11/9/26

COMP 311: *Computer Organization!*

Lecture 22: Pipelining the RISC-V CPU, Hazards



The University
of North Carolina
at Chapel Hill

https://tinyurl.com/comp311-fa26

Logistics Update



- Lab 4 is out
  - Due 11/23
- WA5 released today.

Plan for Class Today



- 3:35-4:00: Finish going over single-cycle RISC-V Datapath
- 4:00-4:15: Pipelining the RISC-V CPU?
- 4:15-4:20: BREAK
- 4:20-4:50: Pipeline Hazards

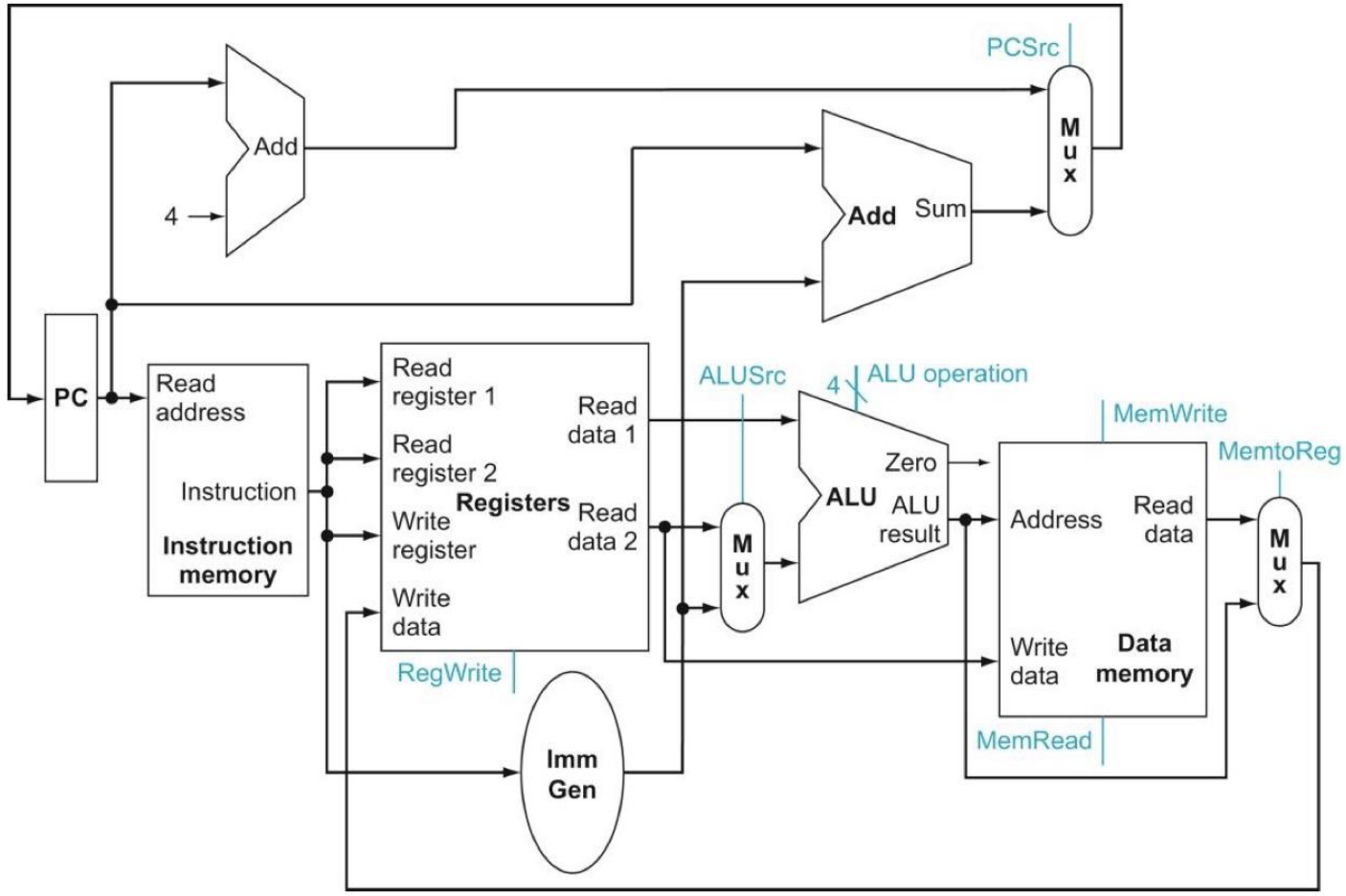
Guiding Question ☺

What happens when there are dependencies between instructions being computed at the same time in a CPU?

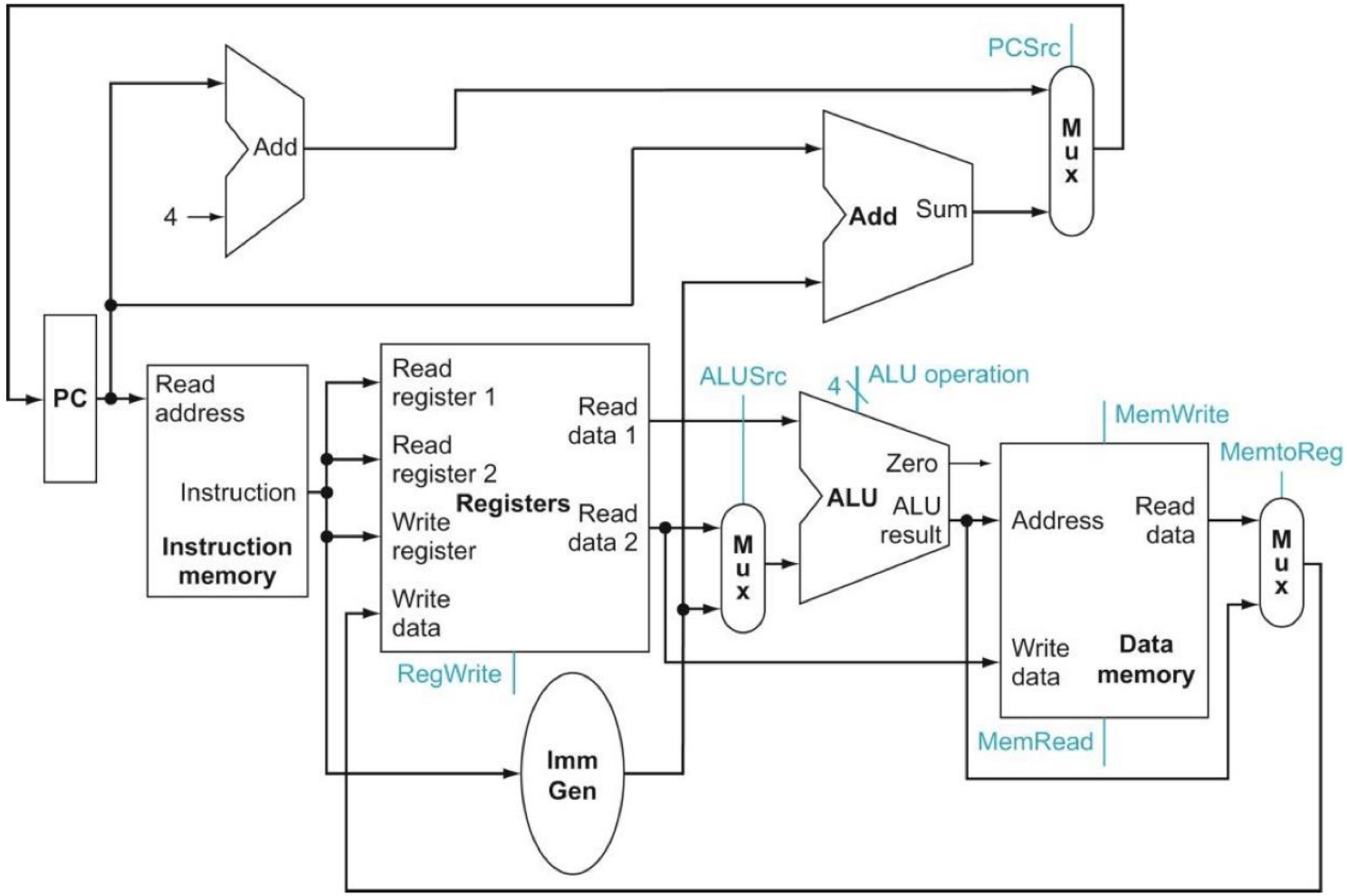


Review: The RISC-V Datapath

The RISC-V Datapath



ADD | Start with known state



add x5, x1, x11

add x5, x1, x2

PC = 0x1000

x1 = 7

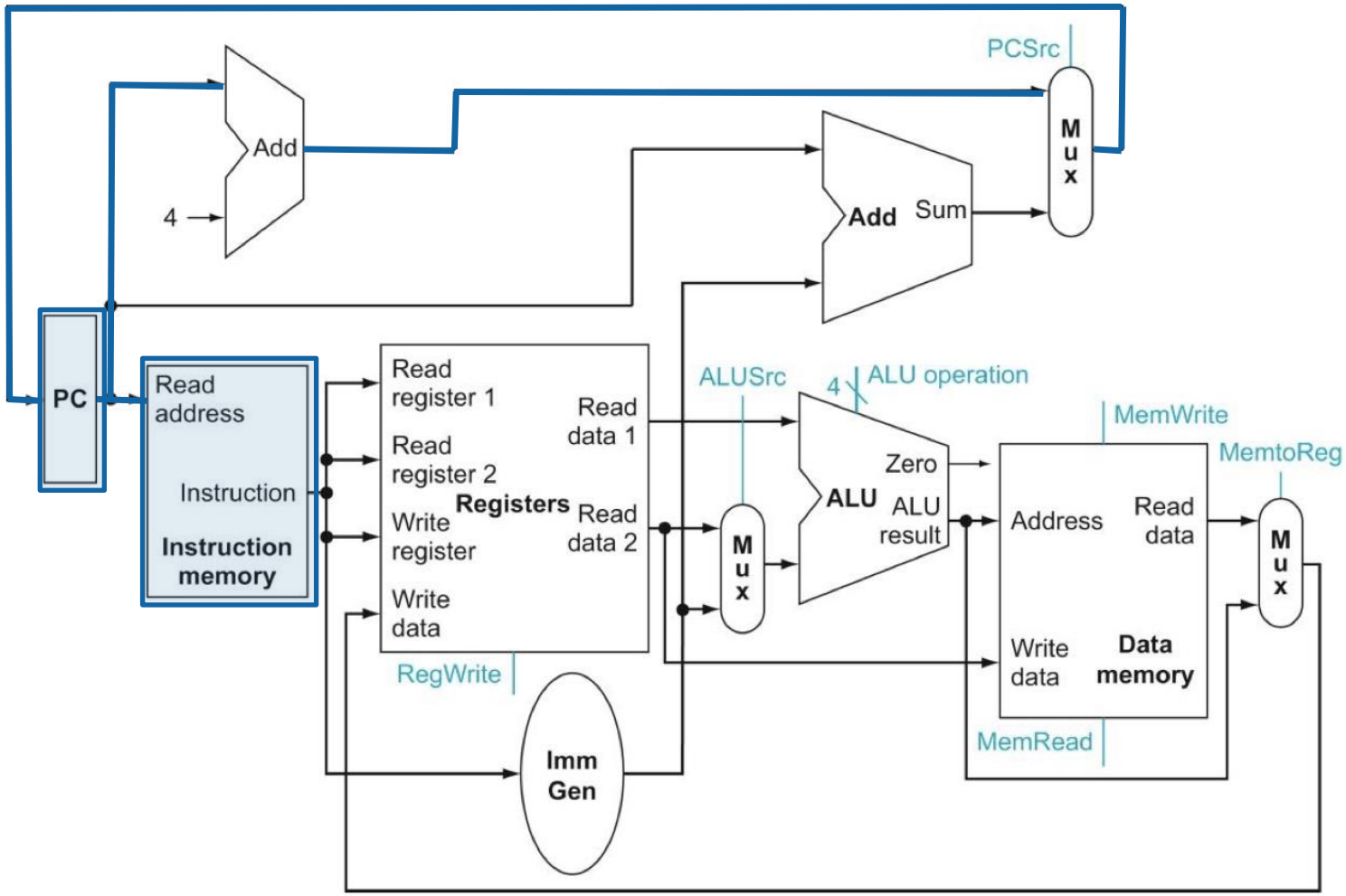
Immediate = 11

Predict: what changes?

Expected result: x5 = 18

Single-cycle datapath • steps show signal flow, not separate clock cycles.

ADD | 1 Fetch



add x5, x1, 11

PC addresses instruction memory.

