COMP311: *COMPUTER ORGANIZATION!*

Lecture 22: Source Bypassing

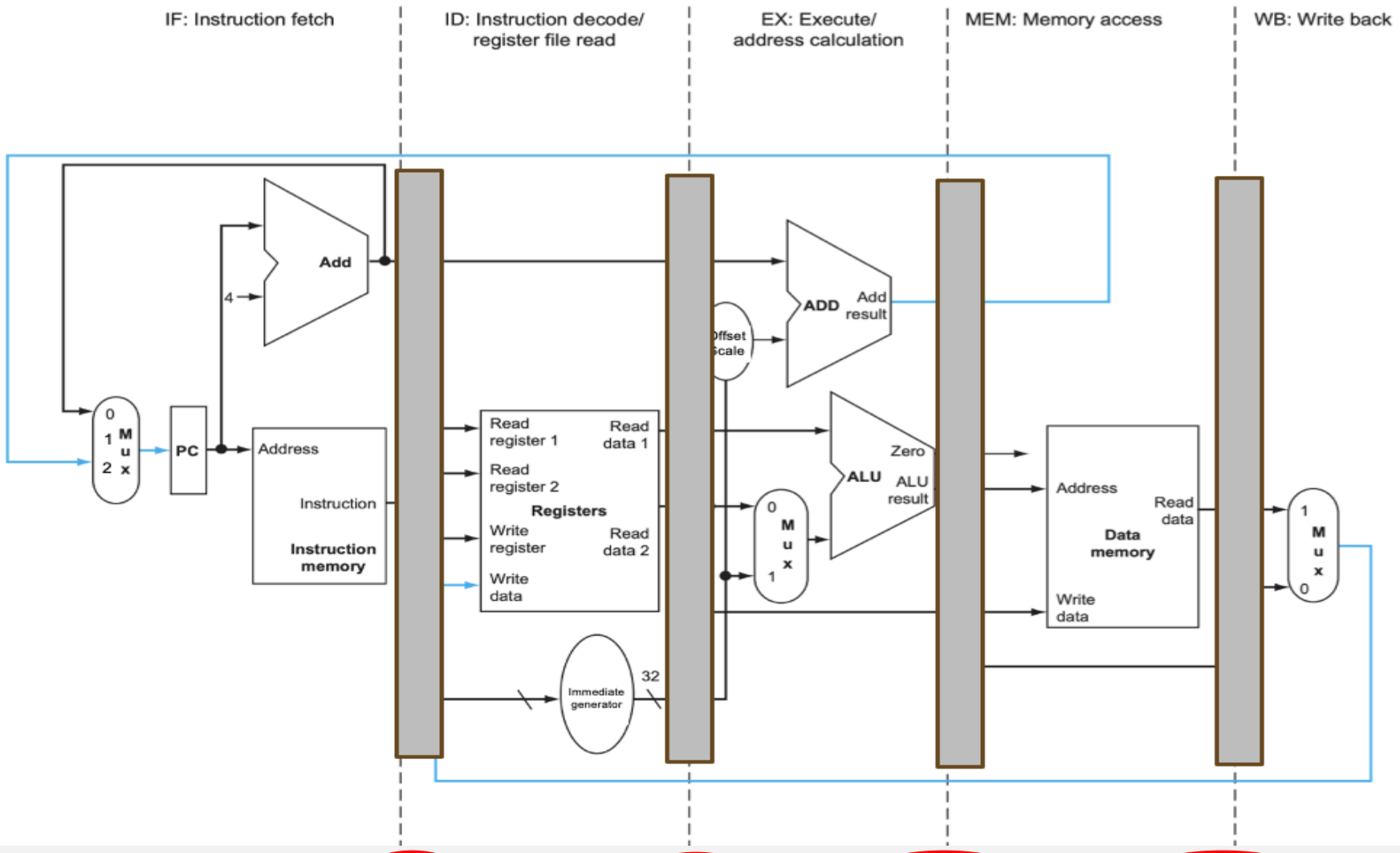
tinyurl.com/comp311-sp26

Logistics Update!

- Lab 5 Starter Code Updated if you pulled early
  - *Due 4/27 (LDOC)*
- Written Assignment 6
  - *Due Tuesday (4/21)*
- Office Hours Ending LDOC
- Final Review Session on First Reading Day

