CSE 451: Operating Systems
Spring 2020

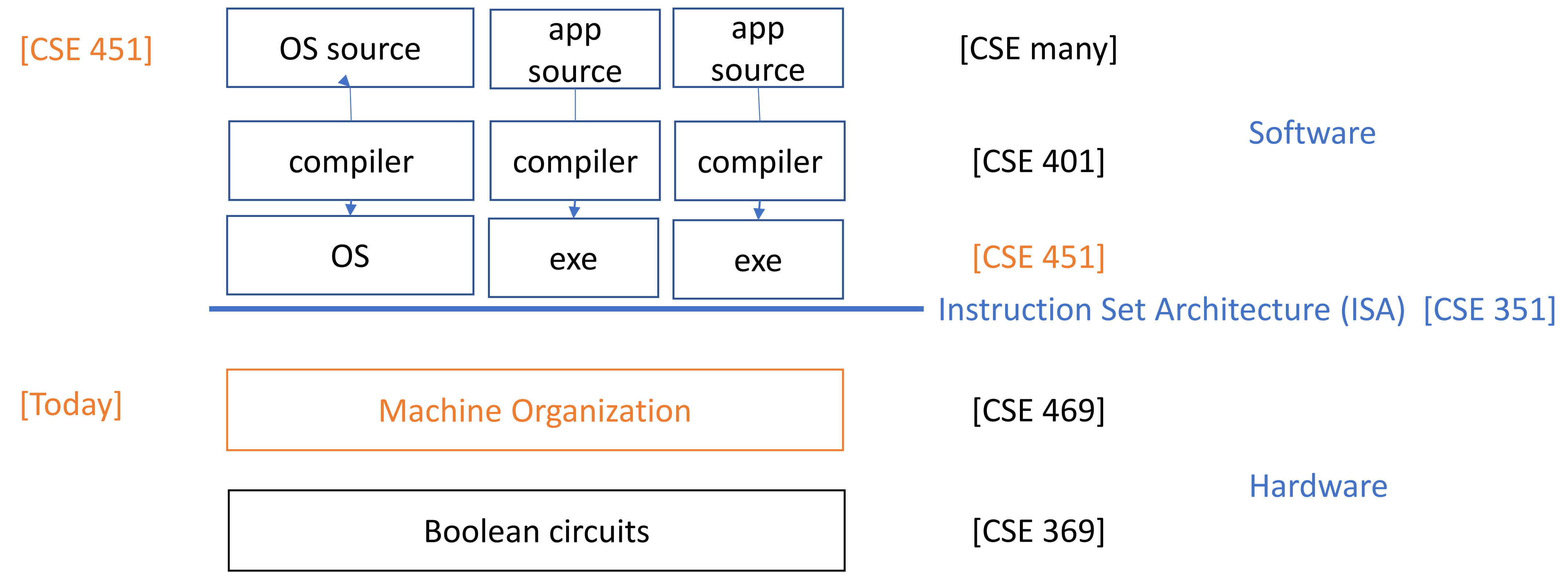
Module -1

(Instruction Level) Parallelism

Today: Overview

- Nothing we’re doing will directly be part of anything graded
    - I’m mainly hoping it’s interesting ...
    - and vaguely relevant to some concepts/mechanism we’ll see later...
    - and while we’ll be mainly talking about specifics, some part of the material is very general

Today: Overview



Terminology

- Parallelism
  - Doing more than one thing at a time
    - Usually for performance reasons
    - Multiple hardware resources in use at once
- Concurrency
  - Doing more than one than thing during the same time period
    - Can be for performance
      - Avoid synchronization
      - Increase utilization of (expensive) hardware resources
    - Often to simplify software implementation
    - May or may not involve multiple hardware resources

A Ridiculously Simplified History of System Software

- Era 0: There’s one compute in the world, it’s in one location, and it’s running one program at a time (or maybe just one program, period).
- 1: They’re handy, so more computers are built. Each runs one program at a time. They’re incredibly expensive, so there are still only a small number of them in the world.
- 2: Computers have CPUs and I/O devices. Because they’re expensive, people want to use both at once. Individual programs try to parallelize their I/O and computation.
- 3: It’s hard to parallelize individual apps. OS’s are written that overlap loading into memory the next program to run with running the current program.

A Ridiculously Simplified History of Computing

- 4: It’s hard to parallelize individual apps. OS’s allow many programs to be loaded and run concurrently, so that computation of one can be overlapped with I/O of another. This allows programs to be written using sequential semantics (each run sequentially, not in parallel), but the system achieves parallel execution (if you think of all the programs together as one execution)
- 5: If you want an individual program to run faster, the programmer has to write a parallel or concurrent program.
- 6: Parallelizing compilers, multicore processors, networks and distributed systems, the cloud...

What Was Happening With Hardware?

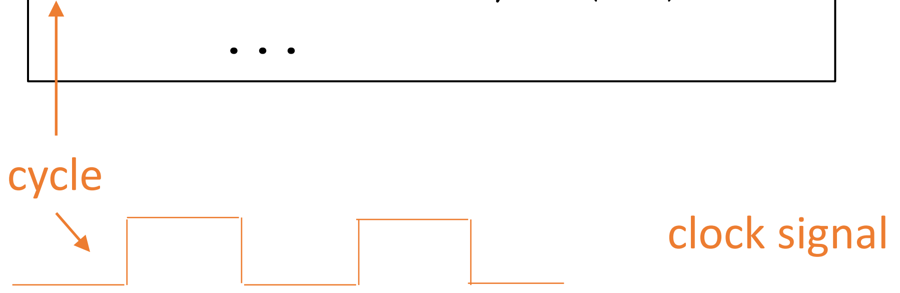
- Era 0: Sequential execution of hardware instructions