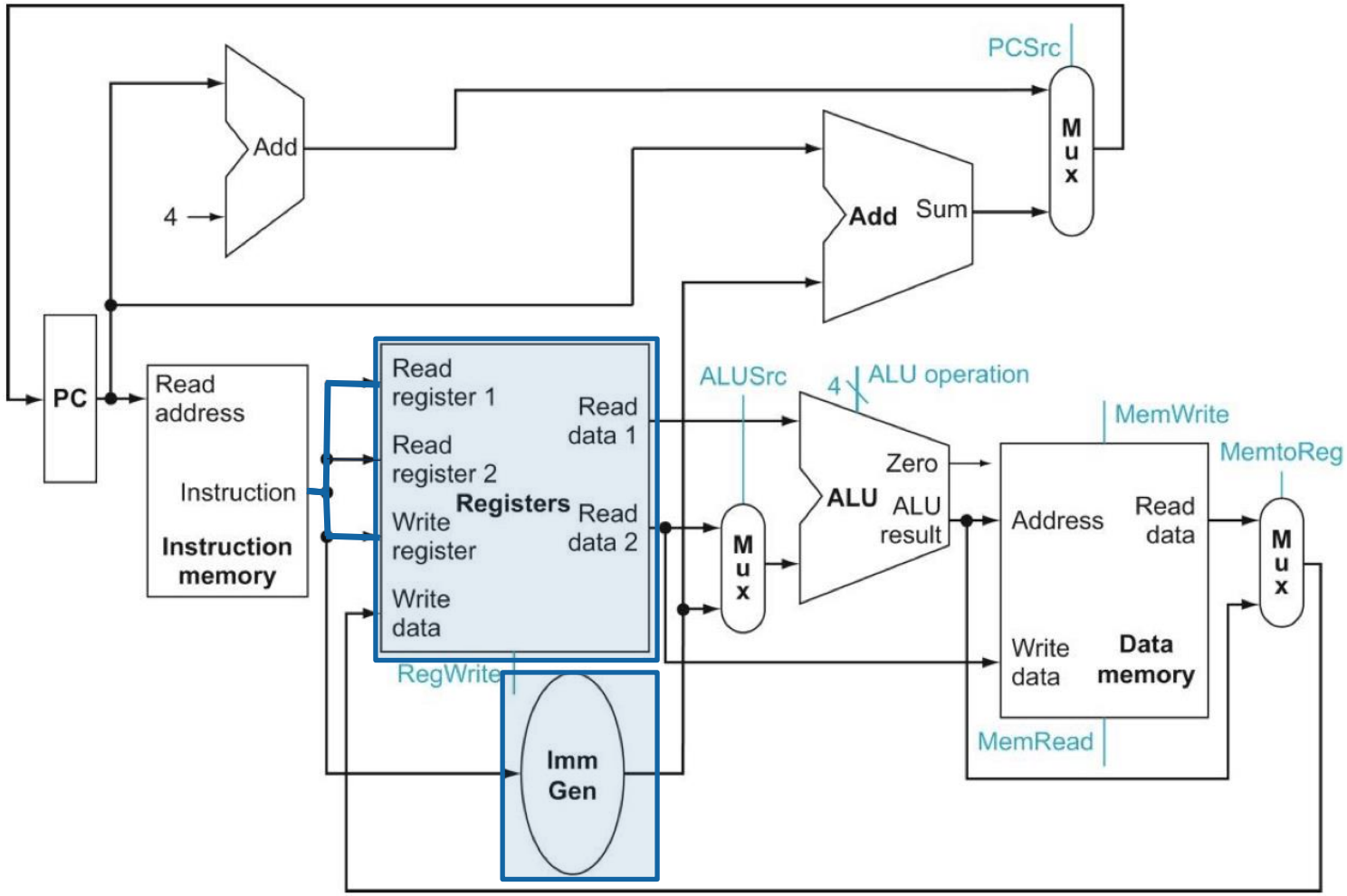
Fetch add x5, x1, 11.

The PC adder computes 0x1004.

PCSrc = 0 → select PC + 4

Single-cycle datapath • steps show signal flow, not separate clock cycles

ADD | 2 Decode / read



add x5, x1, 11

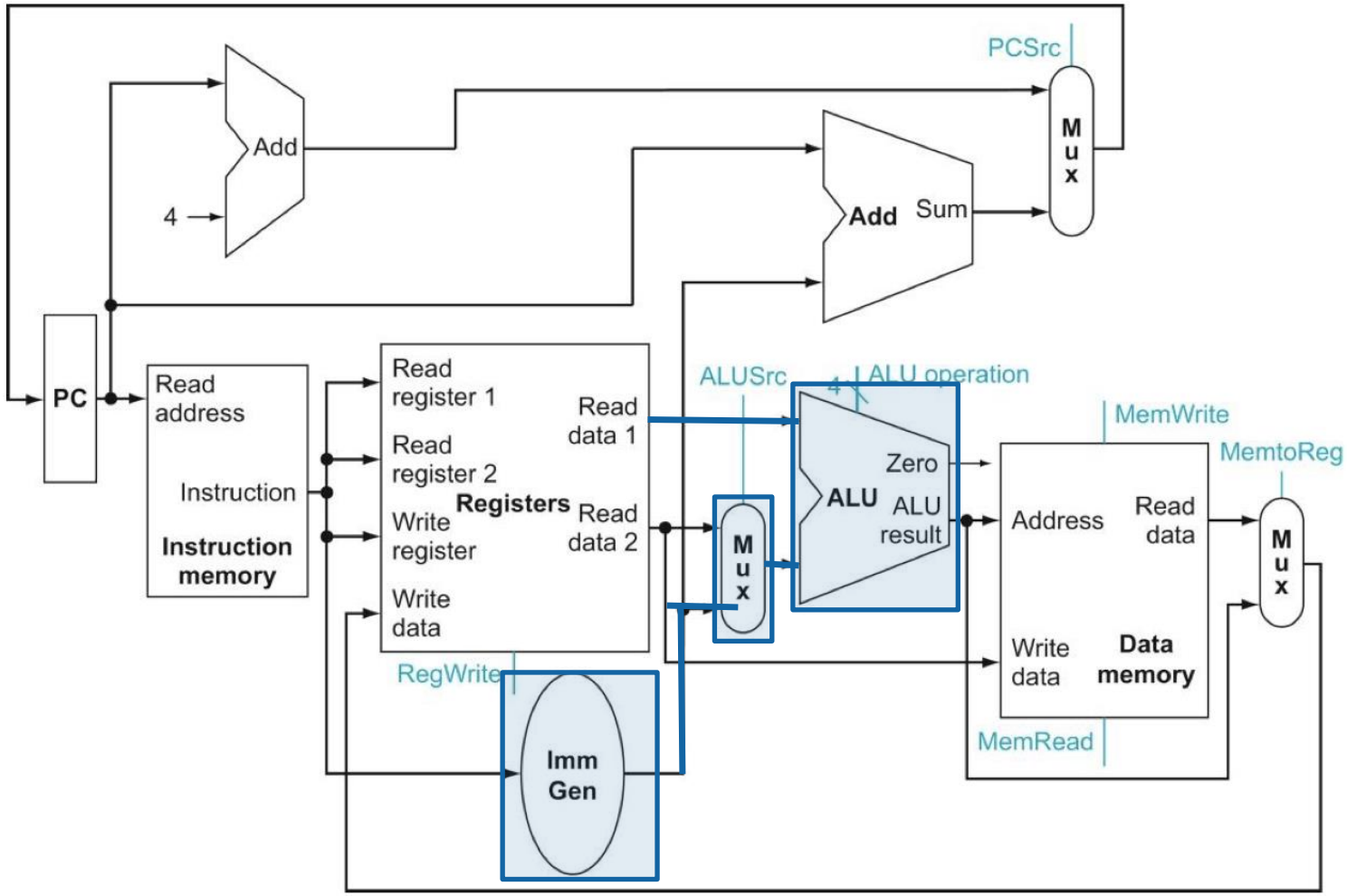
rs1 = 1 → read x1 = 7

Immediate: 11

rd = 5 → destination x5

Control decodes an R-type ADD.

ADD | 3 Execute



add x5, x1, 11

ALU input A = 7

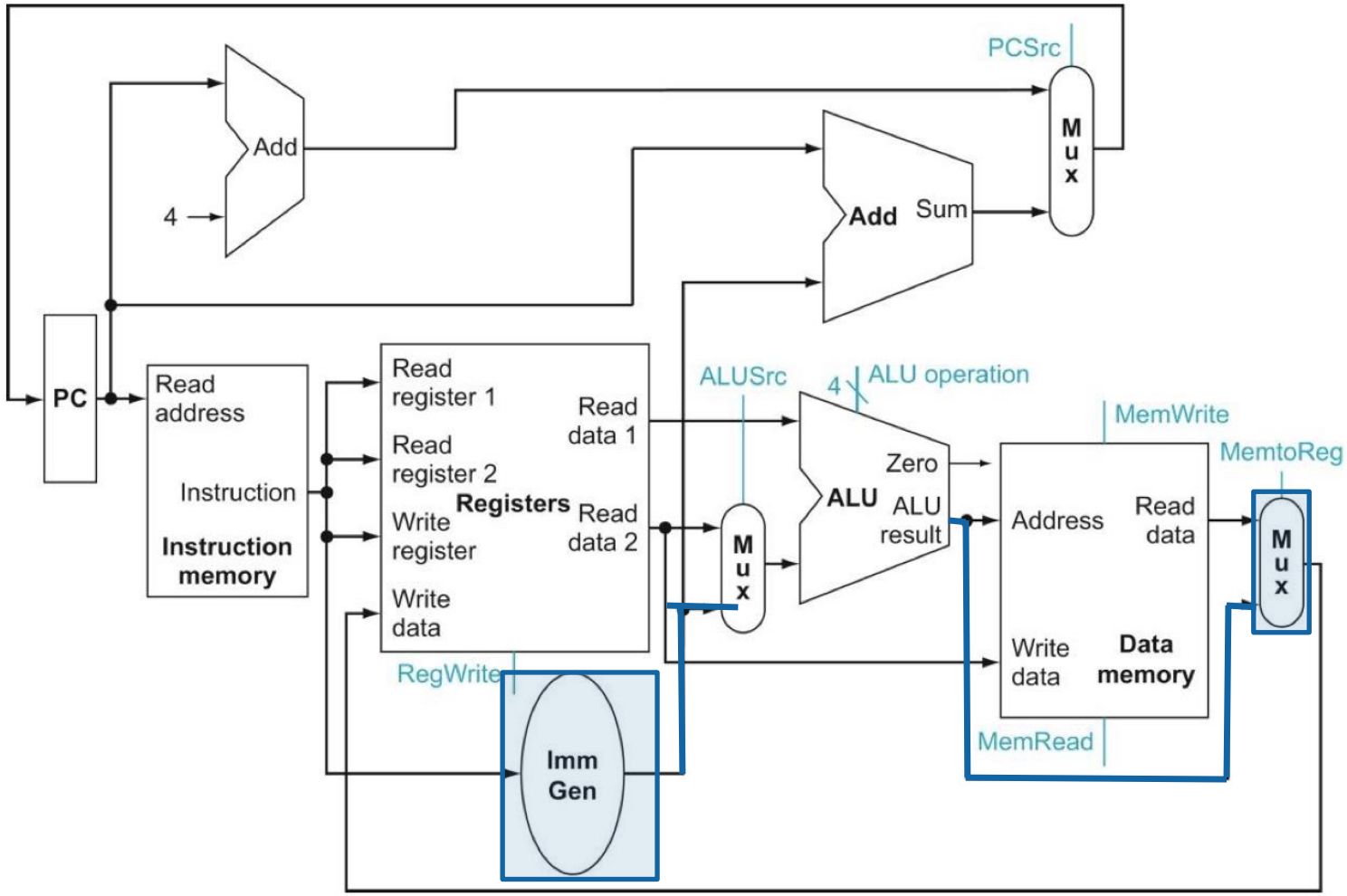
ALU input B = 11

ALU result = 7 + 11 = 18

ALUSrc = 1; ALU operation = ADD

Single-cycle datapath • steps show signal flow, not separate clock cycles.

ADD | 4 Bypass data memory



add x5, x1, 11

The result 18 travels around data memory.