QUICK REVIEW: PIPELINE HAZARDS

Each pipeline register name is based on the stages it separates



Register Names

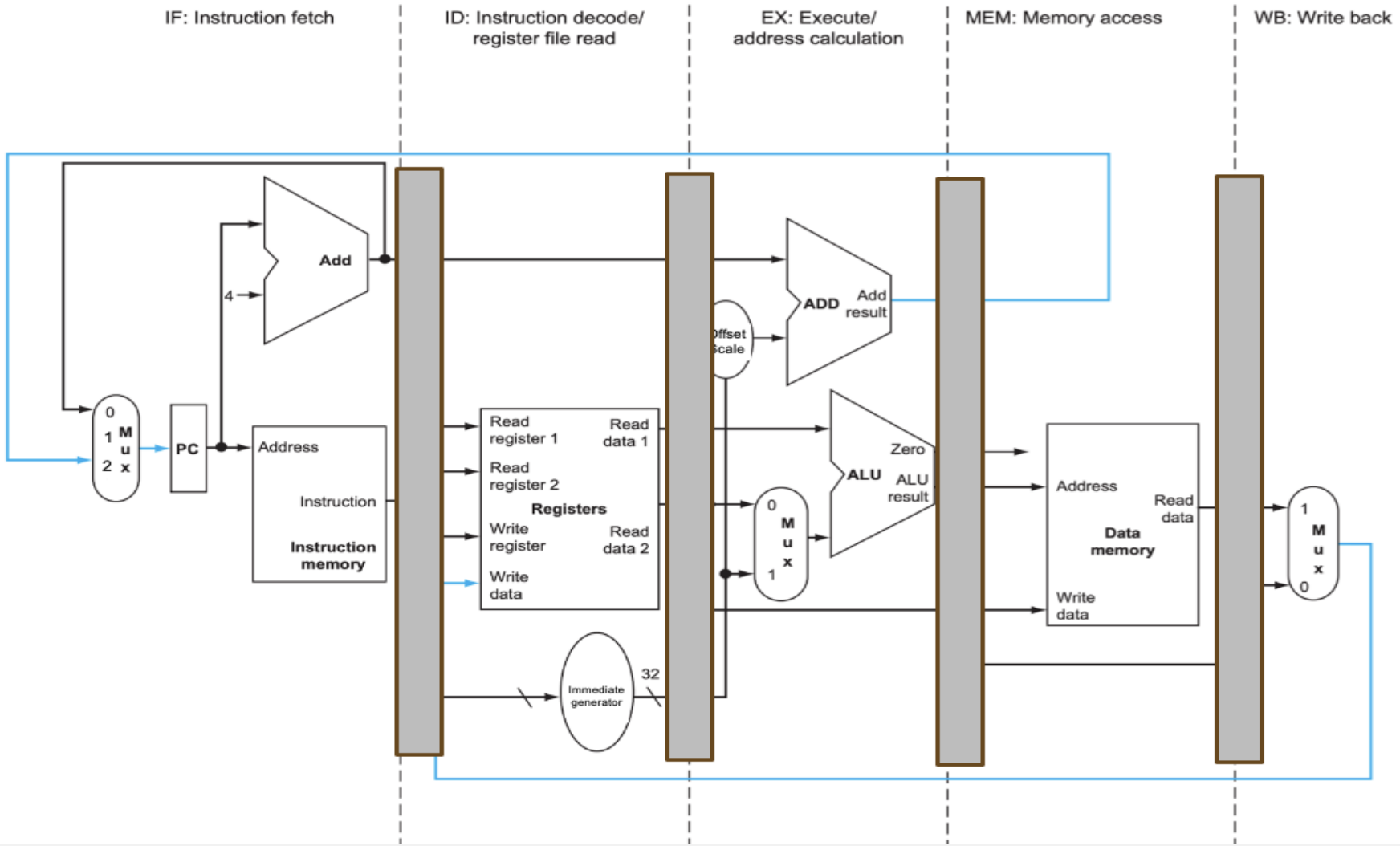
IF/ID

ID/EX

EX/MEM

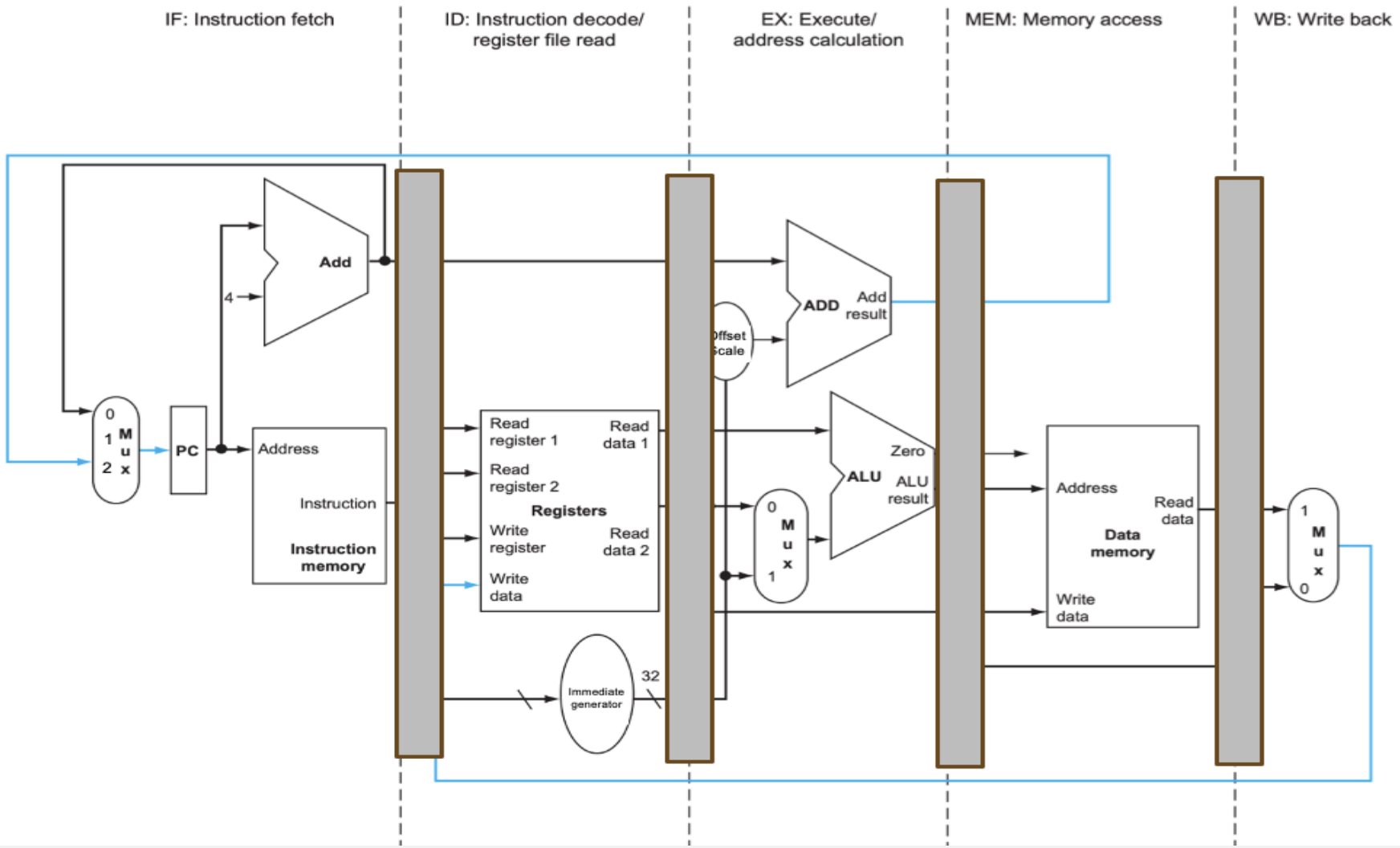
MEM/WB

Cycle 0



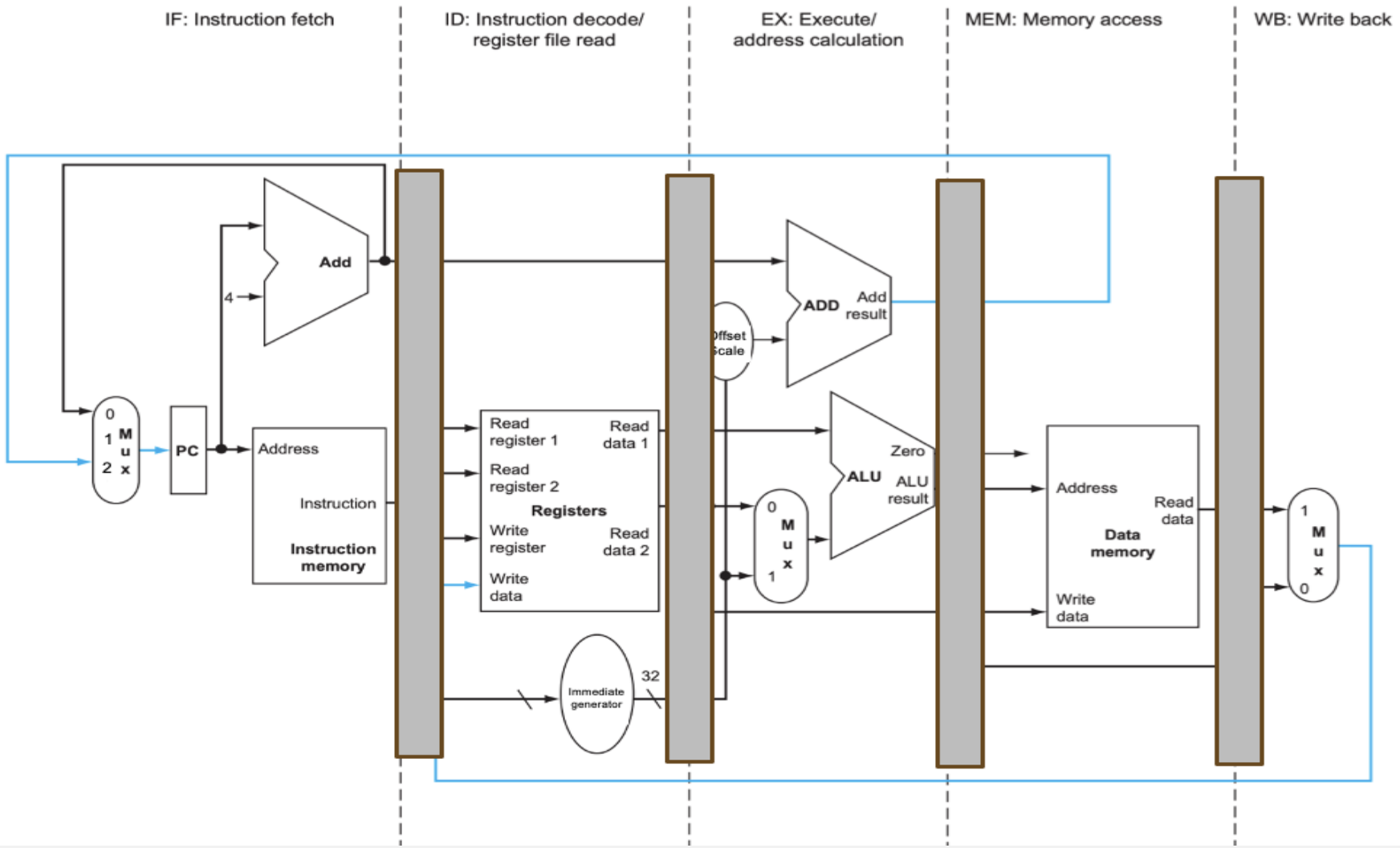
add x5, x3, x4

Cycle 1



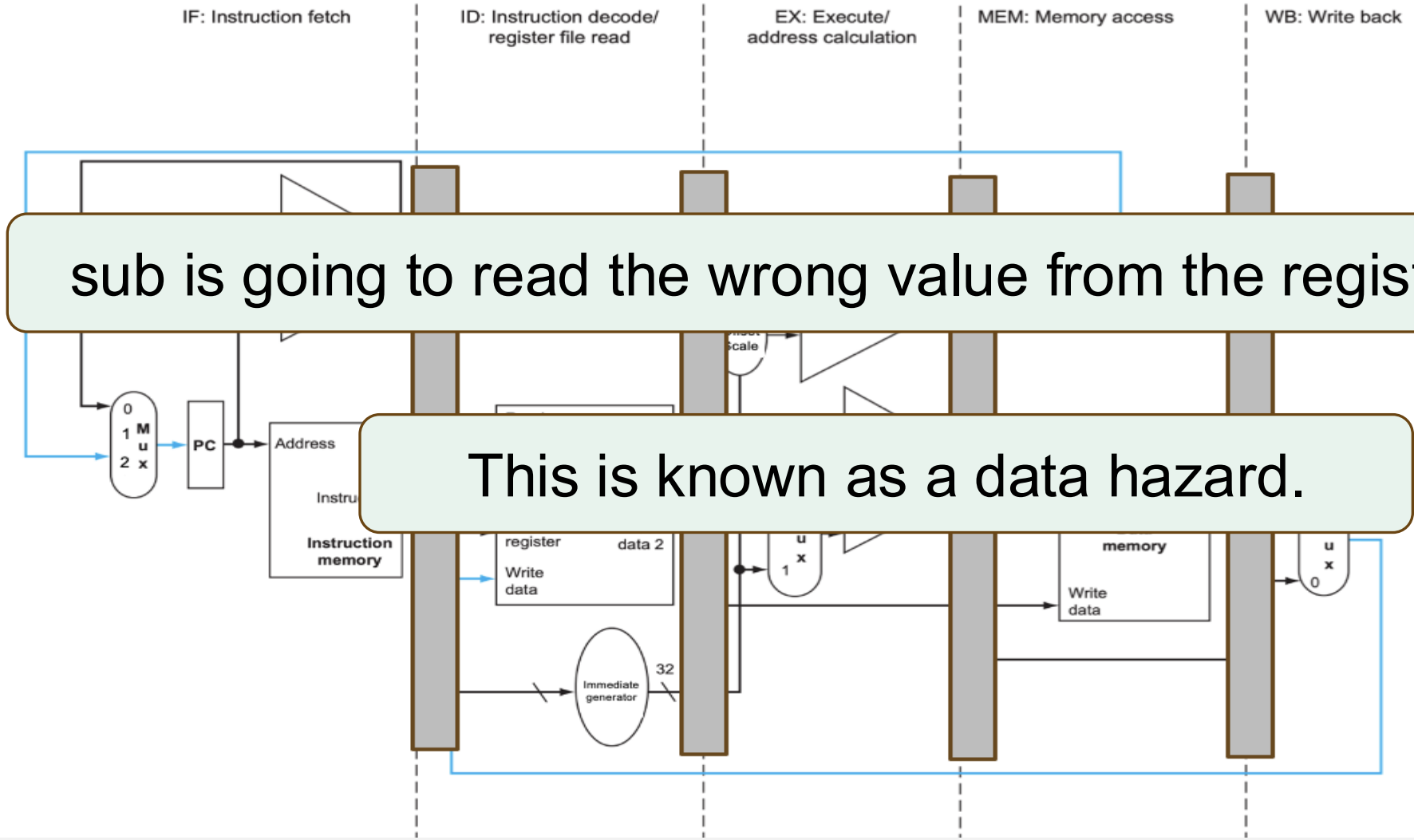
sub x7, x5 x2 add x5, x3, x4

Cycle 2



sub x7, x5 x2 add x5, x3, x4

Cycle 2

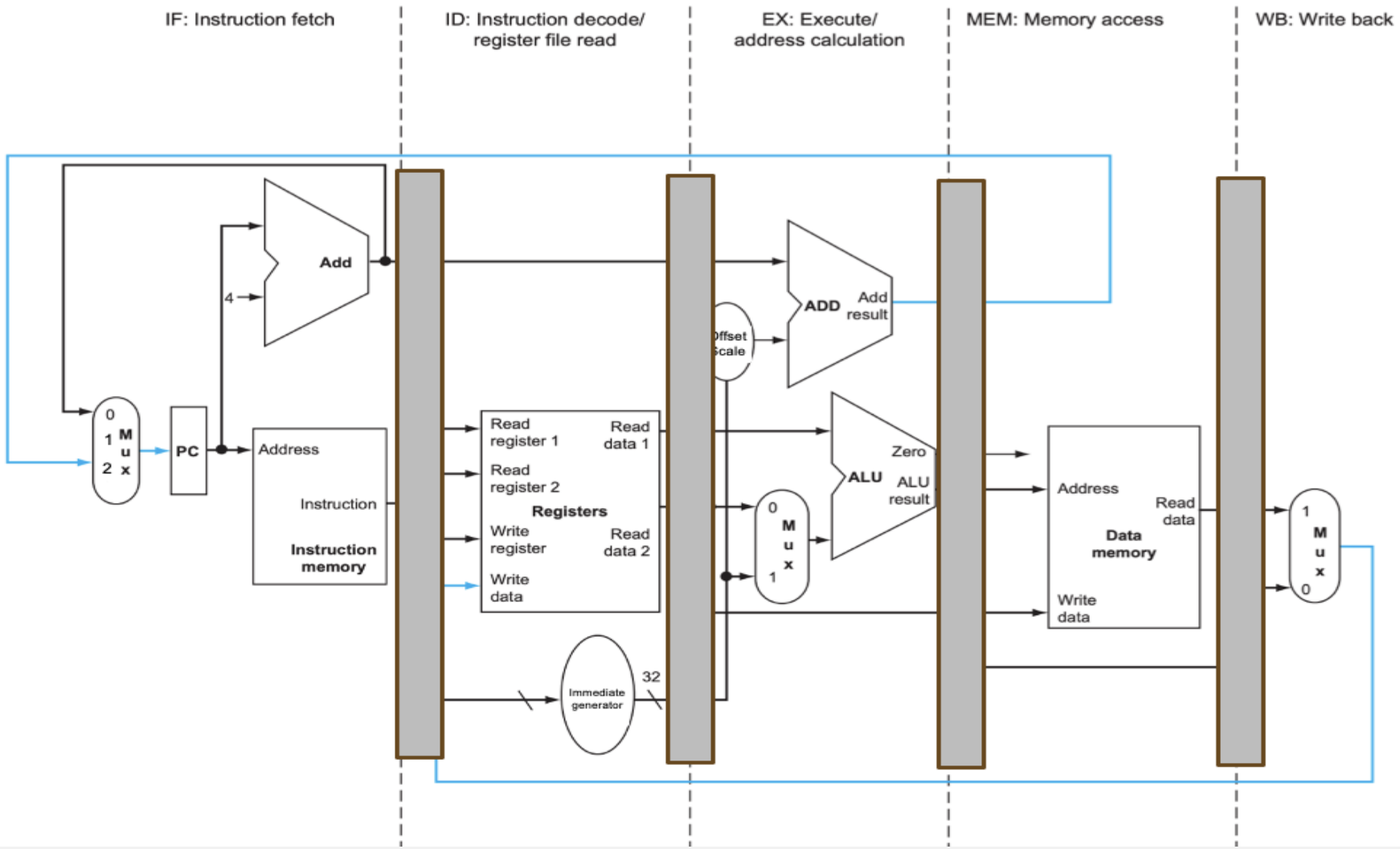


sub x7, x5, x2 add x5, x3, x4

Pipeline Stalls

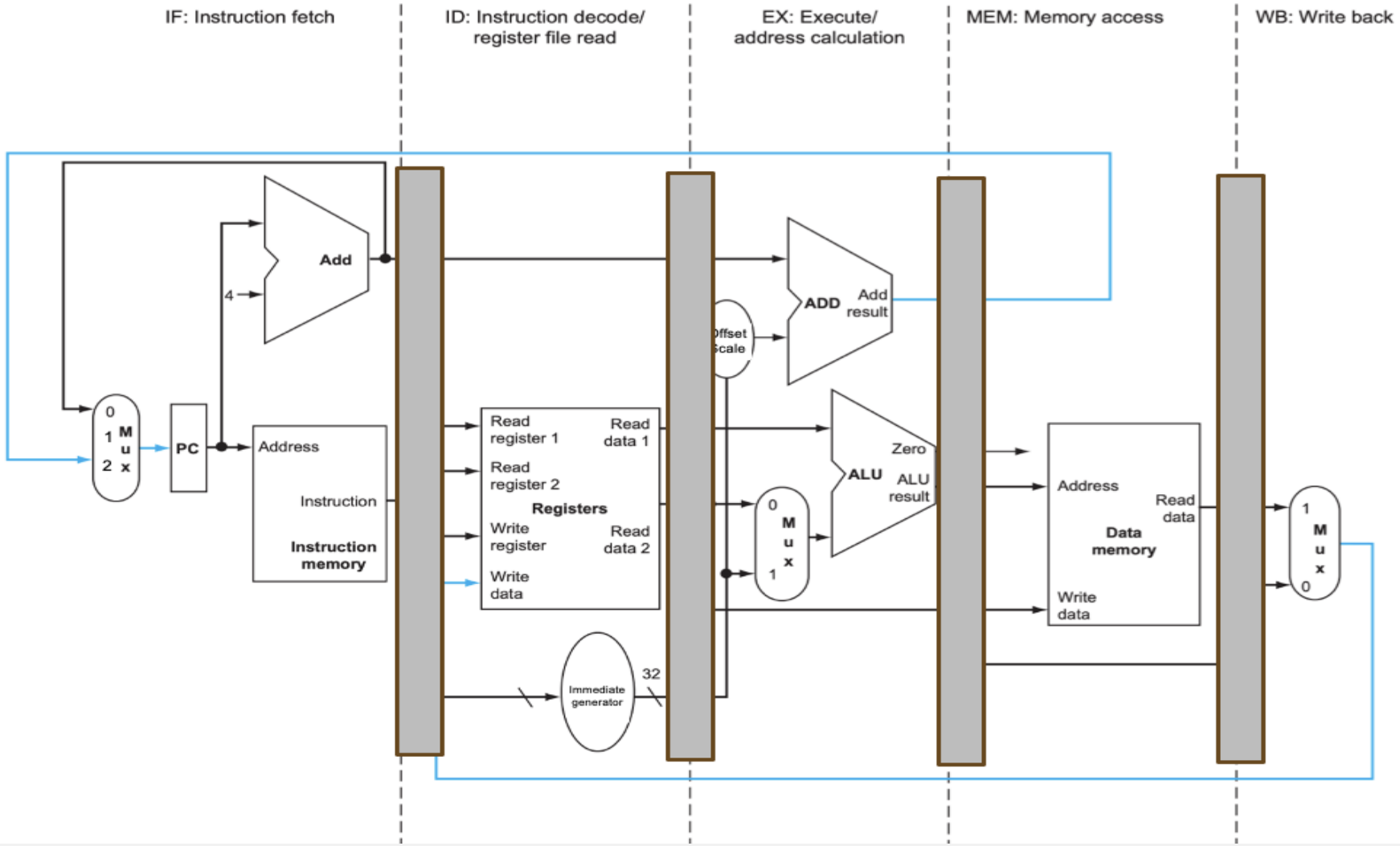
- Also known as
  - bubbles
  - nops (no operation)
- An instruction that does nothing
- Provides time for the correct value to be written to the register file

Cycle 2



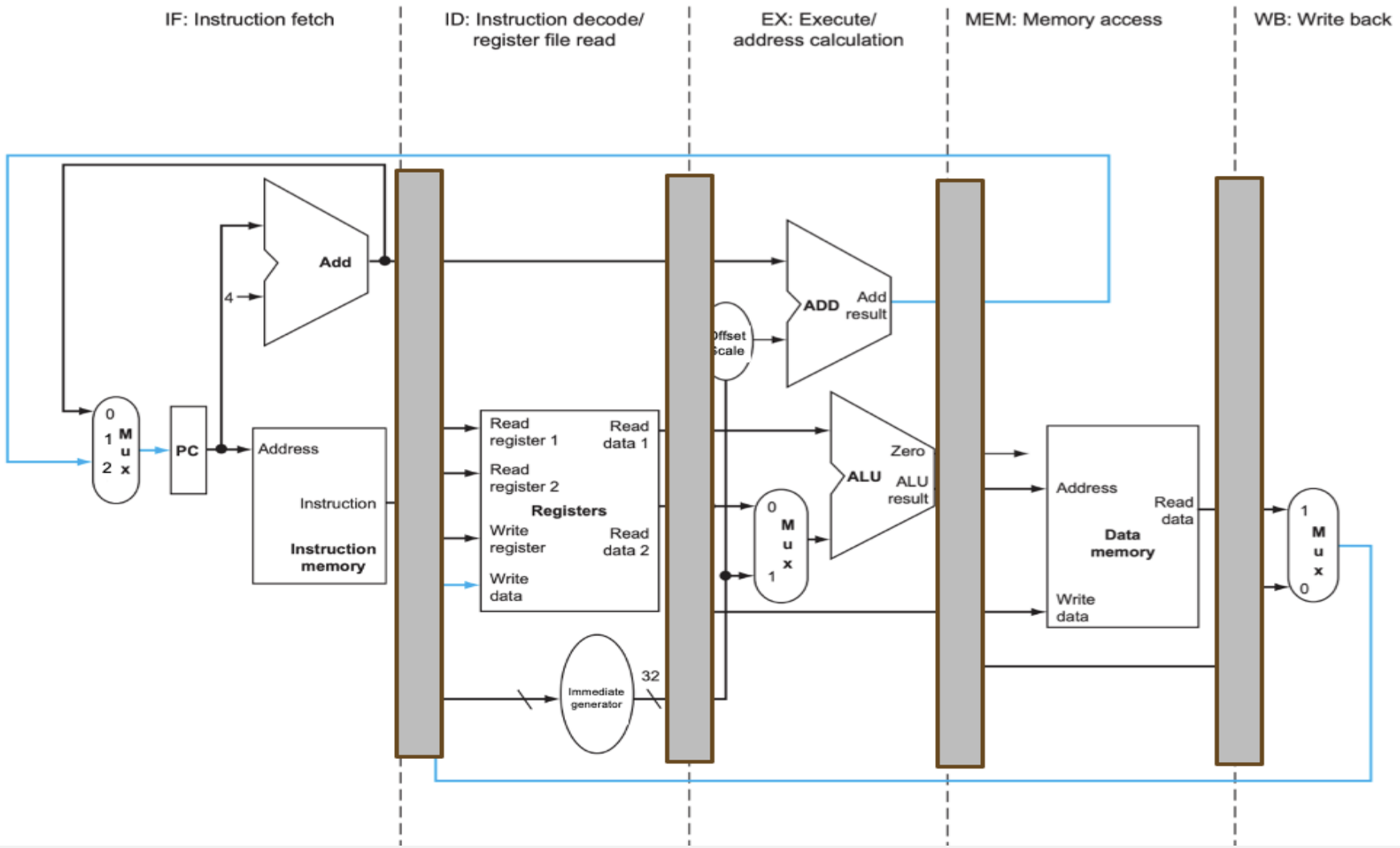
sub x7, x5, x2add x5, x3, x4

Cycle 3



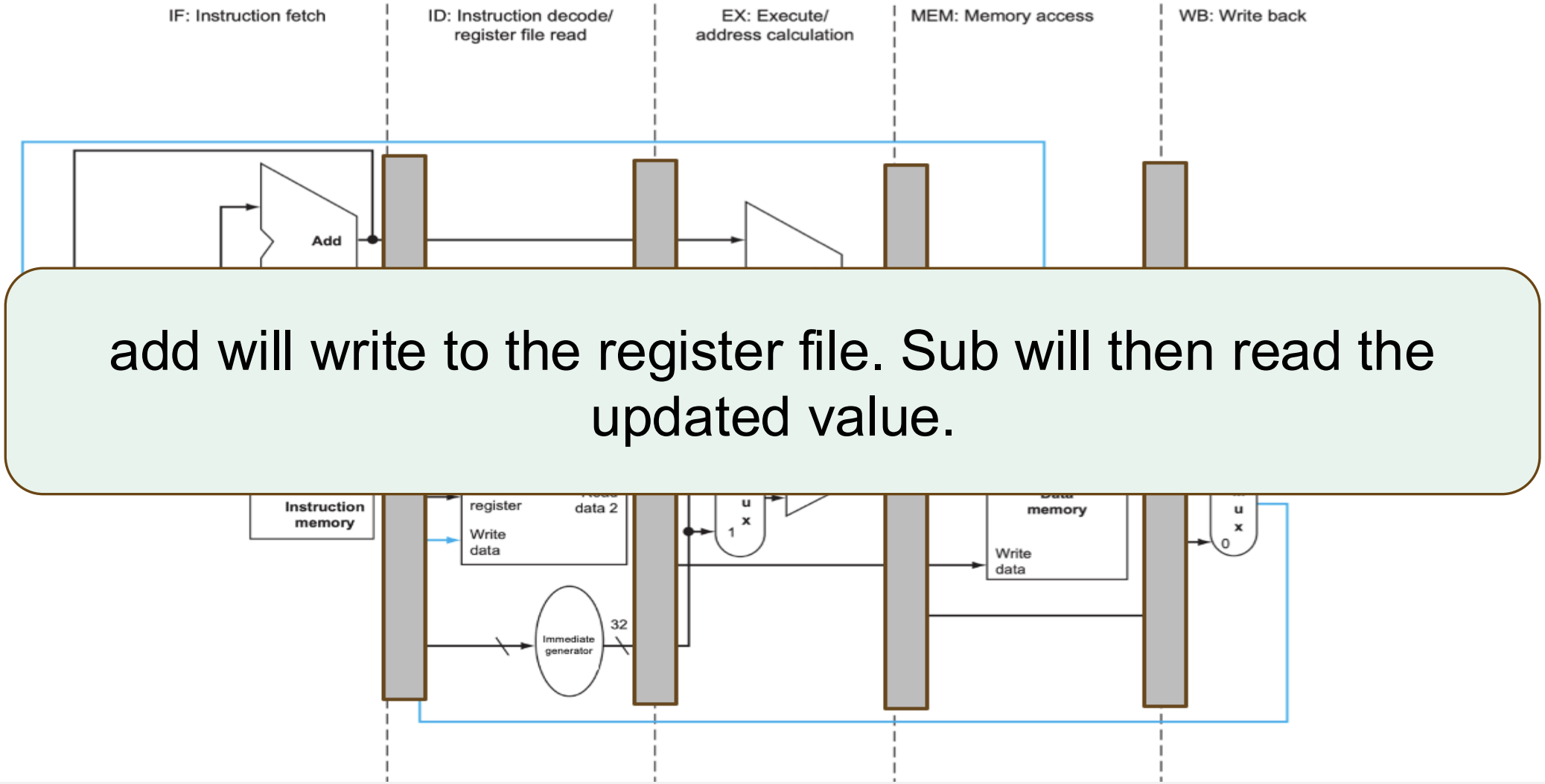
sub x7, x5, x2 ***STALL*** add x5, x3, x4

Cycle 4



sub x7, x5, x2 ***STALL*** ***STALL*** add x5, x3, x4

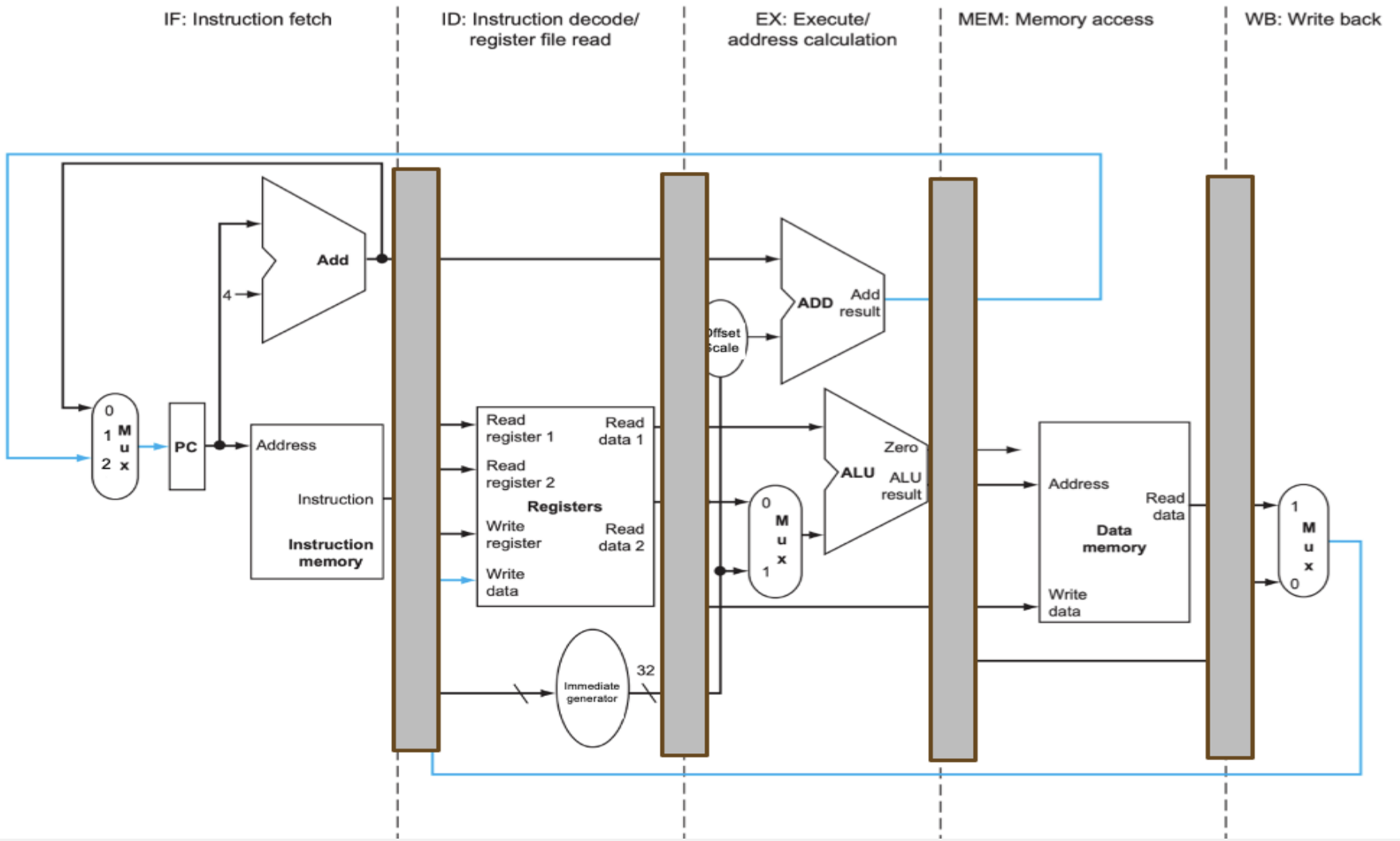
Cycle 4



add will write to the register file. Sub will then read the updated value.

