

CS110A Pipeline

Chundong Wang

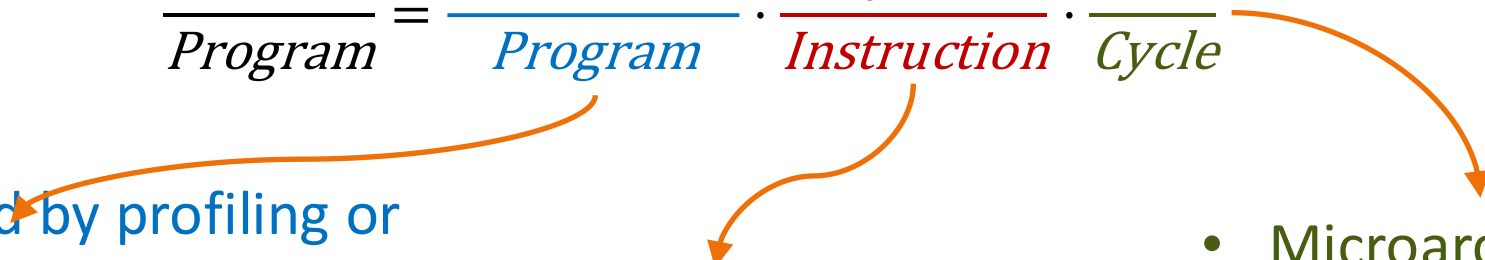
Performance

Recall the great ideas in CA

- Abstraction (layers of representation/interpretation)
- Moore's law (designing through trends)
- Make the common case fast
- Principle of locality (memory hierarchy)
- Parallelism (pipeline as a special case)
- **Performance measurement & improvement**
- Dependability via redundancy

“Iron law” of performance

- CPU (execution) time (ignore I/O, operating system overhead etc.)

$$\frac{\textit{Time}}{\textit{Program}} = \frac{\textit{Instructions}}{\textit{Program}} \cdot \frac{\textit{Cycles}}{\textit{Instruction}} \cdot \frac{\textit{Time}}{\textit{Cycle}}$$


- Can be obtained by profiling or hardware counter

- ISA (e.g., RISC vs. CISC)
- The program itself
- Compiler
- Programming language
- etc.

- Short as “CPI”
- Microarchitecture implementation or circuit design/ISA
- Compiler
- Program
- Programming language

- Microarchitecture implementation or circuit design/ISA

“Iron law” Example

- **Calculate average CPI**

Program A	A-instruction	B-instruction	C-instruction
CPI	2	2	4
Percentage	20%	40%	40%

- **Calculate CPU (execution) time**
 - CPU frequency 2.5 GHz
 - Program A has in total 10^6 instructions

“Iron law” Example

- Calculate average CPI

Program A	A-instruction	B-instruction	C-instruction
CPI	2	2	4
Percentage	20%	40%	40%

$$\text{Average CPI} = 2 \cdot 20\% + 2 \cdot 40\% + 4 \cdot 40\% = 2.8$$

- Calculate CPU (execution) time
 - CPU frequency 2.5 GHz
 - Program A has in total 10^6 instructions

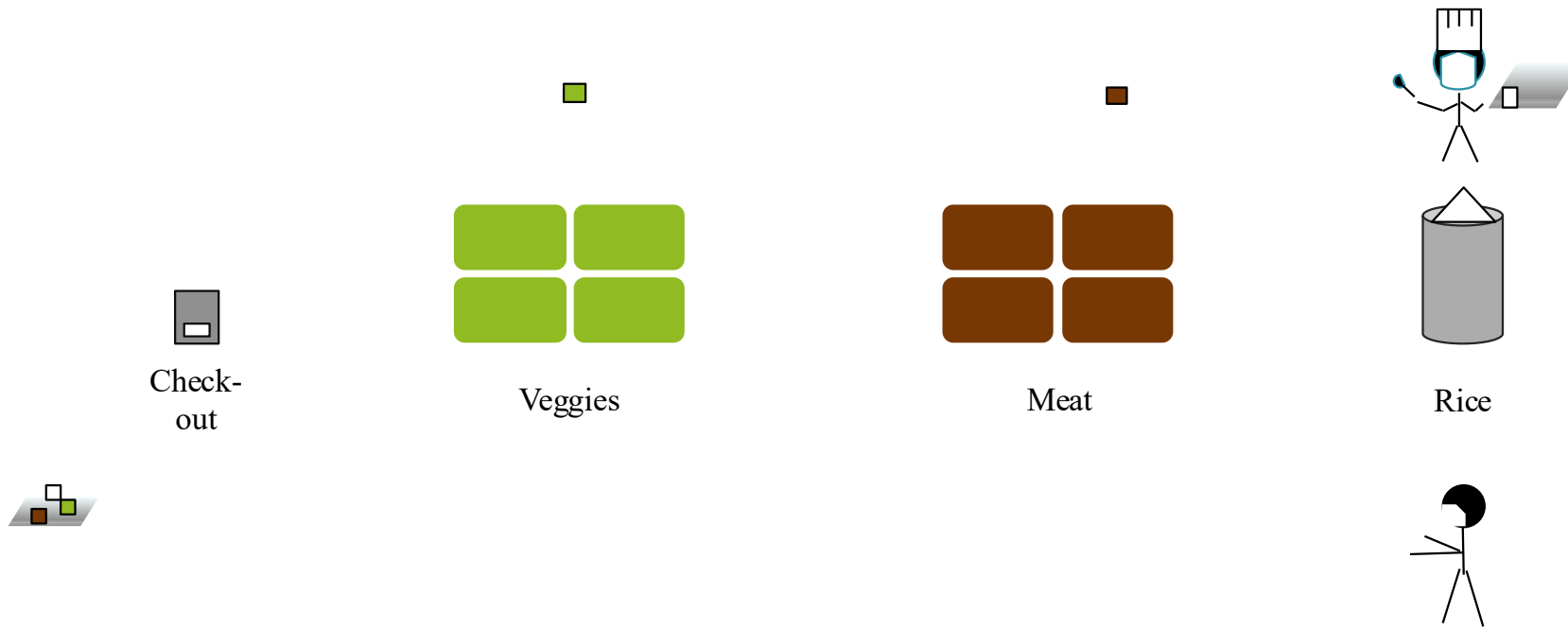
Different equation
if given IPC!

$$\begin{aligned}\text{CPU time} &= \text{Instruction count} * \text{Average CPI} * \text{Clock cycle} \\ &= 10^6 * 2.8 * 1/2.5\text{G} \\ &= 1.12 * 10^{-3} = 1.12 \text{ ms}\end{aligned}$$