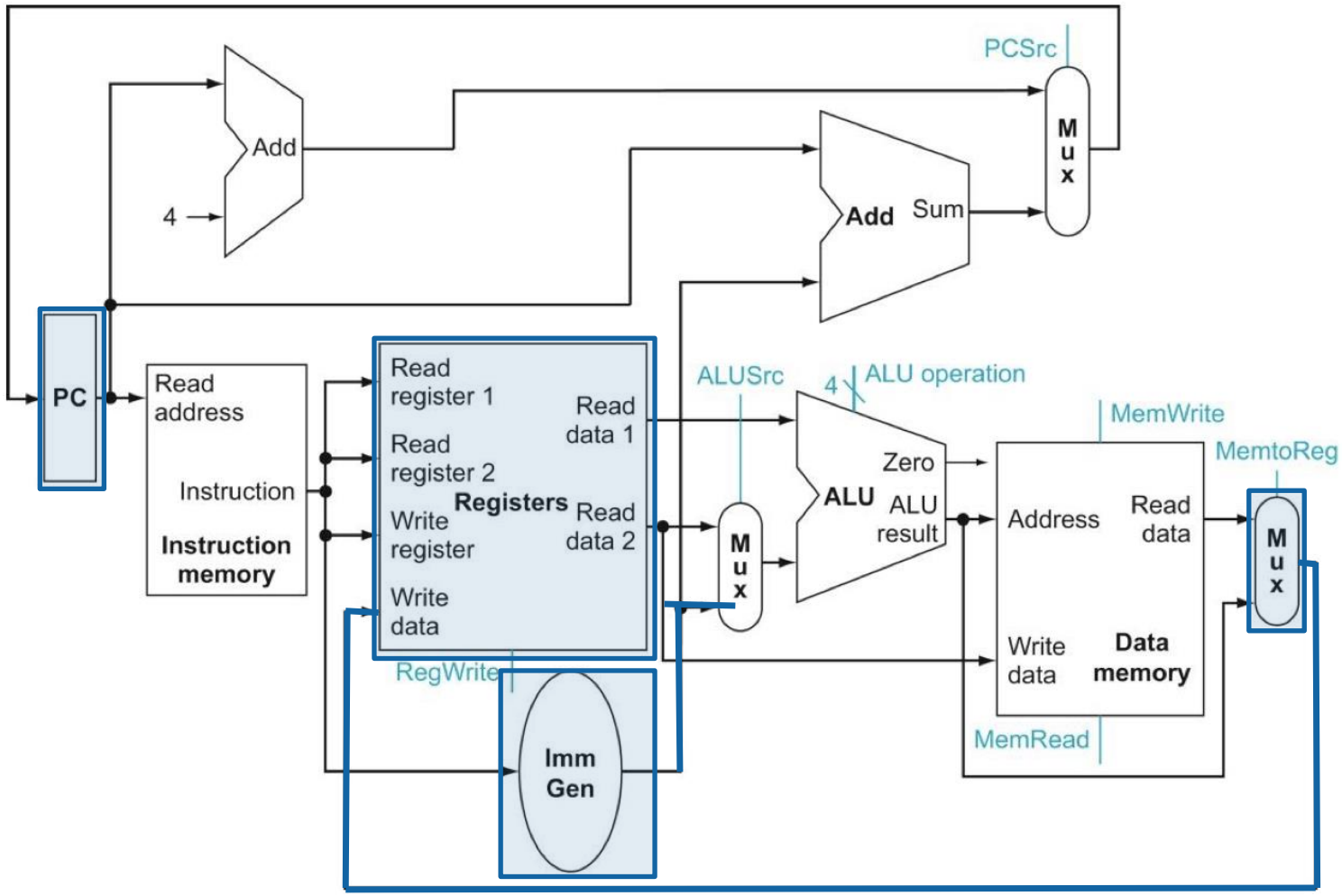
No data-memory read or write.

MemRead = 0; MemWrite = 0

MemtoReg = 0 → ALU result

Single-cycle datapath • steps show signal flow, not separate clock cycles.

ADD | 5 Write back at the edge



add x5, x1, 11

Write 18 to destination x5.

PC becomes 0x1004.

x1 = 7 and x2 = 11 stay unchanged.

RegWrite = 1; rd = 5

Single-cycle datapath • steps show signal flow, not separate clock cycles

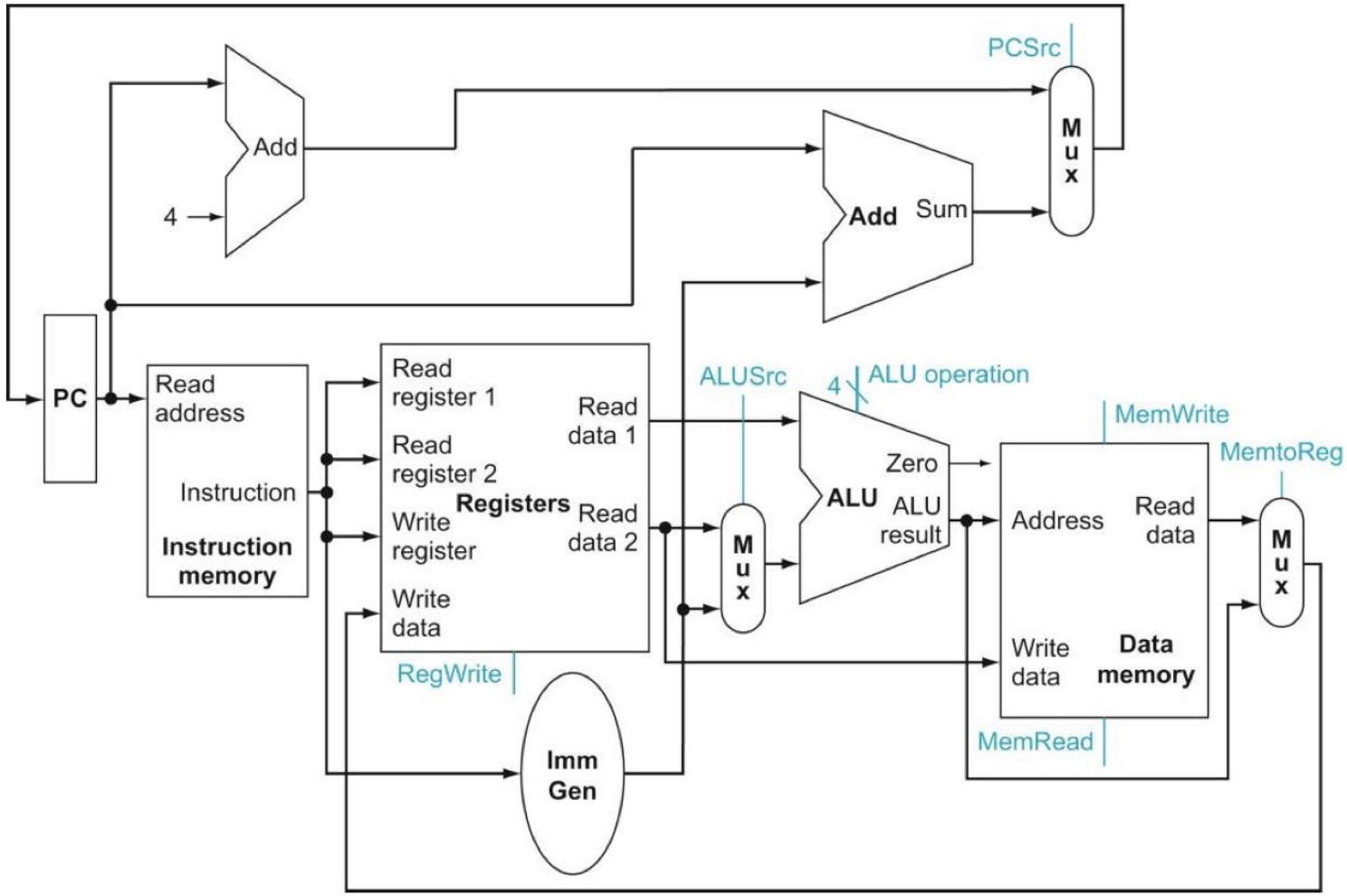
ADD vs ADDI Control Signals



Control Signal	add x5, x1, x2	addi x5, x1, 11
ALUSrc	0 (register)	**1 (immediate)**
ALU operation	ADD	ADD
RegWrite	1	1
MemRead	0	0
MemWrite	0	0
MemtoReg	0 (ALU)	0 (ALU)
Imm Gen	Unused	11
ALU input A	x1 = 7	x1 = 7
ALU input B	x2 = 11	**Immediate = 11**
Result (x5)	18	**18**

Key Takeaway: different source for operand B!

LW | Next instruction: load



lw x6, 8(x3)

lw x6, 8(x3)

PC = 0x1004; x3 = 0x2000

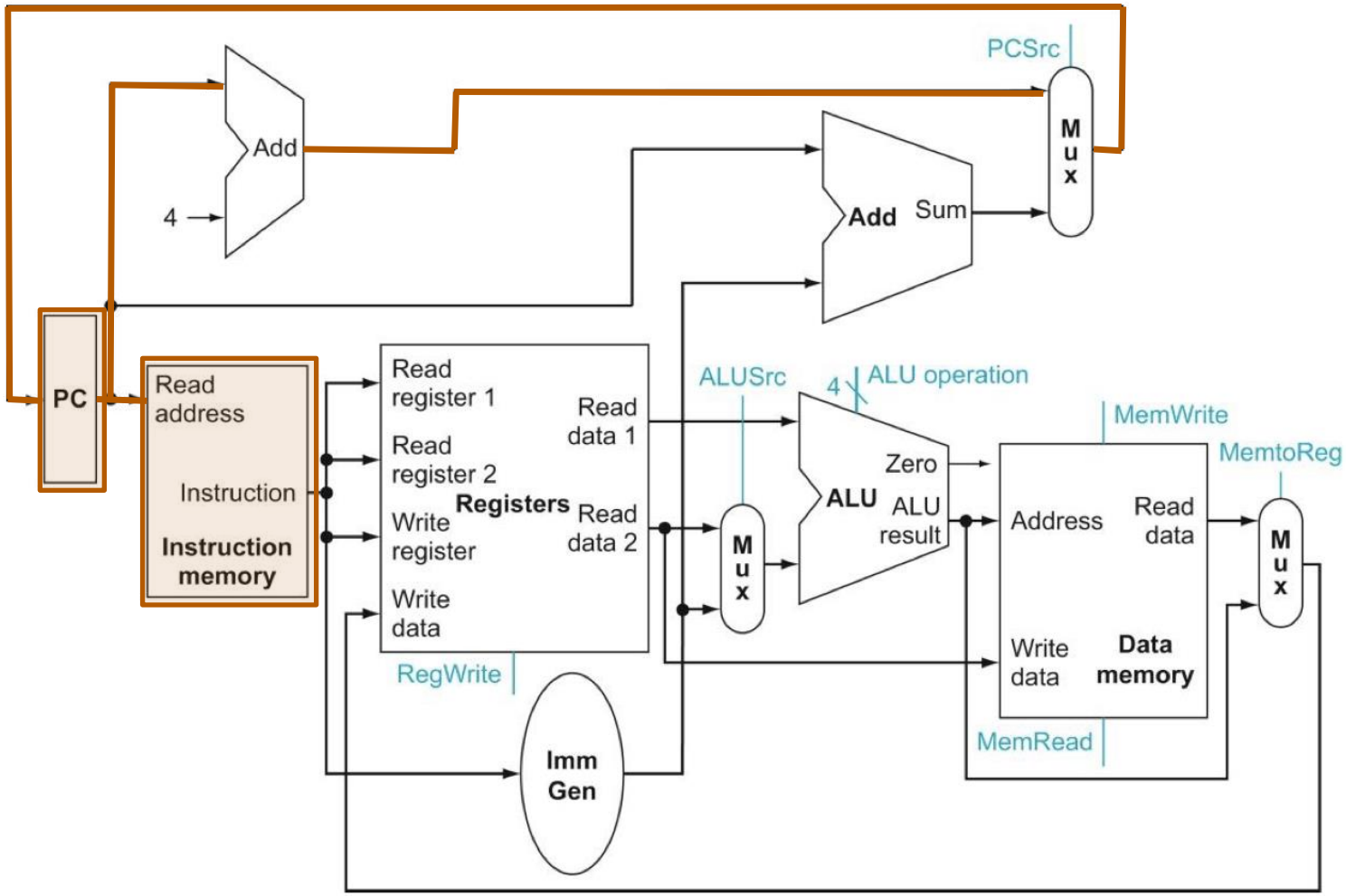
Memory word at 0x2008 = 42

Predict: does x6 get 0x2008 or 42?

Keep x5 = 18 from the ADD.

Single-cycle datapath • steps show signal flow, not separate clock cycles.

LW | 1 Fetch



lw x6, 8(x3)

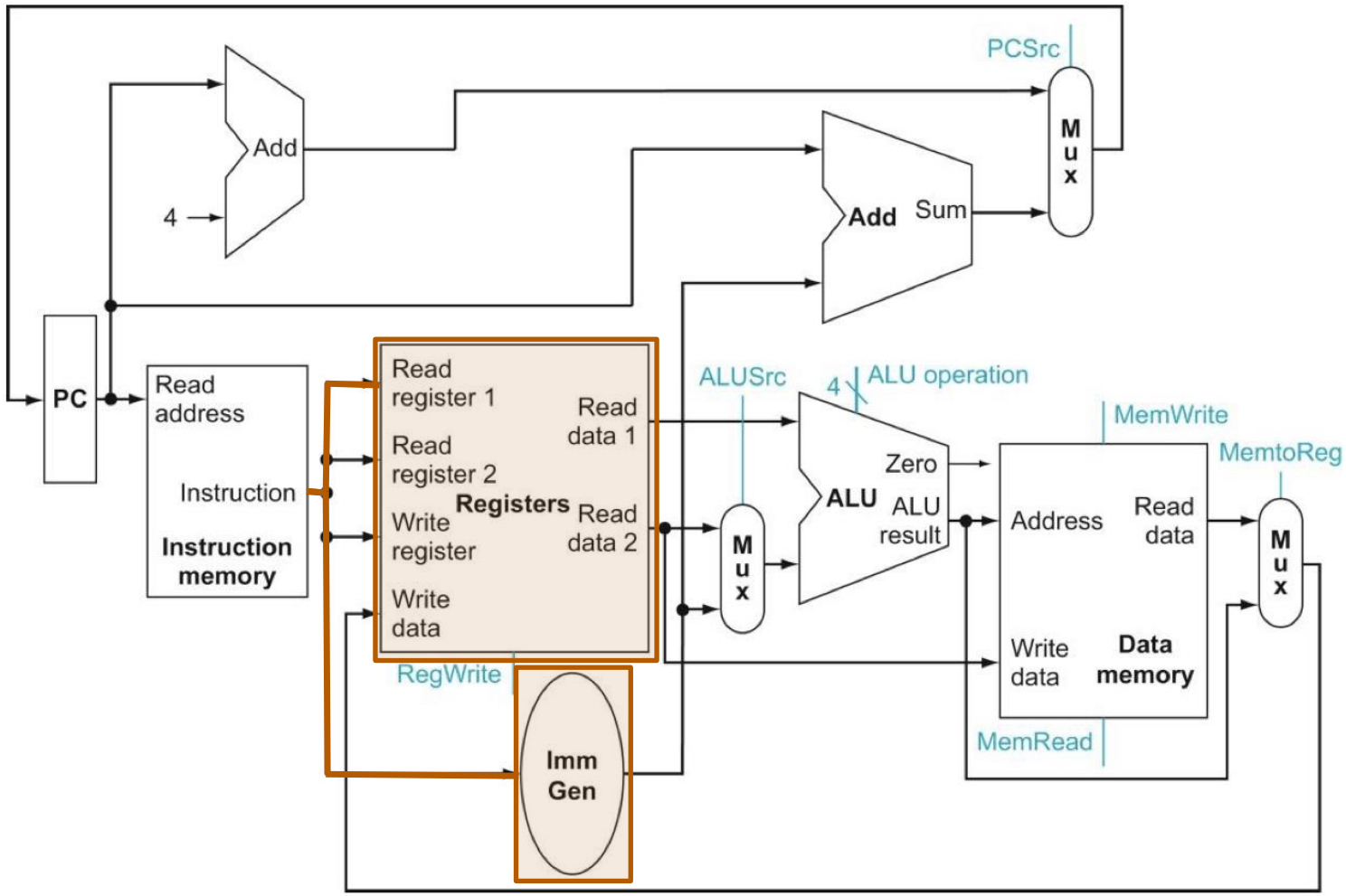
Fetch lw x6, 8(x3) from instruction memory.