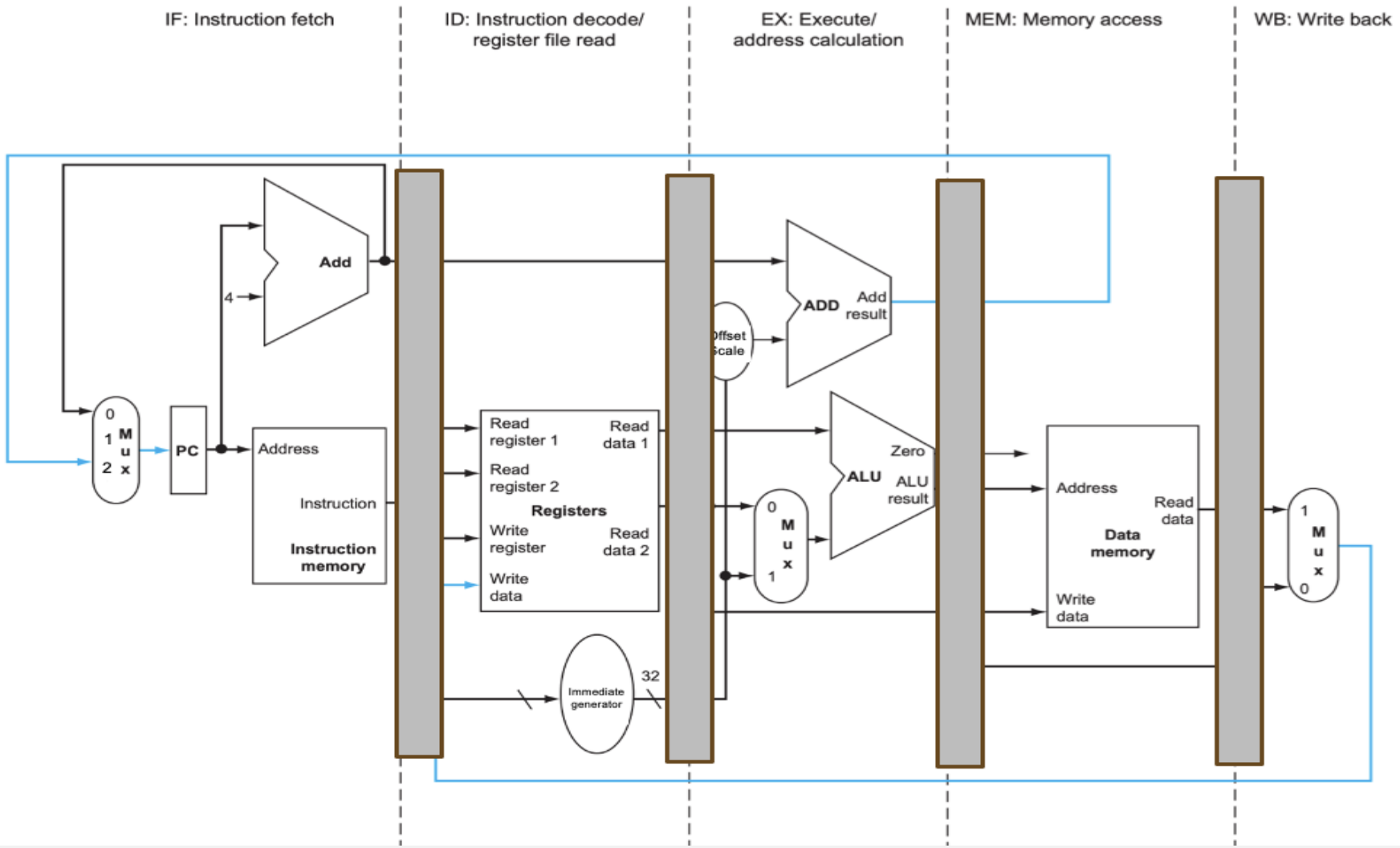
sub x7, x5, x2 ***STALL*** ***STALL*** add x5, x3, x4

Cycle 5



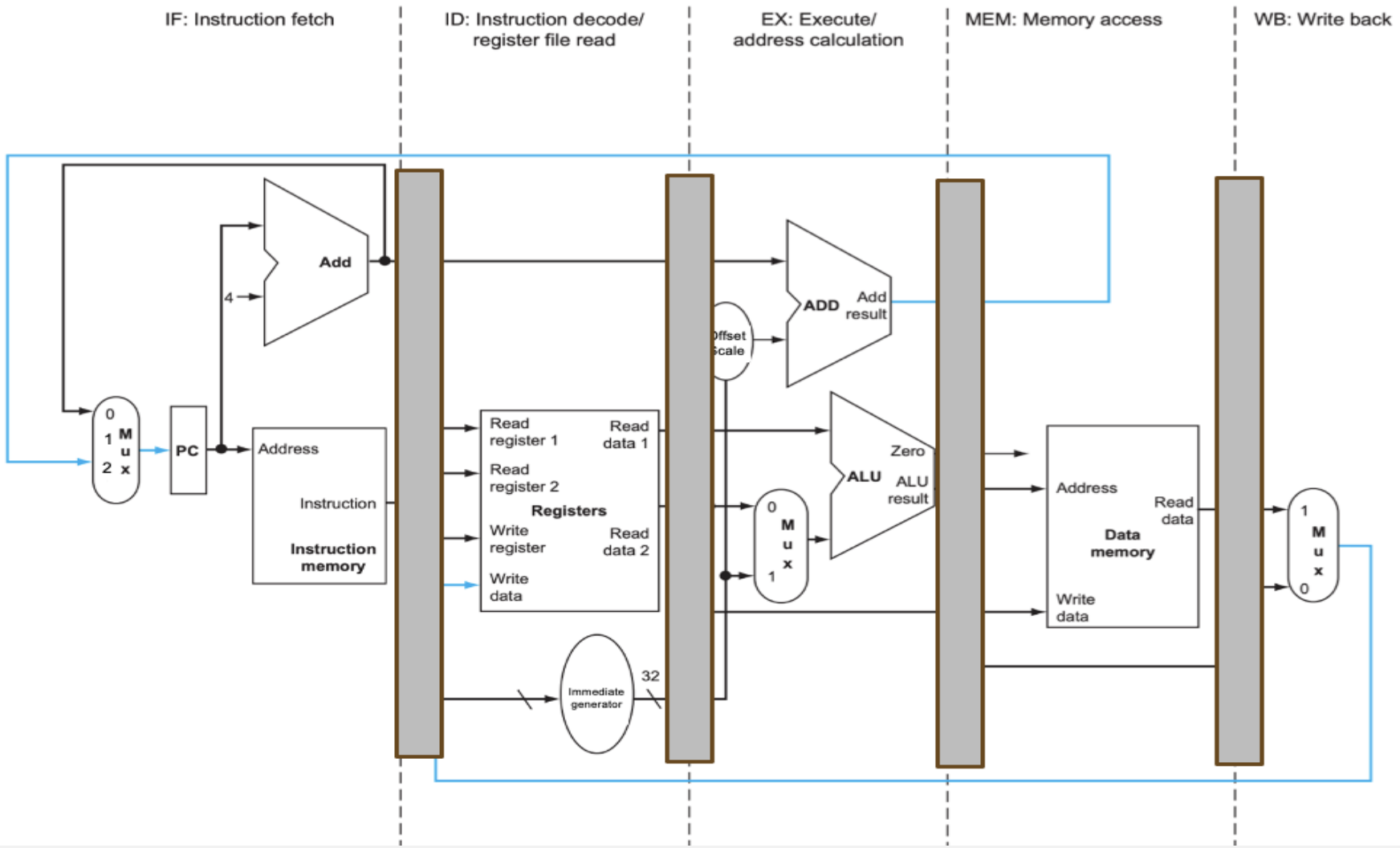
sub x7, x5, x2 ***STALL*** ***STALL***

Cycle 6



sub x7, x5, x2 ***STALL***

Cycle 7



sub x7, x5, x2

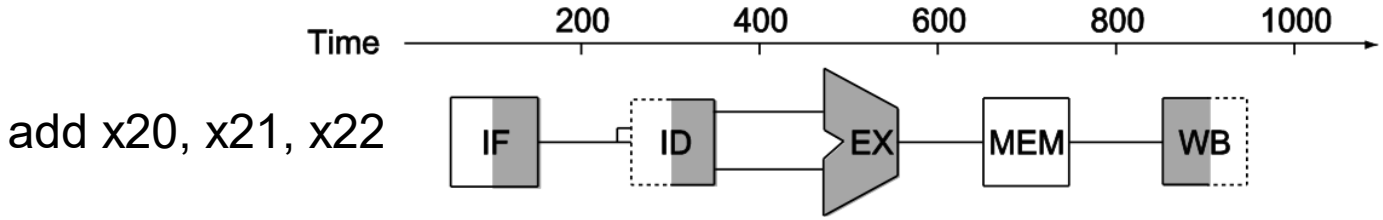
Pipeline Diagram

Instruction	Cycle #													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
add x20, x21, x22	F	D	X	M	W										
sub x28, x20, x15		F	D	D	D	X	M	W							
and x27, x5, x18															
or x9, x28, x3															

sub will remain in the decode stage until add writes back

F = Fetch
D = Decode
X = Execute
M = Memory
W = Writeback

Question from last time: Why can Decode & WriteBack happen in the same clock cycle?



The bold/dark parts are showing when the stage is actively using its inputs / producing outputs within the cycle.

Light parts are when we access memory.

Instruction

Cycle #

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
add x20, x21, x22	F	D	X	M	W										
sub x28, x20, x15		F	D	D	D	X	M	W							

2 stalls

RISC-V design:
Register file **writes** in the **first half** of the cycle.
Allows for **reads** in **second half**

Pipeline Diagram

Instruction	Cycle #													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
add x20, x21, x22	F	D	X	M	W										
sub x28, x20, x15		F	D	D	D	X	M	W							
and x27, x5, x18			F	F	F	D	X	M	W						
or x9, x28, x3						F	D	D	X	M	W				

sub will remain in the decode stage until add writes back

and will remain in the fetch stage while sub is in the decode state

F = Fetch
D = Decode
X = Execute
M = Memory
W = Writeback

Pipeline Diagram

Instruction	Cycle #													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
add x20, x21, x22	F	D	X	M	W										
sub x28, x20, x15		F	D	D	D	X	M	W							
and x27, x5, x18			F	F	F	D	X	M	W						
or x9, x28, x3						F	D	D	X	M	W				

or will remain in the decode stage until *sub* writes back

F = Fetch
D = Decode
X = Execute
M = Memory
W = Writeback

Performance Metric: CPI (Cycles per instruction)

$$CPI = \frac{total\ number\ of\ cycles\ to\ execute\ program}{number\ of\ instructions\ in\ program}$$

Performance Metric: CPI (Cycles per instruction)

$$CPI = \frac{total\ number\ of\ cycles\ to\ execute\ program}{number\ of\ instructions\ in\ program}$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
add x20, x21, x22	F	D	X	M	W										
sub x28, x20, x15		F	D	D	D	X	M	W							
and x27, x5, x18			F	F	F	D	X	M	W						
or x9, x28, x3						F	D	D	X	M	W				

Performance Metric: CPI (Cycles per instruction)

$$CPI = \frac{total\ number\ of\ cycles\ to\ execute\ program}{number\ of\ instructions\ in\ program}$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
add x20, x21, x22	F	D	X	M	W										
sub x28, x20, x15		F	D	D	D	X	M	W							
and x27, x5, x18			F	F	F	D	X	M	W						
or x9, x28, x3						F	D	D	X	M	W				

$$CPI = \frac{11}{4} = 2.75$$

Ideal CPI (Cycles per instruction) = 1

$$CPI = \frac{\#\ of\ cycles}{\#\ of\ instructions}$$

Ideal CPI (Cycles per instruction) = 1

In the ideal case, we will have one instruction finishing on every cycle.
If one instruction finishes every cycle, the total number of cycles needed to execute the program is (4 + # of instructions).

$$CPI = \frac{\#\ of\ cycles}{\#\ of\ instructions} = \frac{4+\#\ of\ instructions}{\#\ of\ instruciotns}$$

(this equation assumes no stalls)

Stalls are going to hurt our CPI. We are going to see later in the lecture how to reduce the number of stalls.

Pipeline Diagram with Stalls

1. Fill out the pipeline diagram for the following set of instructions.
2. What is the CPI?
3. What is the speedup compared to a single-cycle datapath?
   - Assume pipelined clock period = 220ps and single-cycle clock period = 820ps

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
add x3, x2, x2															
or x4, x2, x5															
and x9, x4, x10															
sub x11, x4, x14															
add x11, x15, x18															

Recall: Data Hazards

Data hazard: instruction depends on previous result that is still in the pipeline

Problem: When a register source is needed from a later stage of the pipeline before it is written.

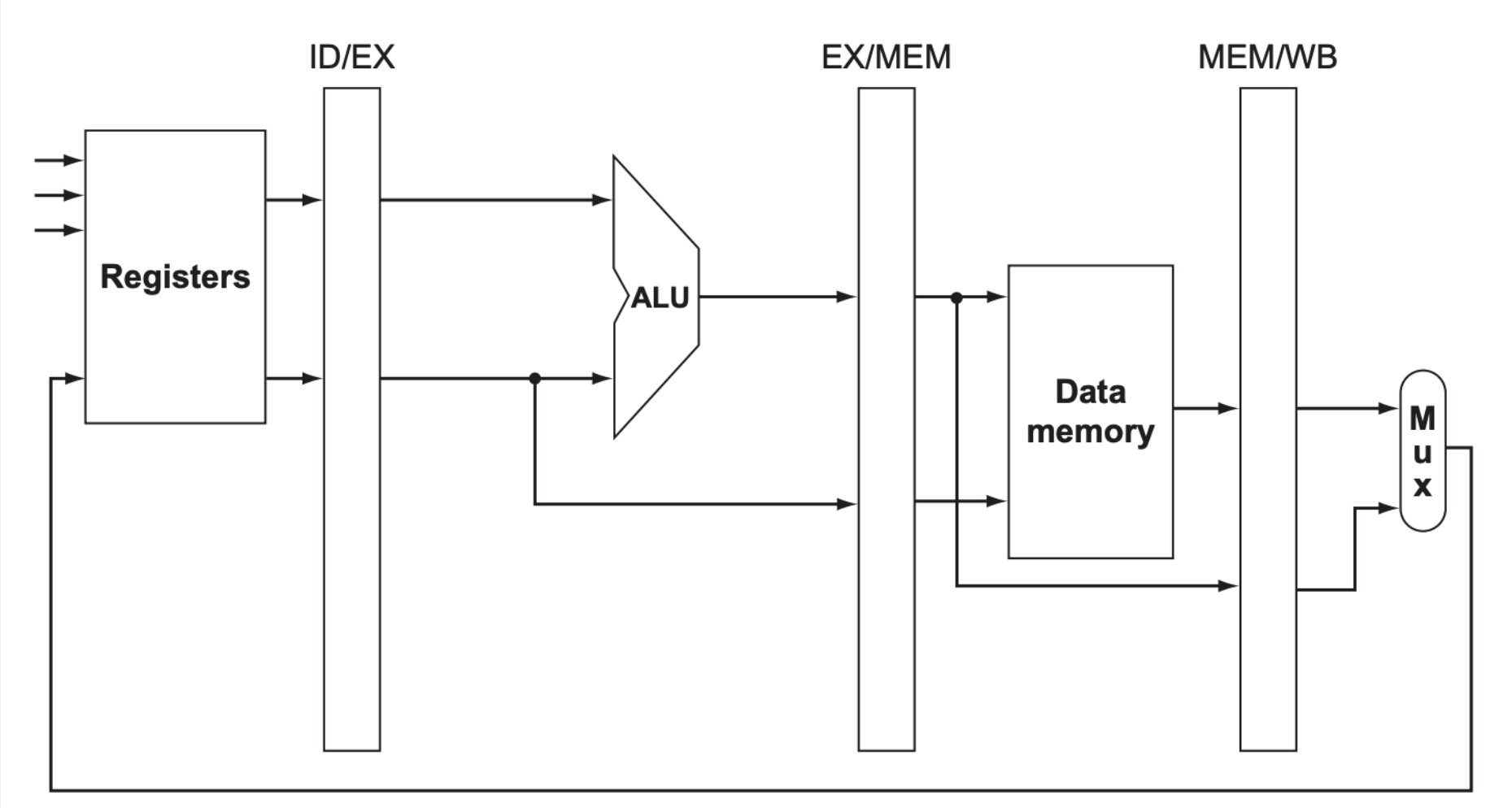
Solutions:

1. Program around it. One could document the weird semantics-- "You can't reference the destination register of an instruction in the immediately following instruction." Would make make assembly language even harder to understand. Would expose the pipeline, once again making future improvements difficult to implement while maintaining code compatibility.