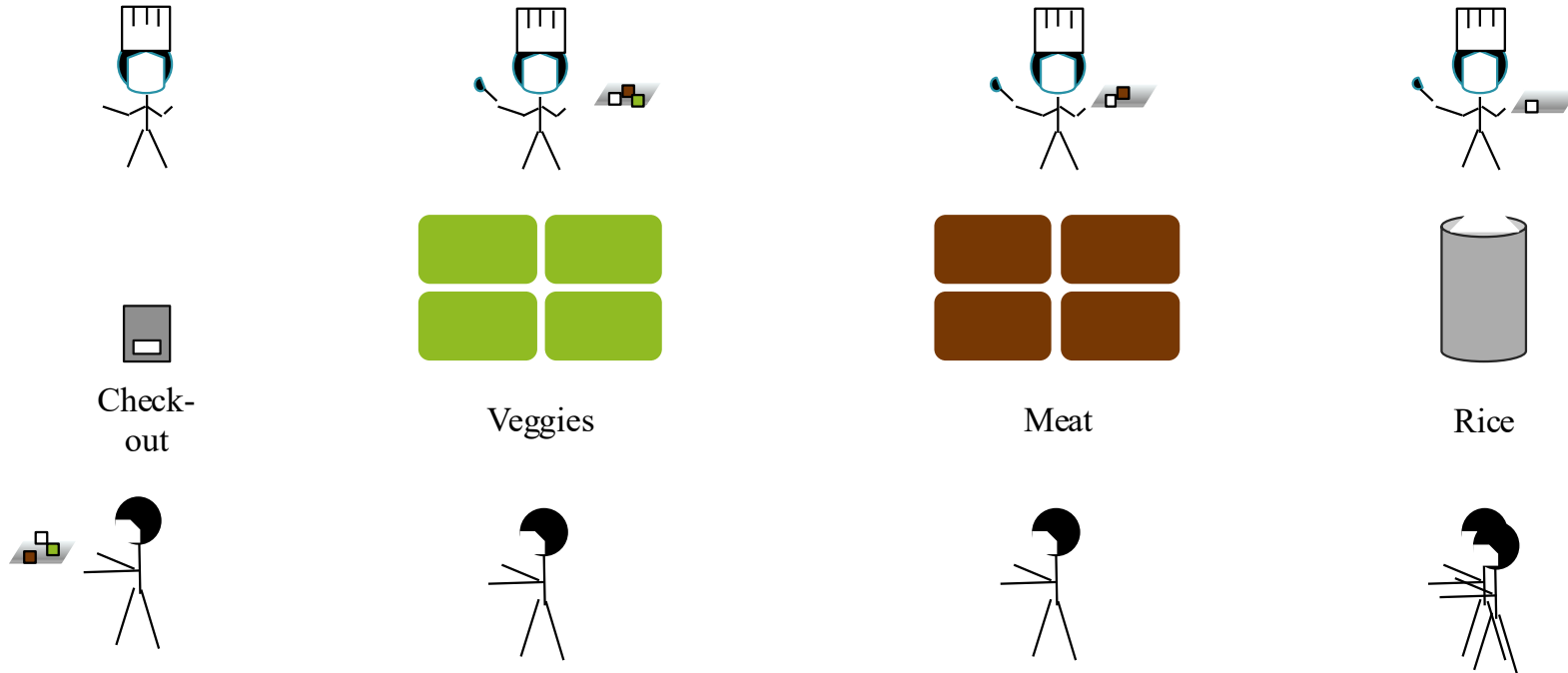
Pipeline

One way to parallelism

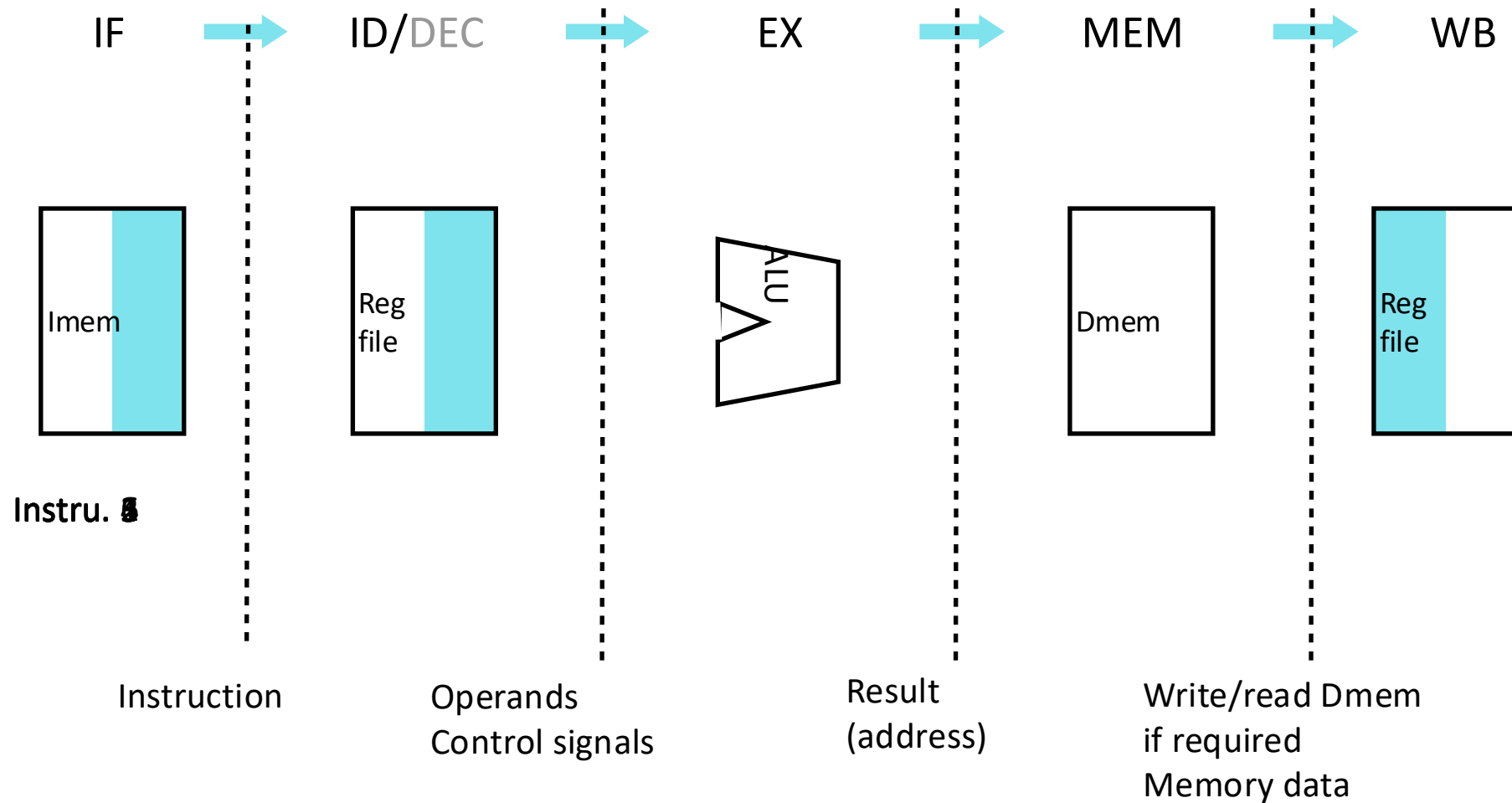
Pipe that line!



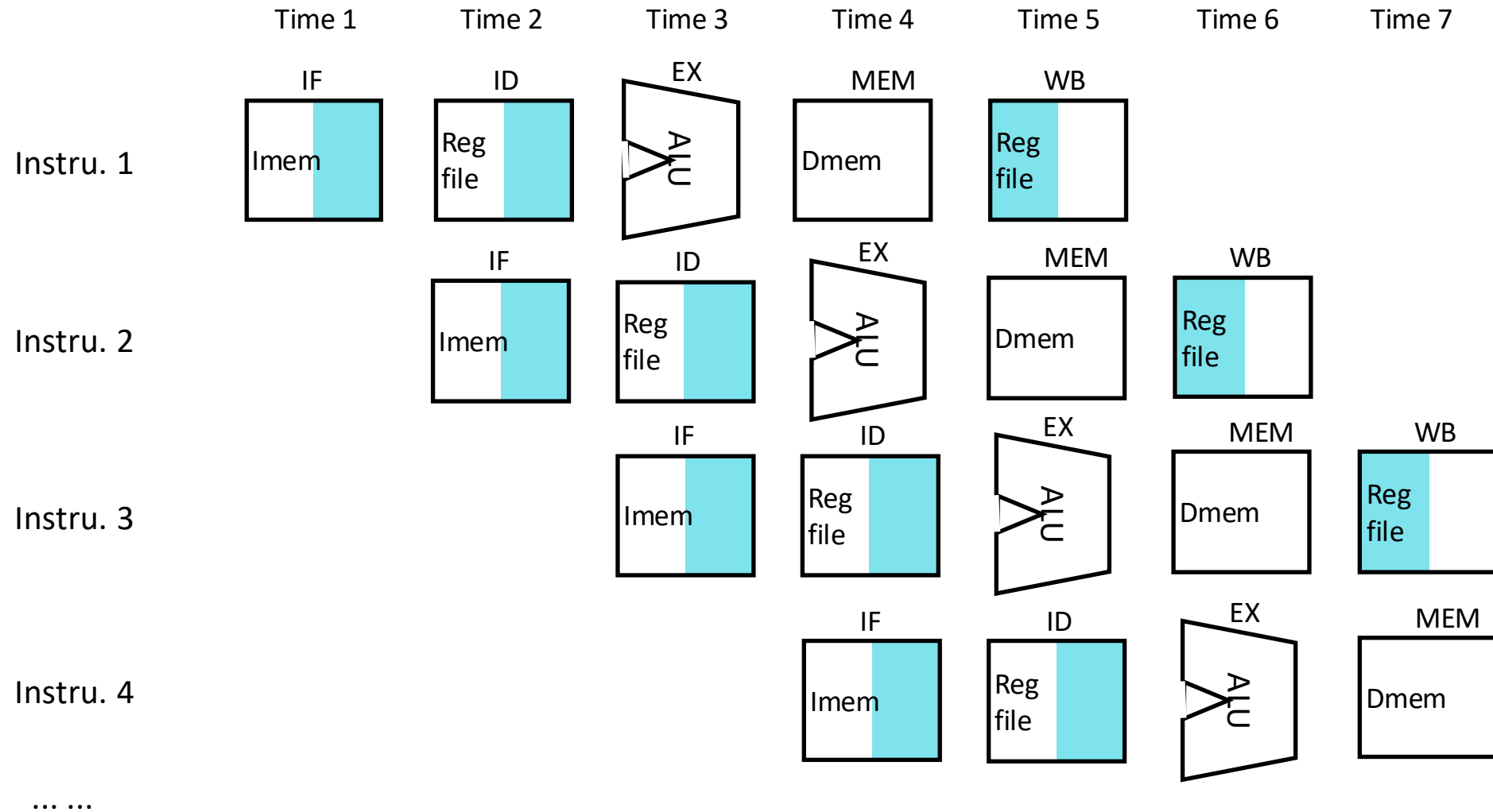
Pipe that line!