The PC adder computes 0x1008.

PCSrc = 0 → select PC + 4

Single-cycle datapath • steps show signal flow, not separate clock cycles.

LW | 2 Decode / read



lw x6, 8(x3)

rs1 = 3 → base x3 = 0x2000

Sign-extend immediate 8.

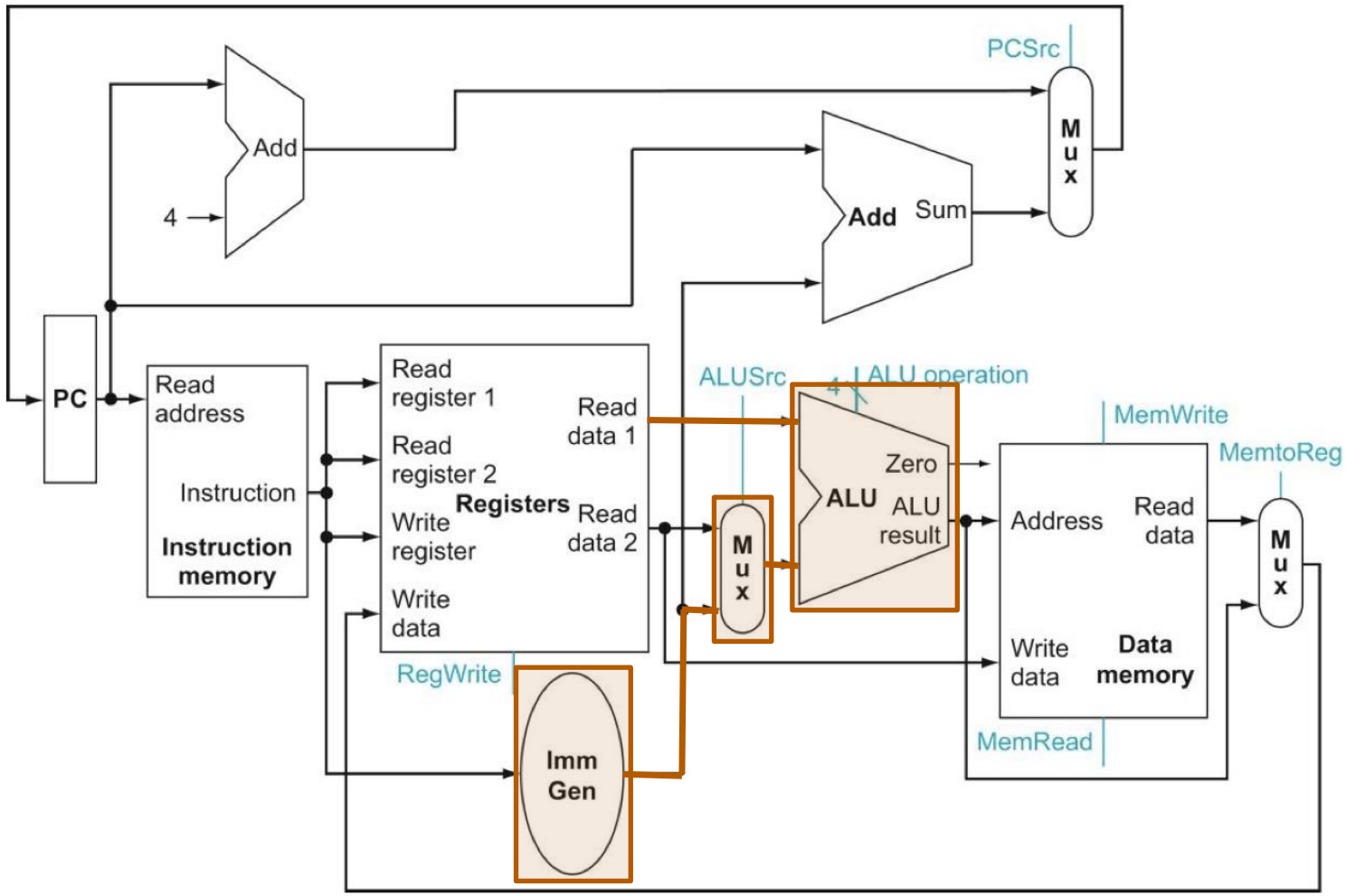
rd = 6 → destination x6

No rs2 operand is needed.

I-type load: base + byte offset

Single-cycle datapath • steps show signal flow, not separate clock cycles.

LW | 3 Compute the address



lw x6, 8(x3)

ALU input A = 0x2000

ALU input B = 8

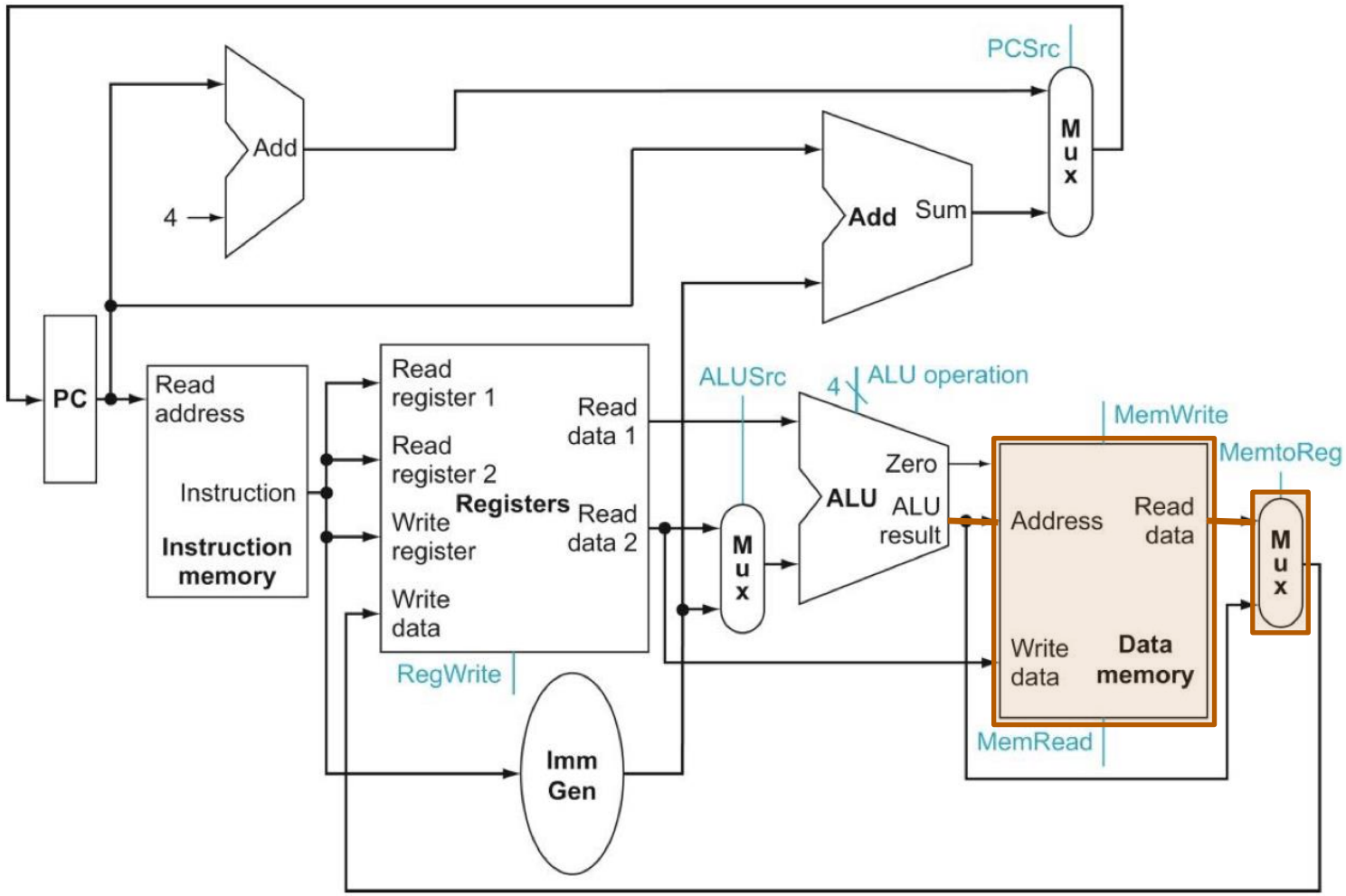
Effective address = 0x2008

This is an address, not the loaded value.

ALUSrc = 1; ALU operation = ADD

Single-cycle datapath • steps show signal flow, not separate clock cycles.

LW | 4 Read data memory



lw x6, 8(x3)

Address = 0x2008

Read the word stored there: 42.

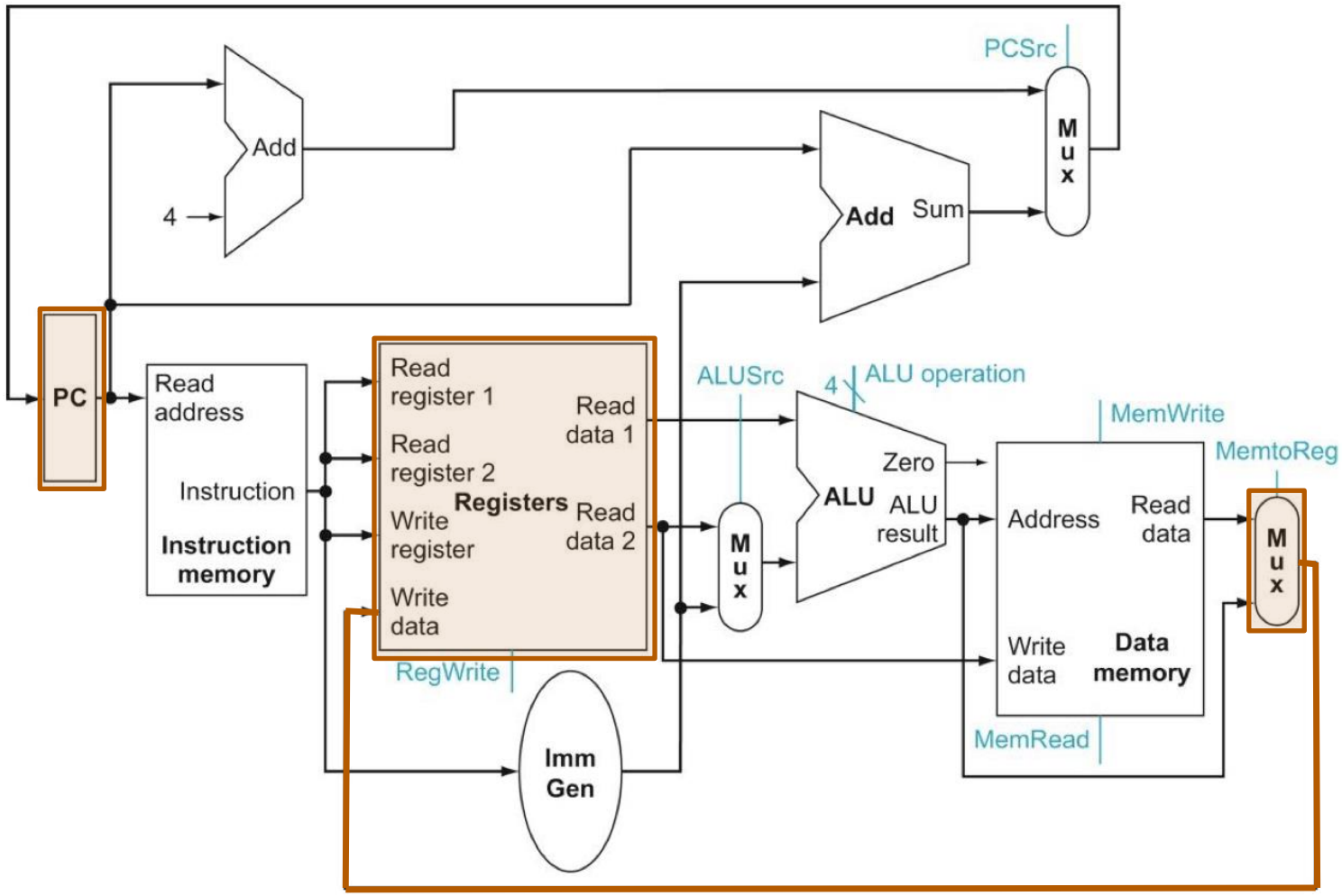
Select memory data for writeback.