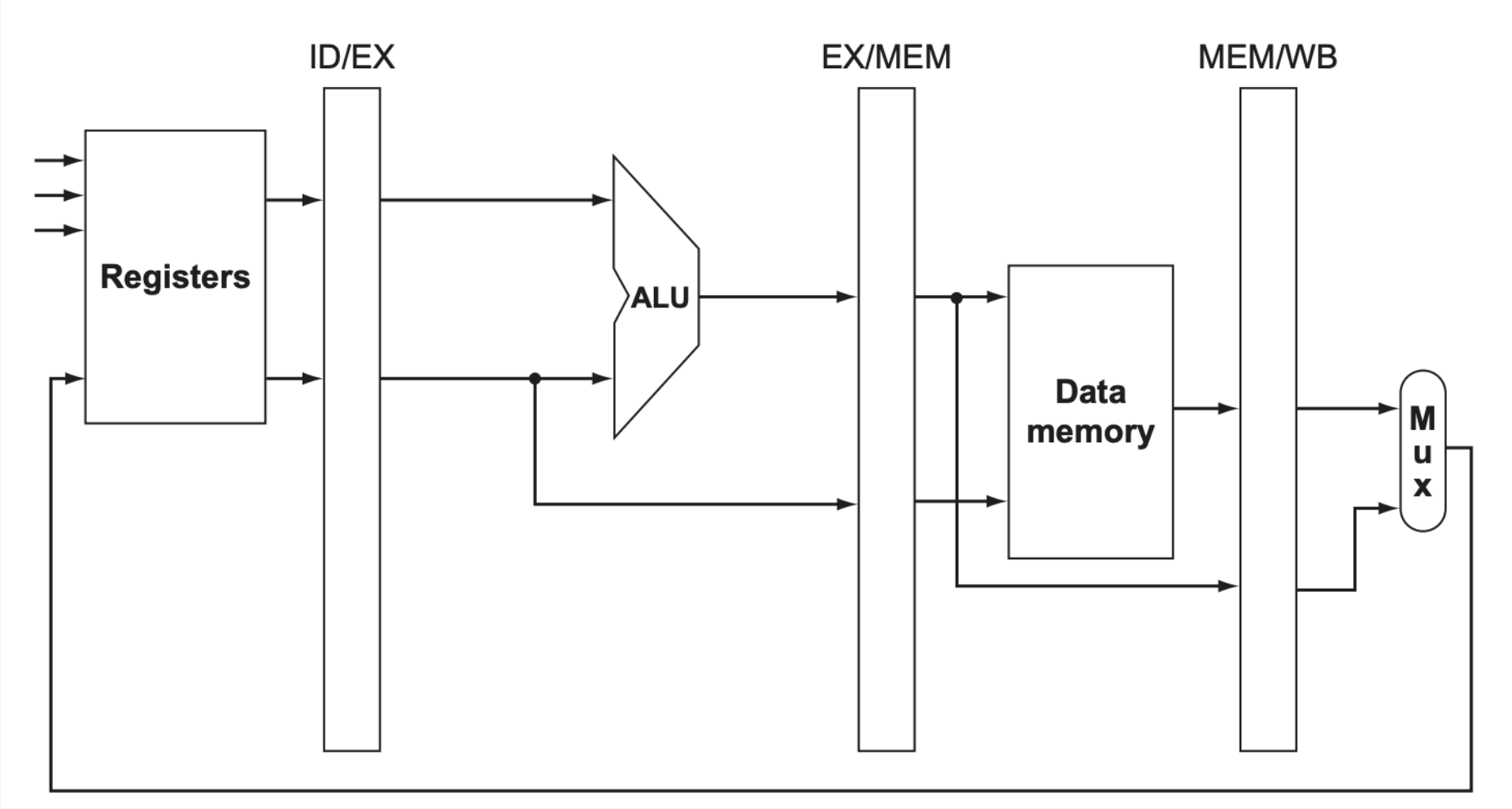
2. **Hardware bypass multiplexers.**

SOURCE BYPASSING

Simplified datapath to focus on forwarding for r-format



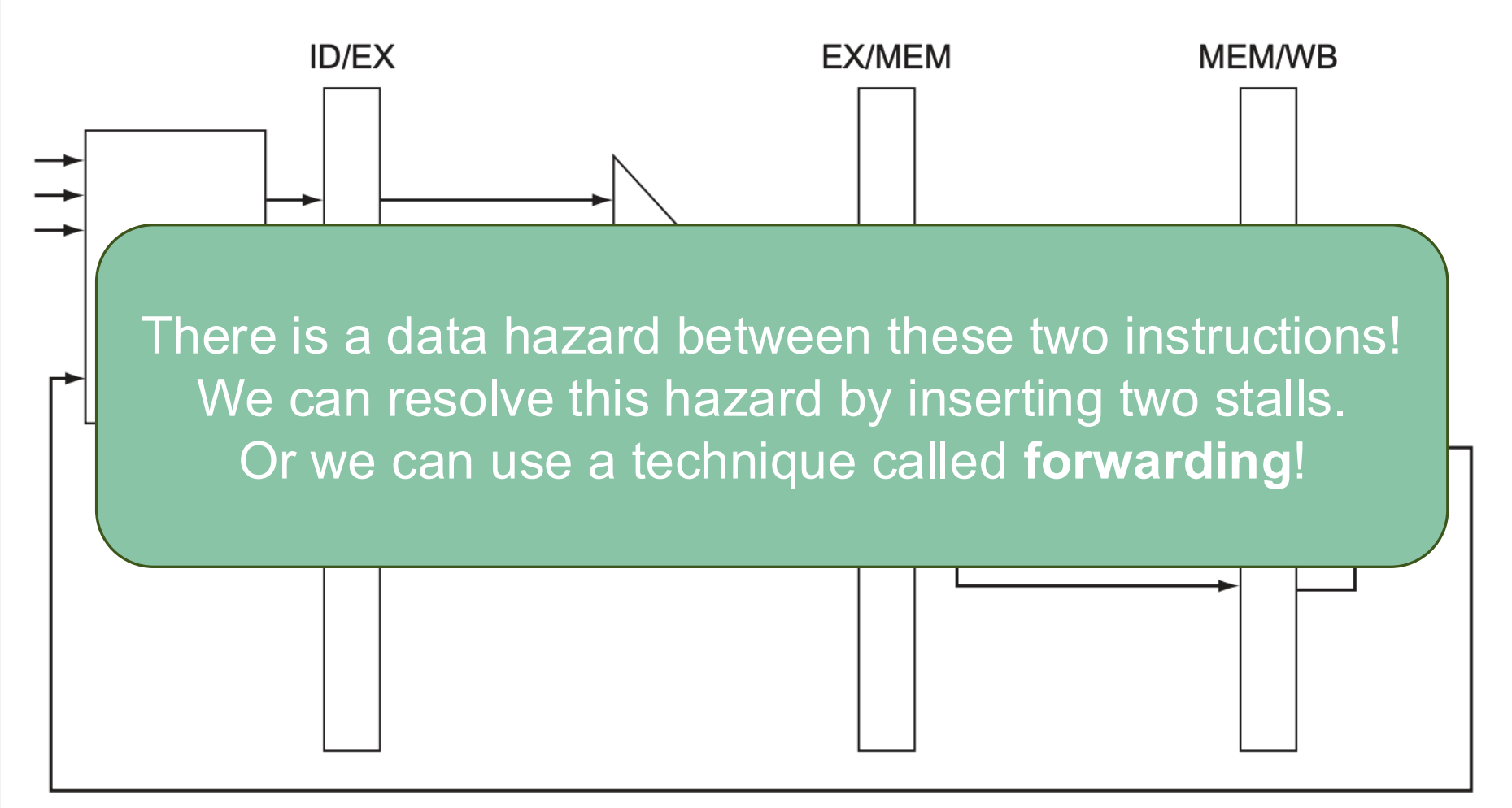
Forwarding rs1



sub x7, x5, x2

add x5, x3, x4

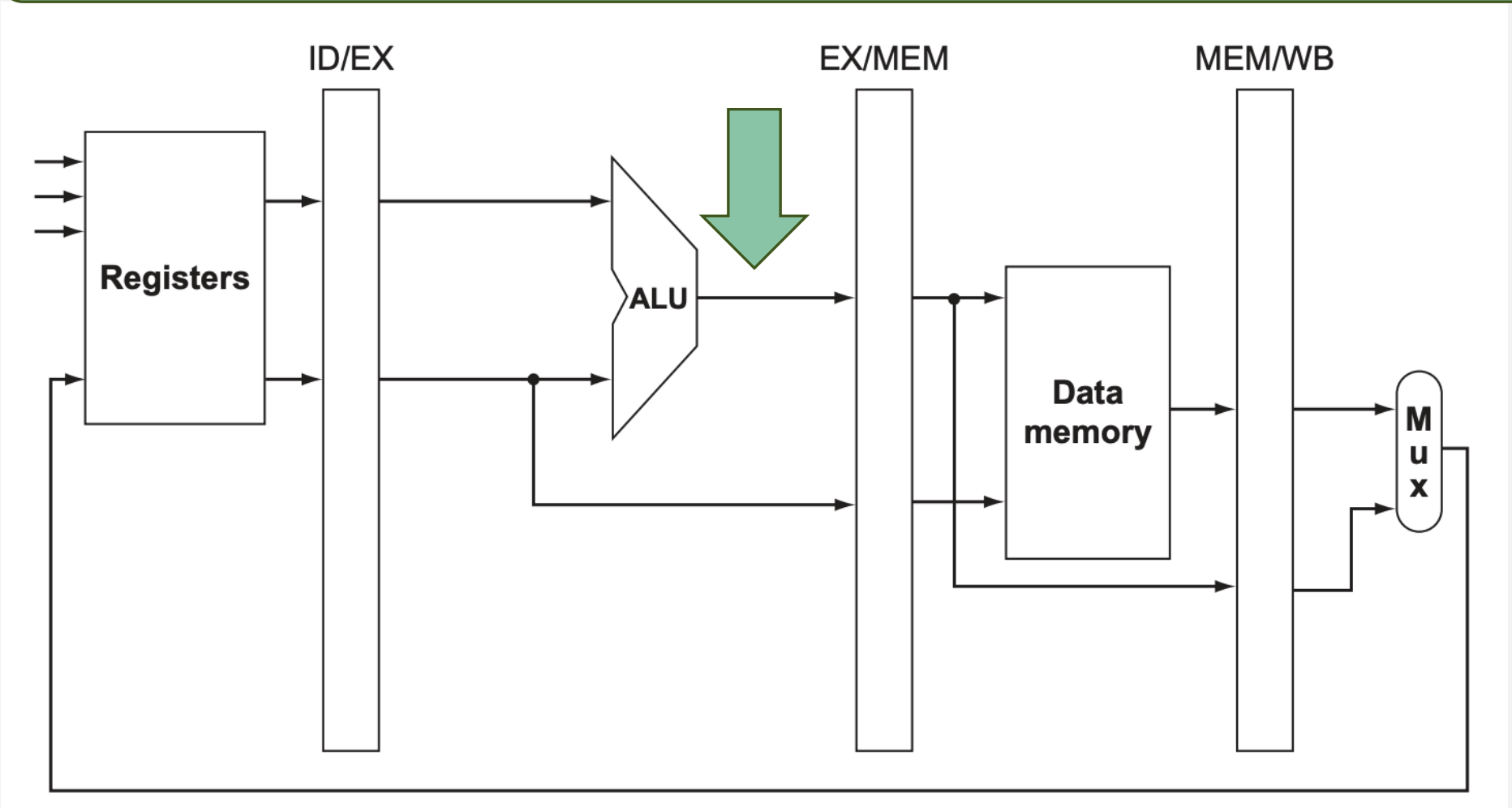
Forwarding rs1



sub x7, x5, x2

add x5, x3, x4

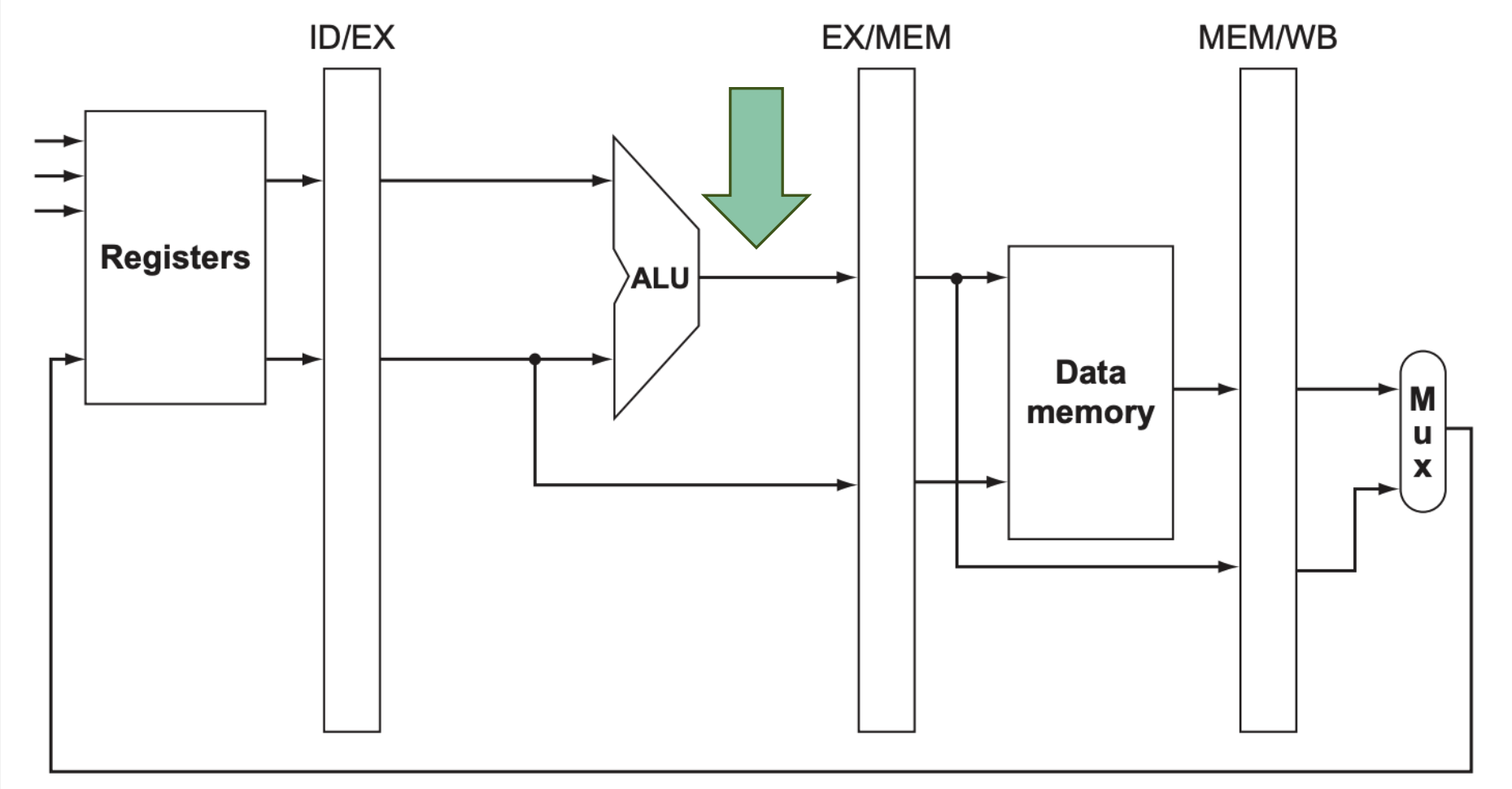
The value that is to be written to x5 is available at the ALU output



sub x7, x5, x2

add x5, x3, x4

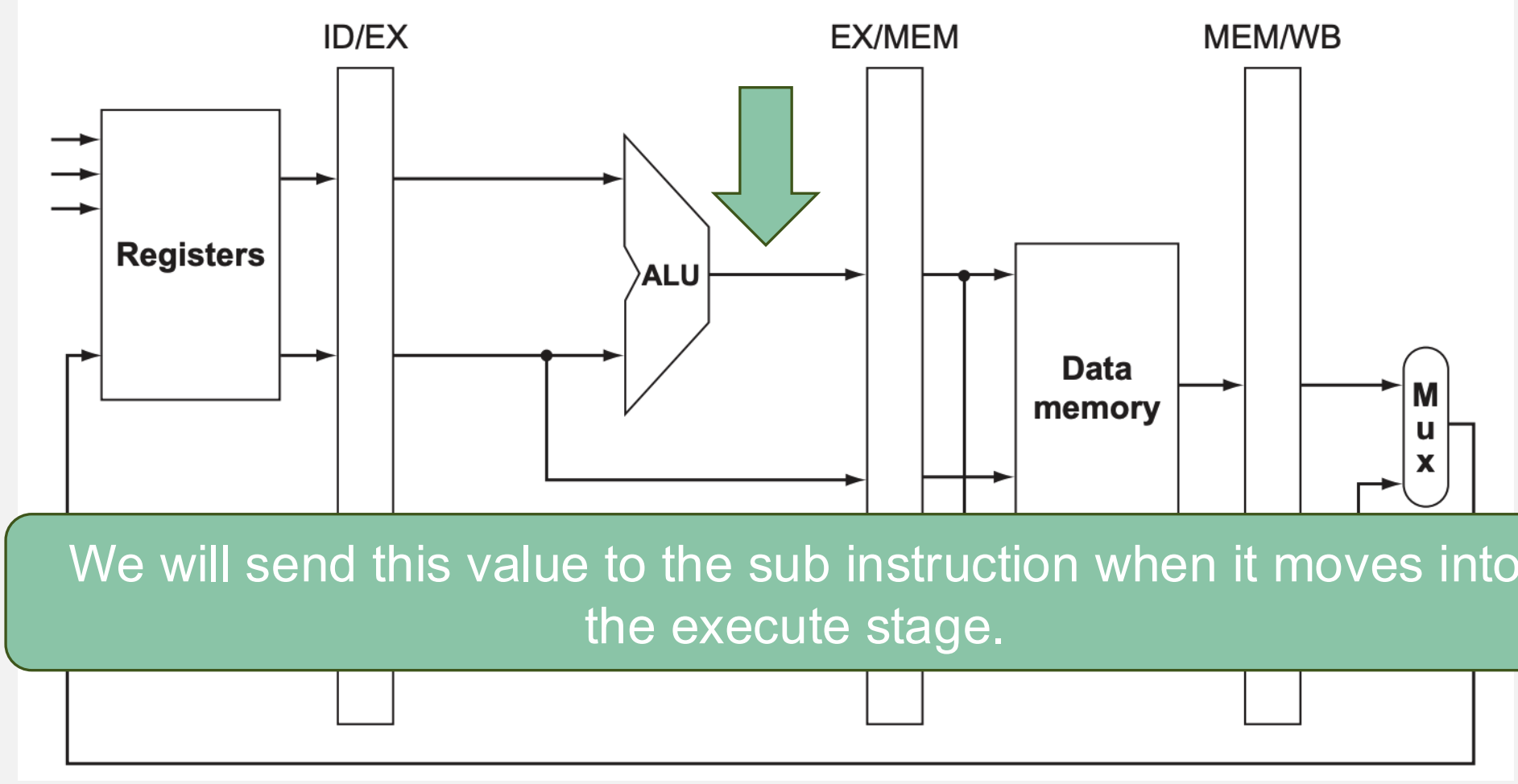
Instead of waiting for this value to be written back to the register file, we can send it back to the sub instruction!



sub x7, x5, x2

add x5, x3, x4

Instead of waiting for this value to be written back to the register file, we can send it back to the sub instruction!