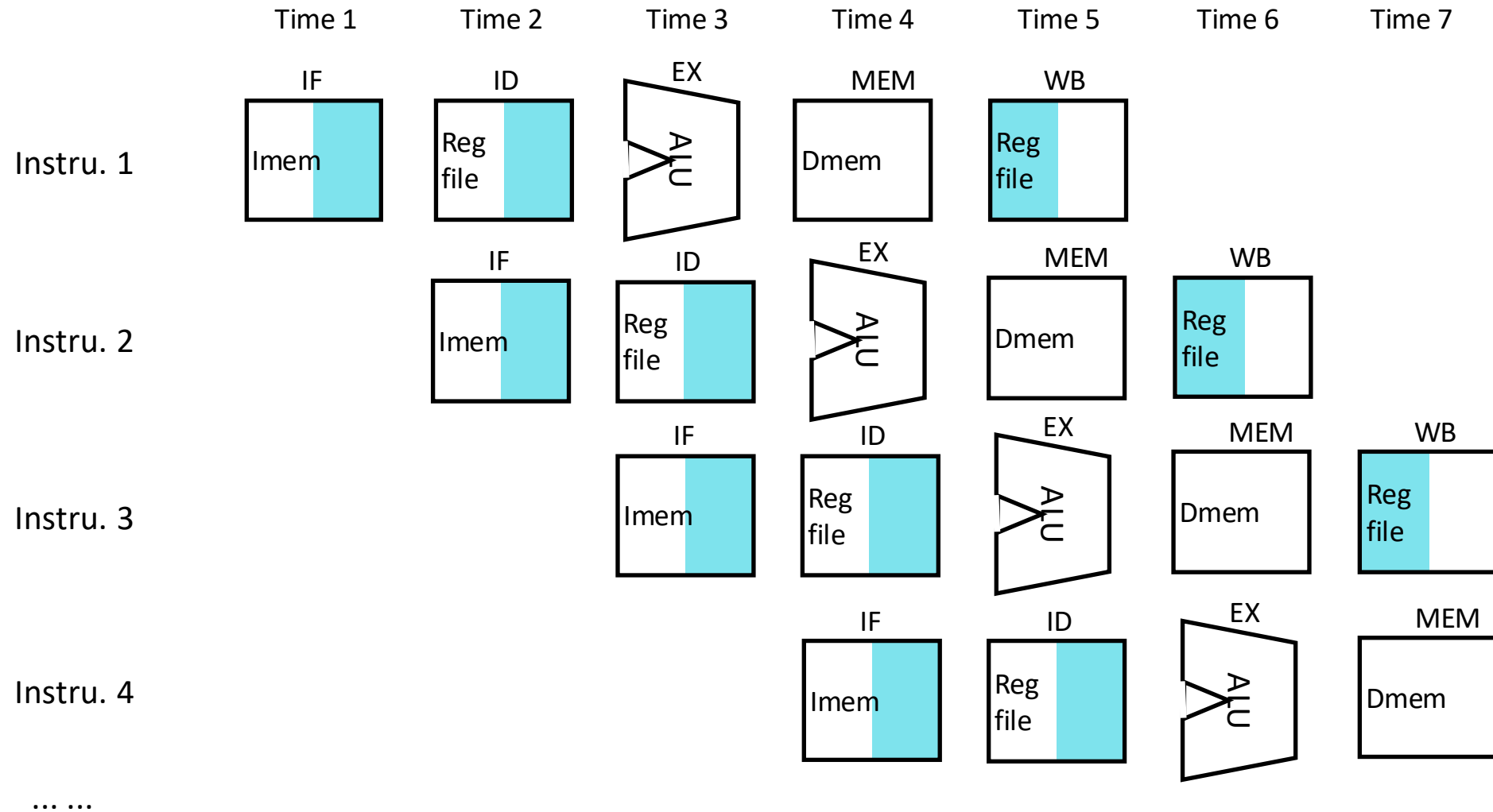
Simplify the CPU with 5 stages



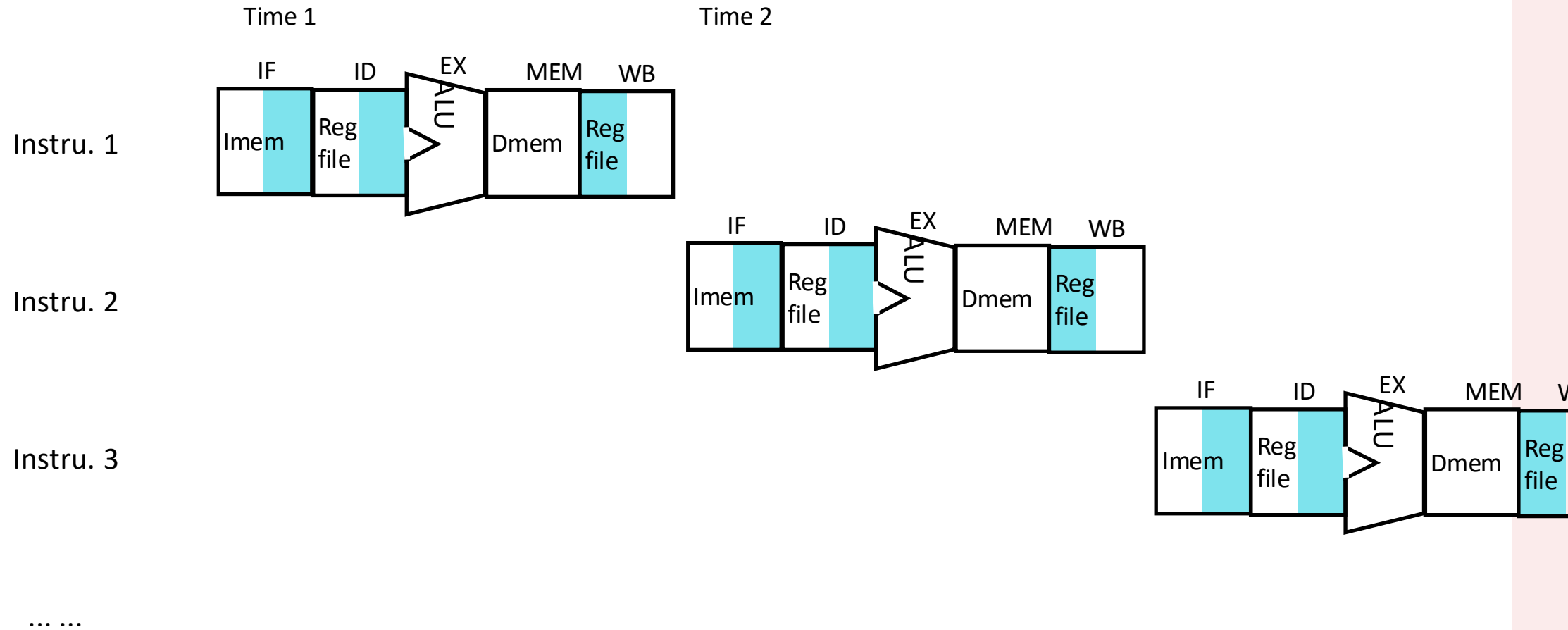
Make an analogy



Make an analogy



Previously for a single-cycle CPU



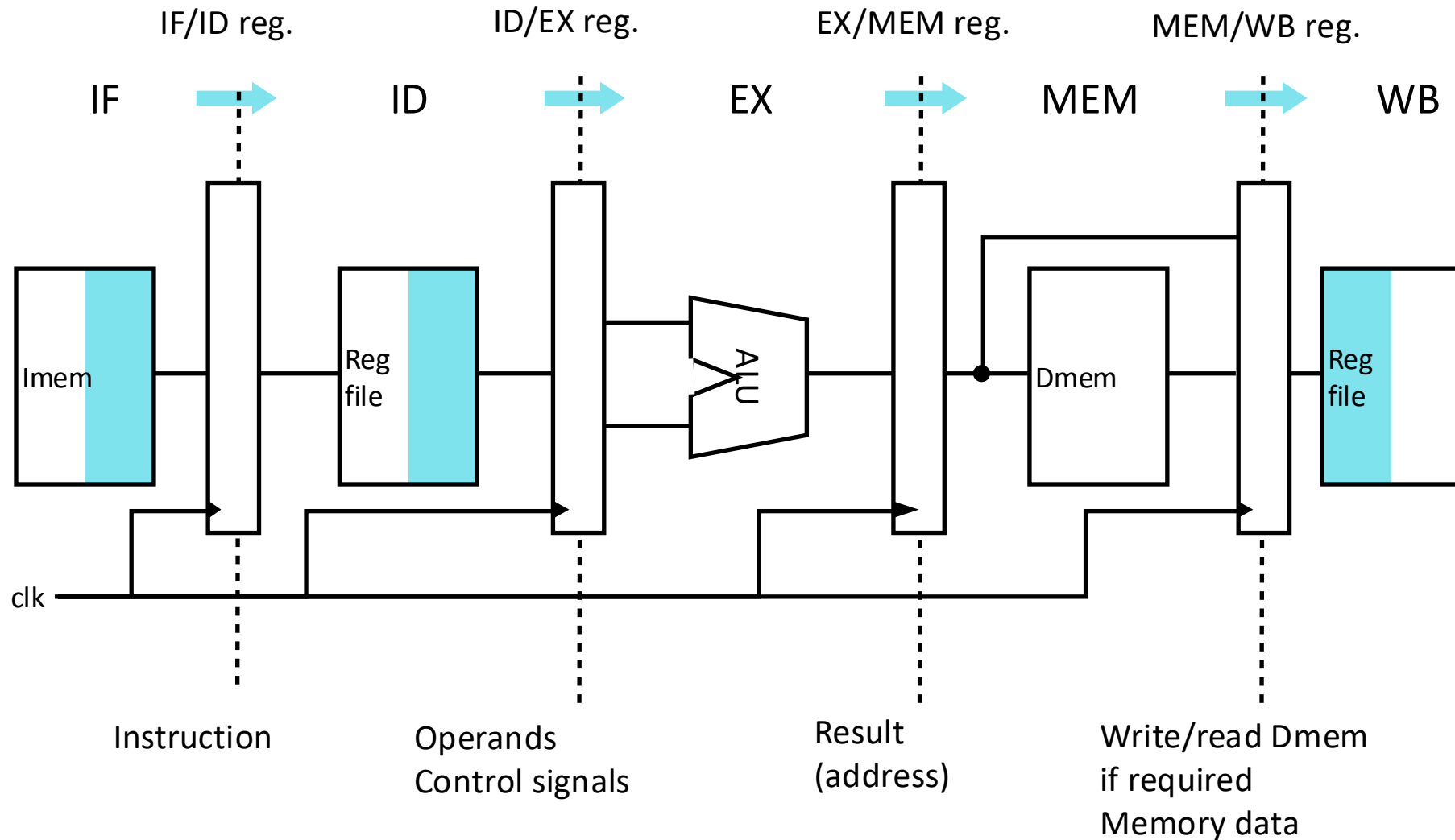
Observations

- Each instruction still take 5 stages/time slots to complete, no matter pipelined or not
- Period of the time slot is different
 - Single-cycle CPU: $t_{IF} + t_{ID} + t_{EX} + t_{MEM} + t_{WB}$
 - Pipelined CPU: $\max\{t_{IF}, t_{ID}, t_{EX}, t_{MEM}, t_{WB}\}$
- Use the “iron law” of performance, pipelined CPU is faster

$$\frac{\textit{Time}}{\textit{Program}} = \frac{\textit{Instructions}}{\textit{Program}} \cdot \frac{\textit{Cycles}}{\textit{Instruction}} \cdot \frac{\textit{Time}}{\textit{Cycle}}$$

- How to make the CPU pipelined?

Insert pipeline registers



Critical path decided by the slowest stage

Detailed considerations

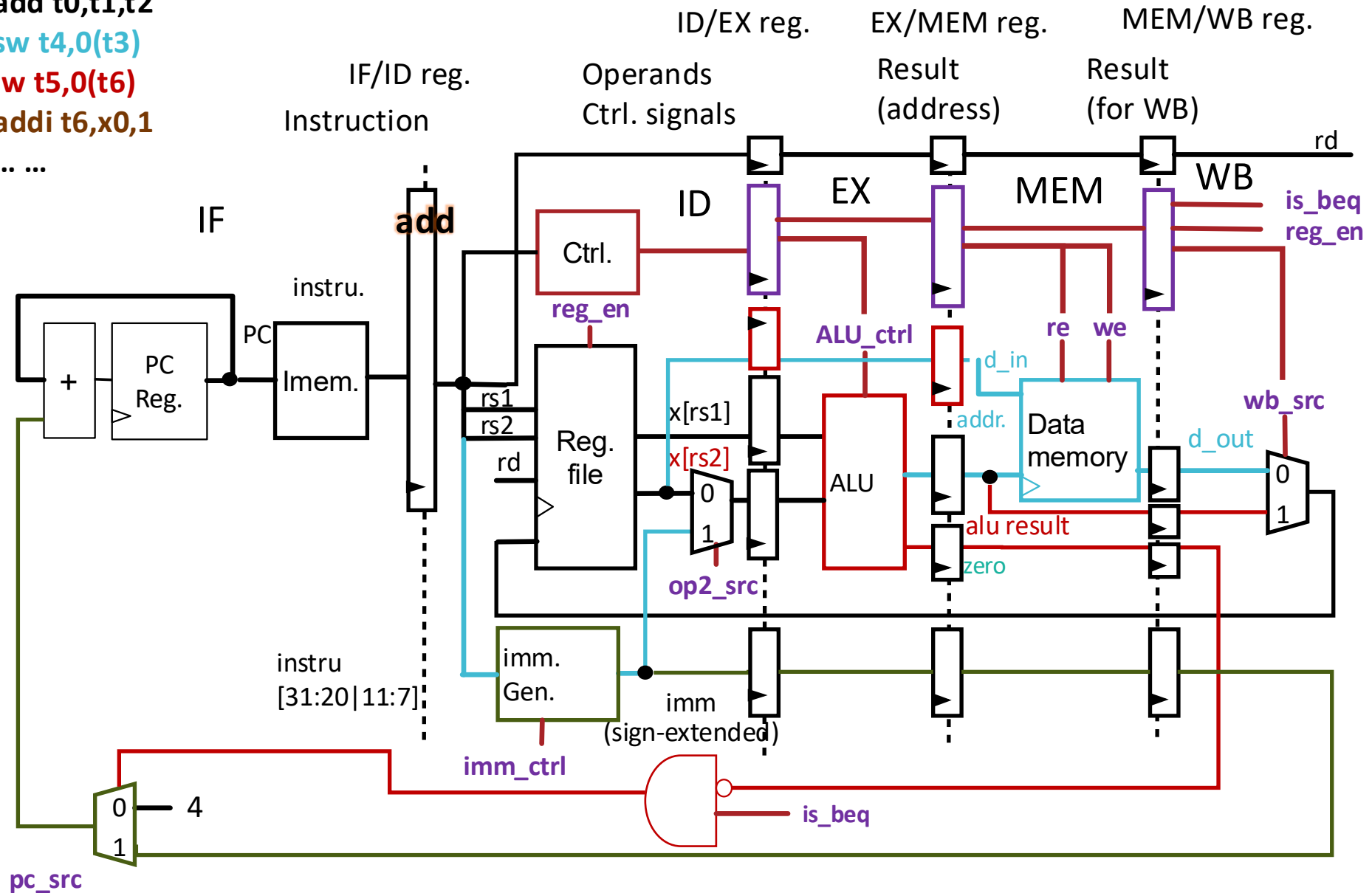
add t0,t1,t2

sw t4,0(t3)

lw t5,0(t6)

addi t6,x0,1

... ..