MemRead = 1; MemWrite = 0

MemtoReg = 1 → memory data

Single-cycle datapath • steps show signal flow, not separate clock cycles.

LW | 5 Write back at the edge



lw x6, 8(x3)

Write 42 to destination x6.

PC becomes 0x1008.

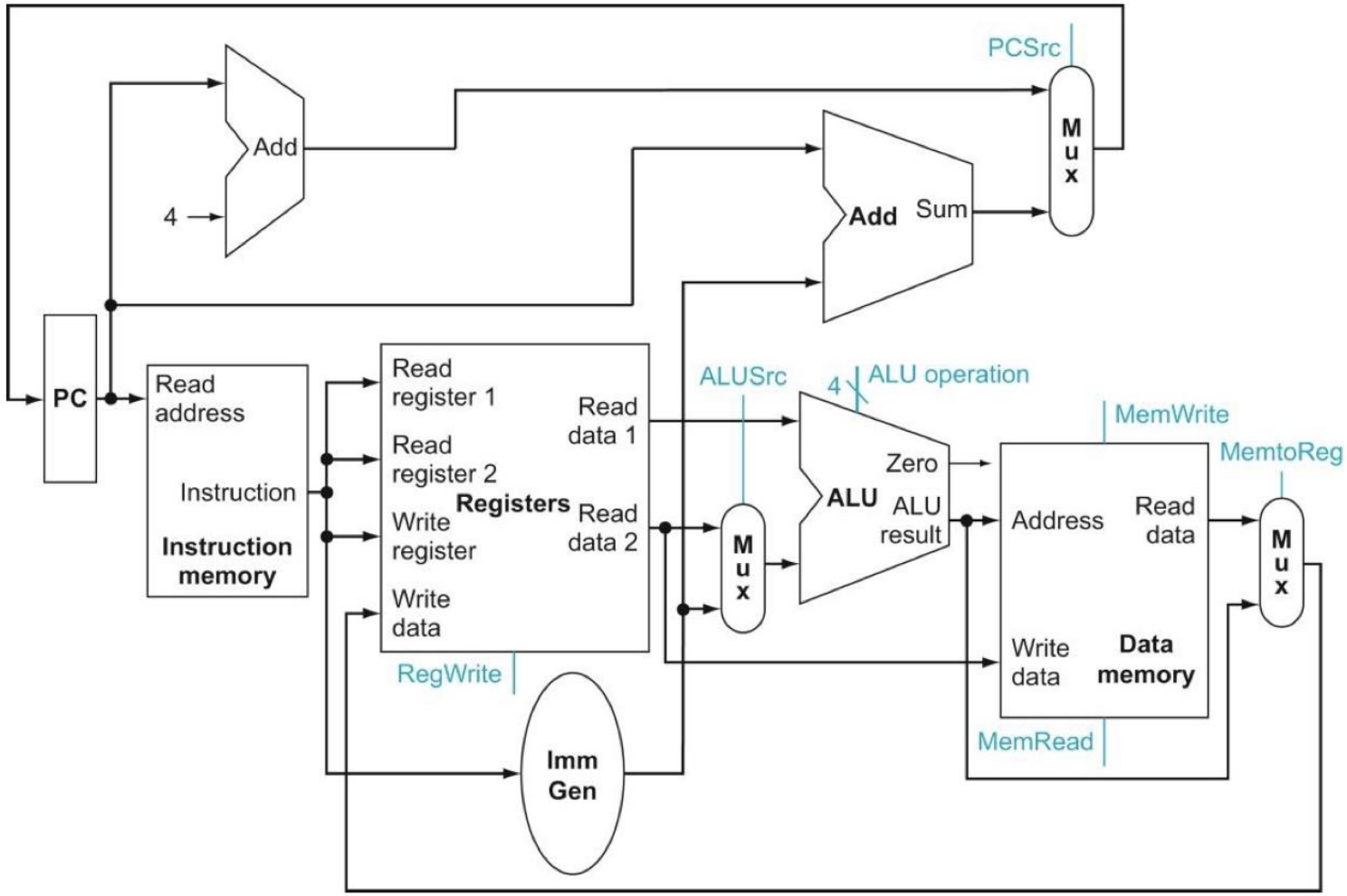
x3 remains 0x2000.

Memory at 0x2008 remains 42.

RegWrite = 1; rd = 6

Single-cycle datapath • steps show signal flow, not separate clock cycles.

SW | Next instruction: store



sw x6, 12(x3)

PC = 0x1008

x3 = 0x2000; x6 = 42

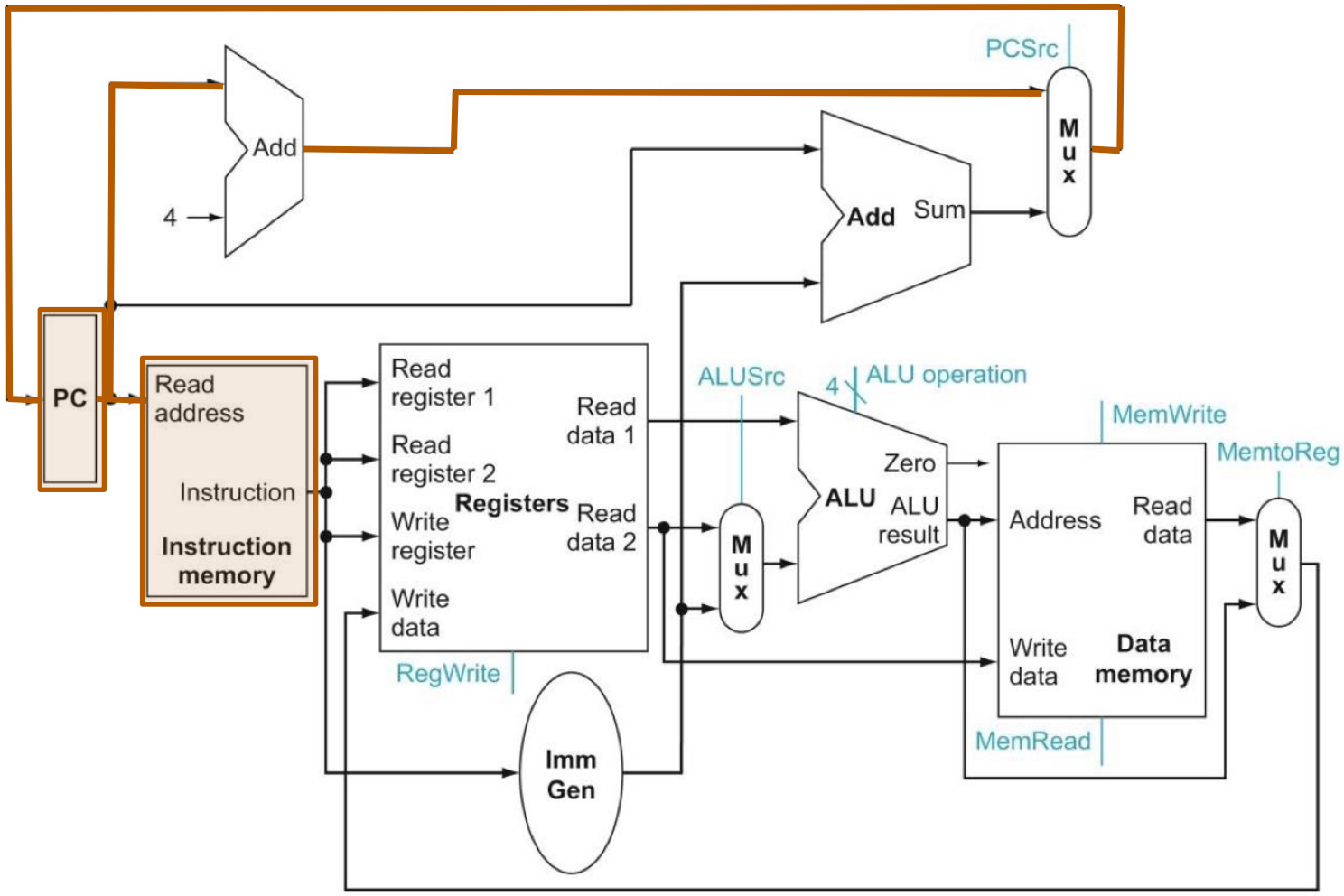
Memory at 0x200C = 0

Predict: what will change?

Store x6 at address x3 + 12.

Single-cycle datapath • steps show signal flow, not separate clock cycles.

SW | 1 Fetch



sw x6, 12(x3)

Fetch sw x6, 12(x3).

Instruction address: 0x1008

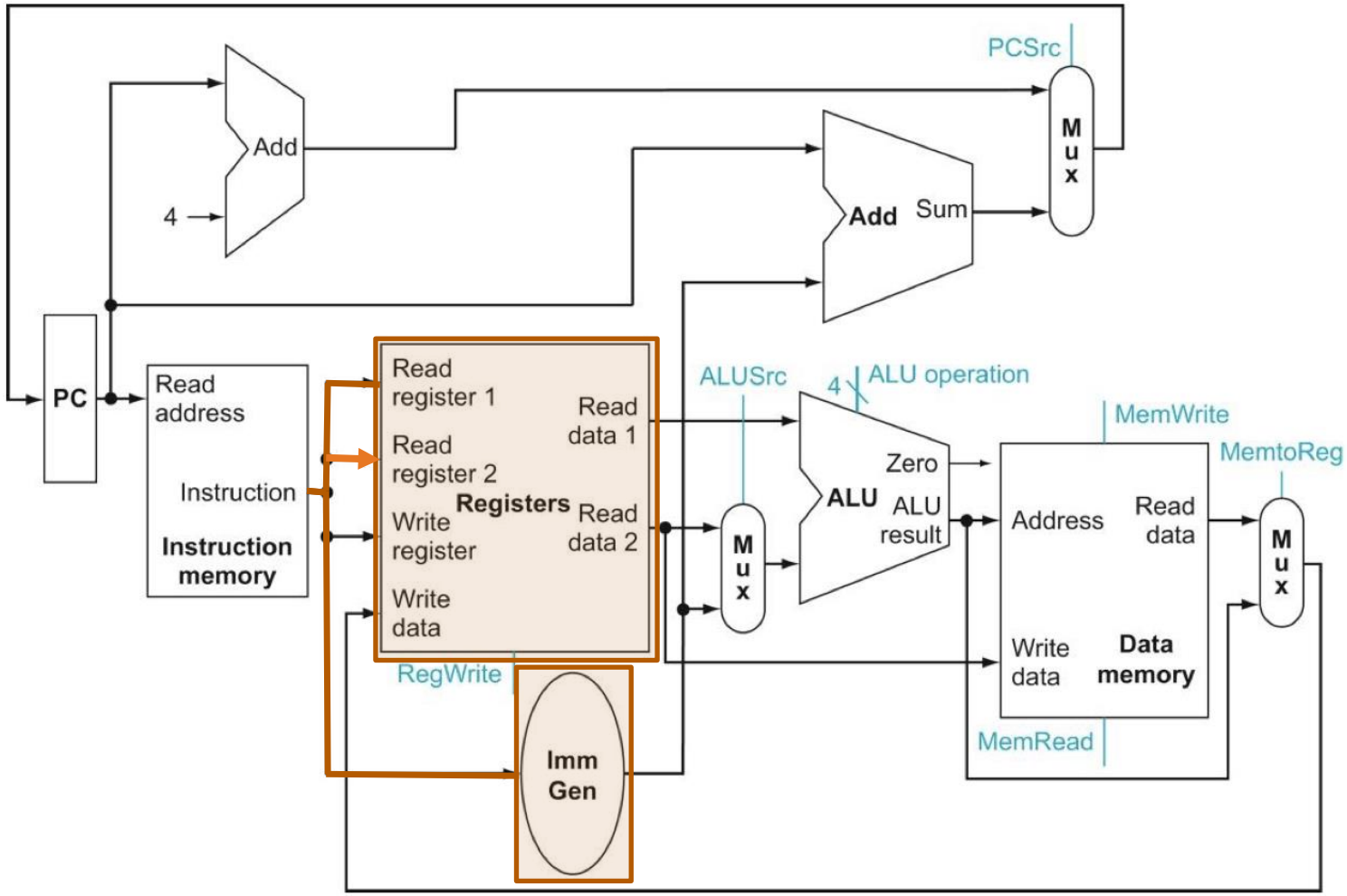
PC + 4 = 0x100C

PC changes at the edge.