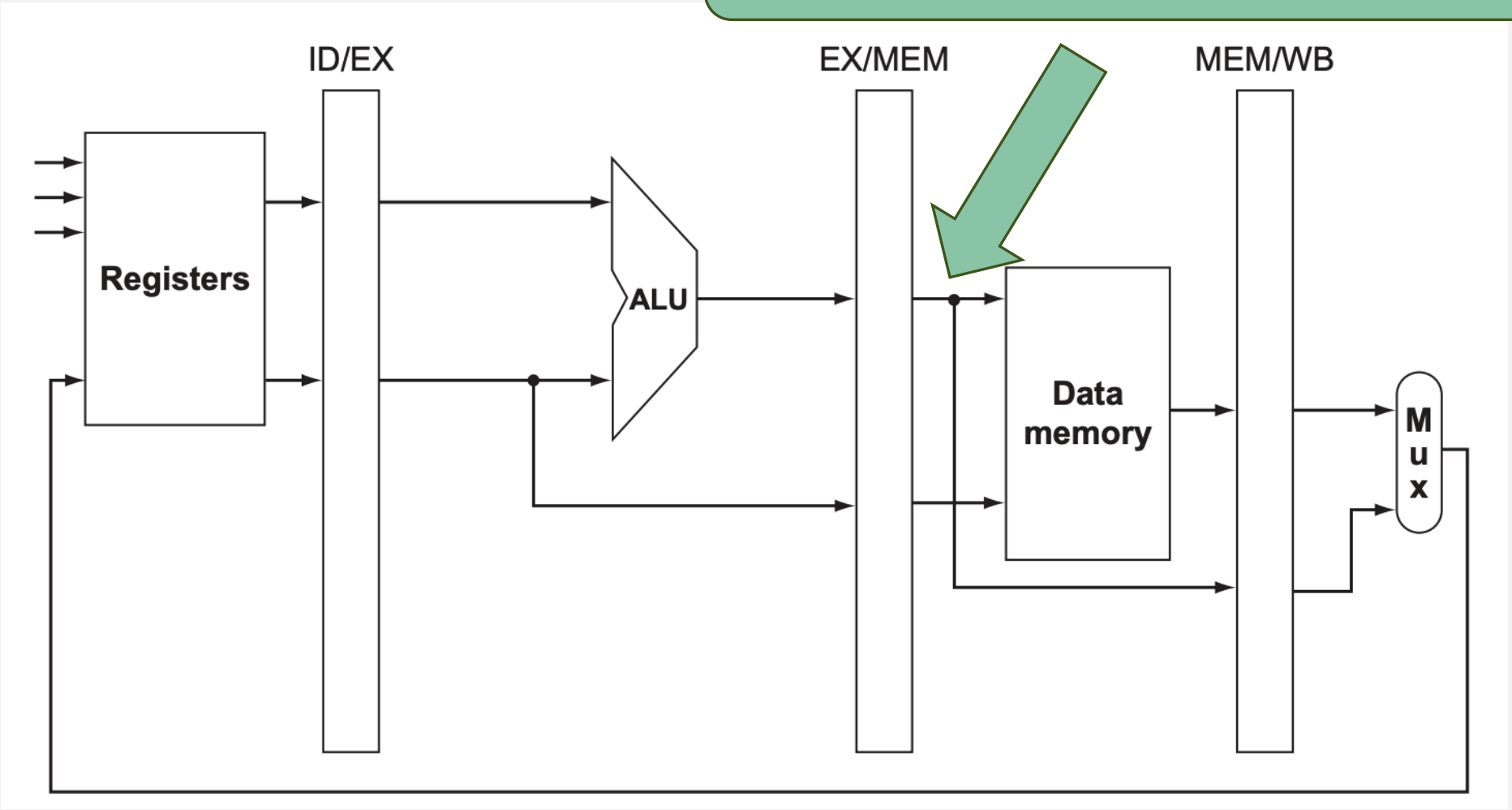
We will send this value to the sub instruction when it moves into the execute stage.

sub x7, x5, x2

add x5, x3, x4

Forwarding rs1

The value to be written to x5 is here

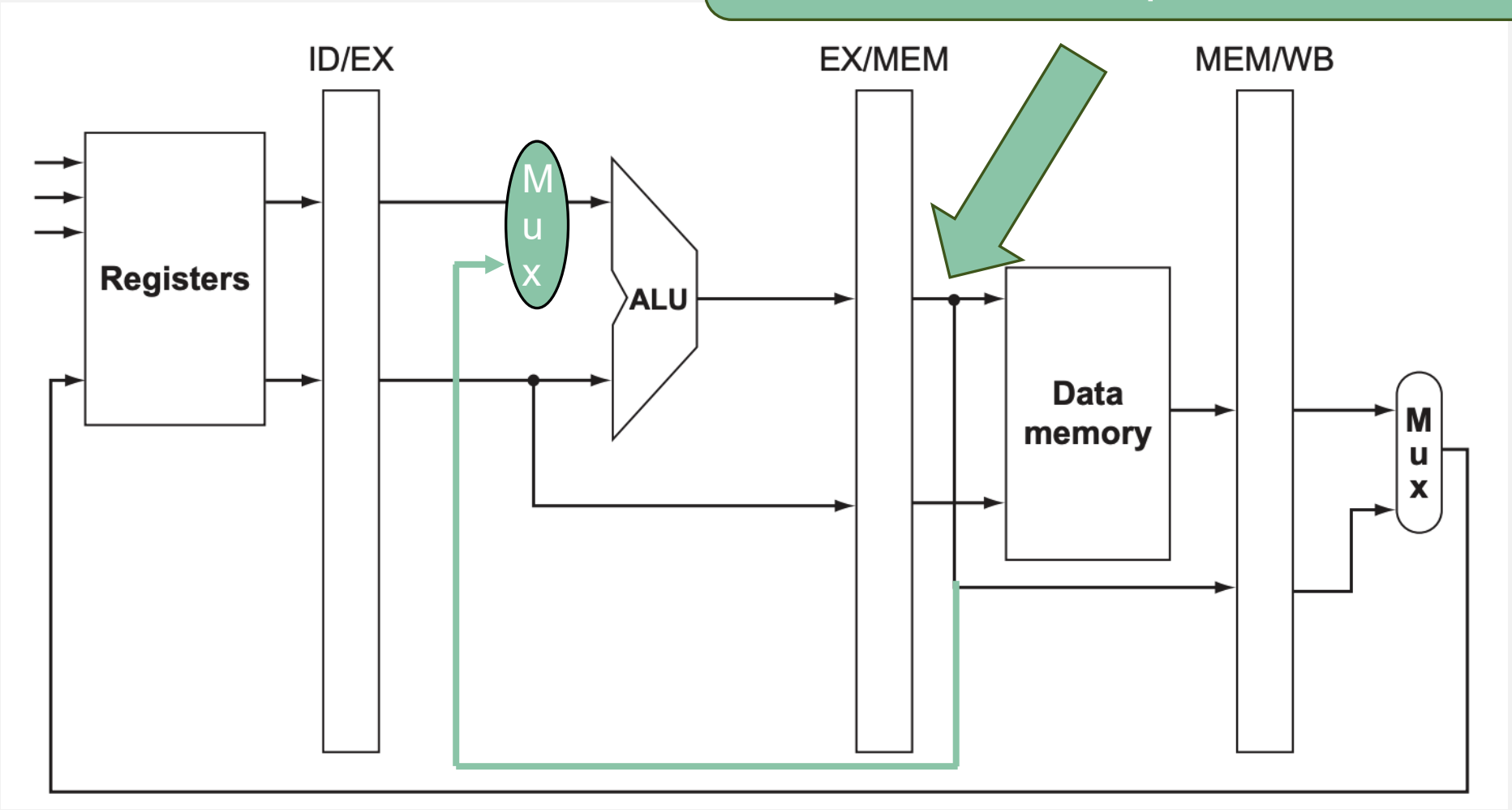


sub x7, x5, x2

add x5, x3, x4

Forwarding rs1

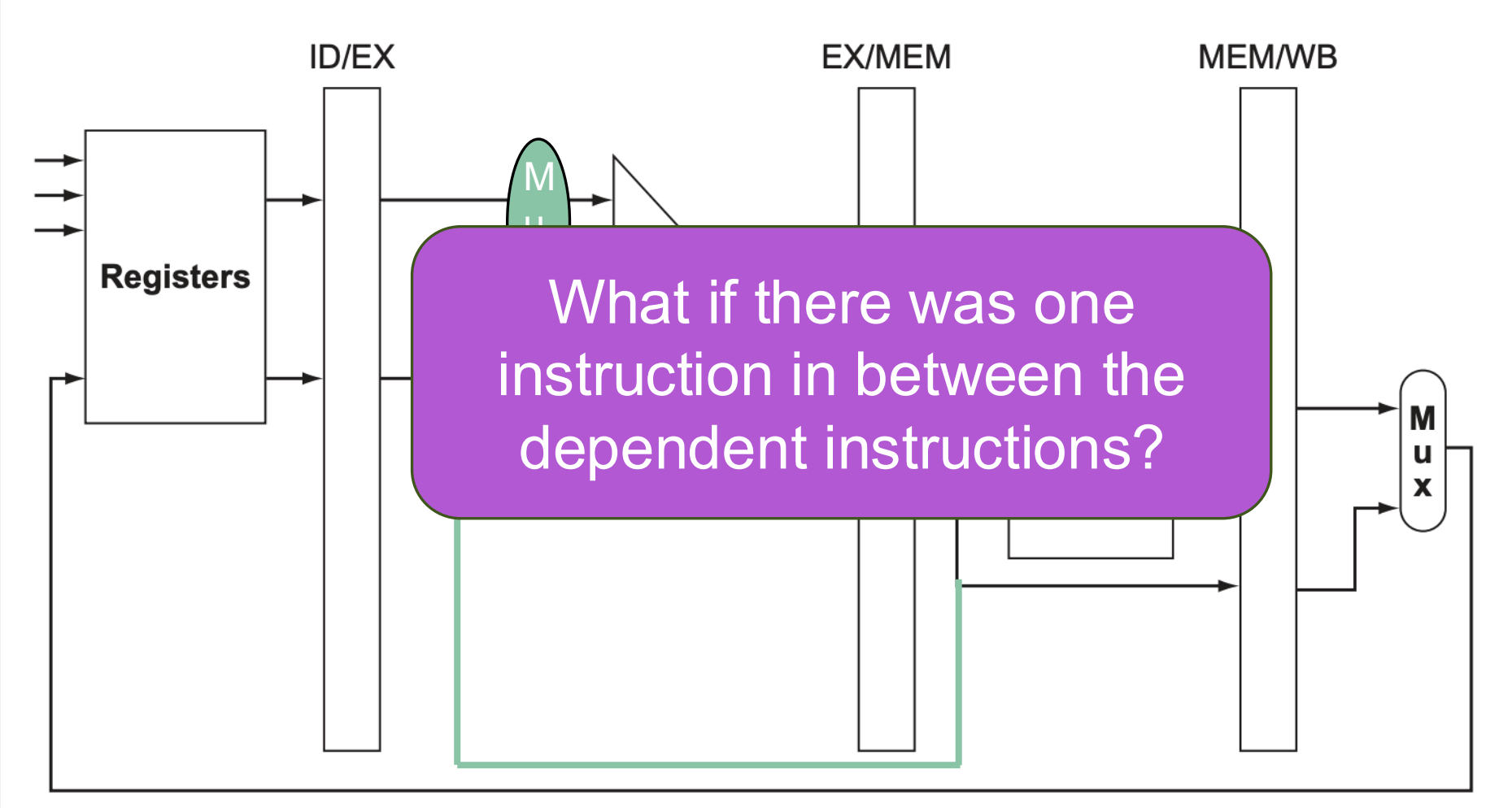
We are forwarding the value to be written to x5 to the dependent instruction.