Detailed considerations

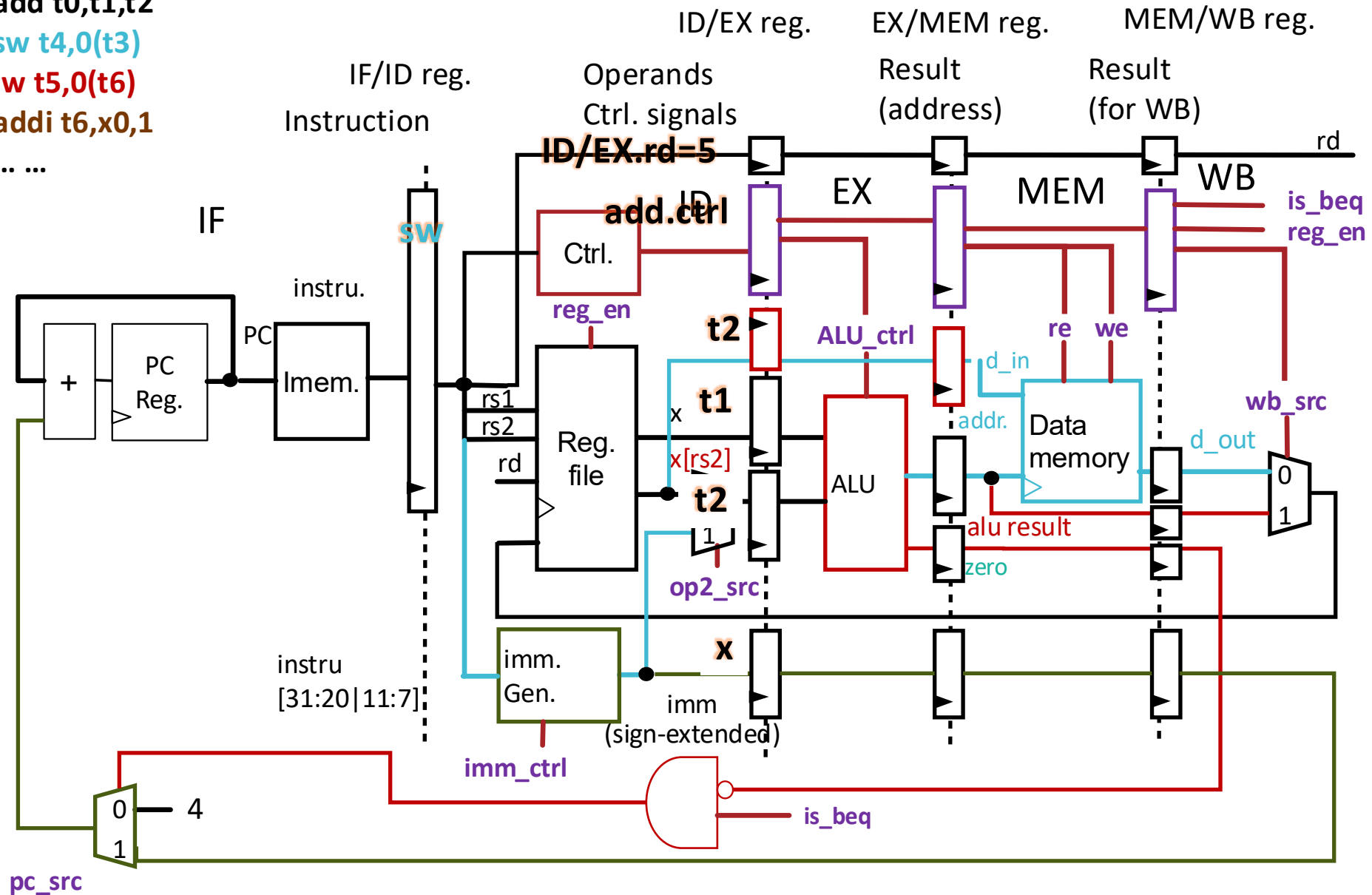
add t0,t1,t2

sw t4,0(t3)

lw t5,0(t6)

addi t6,x0,1

... ..



...

MEM/WB reg.

Instruction



Detailed considerations

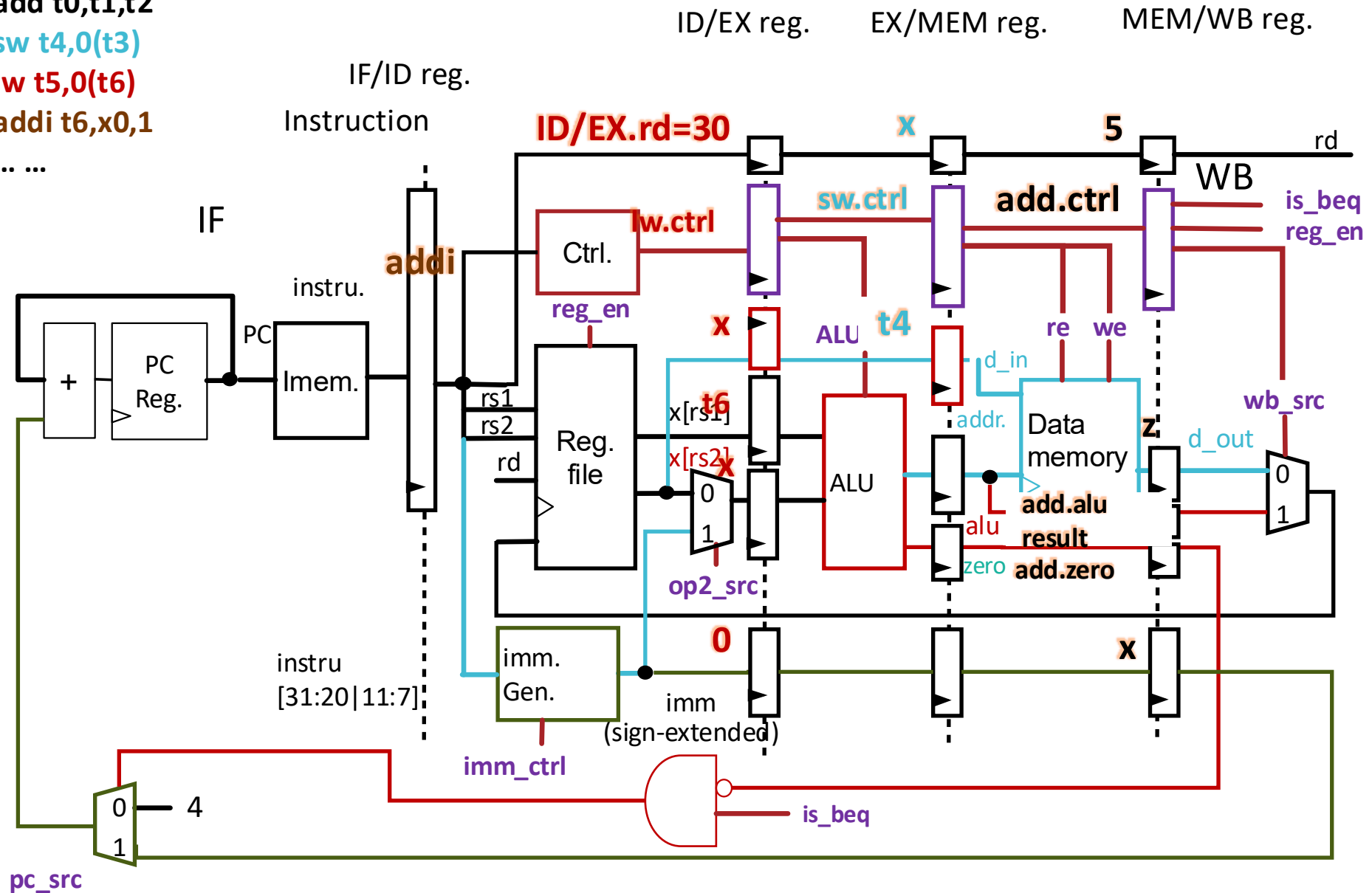
add t0,t1,t2

sw t4,0(t3)

lw t5,0(t6)

addi t6,x0,1

... ..