Program Source

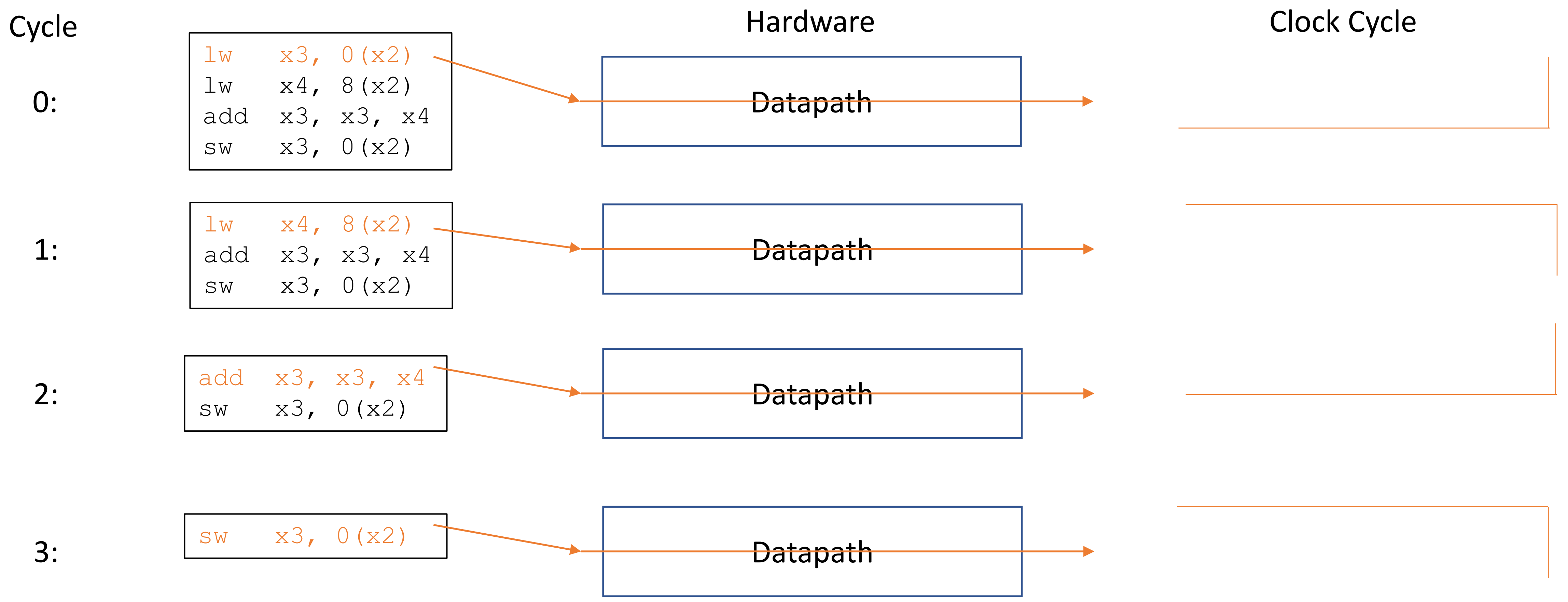
```
        lw          x3, 0(x2)
        lw          x4, 8(x2)
        add         x3, x3, x4
        sw          x3, 0(x2)
        ...
```

Program Execution

```
0:         lw       x3, 0(x2)
1:         lw       x4, 8(x2)
2:         add      x3, x3, x4
3:         sw       x3, 0(x2)
          ...
```



Single Cycle Execution



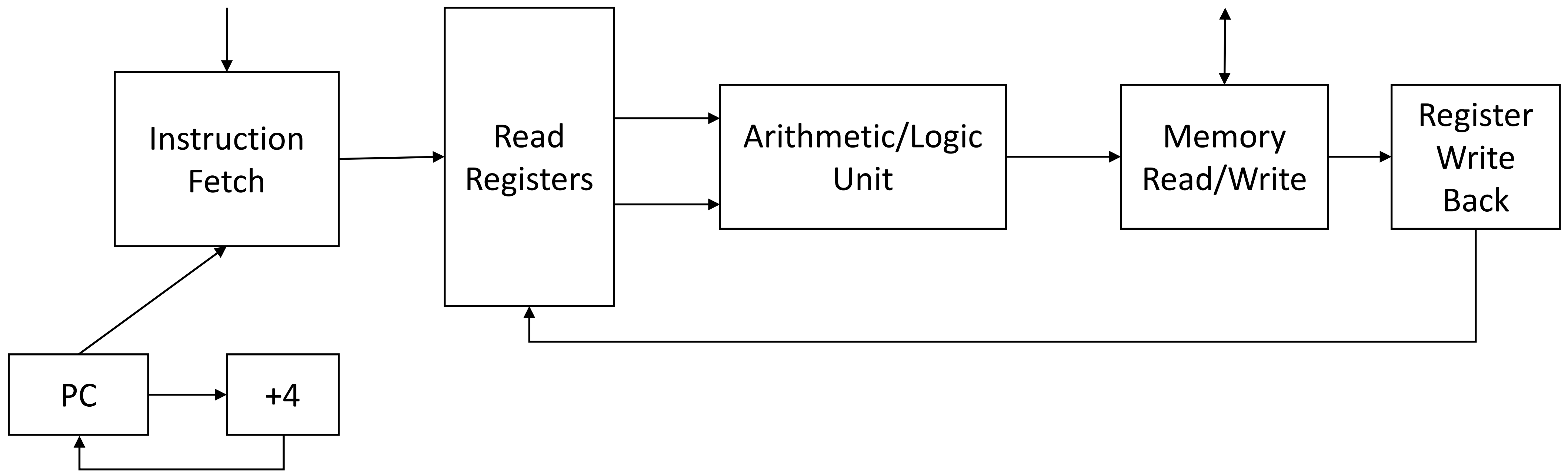
```
lw    x3, 0(x2)
lw    x4, 8(x2)
add   x3, x3, x4
sw    x3, 0(x2)
```

```
lw    x4, 8(x2)
add   x3, x3, x4
sw    x3, 0(x2)
```