sub x7, x5, x2

add x5, x3, x4

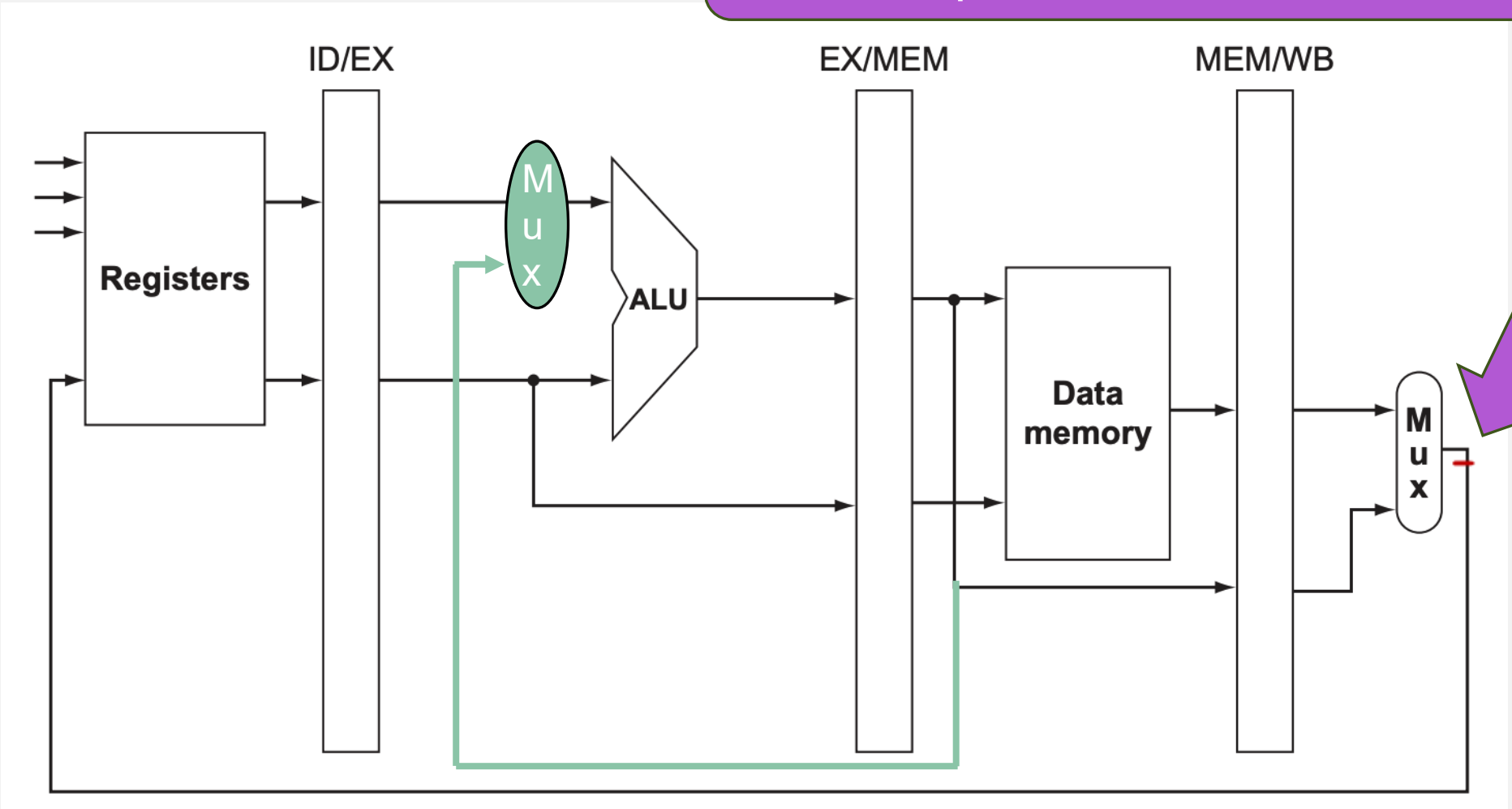
Forwarding rs1



sub x7, x5, x2 or x8, x9, x10 add x5, x3, x4

Forwarding rs1

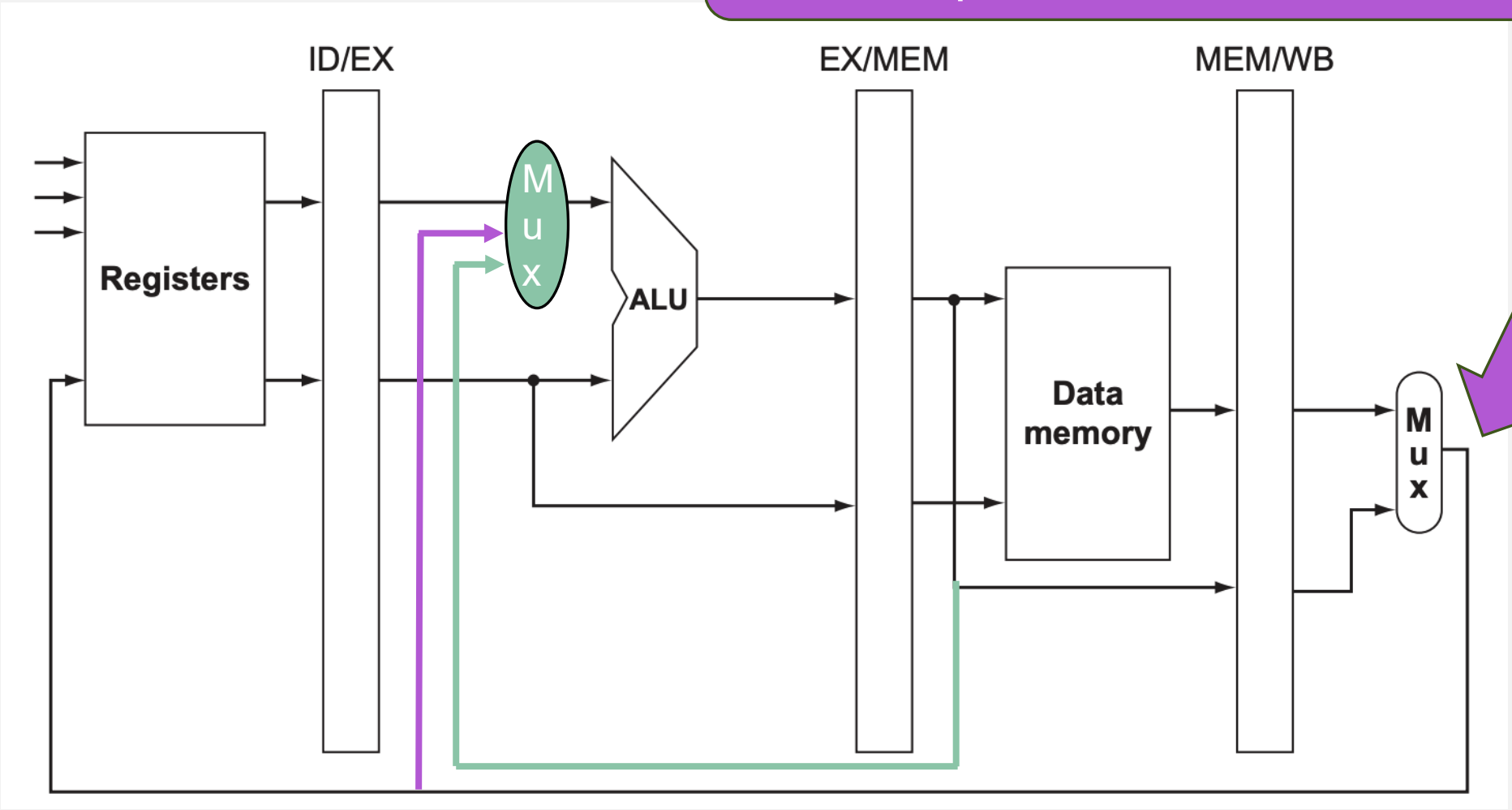
The value to be written to x5 is at the output of the writeback mux.



sub x7, x5, x2 or x8, x9, x10 add x5, x3, x4

Forwarding rs1

The value to be written to x5 is at the output of the writeback mux.

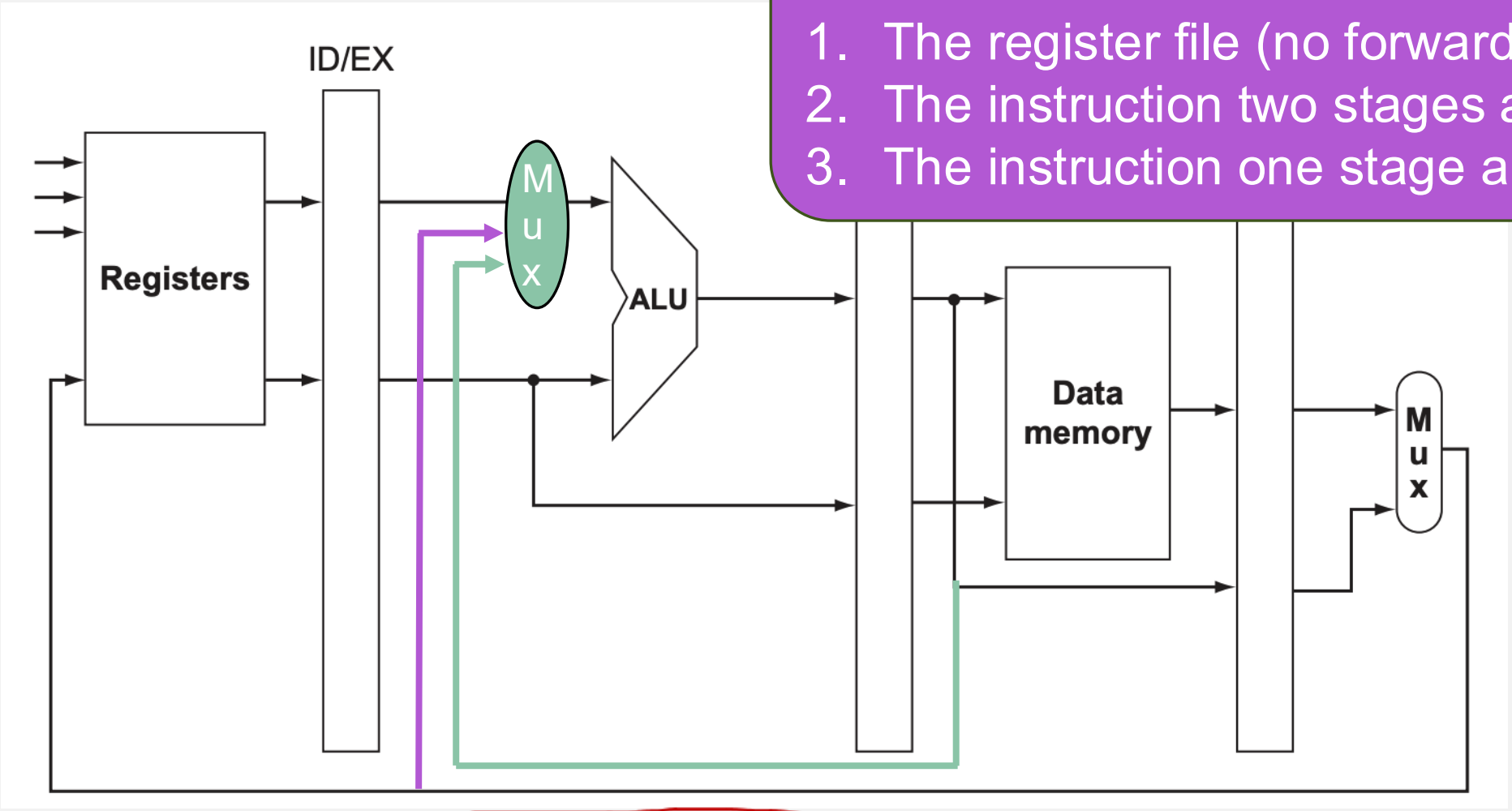


sub x7, x5, x2 or x8, x9, x10 add x5, x3, x4

Forwarding rs1

Now, we can choose if the top input of the ALU should come from

1. The register file (no forwarding)
2. The instruction two stages ahead
3. The instruction one stage ahead



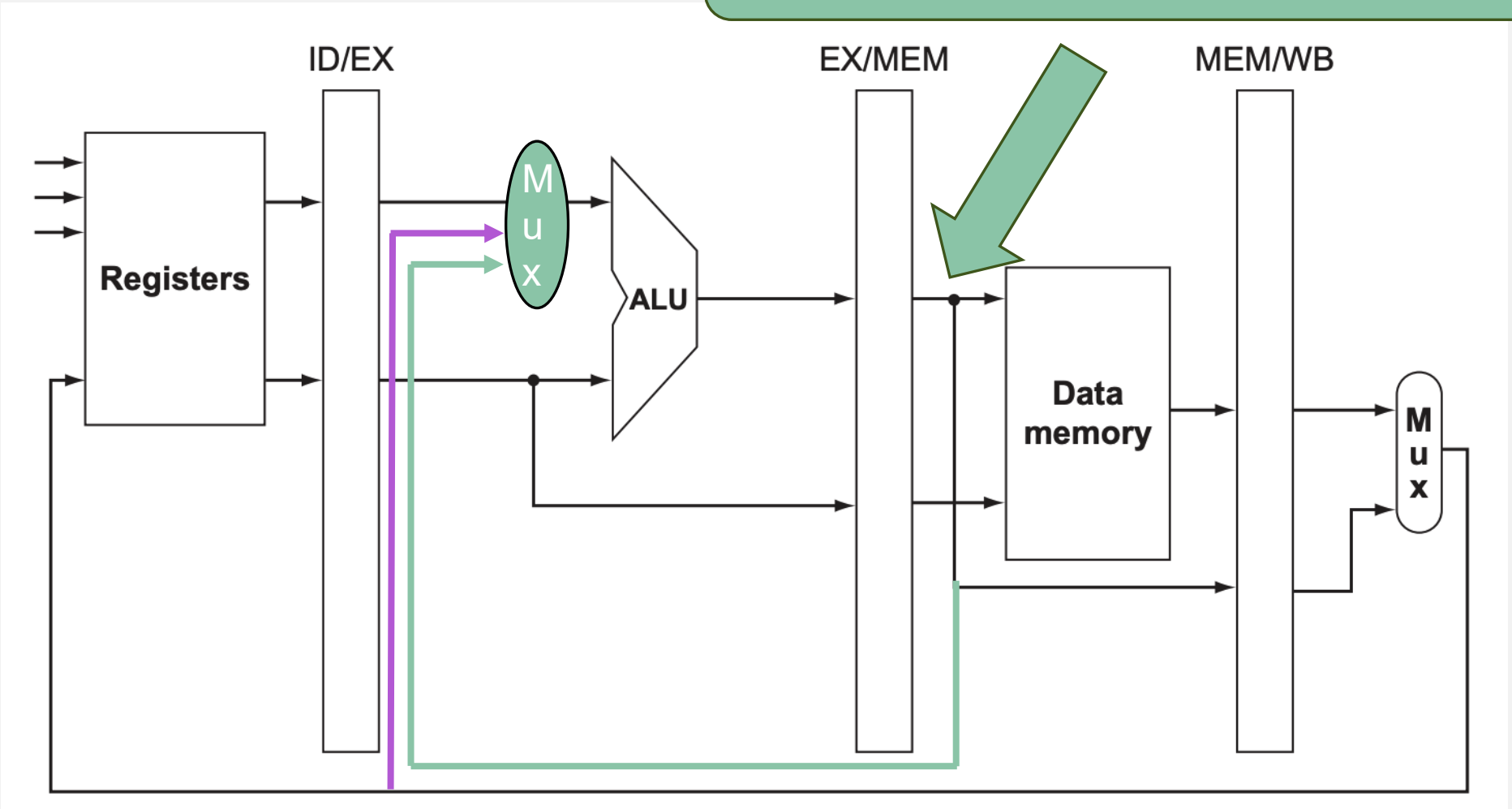
sub x7, x5, x2

or x8, x9, x10

add x5, x3, x4

Forwarding rs2

The value to be written to x5 is here

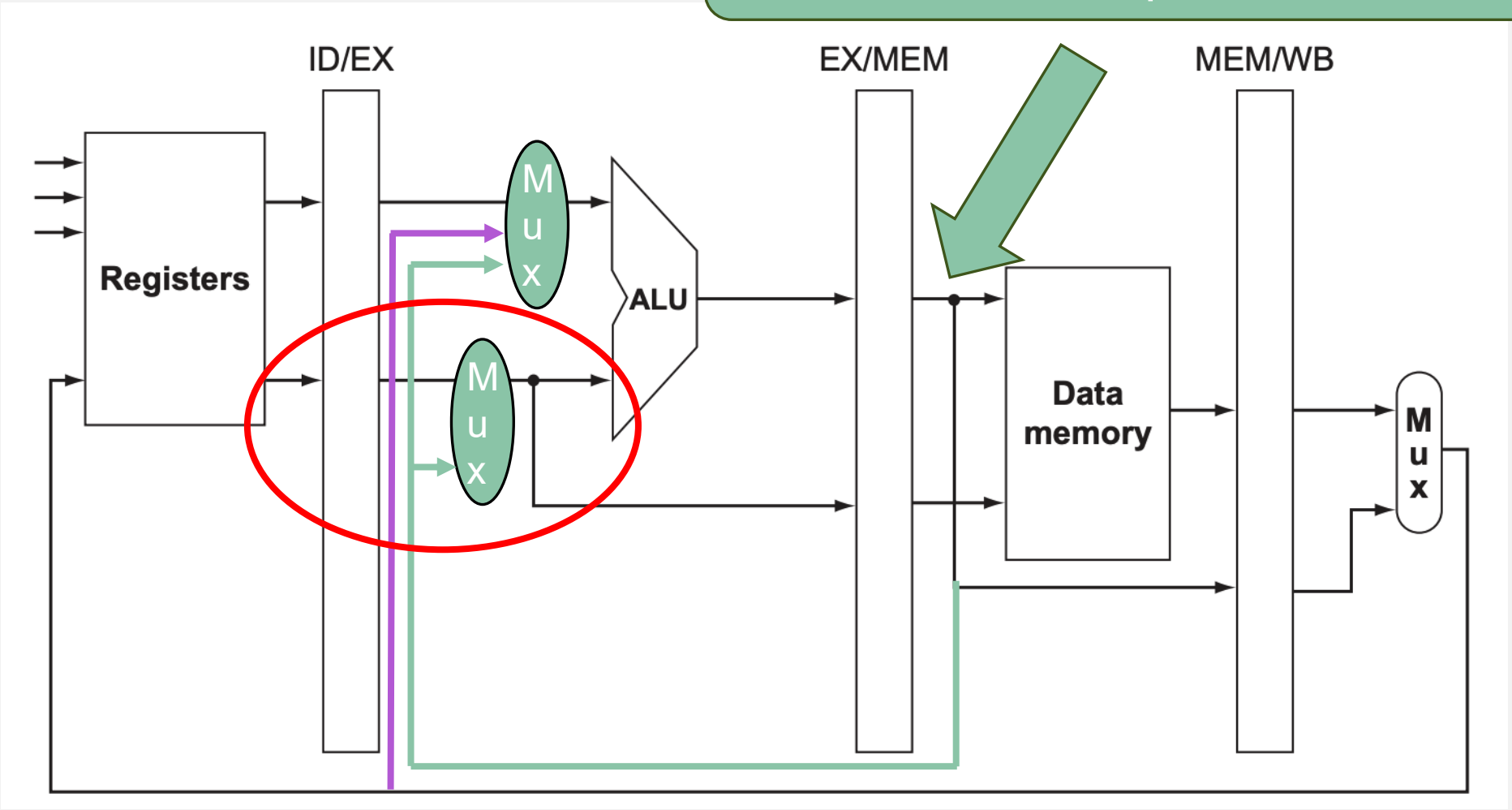


sub x7, x2, x5

add x5 x3, x4

Forwarding rs2

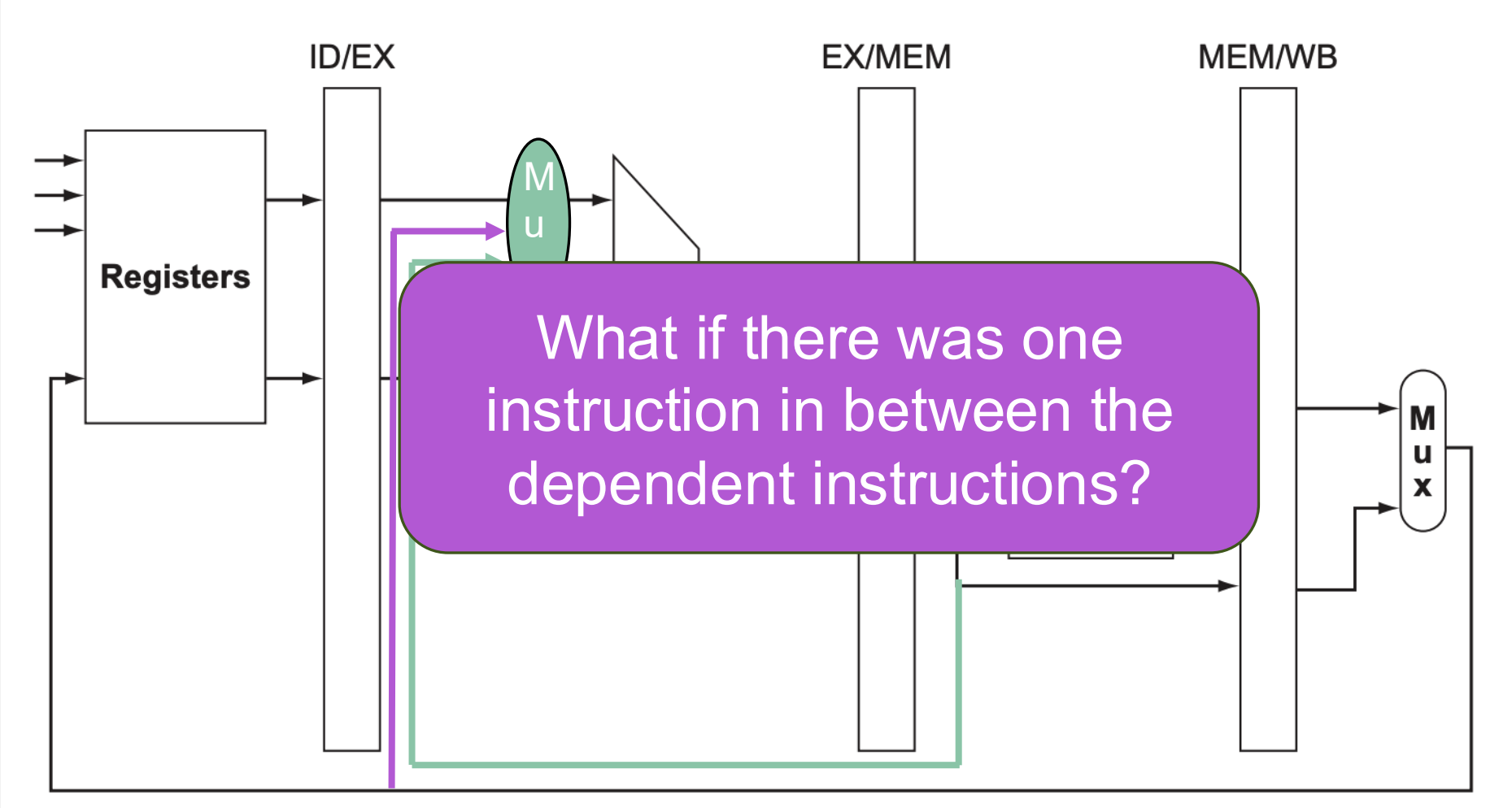
We are forwarding the value to be written to x5 to the dependent instruction.