Hazards ahead!!!

add t0,t1,t2

sw t4,0(t3)

lw t5,0(t6)

```
addi t6,x0,1
```

• • • • •

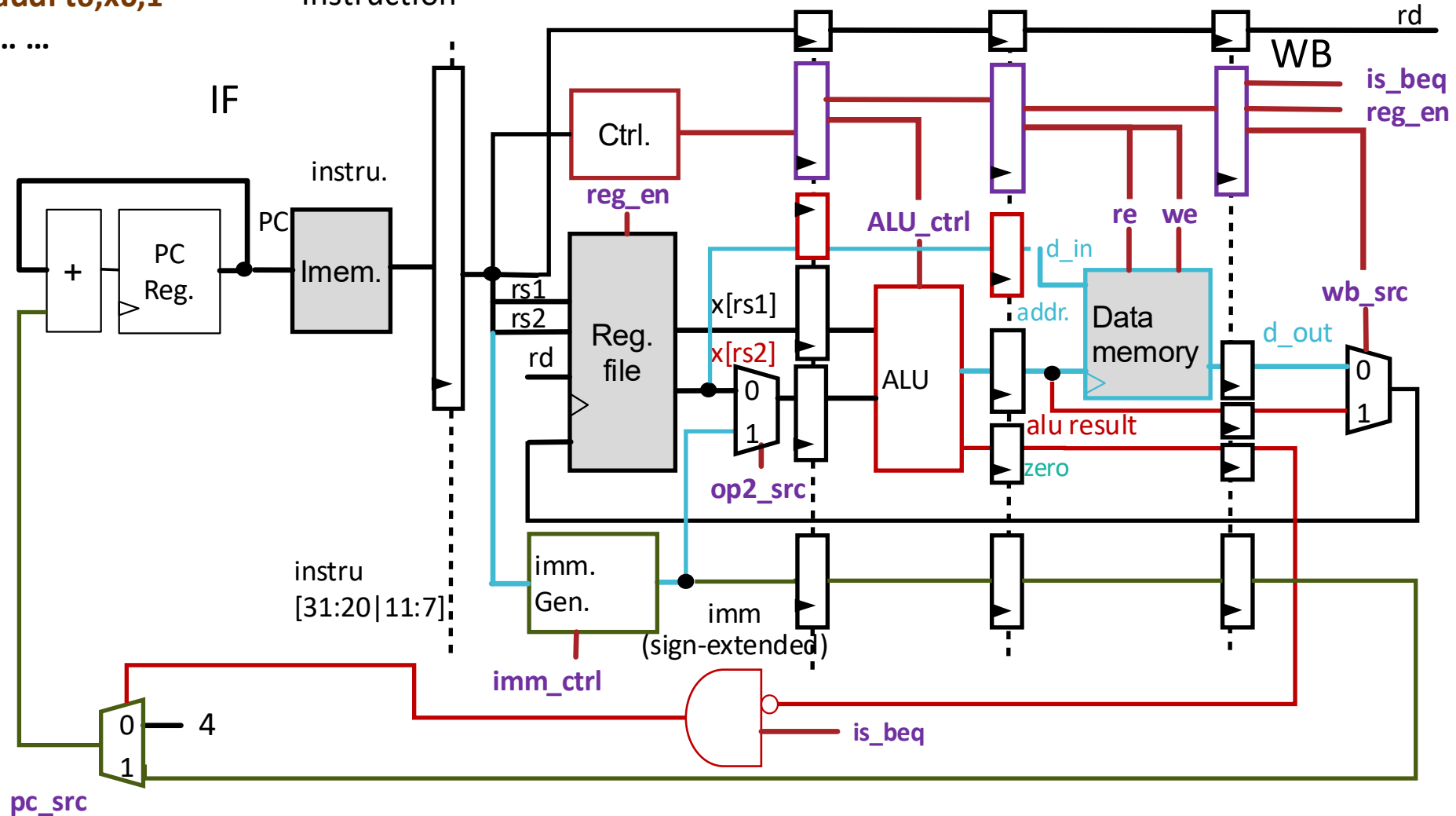
ID/EX reg.

EX/MEM reg.

MEM/WB reg.

IF/ID reg.

Instruction

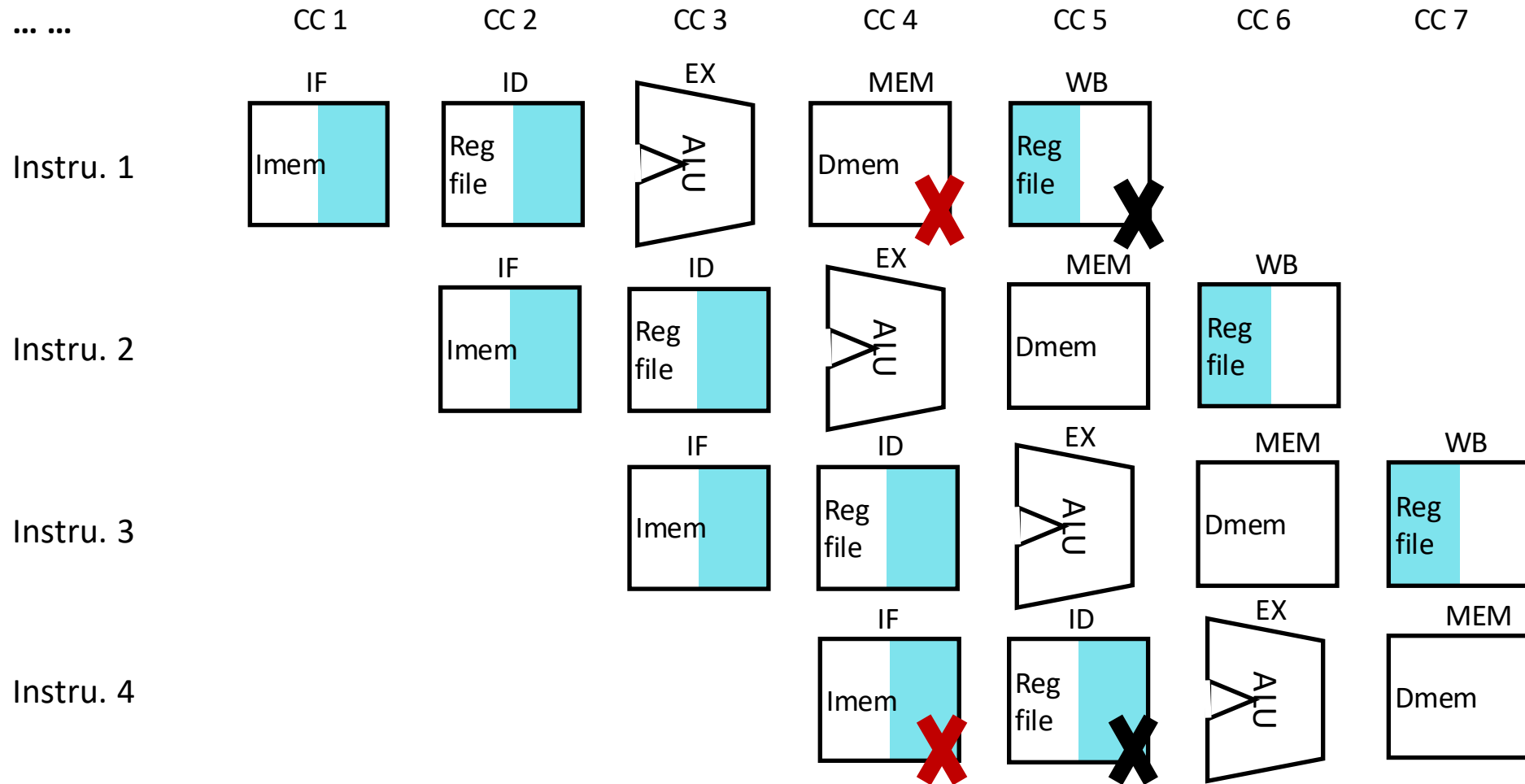


Hazards ahead!!!

- A **hazard** is a situation in which a planned instruction cannot execute in the “proper” clock cycle.
- **Structural hazards**
- Data hazards
- Control hazards

Structural hazards

lw t0,0(t2)
sw t0,0(t3)
lw t5,0(t6)
addi t6,t0,1



Structural hazards

- Caused by hardware limitations. Two or more instructions in the pipeline compete for a single physical resource
- Can be solved by
 - Seperate instruction and data memory (real CPU uses instruction cache and data cache)
 - or using dual-port memory (input multiple addresses, output multiple data) ([general ways to solve structural hazards, add more hardware](#))
 - Assume register file write at rising edge (in the textbook “the first half clock cycle”), read arbitrarily (in the textbook “the second half clock cycle”), and design the hardware with this feature
 - Instructions take turns to use the physical resource (wait/stall)

Hazards ahead!!!

- Structural hazards
- **Data hazards**
- Control hazards

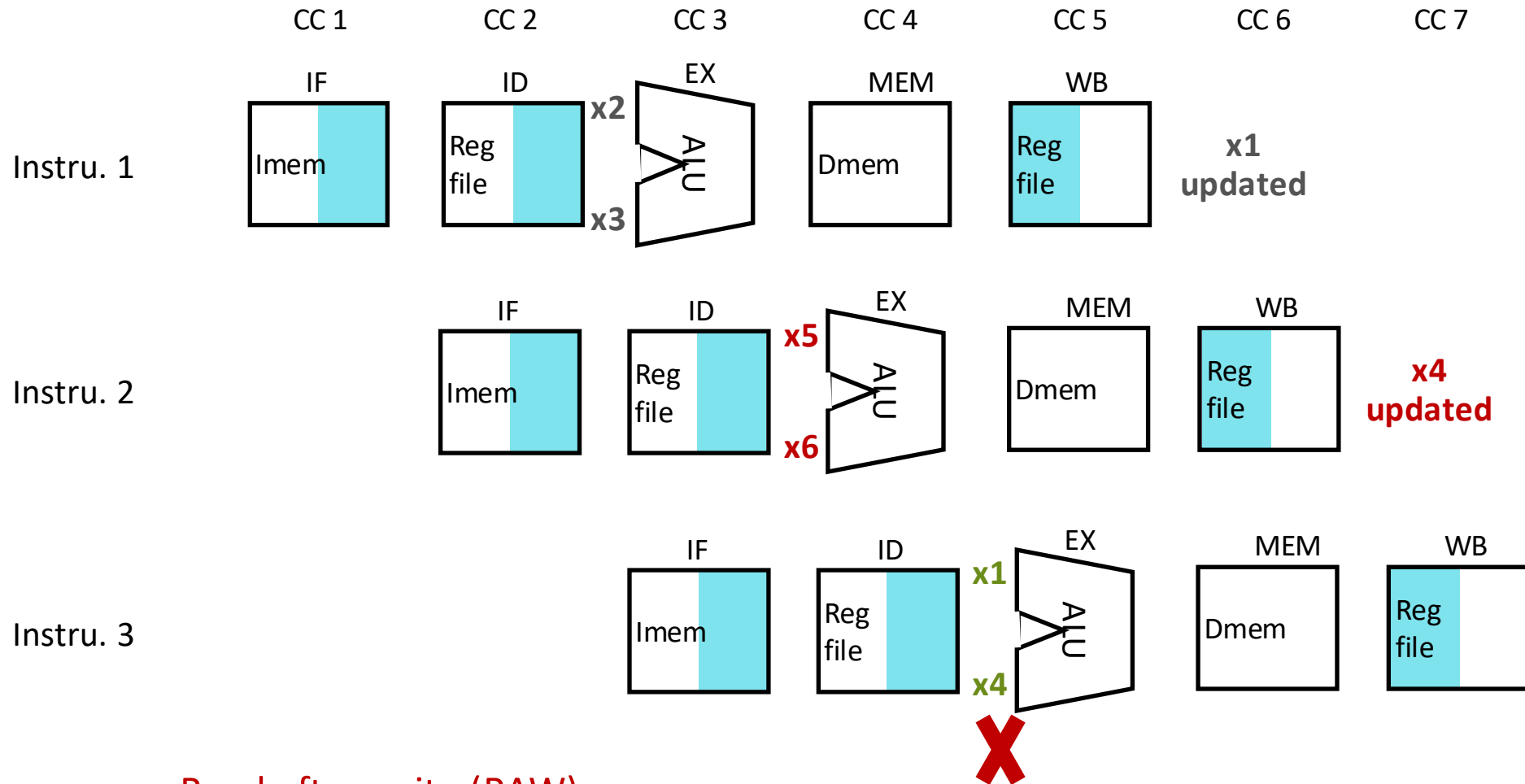
Data hazards

add x1, x2, x3

add x4, x5, x6

add x7, x1, x4

R-type



Read after write (RAW)