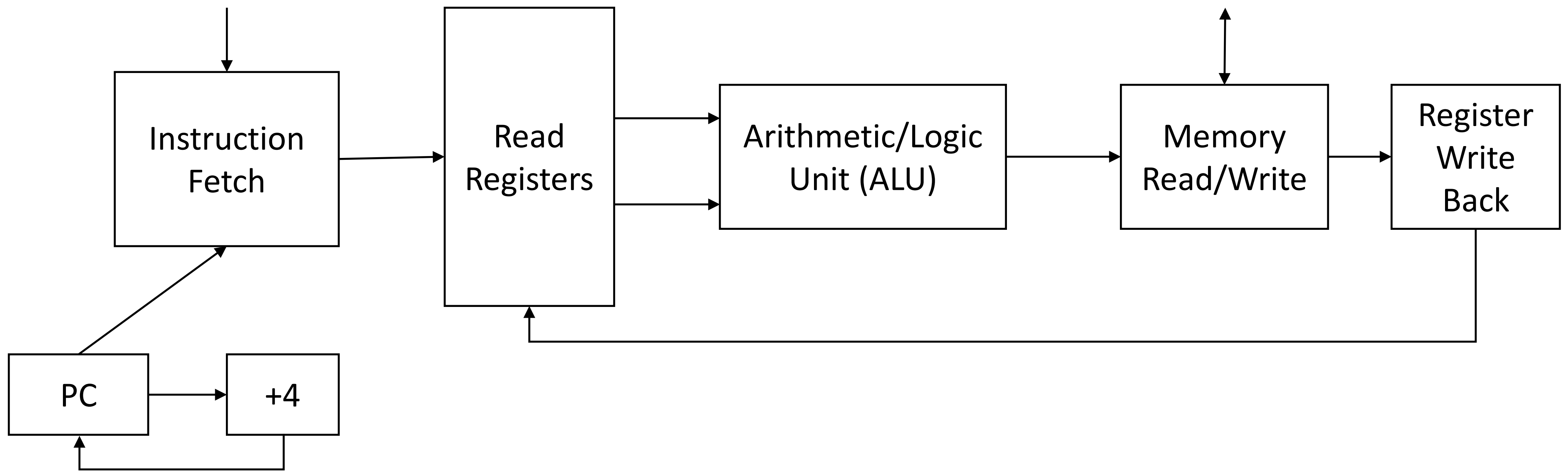
```
add   x3, x3, x4
sw    x3, 0(x2)
```

```
sw    x3, 0(x2)
```

Single Cycle Datapath (RISC)



```
add    x3, x3, x4
```



Fetch instruction

Get contents of x3 and x4

Add contents of x3 and x4

Pass through add result

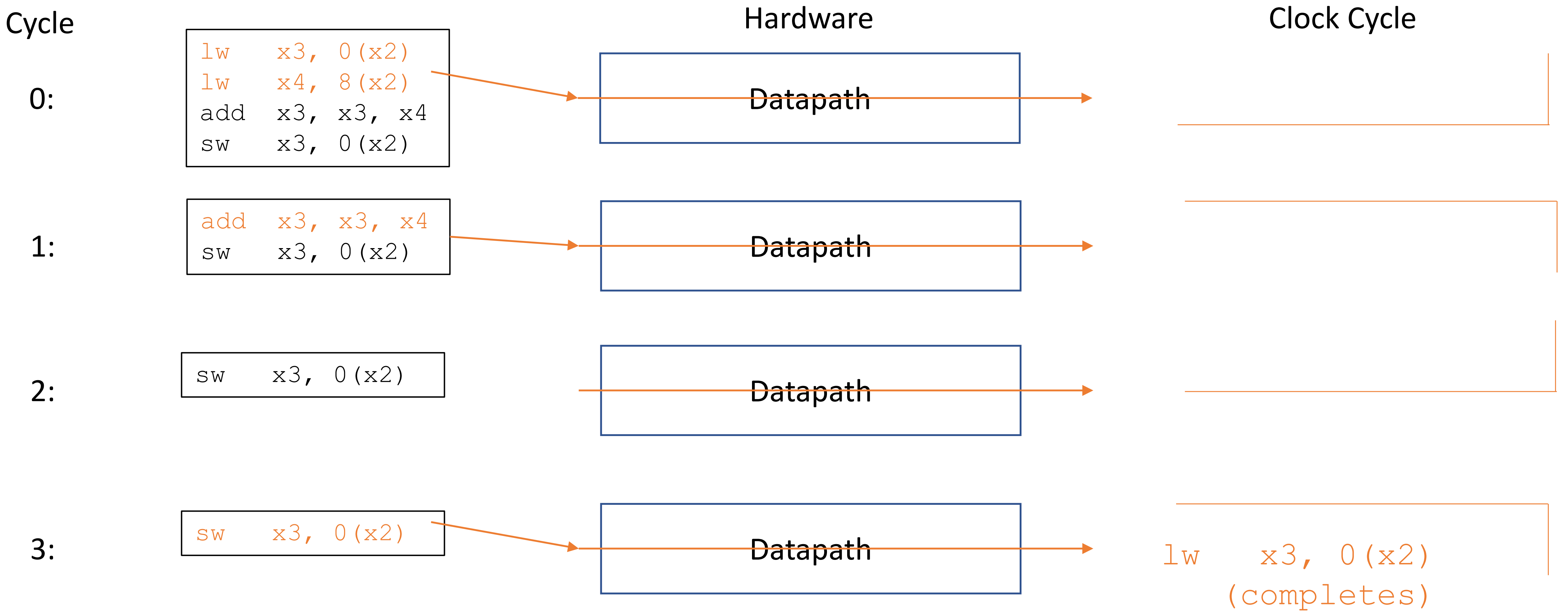
Write to x3

Making the CPU faster

- Shorten the clock cycle
  - Basically, keep the circuit you have but make it operate faster
    - make the “gates” (basic components of the circuit) smaller, and/or
    - increase the voltage
      - increase the heat...
- Execute more than one instruction at a time
  - ILP – instruction level parallelism
  - IPC – average number of instructions completed per cycle

IPL – Artist's Rendition

(Pretty Much No Detail Here is Accurate!)



Cycle 0:

```
lw   x3, 0(x2)
lw   x4, 8(x2)
add  x3, x3, x4
sw   x3, 0(x2)
```

Cycle 1:

```
add  x3, x3, x4
sw   x3, 0(x2)
```

Cycle 2:

```
sw   x3, 0(x2)
```

Cycle 3:

```
sw   x3, 0(x2)
```

```
lw   x3, 0(x2)
  (completes)
```

How Can We Implement ILP?