sub x7, x2, x5

add x5 x3, x4

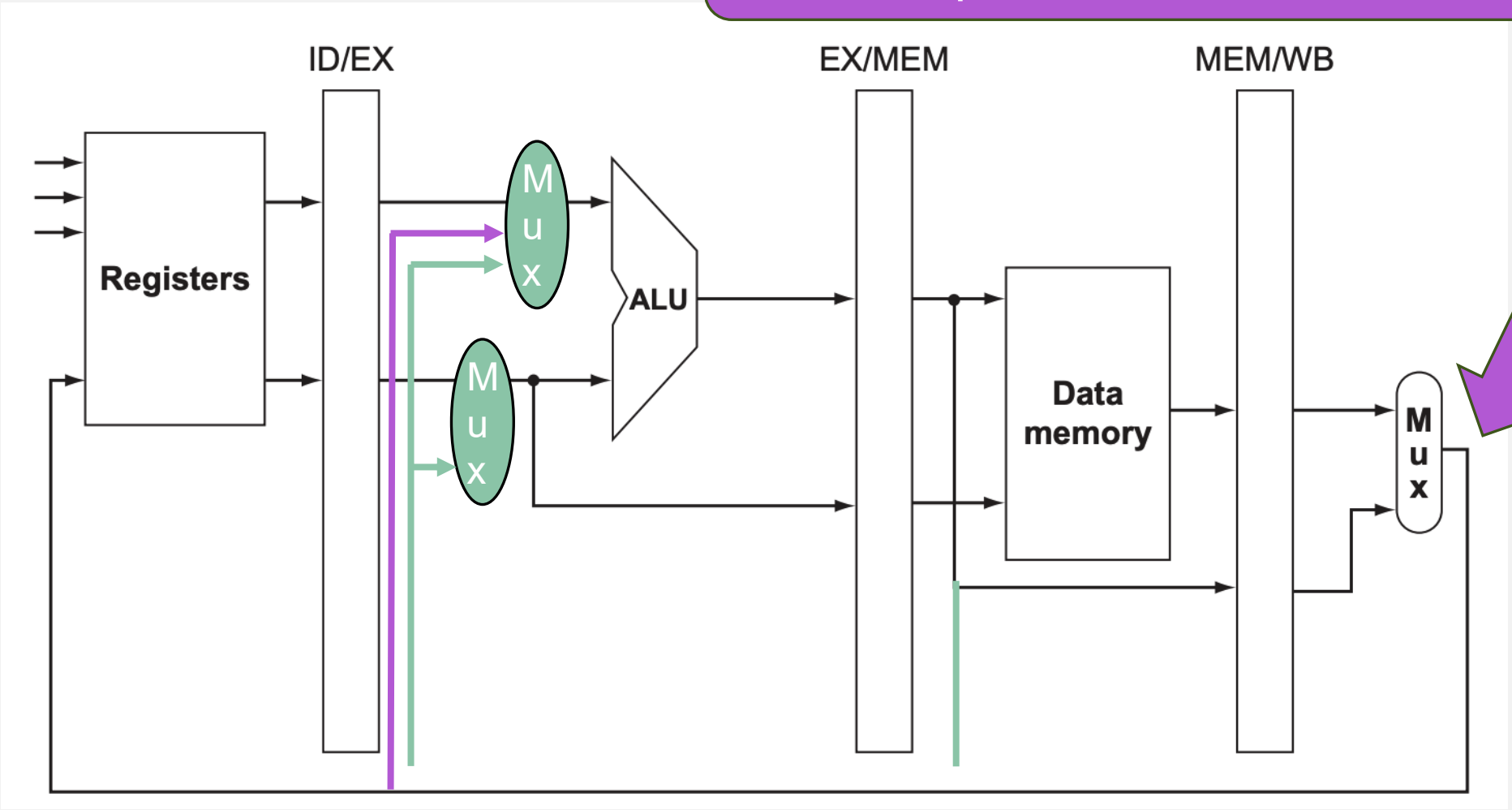
Forwarding rs2



sub x7, x2, x5 or x8, x9,x10 add x5,x3, x4

Forwarding rs2

The value to be written to x5 is at the output of the writeback mux.