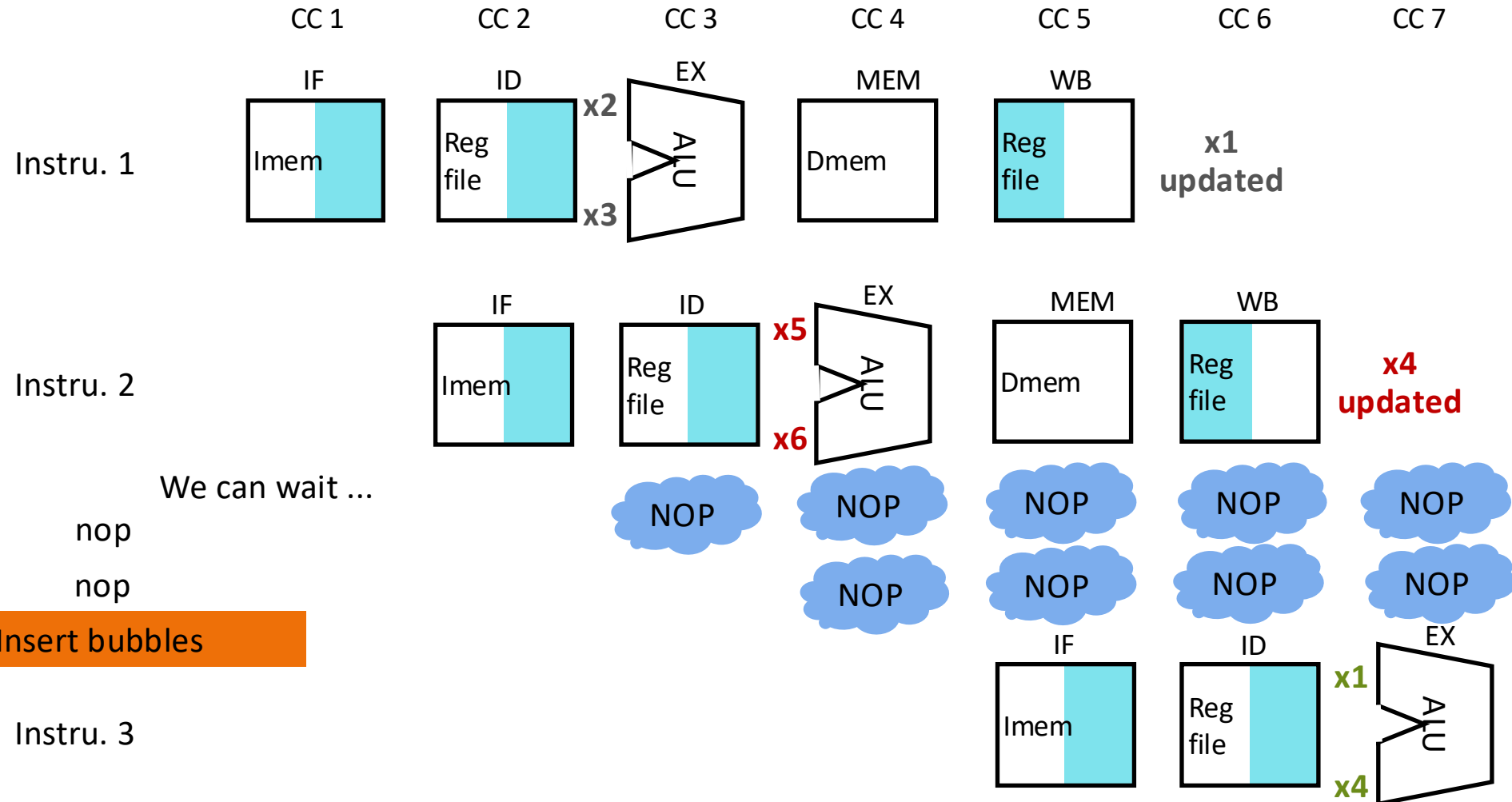
Data hazards -- solution 1

add x1, x2, x3

add x4, x5, x6

add x7, x1, x4

R-type

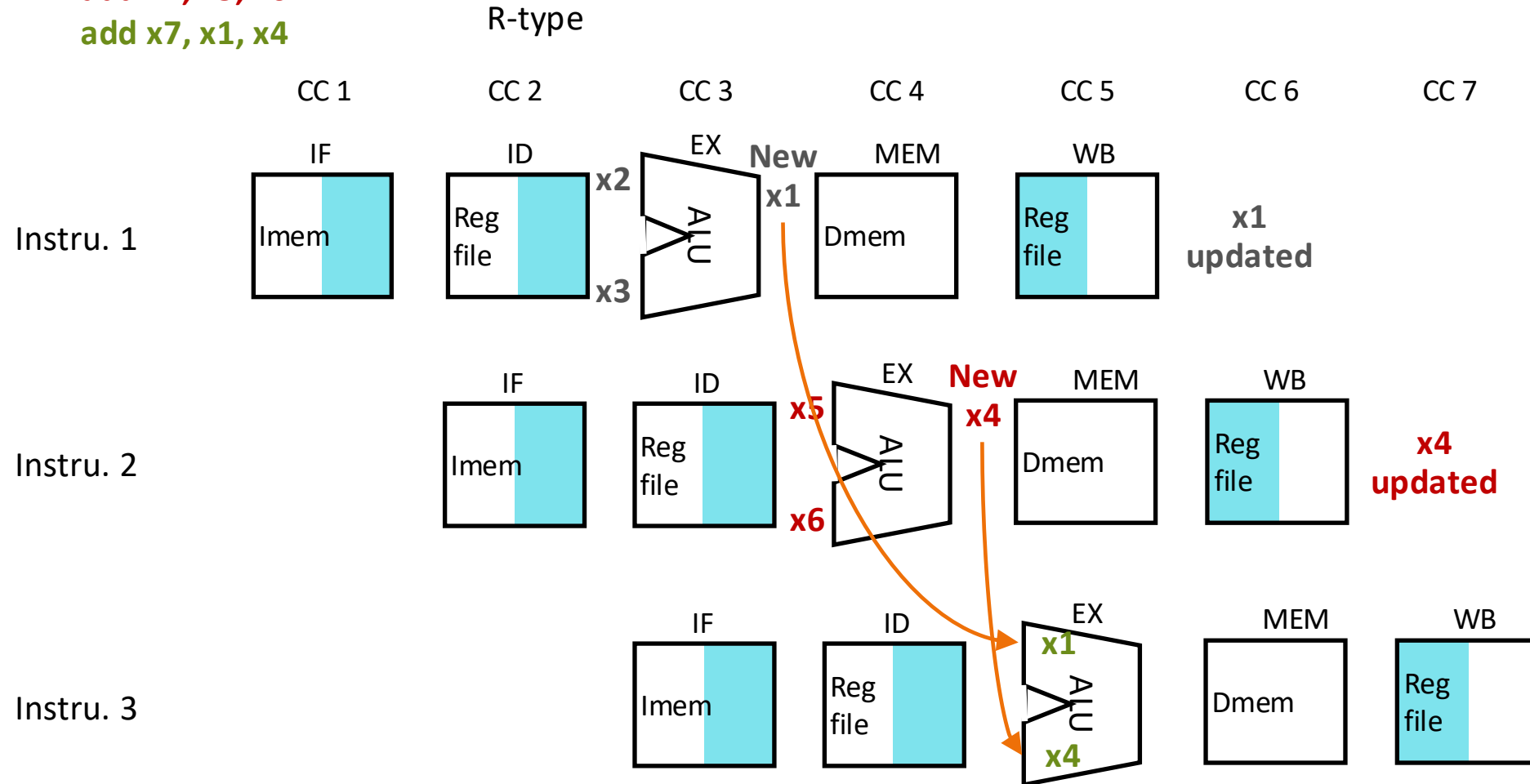


Data hazards -- solution 2

add x1, x2, x3

add x4, x5, x6

add x7, x1, x4



Forwarding or bypass

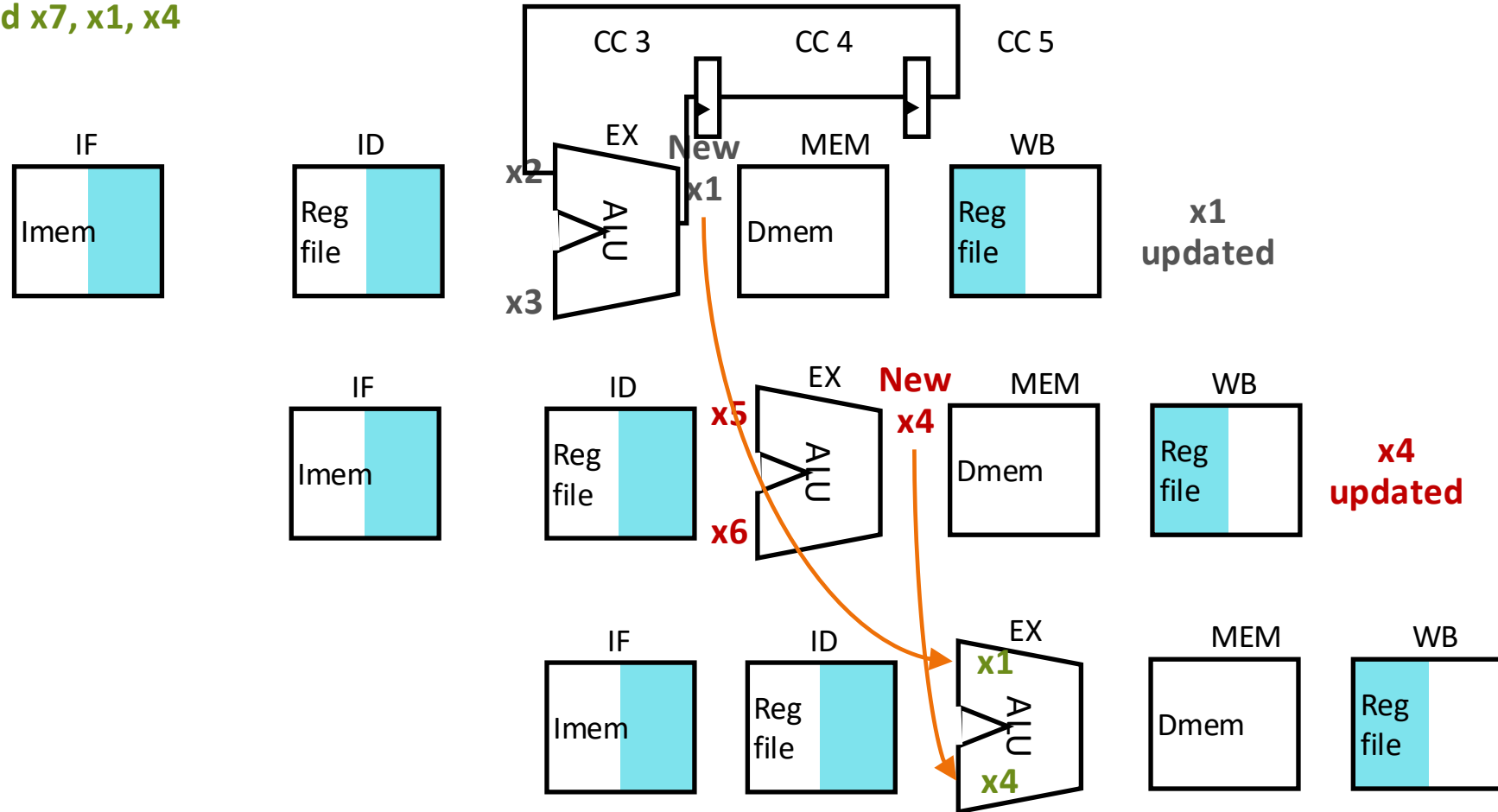
Data hazards -- solution 2

add x1, x2, x3

add x4, x5, x6

add x7, x1, x4

Add registers to store the updated values
(actually can access from pipeline reg.)