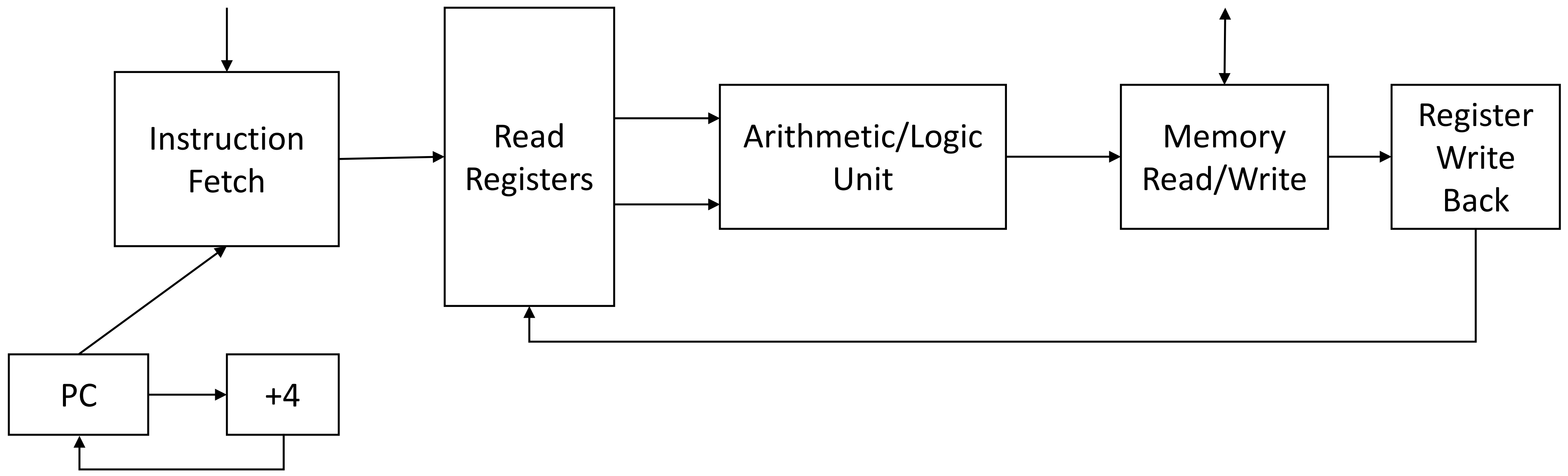
- The simplest approach is pipelining

vs.