PCSrc = 0 → select PC + 4

Single-cycle datapath • steps show signal flow, not separate clock cycles.

SW | 2 Decode / read



sw x6, 12(x3)

rs1 = 3 → base = 0x2000

rs2 = 6 → store data = 42

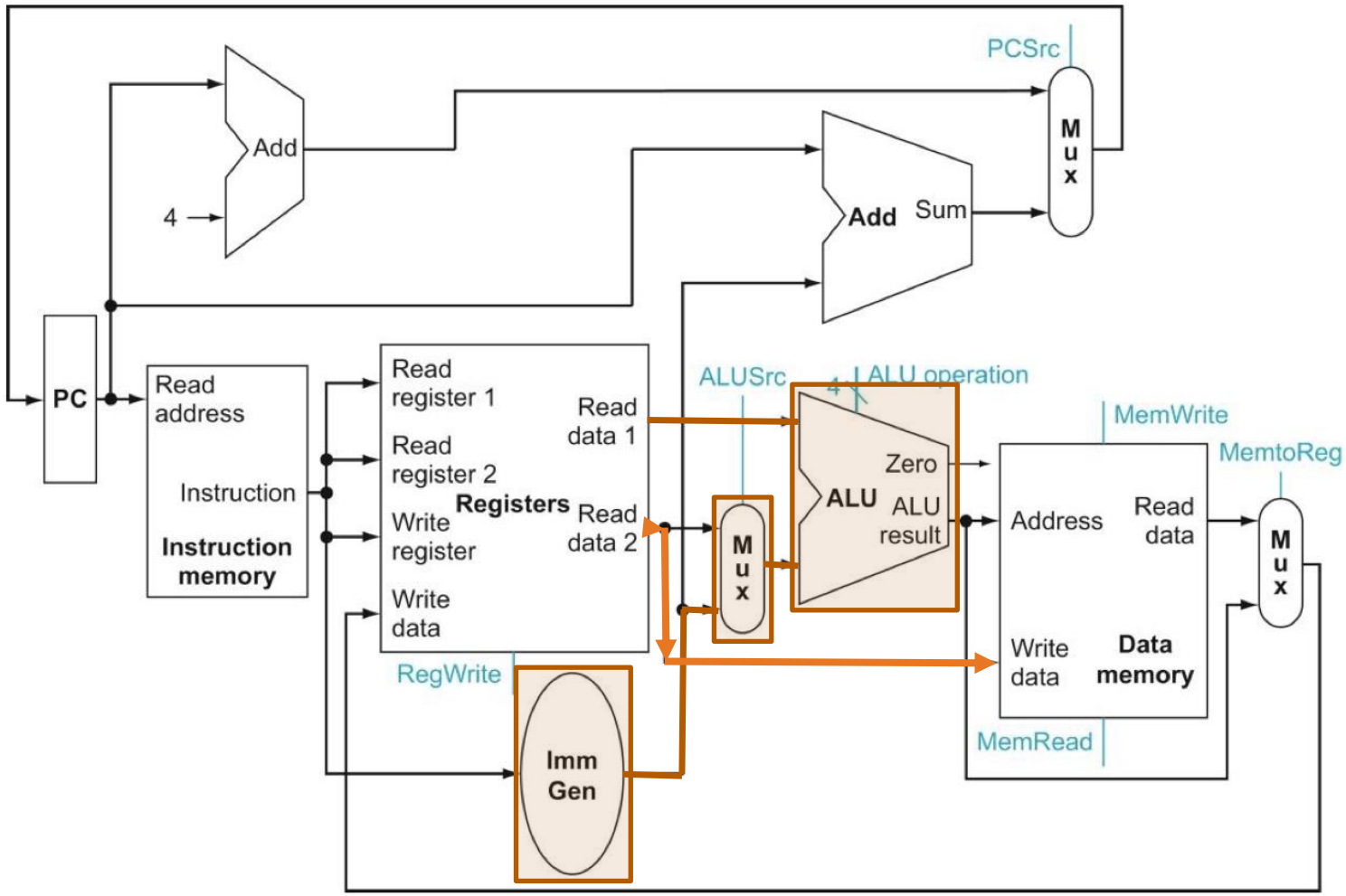
S-type immediate = 12

No destination register rd.

Join bits 31:25 and 11:7; then sign-extend.

Single-cycle datapath • steps show signal flow, not separate clock cycles.

SW | 3 Compute the address



sw x6, 12(x3)

ALU input A = 0x2000

ALU input B = 12 (0xC)

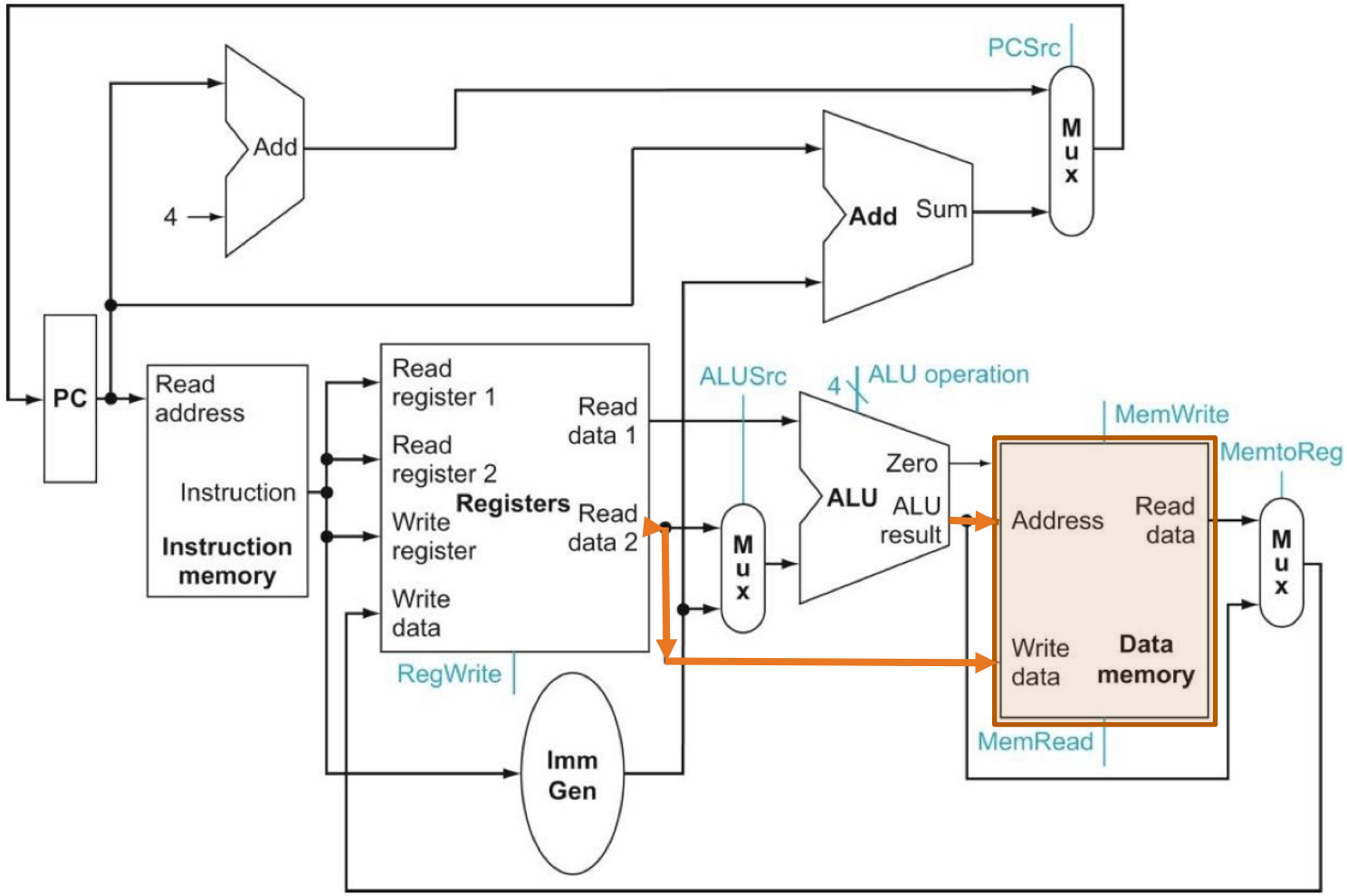
Address = 0x200C

42 follows a separate path.

ALUSrc = 1; ALU = ADD

Single-cycle datapath • steps show signal flow, not separate clock cycles.

SW | 4 Prepare the memory write



sw x6, 12(x3)

Address ← ALU: 0x200C

Write data ← x6: 42

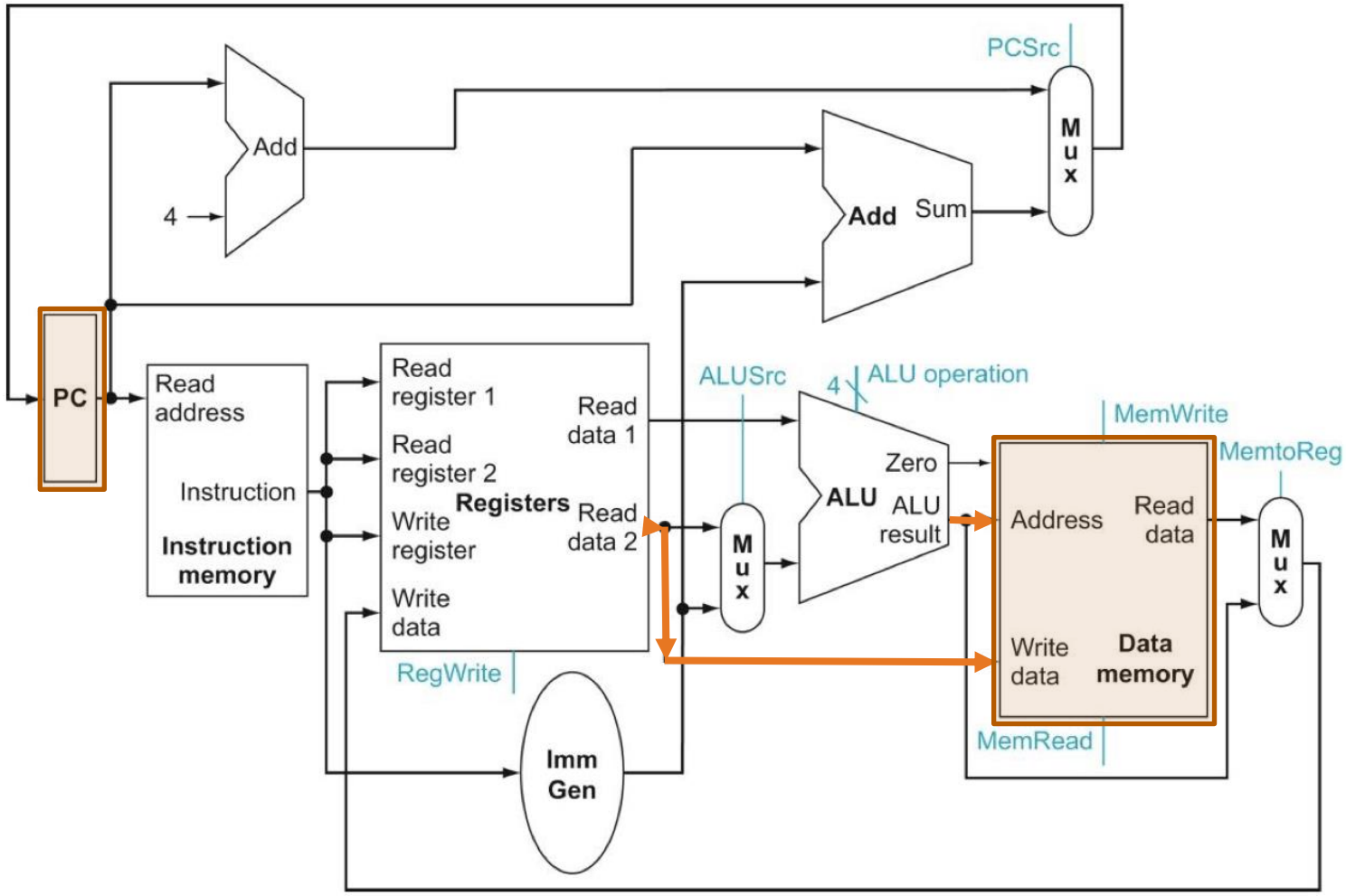
Memory stores the data, not the address.

MemWrite = 1; MemRead = 0

Write commits at the edge.

Single-cycle datapath • steps show signal flow, not separate clock cycles.

SW | 5 Commit at the edge



sw x6, 12(x3)

Memory[0x200C] ← 42

PC ← 0x100C

x3 = 0x2000; x6 = 42

No register is written.

RegWrite = 0

MemtoReg = X (unused)

Single-cycle datapath • steps show signal flow, not separate clock cycles.



5 Stage RISC-V CPU!

Splitting our CPU Datapath into 5 Stages



- Instruction Fetch
  - Reading the instruction from memory
- Instruction Decode
  - Splitting up the instruction, generating control signals, and reading the register file
  - Jump address calculation
- Execute
  - ALU Operation
  - Branch address calculation
- Memory
  - Reading/writing data memory
- Writeback
  - Writing the result to the register file



Instructions pass through *all 5 stages*, but different work happens in each stage depending on the instruction type.