Forwarding or bypass

Data hazards -- solution 2

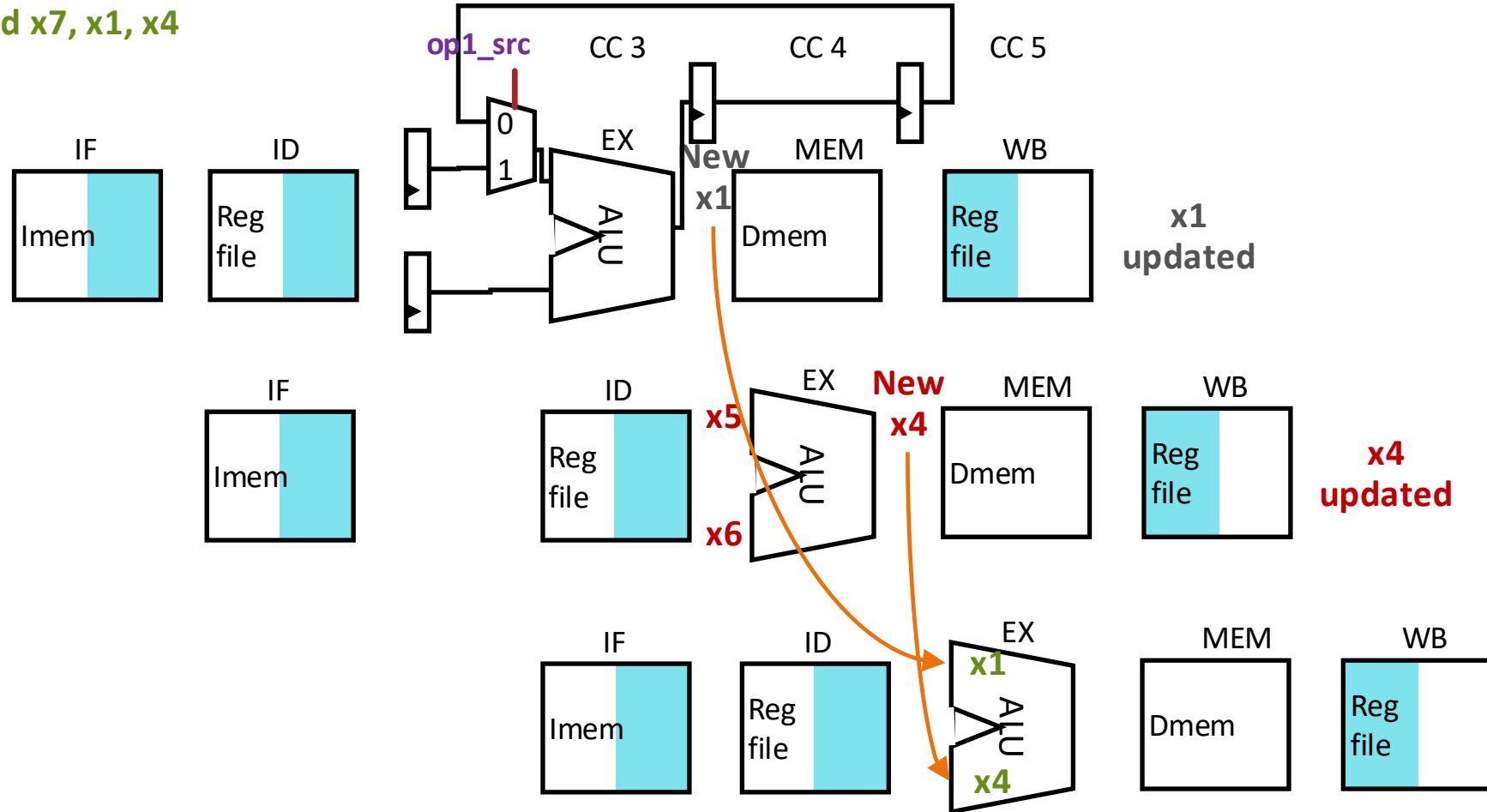
add x1, x2, x3

add x4, x5, x6

add x7, x1, x4

Add registers to store the updated values

(actually can access from pipeline reg.)

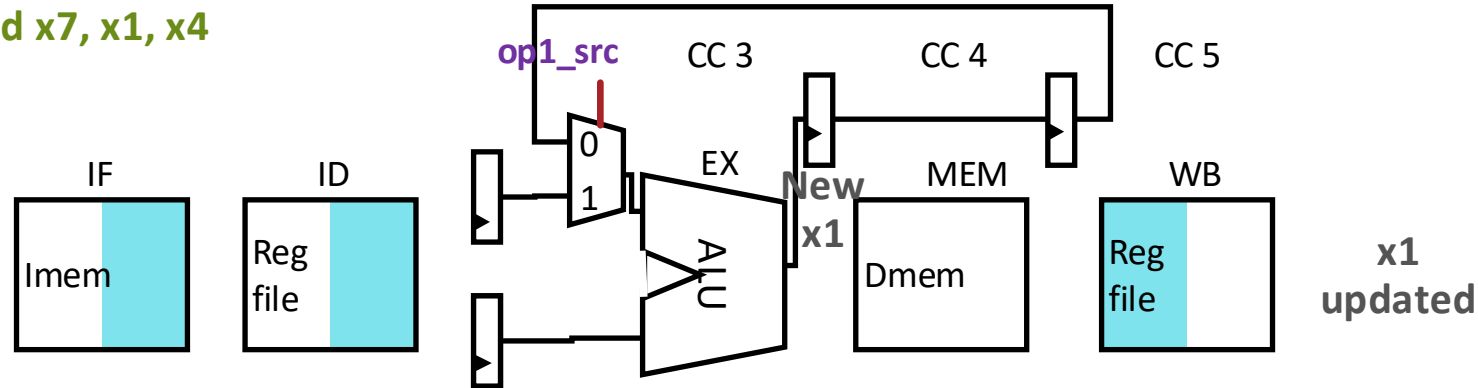


Forwarding or bypass

Data hazards -- solution 2

add x1, x2, x3
add x4, x5, x6
add x7, x1, x4

Add registers to store the updated values
(actually can access from pipeline reg.)

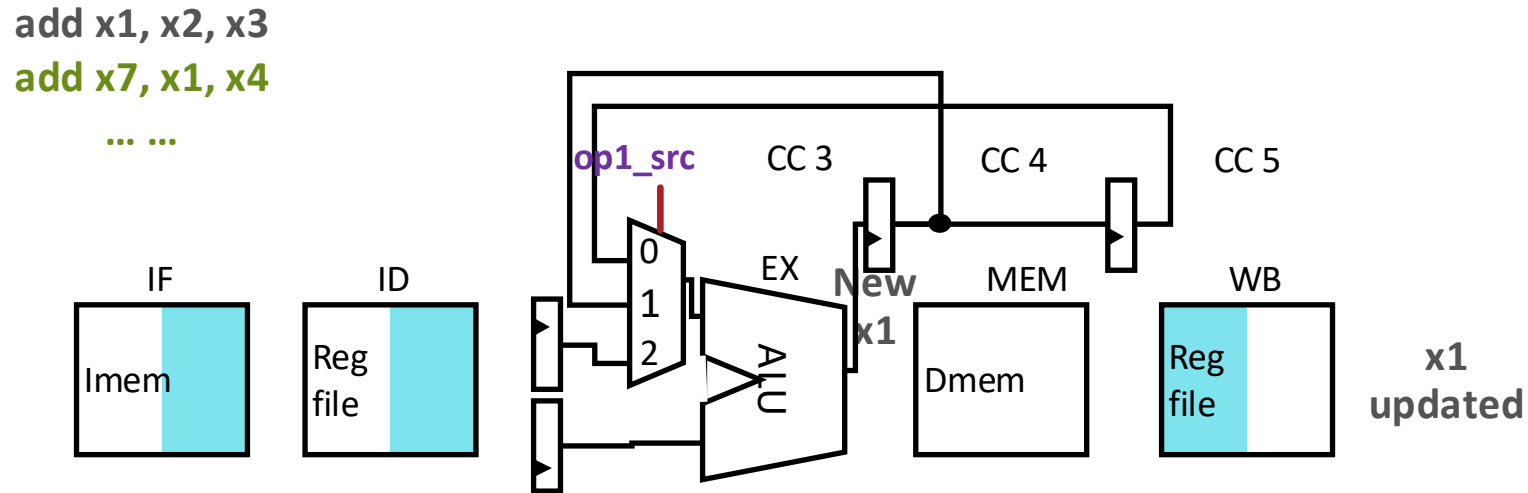


- How to decide **op1_src**?
 - Select the forwarded value or the value from ID/EX register
 - 1. rd of the add instruction (may be the other type instructions) equals rs1 in **add** (may be the other type instructions) instruction
 - 2. Ignore write to x0
 - 3. The first instruction must write the register and the third instruction must read the register