Not Pipelining

```
lw   x4, 8(x2)
add  x3, x3, x4
sw   x3, 0(x2)
```

Cycle time

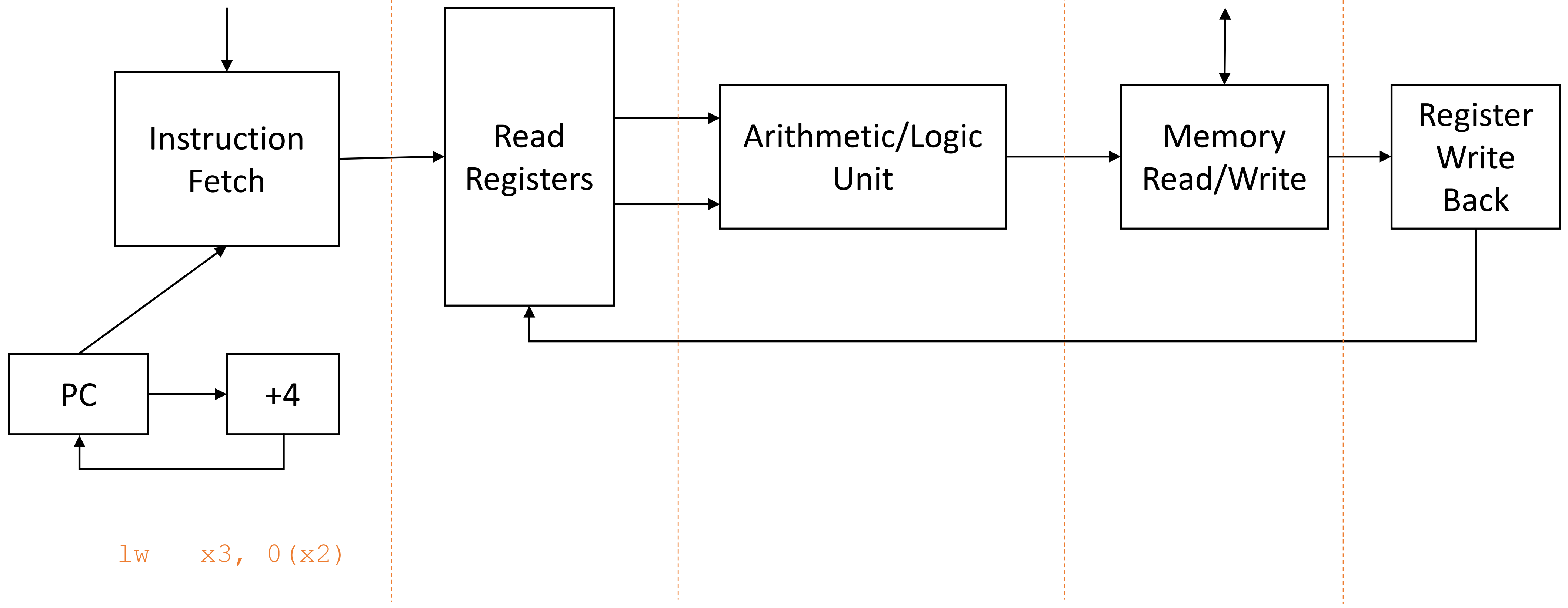


```
lw   x3, 0(x2)
```

(5-stage) Pipelining

Cycle time

```
lw    x4, 8(x2)
add   x3, x3, x4
sw    x3, 0(x2)
```

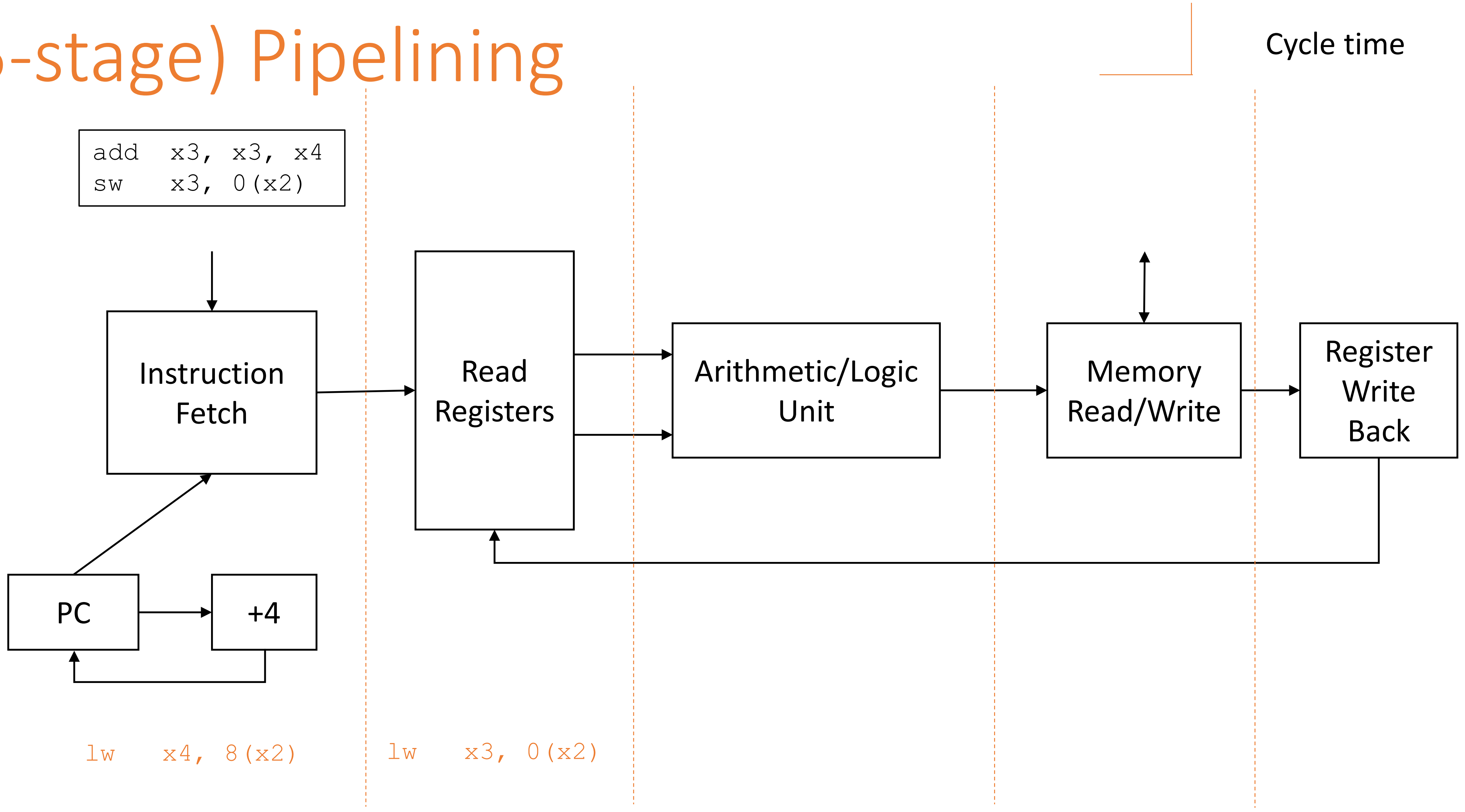


```
lw    x3, 0(x2)
```

(5-stage) Pipelining

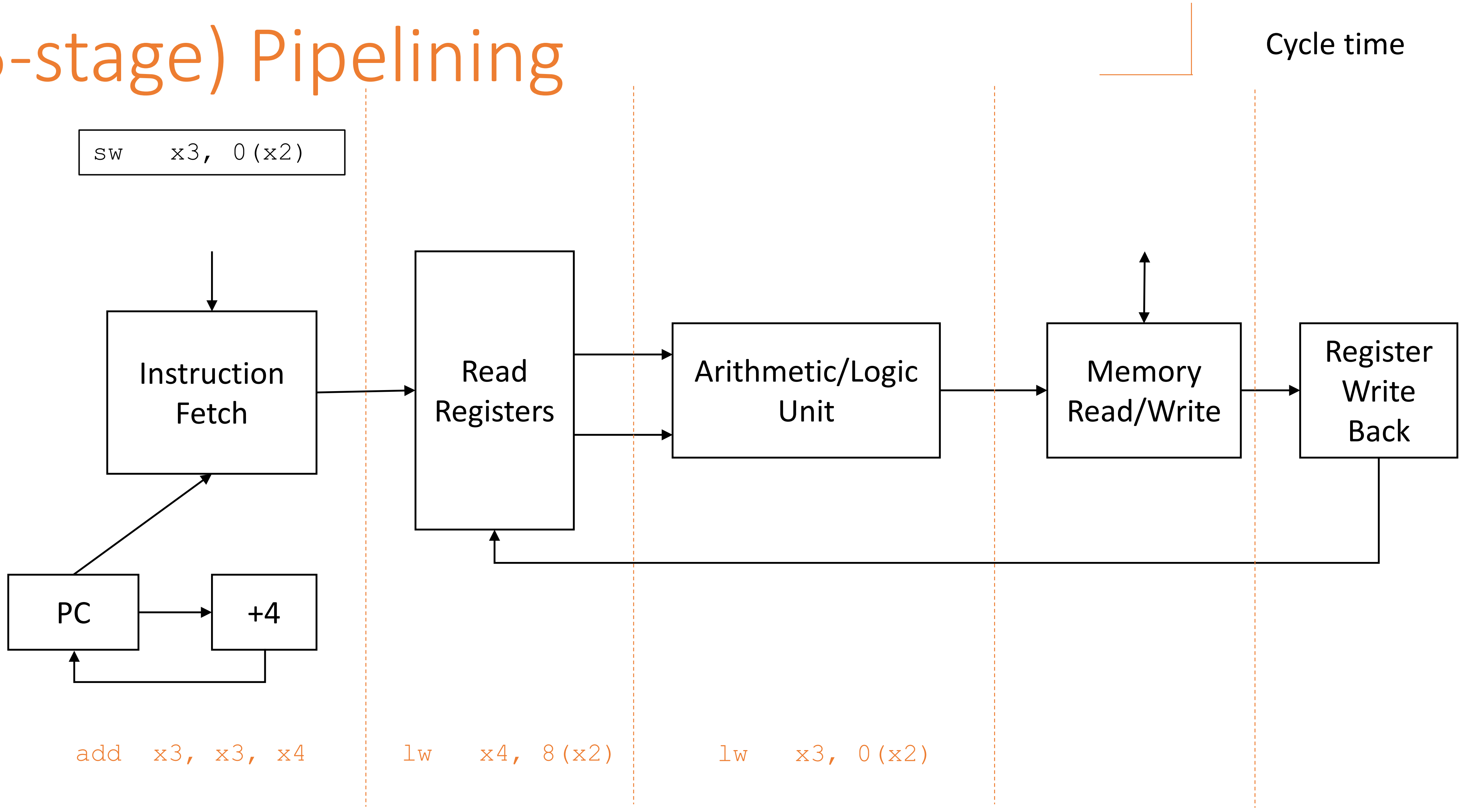
Cycle time

```
add  x3, x3, x4
sw   x3, 0(x2)
```