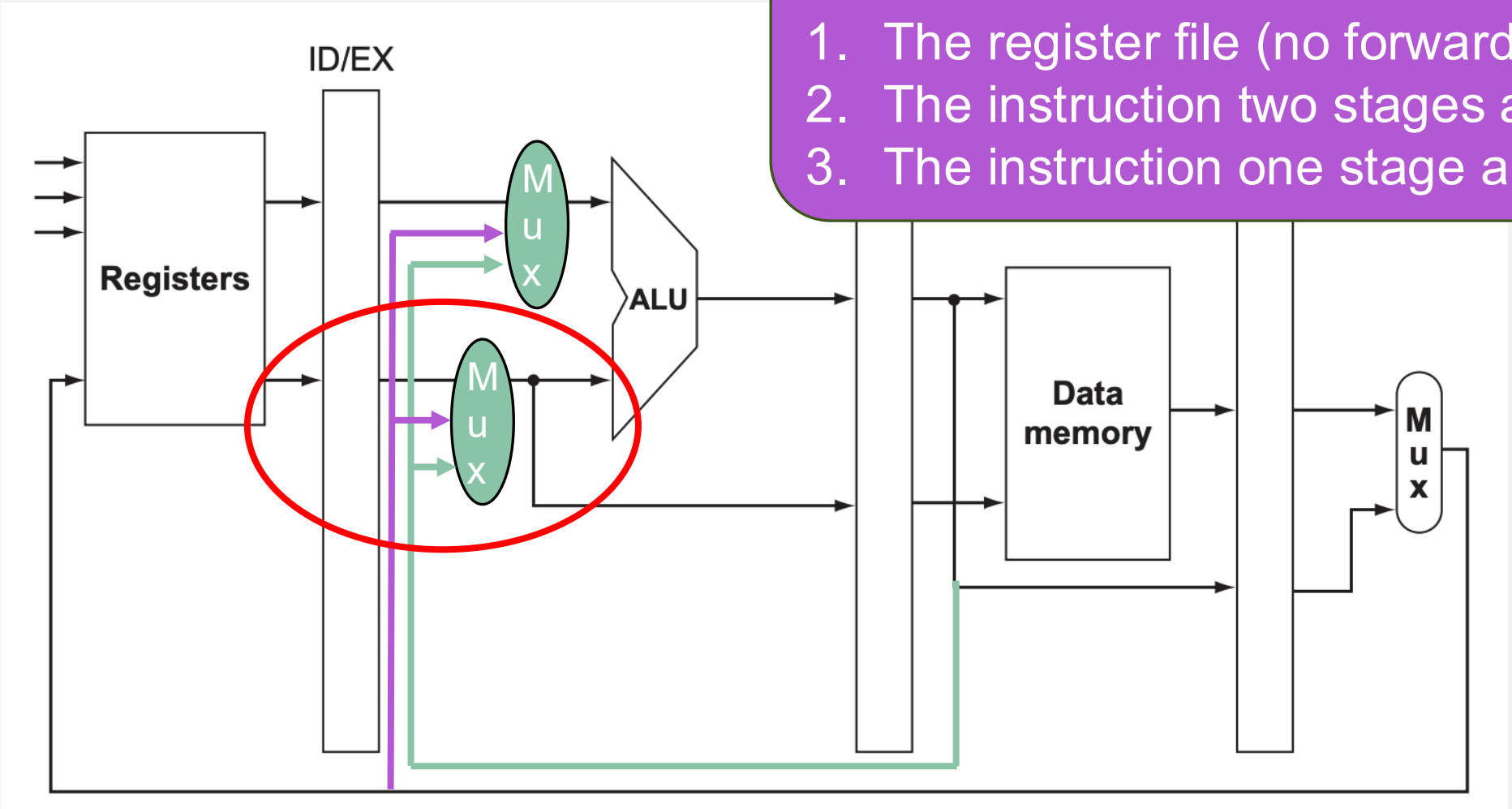


sub x7, x2, x5 or x8, x9,x10 add x5,x3, x4

Forwarding rs2

Now, we can choose if the bottom input of the ALU should come from

1. The register file (no forwarding)
2. The instruction two stages ahead
3. The instruction one stage ahead

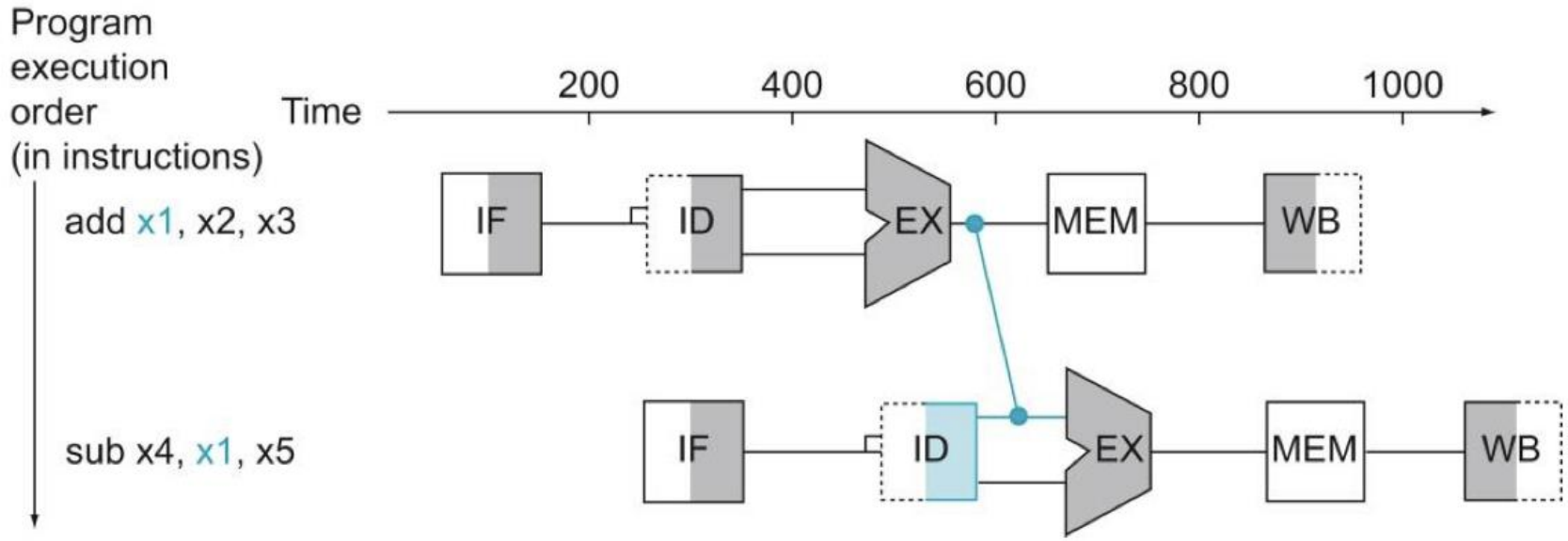


sub x7, x2, x5

or x8, x9,x10

add x5,x3, x4

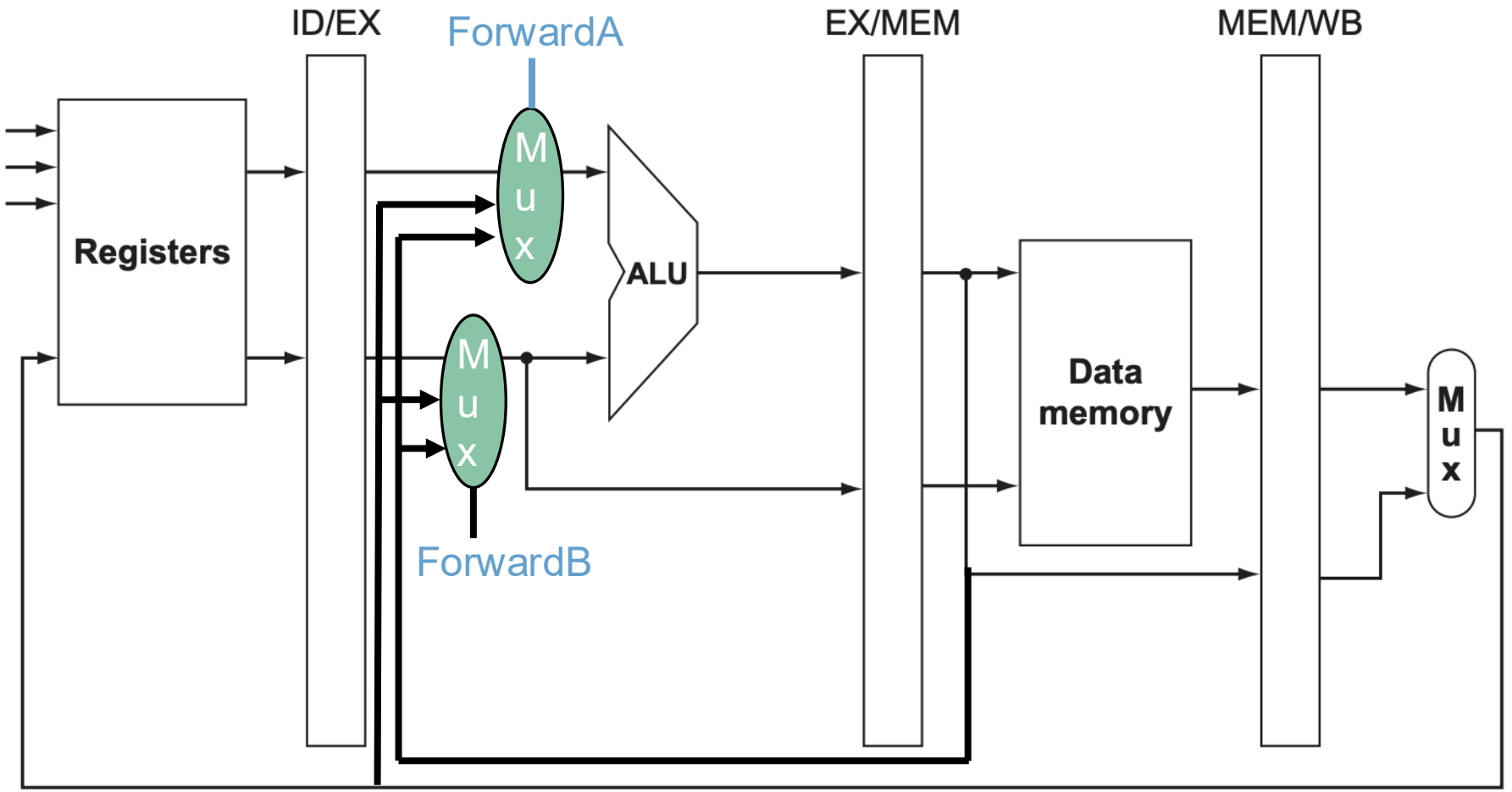
Graphical Representation of Forwarding



Source: https://csweb.wooster.edu/jmusgrave/cs210/Chapter_04.pdf

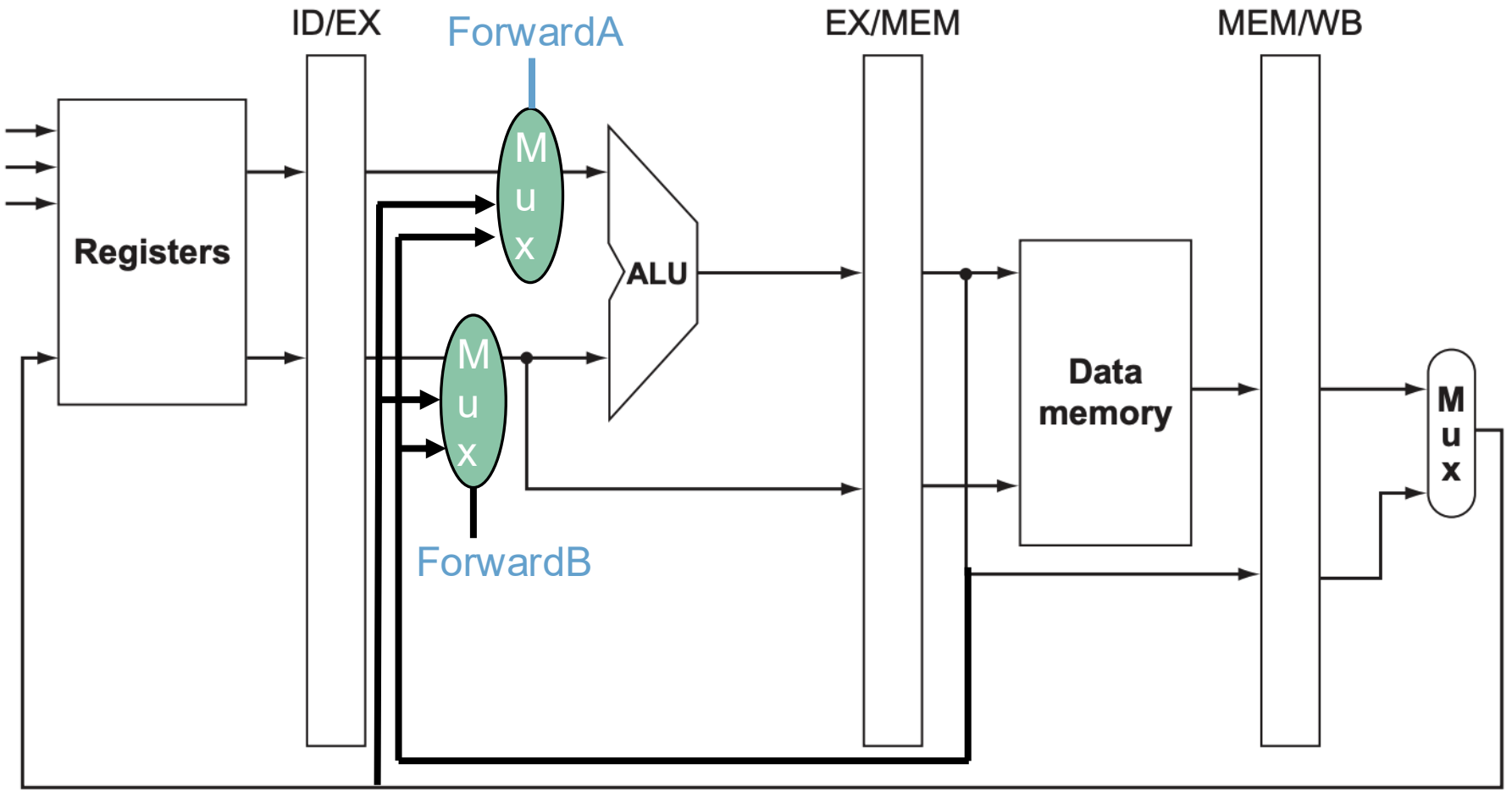
LOAD-USE HAZARD

Load-Use Data Hazard



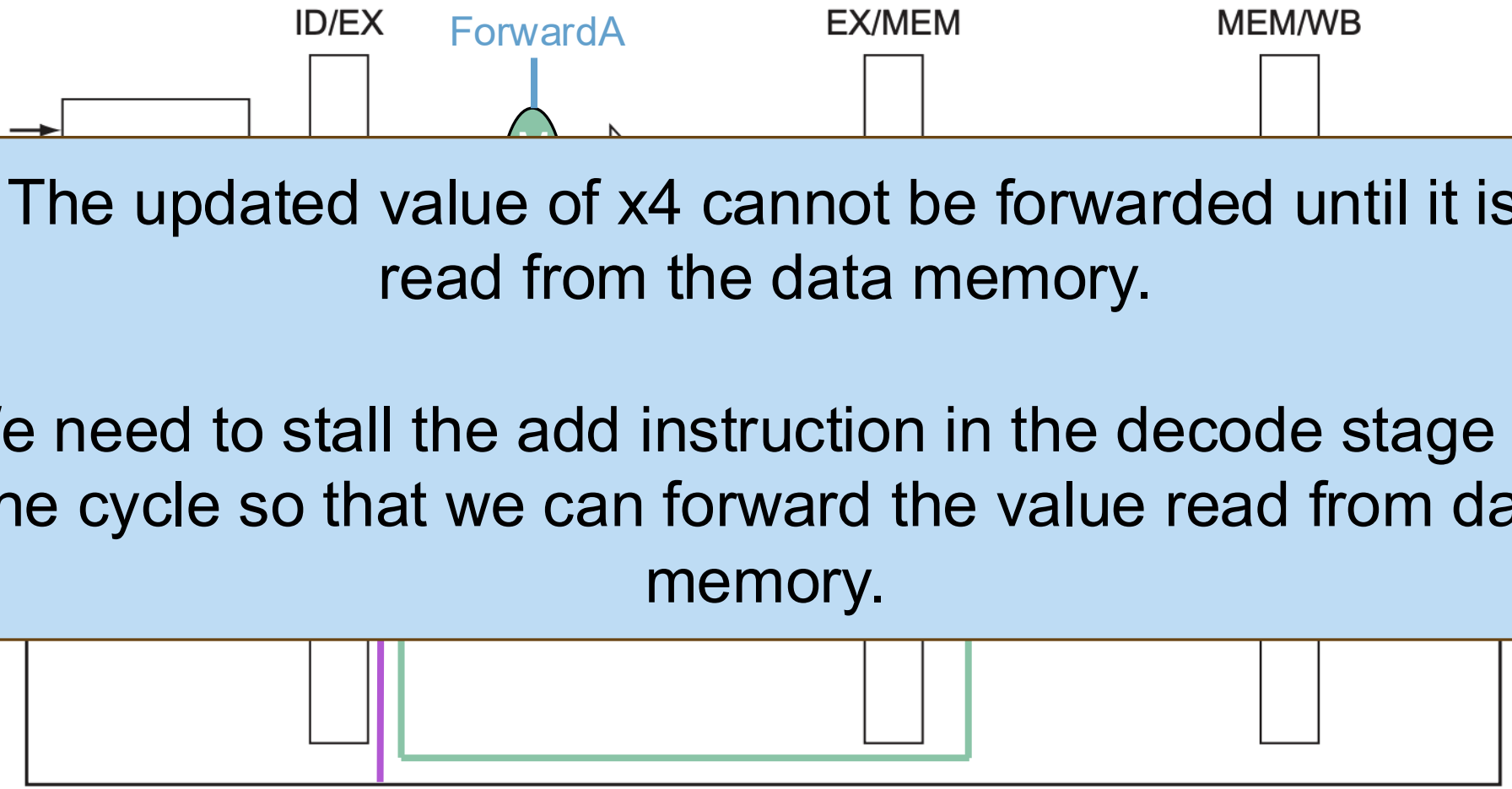
add x5, x4, x3 lw x4, 0(x9)

Load-Use Data Hazard



add x5, x4, x3 lw x4, 0(x9)

Load-Use Data Hazard

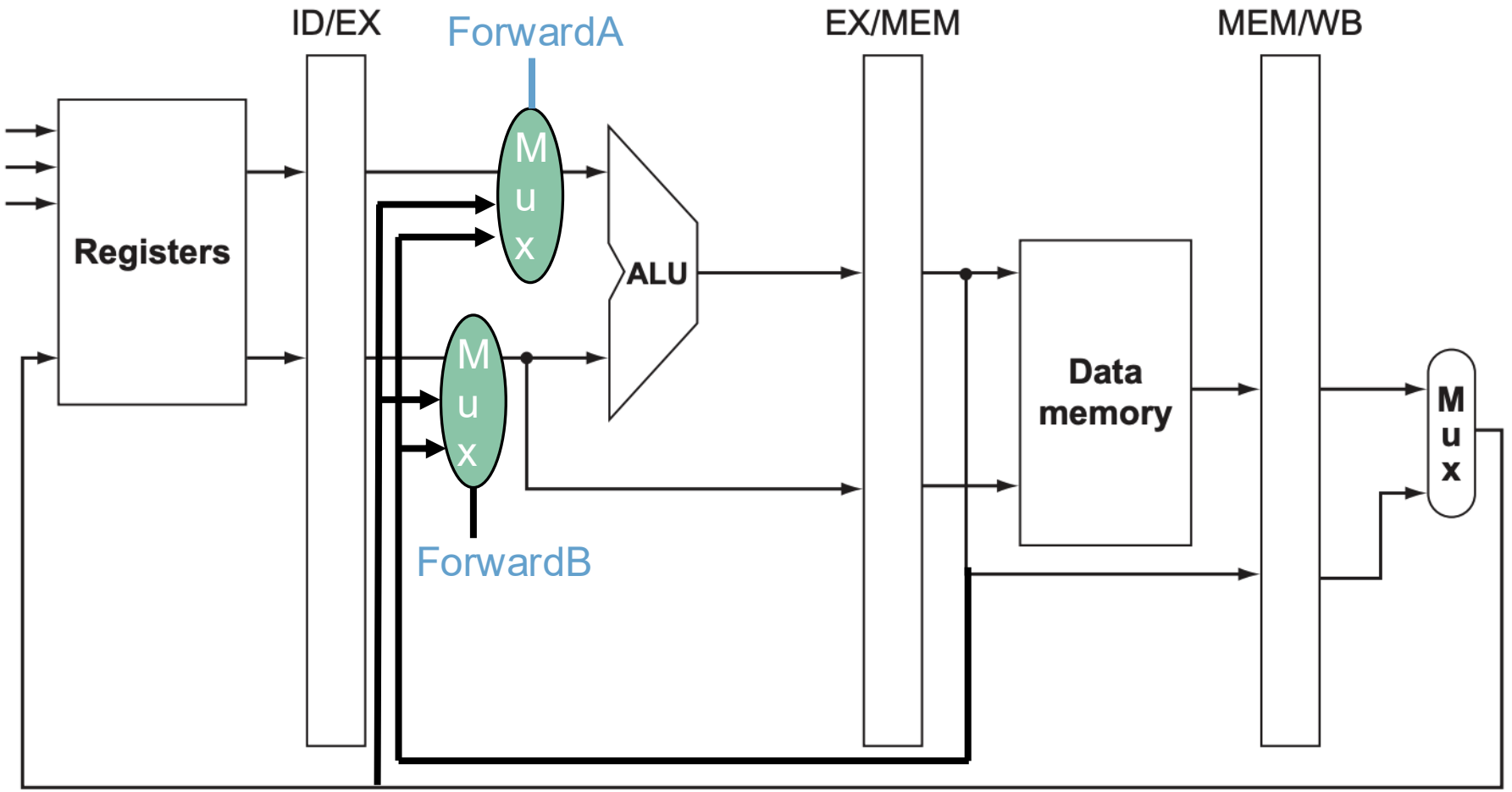


The updated value of x4 cannot be forwarded until it is read from the data memory.

We need to stall the add instruction in the decode stage for one cycle so that we can forward the value read from data memory.

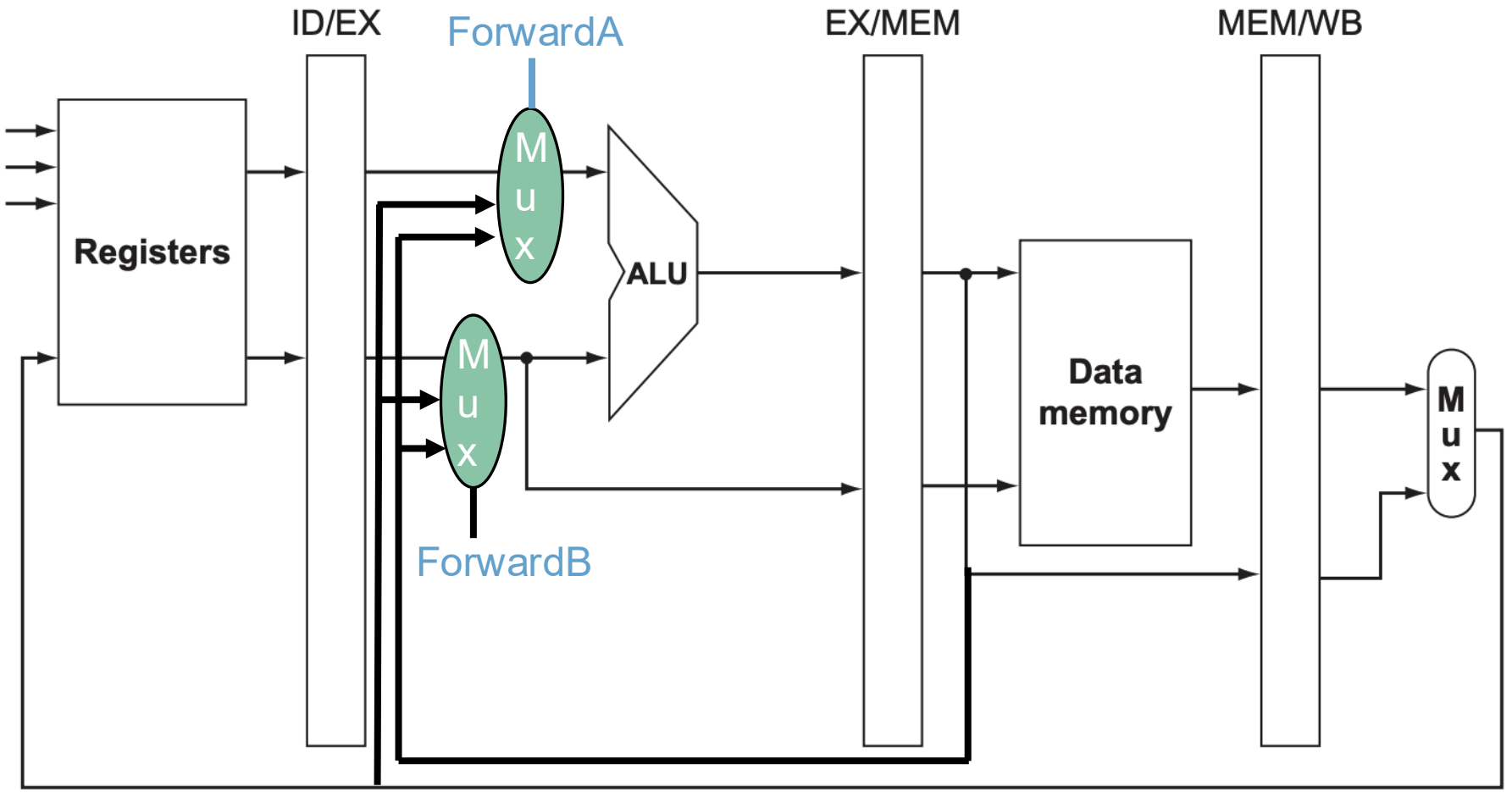
add x5, x4, x3 lw x4, 0(x9)

Load-Use Data Hazard



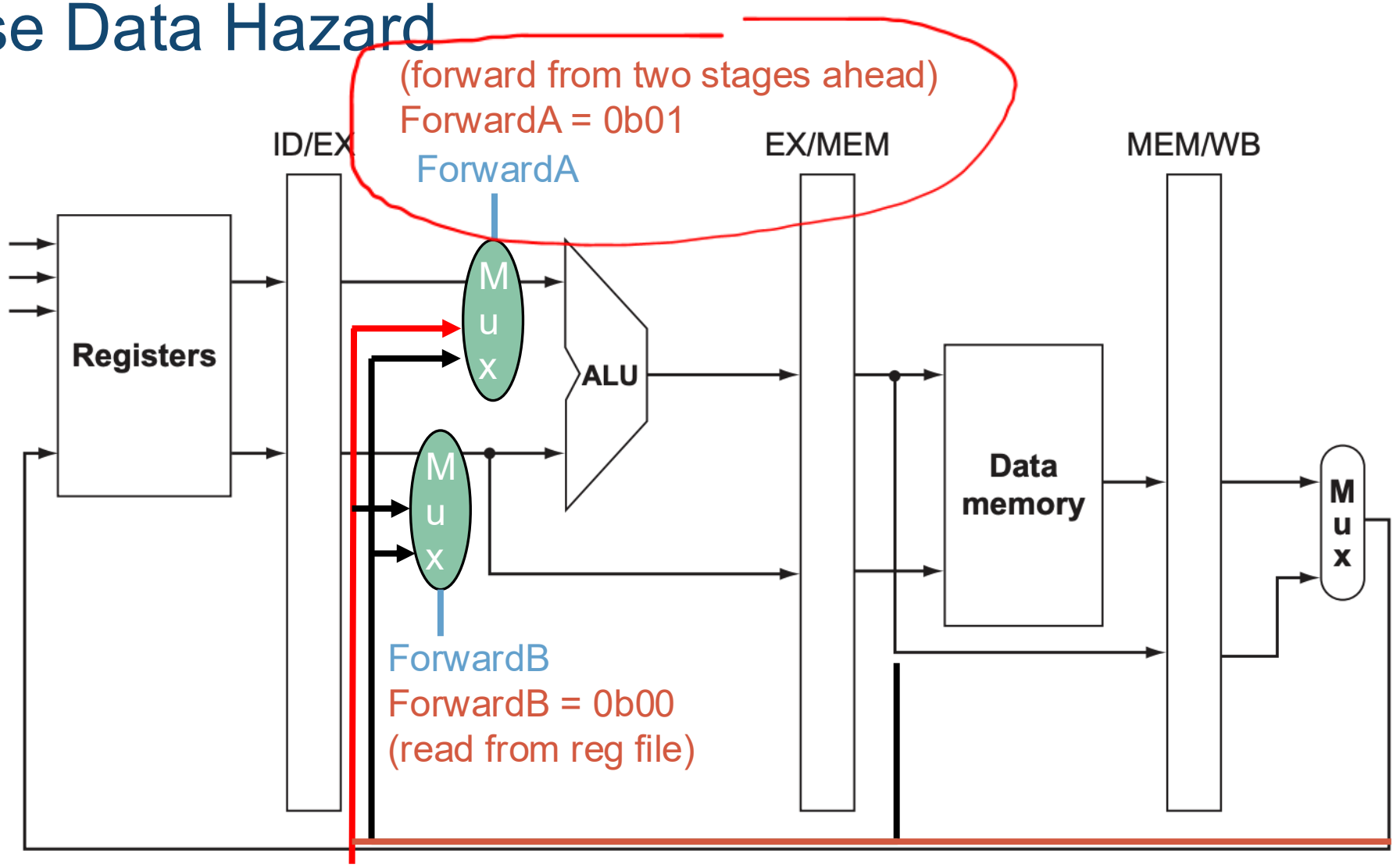
add x5, x4, x3 lw x4 , 0(x9)

Load-Use Data Hazard



add x5, x4, x3 *stall* lw x4 , 0(x9)

Load-Use Data Hazard



add x5, x4, x3

~~stall~~

lw x4, 0(x9)

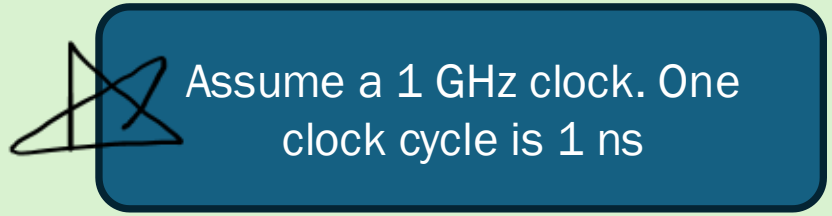
Pipeline Diagram

1. Fill out the pipeline diagram for the following set of instructions.
2. What should ForwardA and ForwardB be set to on each cycle?
   - If the instruction in the execute stage is a stall, set ForwardA and ForwardB to XX.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lw x3, 0(x8)															
add x11, x3, x3															
lw x4, 0(x11)															
add x9, x11, x9															
sub x10, x4, x9															

ForwardA															
ForwardB															

Performance Analysis + Pipelining



1×10^{-9}

- Assume a pipelined implementation where:
  - *Taken branches take 2 cycles.*
  - *Loads and stores take 2 cycles.*
- Assuming approximately 10% of instructions executed are branches, and of those 80% of the time they are taken, and 15% of instructions executed are loads or stores, what sort of real speed-up do we expect?

Some takeaways...

- Design strategy: Focus our attention on placing pipelining registers around the slowest circuit elements (bottlenecks!)
- You can pipeline an RISC-V CPU even more. There exist implementations with 7, 8, and 9 pipeline stages. But the overhead of bypass paths and the number of stall cases increase .
- Memory access time is our real bottleneck!