Forwarding or bypass

Forwarding control logic

Data hazards -- solution 2

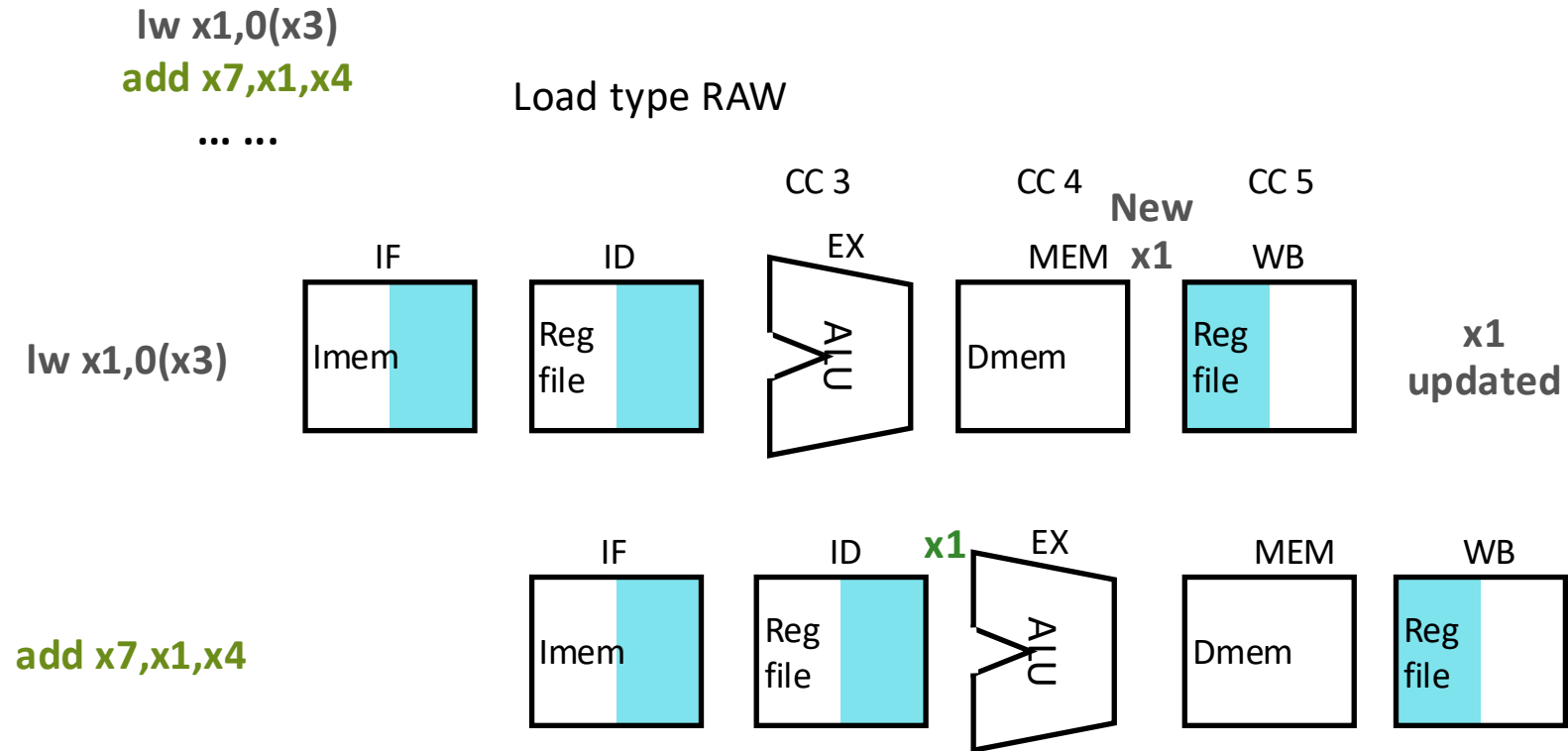


- How to decide **op1_src**?
 - Select the forwarded value or the value from EX/MEM register
 - 1. rd of the add instruction (may be the other type instructions) equals rs1 in **add** (may be the other type instructions) instruction
 - 2. Ignore write to x0
 - 3. The first instruction must write the register and the second instruction must read the register

Forwarding or bypass

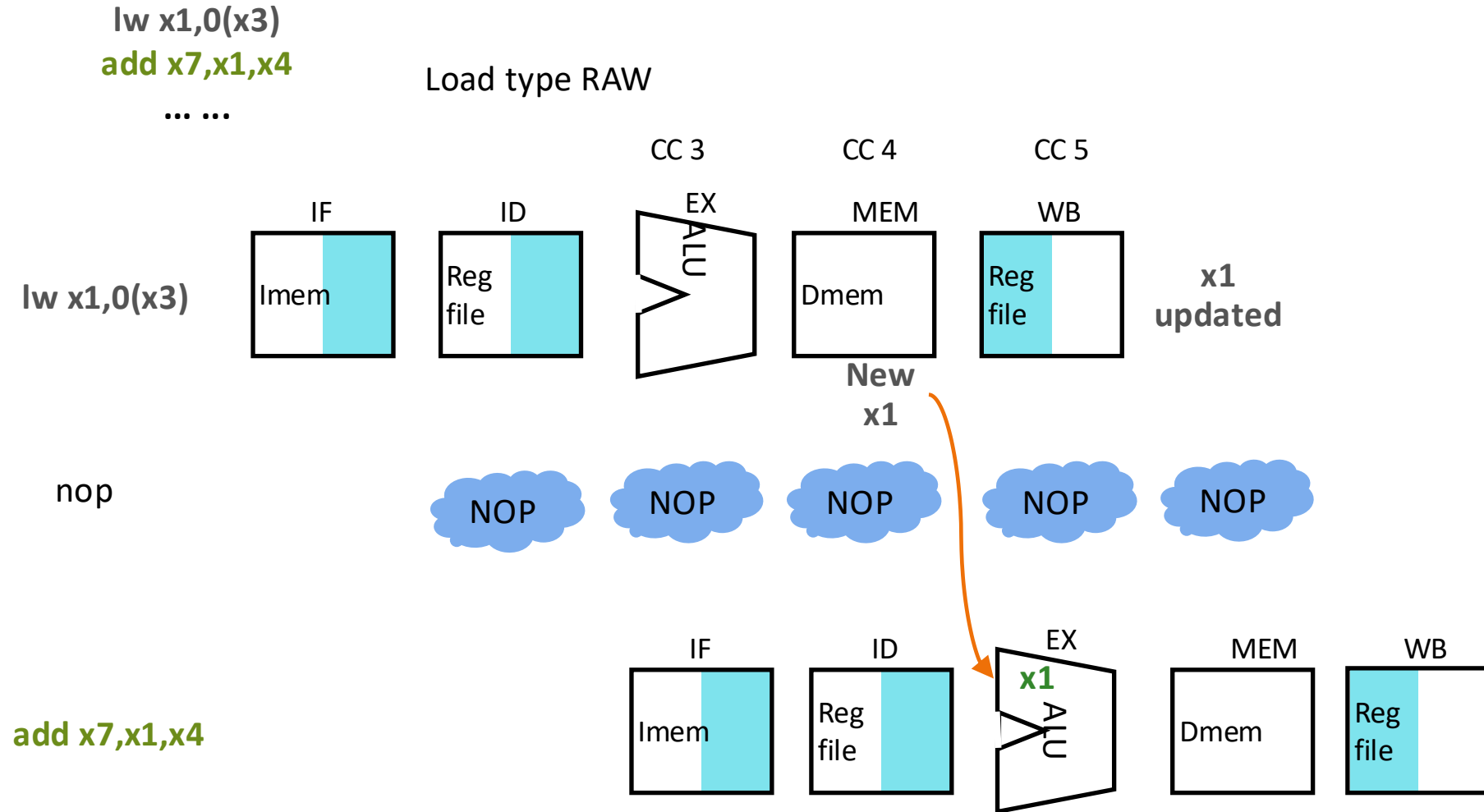
Forwarding control logic

Data hazards



Forwarding cannot solve this, leading to a “load delay slot”

Data hazards -- solution 1



Insert nop and forward x1

Data hazards -- solution 2 warm up

- Identify all the data hazards in the following code

Original Order:

```
lw t1, 0(t0)
lw t2, 4(t0)
add t3, t1, t2
sw t3, 12(t0)
lw t4, 8(t0)
add t5, t1, t4
sw t5, 16(t0)
```

Data hazards -- solution 2 warm up

- Identify all the data hazards in the following code

Original Order:

```
lw t1, 0(t0)
lw t2, 4(t0)
add t3, t1, t2
sw t3, 12(t0)
lw t4, 8(t0)
add t5, t1, t4
sw t5, 16(t0)
```

Data hazards -- solution 2 warm up

- Identify all the data hazards in the following code
- Which of the hazards cannot be completely solved by forwarding?

Original Order:

```
lw t1, 0(t0)
lw t2, 4(t0)
add t3, t1, t2
sw t3, 12(t0)
lw t4, 8(t0)
add t5, t1, t4
sw t5, 16(t0)
```

Data hazards -- solution 2

- Code scheduling: Put unrelated instruction into load delay slot
 - No performance loss!

Original Order:

```
lw t1, 0(t0)
lw t2, 4(t0)
add t3, t1, t2
sw t3, 12(t0)
lw t4, 8(t0)
add t5, t1, t4
sw t5, 16(t0)
```

Code scheduling:

```
lw t1, 0(t0)
lw t2, 4(t0)
lw t4, 8(t0)
add t3, t1, t2
sw t3, 12(t0)
add t5, t1, t4
sw t5, 16(t0)
```

- Code scheduling can be accomplished by the compiler

Data hazards -- solution 2

- Code scheduling: Put unrelated instruction into load delay slot
 - No performance loss!

Original Order:

```
lw t1, 0(t0)
lw t2, 4(t0)
add t3, t1, t2
sw t3, 12(t0)
lw t4, 8(t0)
add t5, t1, t4
sw t5, 16(t0)
```

Code scheduling:

```
lw t1, 0(t0)
lw t2, 4(t0)
lw t4, 8(t0)
add t3, t1, t2
sw t3, 12(t0)
add t5, t1, t4
sw t5, 16(t0)
```

- Code scheduling can be accomplished by the compiler

Data hazards

- How many clock cycles to complete the code before and after code scheduling?
Assume no instruction in the pipeline initially.

Original Order:

```
lw t1, 0(t0)
lw t2, 4(t0)
add t3, t1, t2
sw t3, 12(t0)
lw t4, 8(t0)
add t5, t1, t4
sw t5, 16(t0)
```

Code scheduling:

```
lw t1, 0(t0)
lw t2, 4(t0)
lw t4, 8(t0)
add t3, t1, t2
sw t3, 12(t0)
add t5, t1, t4
sw t5, 16(t0)
```

Hint: Do not miss structural hazard(s)

Summary on data hazards

- Instructions have data dependency
- Occurs when an instruction reads a register before a previous instruction has finished writing to that register (RAW)
- There can also be **WAW/WAR** hazards depending on the pipeline design
- For load-type RAW data hazards, there is a load delay slot unavoidable
- Can be solved by forwarding or code scheduling