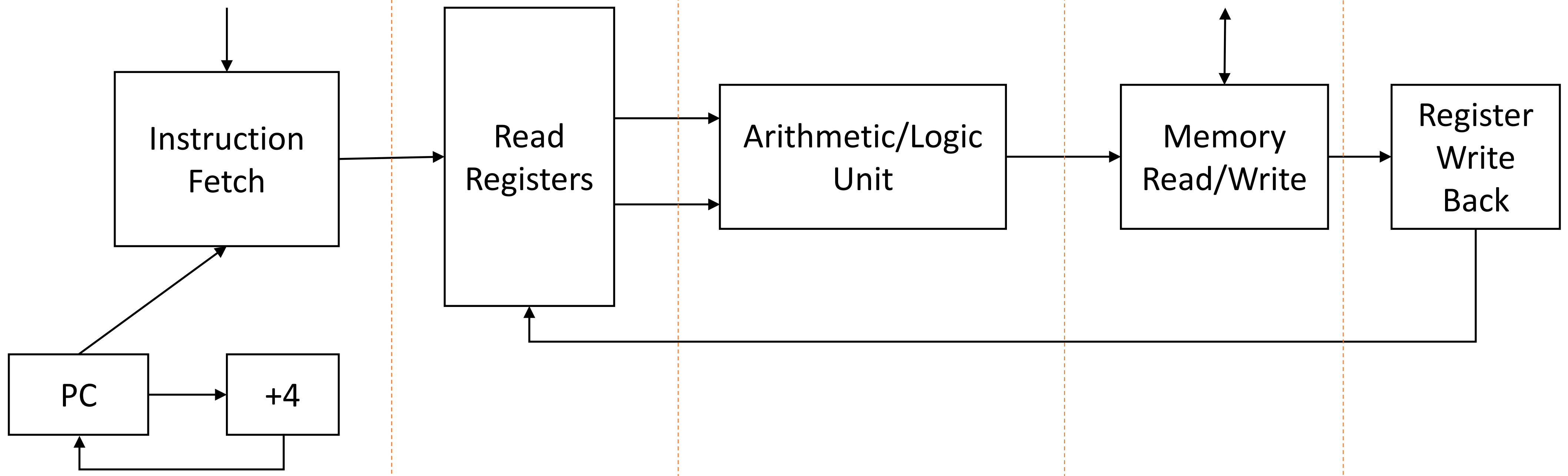
```
lw   x4, 8(x2)        lw   x3, 0(x2)
```

(5-stage) Pipelining



(5-stage) Pipelining

Cycle time



`sw x3, 0(x2)` `add x3, x3, x4` `lw x4, 8(x2)` `lw x3, 0(x2)`

Pipelining

- Idealization
  - Reduce cycle time (compared to single cycle implementation) by a factor equal to the number of stages
  - Still inject one instruction per cycle into the data path
  - Ideally, one instruction per cycle leaves the data path once it fills
- Why is this an idealization, and not reality?
  - Probably can’t divide data path into exactly equal-time stages
  - Hazards

Pipelining Constraints

- Pipelining is a change to the *implementation* of the ISA
- The ISA itself remains unchanged
- The ISA defines the effect of an instruction sequence to be what results from a single cycle implementation – executing each instruction to completion before beginning the next one
- The pipeline executes multiple instructions at once
- But you can’t just execute the instructions in any old order and expect to get the right result
- Which orderings are correct, and which incorrect?

Dependences - RAW

- Dependences restrict the correct orderings
- There are three kinds
- 1) RAW – read-after-write (aka data or flow or true dependence)

```
    lw      x3,  0(x2)
    add     x3,  x3,  x4
```

  - The `add` instruction should read the value produce by the `lw` instruction
  - When the instructions are executed one at a time, no problem
  - When the instructions are executed at the same time, in the general case the `add` may read x3 before the `lw` writes it

Dependences - WAR

- 2) WAR – write-after-read (aka anti-dependence)

```
add    x3, x3, x4
lw     x3, 0(x2)
```

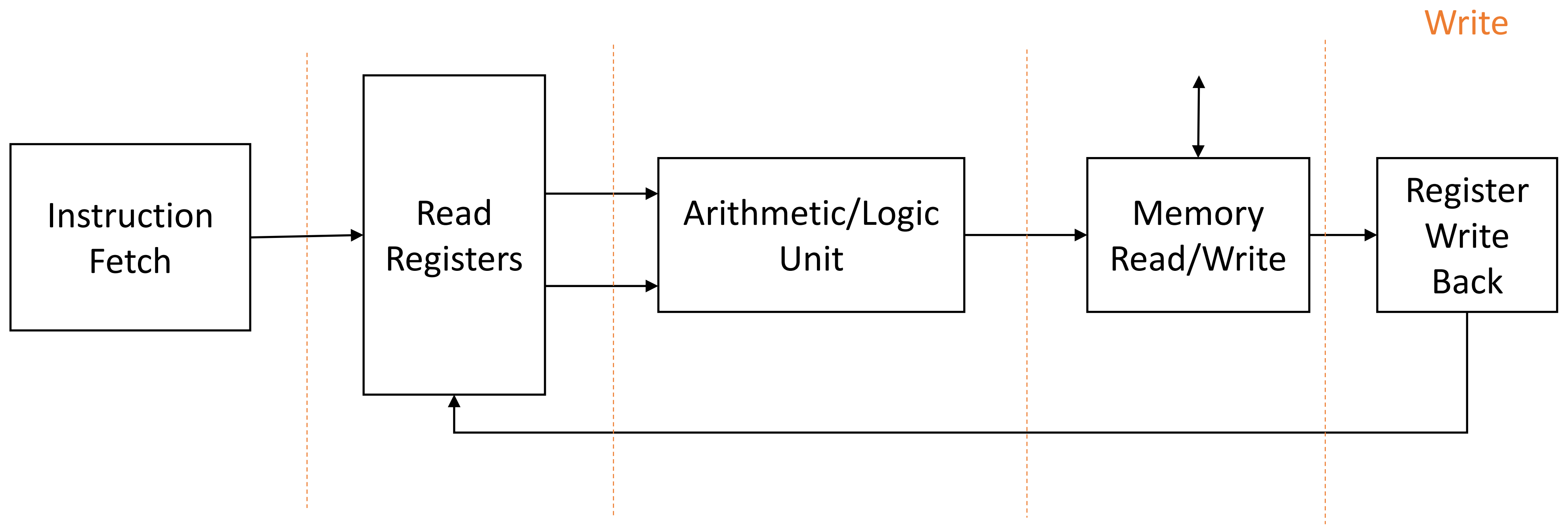
  - The `add` instruction should read the value of x3 before the `lw` instruction writes it

