that updates variables/data structures
  - If multiple threads share adata structure, the code likely needs to explicitly synchronize their execution
    - Synchronization is an overhead, analogous to bubbles in the pipeline
- We "rename" by taking a centralized shared data structure and replacing it with many more similar data structures
  - Often, a "private" data structure per thread and synchronization code that manages the private instances and makes them act as a single instance