- 2) WAW – write-after-write (aka output dependence)

```
add    x3, x3, x4
lw     x3, 0(x2)
```

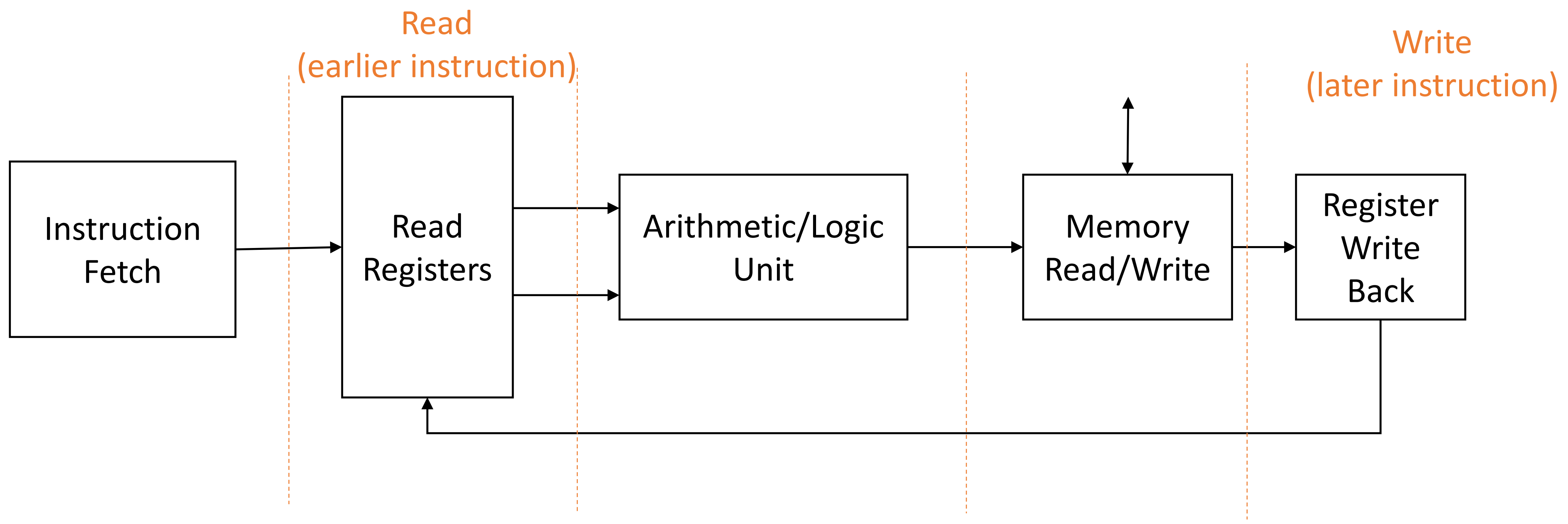
  - The value left in x3 should be the one written by the `lw`

(Register) Dependences and Pipelining: WAW



- Register writes occur in the final stage (write-back). Instruction pass through that stage in execution order.
- So, the pipeline naturally respects WAW ordering contraints

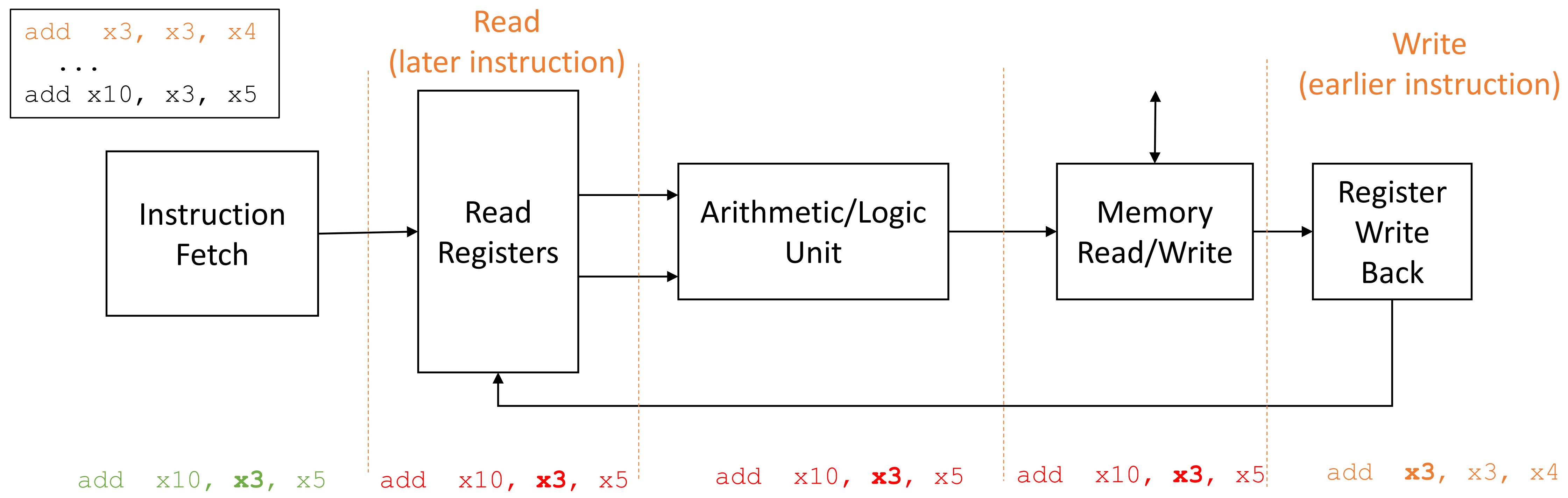
(Register) Dependences and Pipelining: WAR



- The first instruction reads during its second stage
- The instruction that writes trails it in the pipeline and doesn’t write until it reaches stage 5
- So, WAR dependences are always respected by the pipeline

(Register) Dependences and Pipelining: RAW

```
add  x3, x3, x4
  ...
add x10, x3, x5
```



- Trouble / No Trouble
- We call the trouble a *hazard* – we can get the wrong result if we don't do something special
- If the reading instruction is one of the next three instructions after the writing instruction, it will read an incorrect value
  - It gets the value that was in the register before the writing instruction executed

Resolving the Hazard

- There are options...
- 1) Insert bubbles statically
  - Make it the programmer’s (compiler’s) problem
  - The ISA says “don’t do that!”
    - If your code has an instruction that writes a register and any of the next three instructions read the register, the reading instruction gets the old value
  - Compiler must insert “bubbles” to separate producer and consumer of value, if it can’t find anything more useful to do
      - A bubble is a NOP (an instruction that doesn’t change any values)

Resolving the RAW Hazard

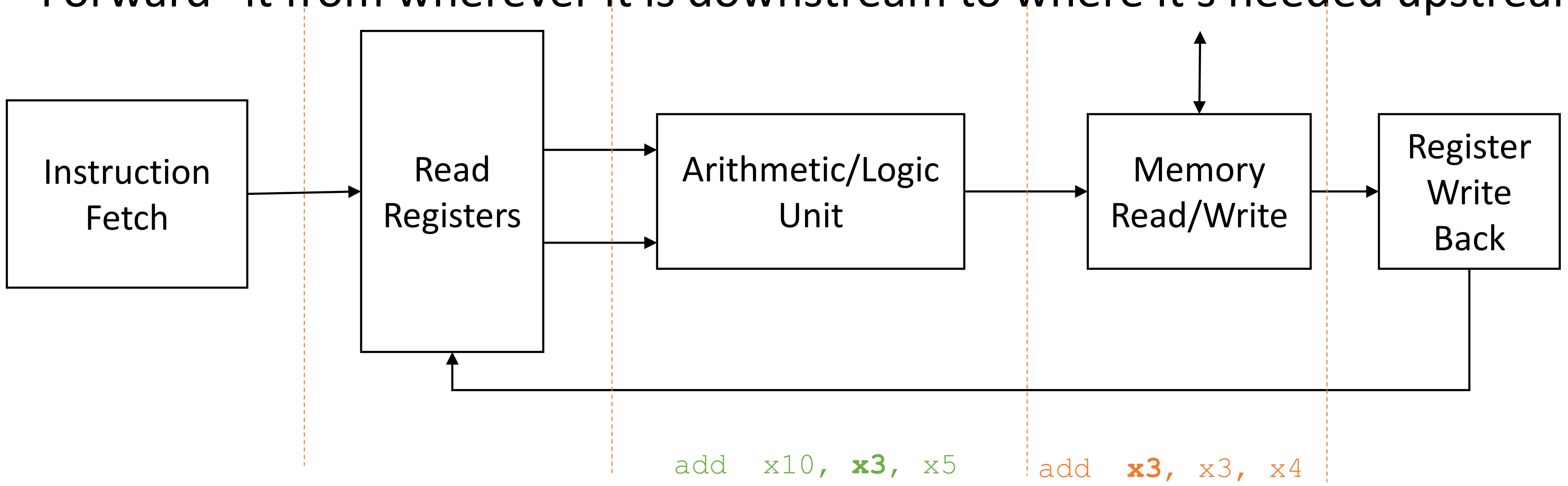
- There are options...
- 1) Insert bubbles statically
  - Make it the programmer’s (compiler’s) problem
  - The ISA says “don’t do that!”
    - If your code has an instruction that writes a register and any of the next three instructions read the register, the reading instruction gets the old value
  - Compiler must insert “bubbles” to separate producer and consumer of value, if it can’t find anything more useful to do
      - A bubble is a NOP (an instruction that doesn’t change any values)
  - Pros: simple (for the hardware, and not too hard for the software)
  - Cons: code is longer (due to NOPs); may have to insert bubbles before branching because you’re not sure exactly where you’re branching to (so execution is slower)

Resolving the RAW Hazard

- 2) Insert bubbles dynamically
  - Make it the hardware's problem
  - The data path notices that the instruction consuming a value follows to closely the instruction that produces that value and stalls – injects bubbles (NOPs) into the pipeline to achieve the required separation
  - Pros: stall only when actually necessary
  - Cons: more complicated hardware; if the compiler is insensitive to this issue may have to stall even though the stall could have been avoided by a smarter compiler

Resolving the RAW Hazard

- 3) Forwarding
  - Note that in many cases the value needed is produced before the consuming instruction absolutely needs it
  - The needed value is somewhere in the data path, just not in the register
  - “Forward” it from wherever it is downstream to where it’s needed upstream



Forwarding (cont.)

- Pros:
  - Can resolve many RAW hazards without introducing bubbles
- Cons:
  - Need more complicated hardware to do the forwarding
  - Still need to be able to inject bubbles when forwarding won’t solve the problem
    - For example, lw x3, 0(x2) followed immediately by add x3, x3, x4

There are further complications

- Control hazards
    - Suppose we fetch a conditional branch instruction during cycle N
    - From what address should we fetch the instruction fetched during cycle N+1
        - If the branch isn't taken, fetch the next instruction in memory (PC+4)
        - If the branch is taken, fetch the instruction at the branch target address
    - During cycle N+1 we don't yet know if the branch will be taken
    - Solution: guess (speculate)
        - Guess the branch won't be taken and fetch next sequential instruction. If later branch is taken, convert the mistakenly fetched instructions into NOPs
        - Keep a "branch prediction table." When you fetch a branch at some PC, remember the PC and what the correct next instruction address is. Next time you fetch that same branch, guess you'll do what you did last time
        - Implement a more sophisticated branch prediction table
            - E.g., branches associated with For loop are taken, taken, taken, ... not taken. Always predicting taken is better than ever predicting not taken

Current Plan for Next Time

- More about ILP
- Pipelining is too restrictive
    - Because instructions can’t pass each other (relative to sequential program execution order), if an instruction stalls all the instructions behind it must stall, even if they could in principle make progress with correct results
    - Some instructions are “harder” than other and take more time
        - E.g., floating point operations vs. integer operations
        - Pipeline effectively stalls the faster instructions
- What to do?
    - Superscalar architectures