Summary Table

Behavior of instructions starts to diverge



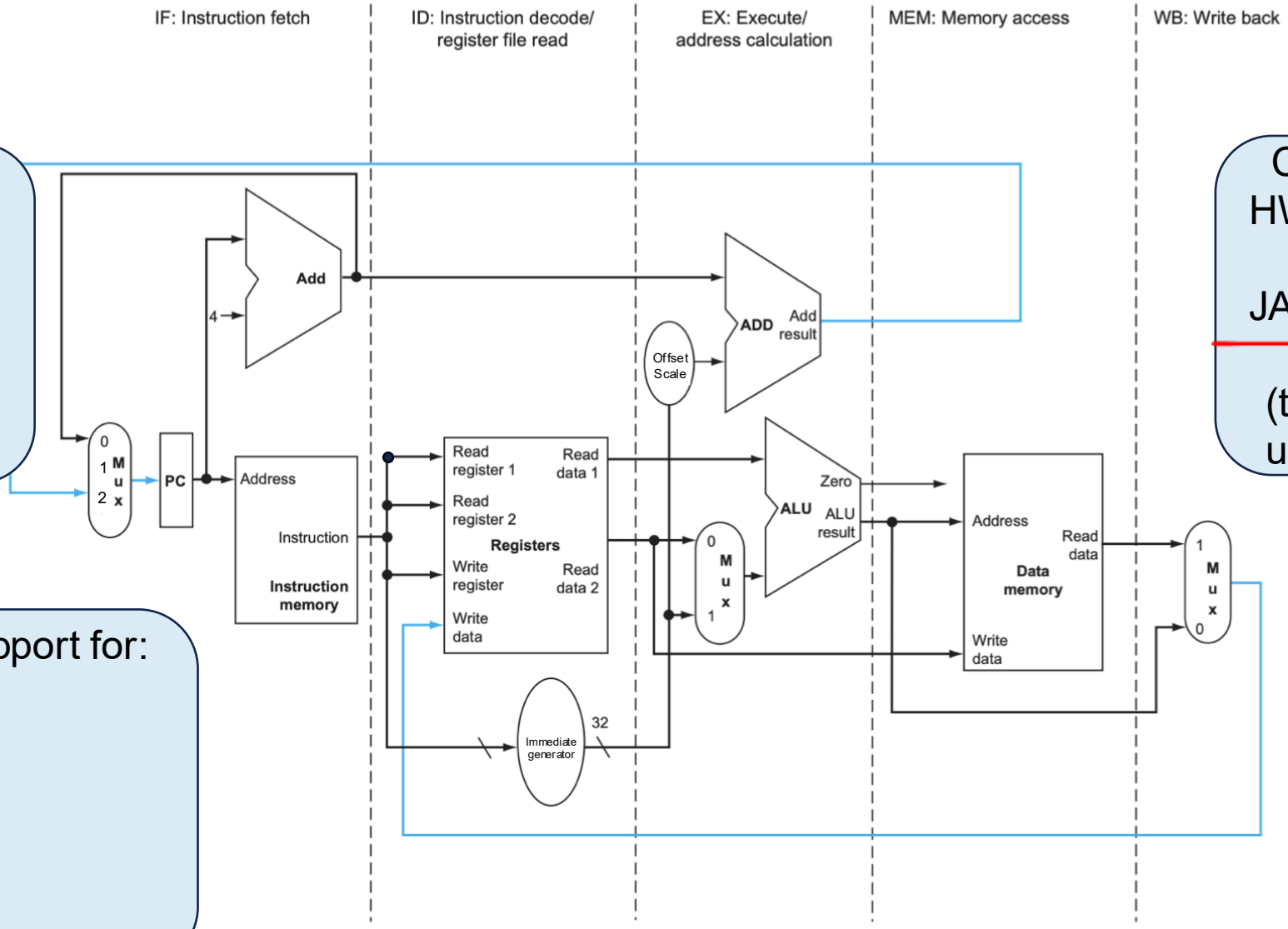
Instruction Type	IF	ID	EX	MEM	WB
`Arithmetic`	✔	✔	ALU	—	Write result
`LW`	✔	✔	Addr calc	Read mem	Write result
`SW`	✔	✔	Addr calc	Write mem	—
`Branch`	✔	✔	Compare + branch	—	—
`Jump`	✔	✔	Jump target*	—	—

EX: Figure out what to do/action to take for specific instruction type

MEM: Only matters for loads/stores

WB: Only matters if we produce a value

Datapath Split into 5 Stages



This version of the full data path has some details abstracted away for readability/tracing purposes.

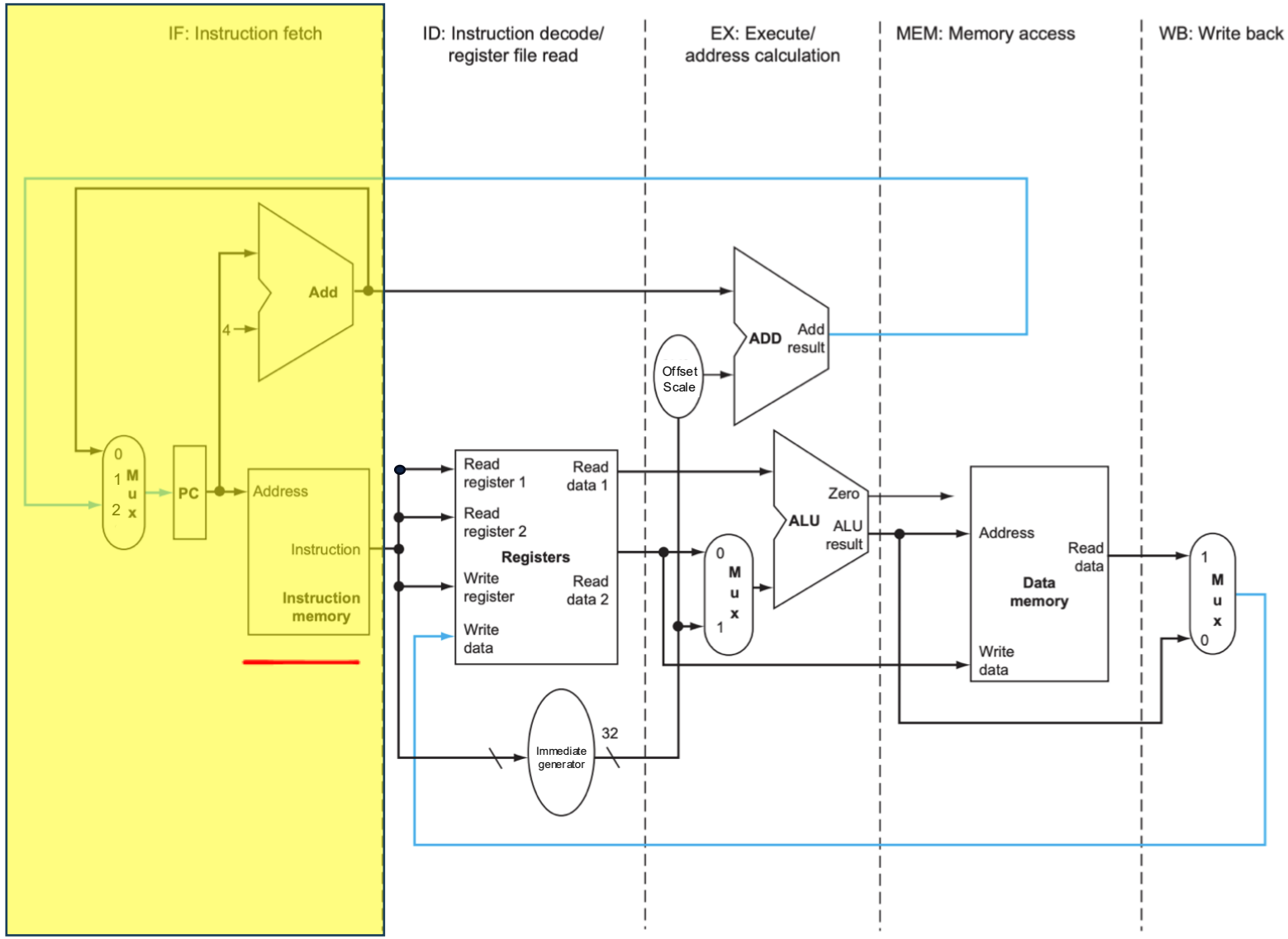
Challenge Q: What HW would need to be added to support JALR? Other types of branches? (the full ALU control unit also missing…)

This datapath has support for:

- R-type arithmetic
- I-type arithmetic
- Loads
- Stores
- BEQ
- JAL

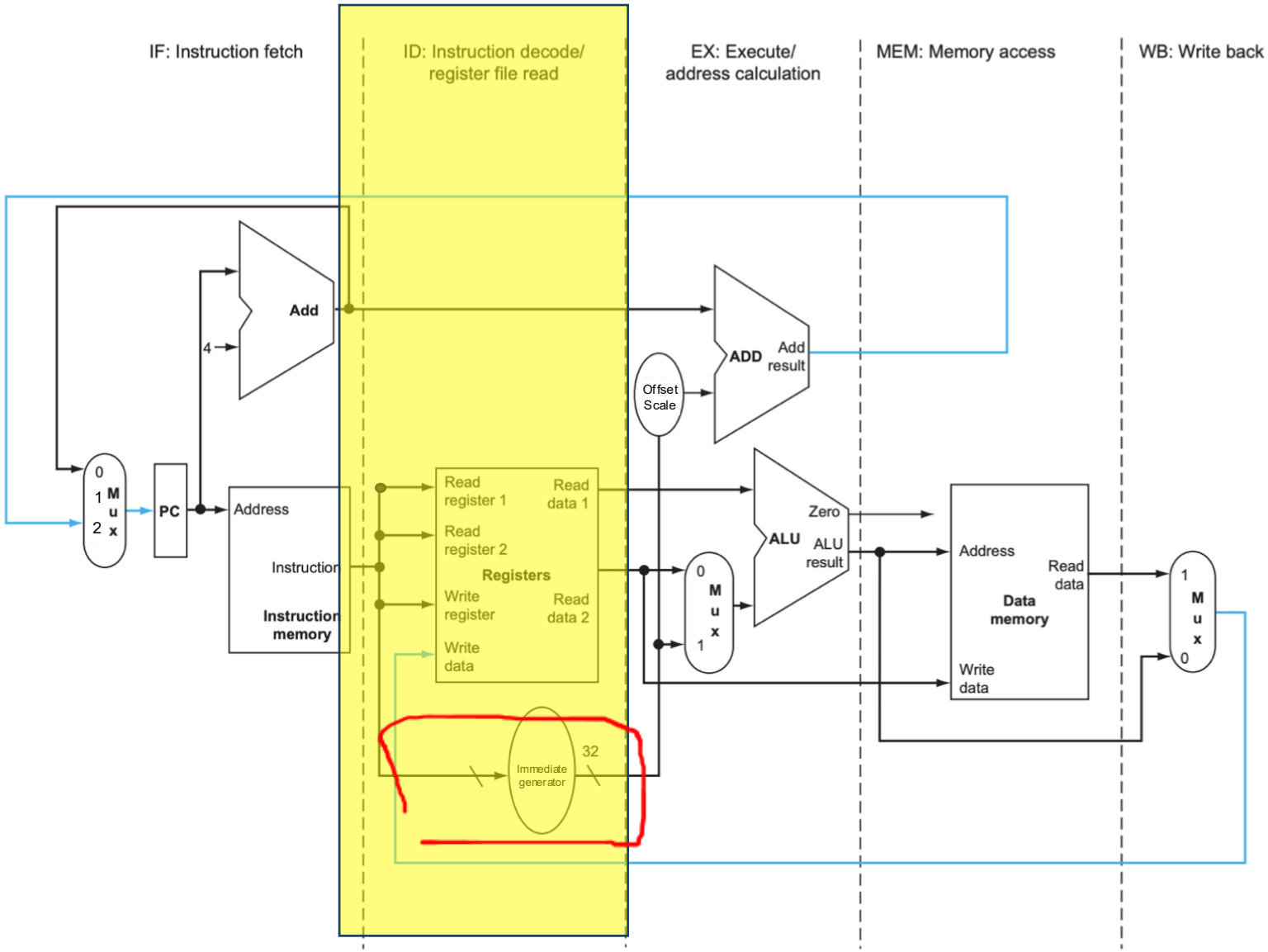
Instruction Fetch

Reading the instruction from memory



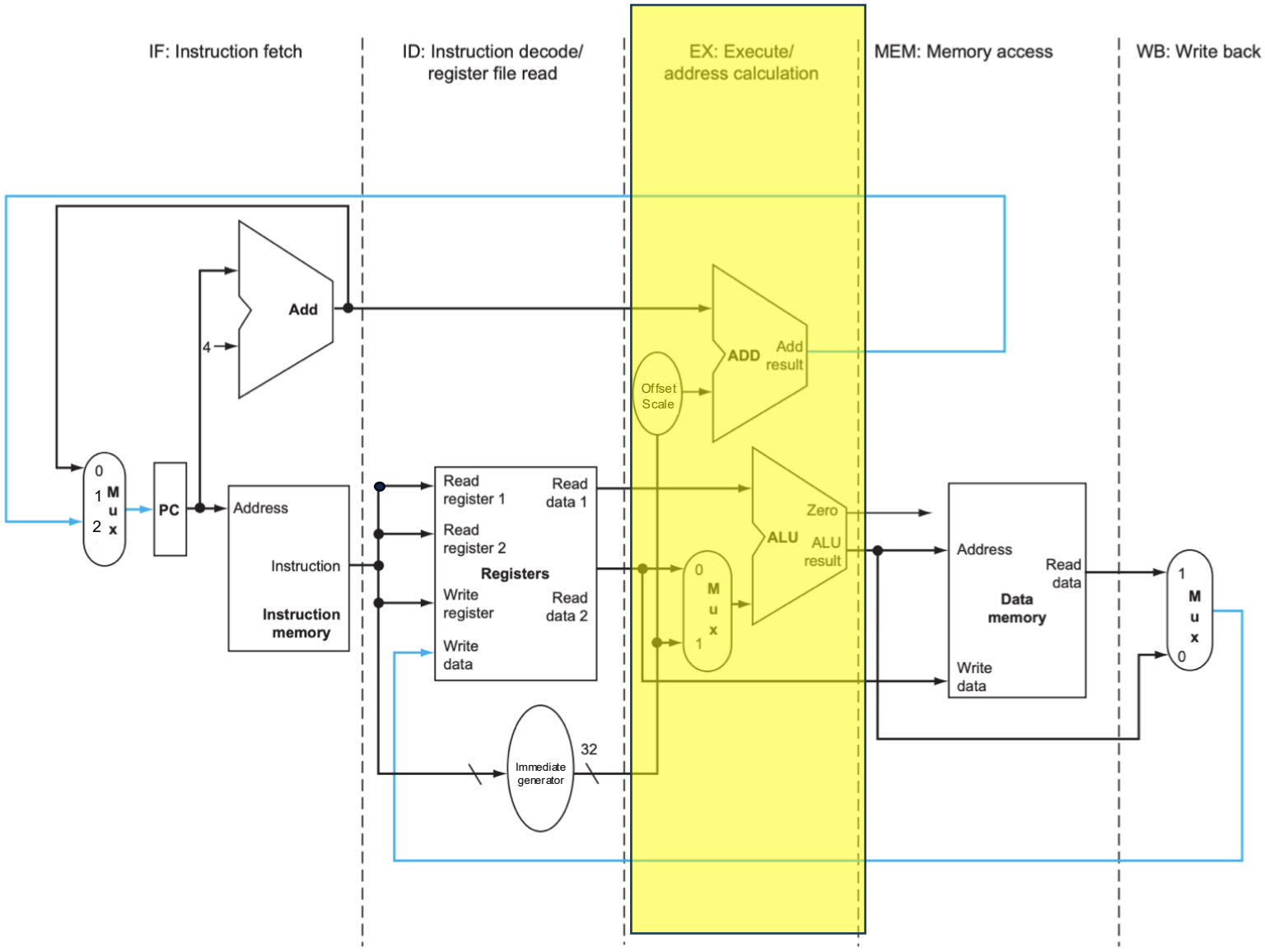
Instruction Decode

- Splitting up the instruction fields
- Generating control signals
- Reading the register file
- Jump address calculation



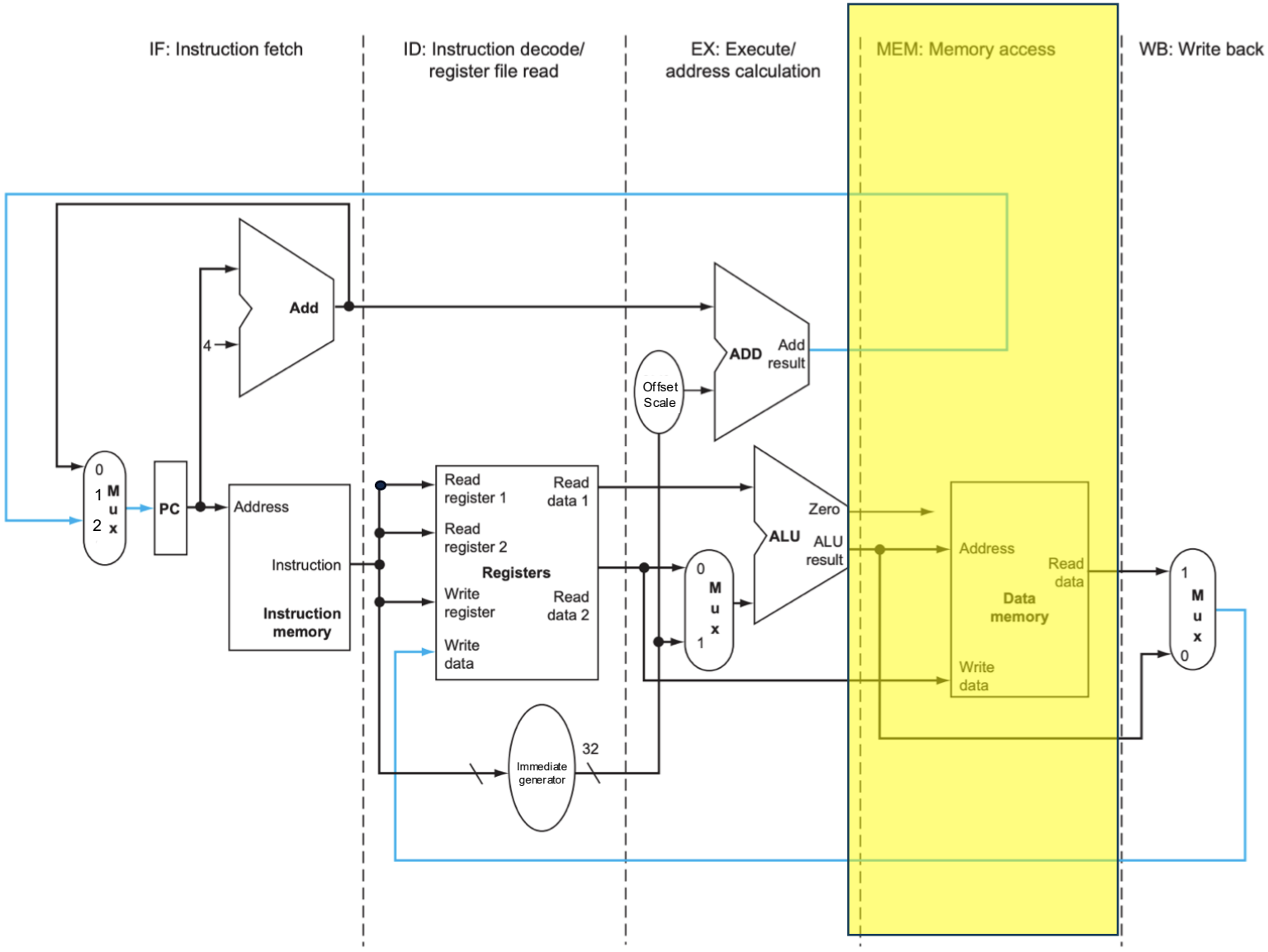
Execute

- ALU Operation
- Branch address calculation



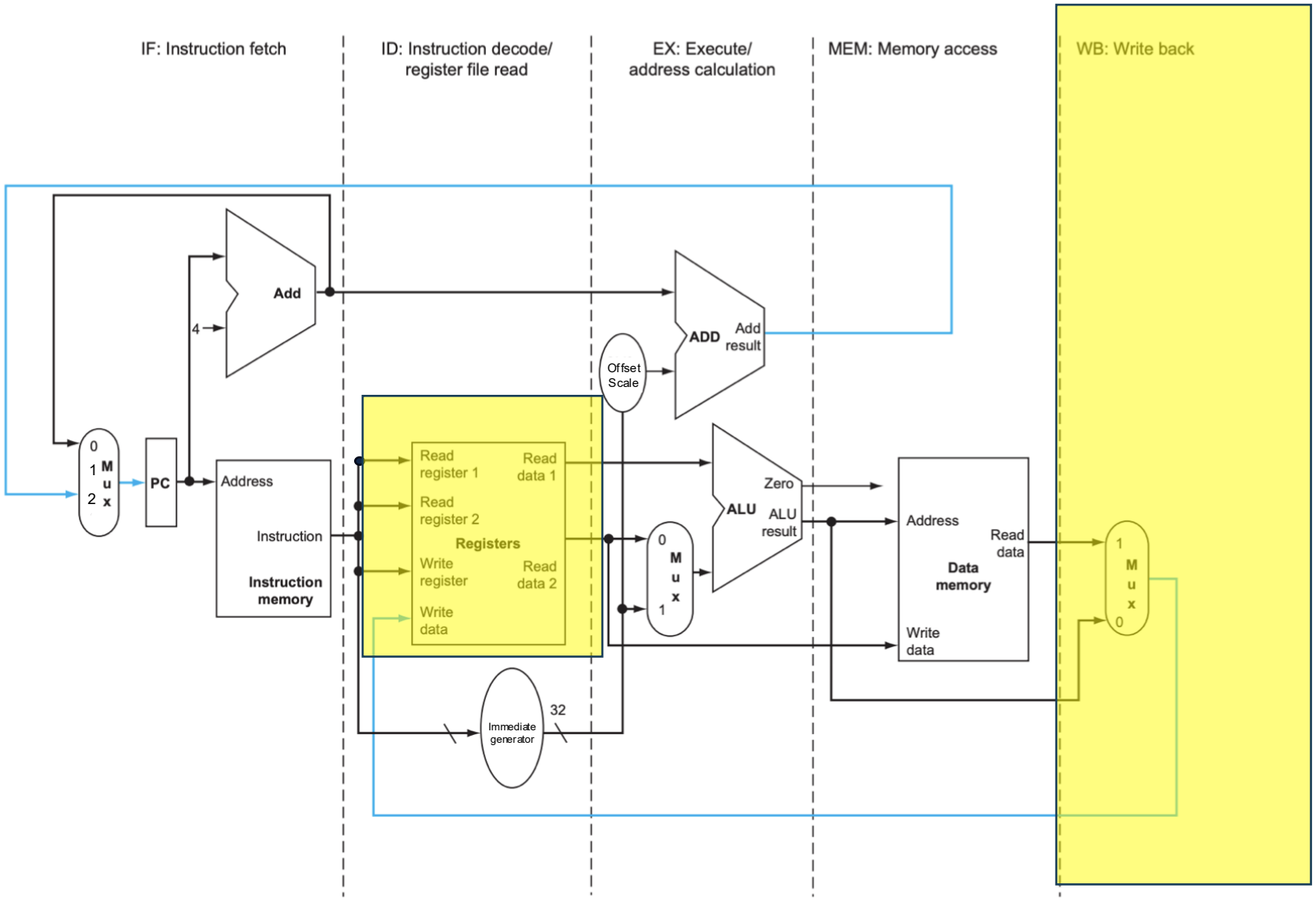
Memory

Reading/writing data memory



Writeback

Writing the result to the register file



Stages Used by Each Instruction



Instruction	Fetch	Decode	Execute	Memory	Writeback
arithmetic and logical (add, addi, or, etc)					
sw					
lw					
branch					
j					

Stages Used by Each Instruction



Instruction	Fetch	Decode	Execute	Memory	Writeback
arithmetic and logical (add, addi, or, etc)	X	X	X		X
sw					
lw					
branch					
j					

Stages Used by Each Instruction



Instruction	Fetch	Decode	Execute	Memory	Writeback
arithmetic and logical (add, addi, or, etc)	X	X	X		X
sw	X	X	X	X	
lw	X	X	X	X	X
branch	X	X	X		
j	X	X			

Time Needed for Each Stage



	Fetch	Decode	Execute	Memory	Writeback
Time	200ps	100ps	200ps	200ps	100ps

Time Needed for Each Instruction



	Fetch	Decode	Execute	Memory	Writeback
Time	200ps	100ps	200ps	200ps	100ps

Instruction	Fetch	Decode	Execute	Memory	Writeback	Total Time
arithmetic and logical (add, addi, or, etc)	X	X	X		X	200ps + 100ps + 200ps + 100ps = 600ps
sw						
lw						
branch						
j						

Time Needed for Each Instruction



	Fetch	Decode	Execute	Memory	Writeback
Time	200ps	100ps	200ps	200ps	100ps

Instruction	Fetch	Decode	Execute	Memory	Writeback	Total Time
arithmetic and logical (add, addi, or, etc)	X	X	X		X	200ps + 100ps + 200ps + 100ps = 600ps
sw	X	X	X	X		200 + 100 + 200 + 200 = 700ps
lw	X	X	X	X	X	200 + 100 + 200 + 200 + 100 = 800ps
branch	X	X	X			200 + 100 + 200 = 500ps
j	X	X				200 + 100 = 300ps

Min Clock Period



- What is the minimum clock period that will work for all instructions?
  - Assuming a clock-to-q delay of 20ps

Even though instructions conceptually pass through all stages in a pipeline, for this question assume a **single-cycle datapath** where only the required components contribute to delay

Single Cycle Datapath Timing

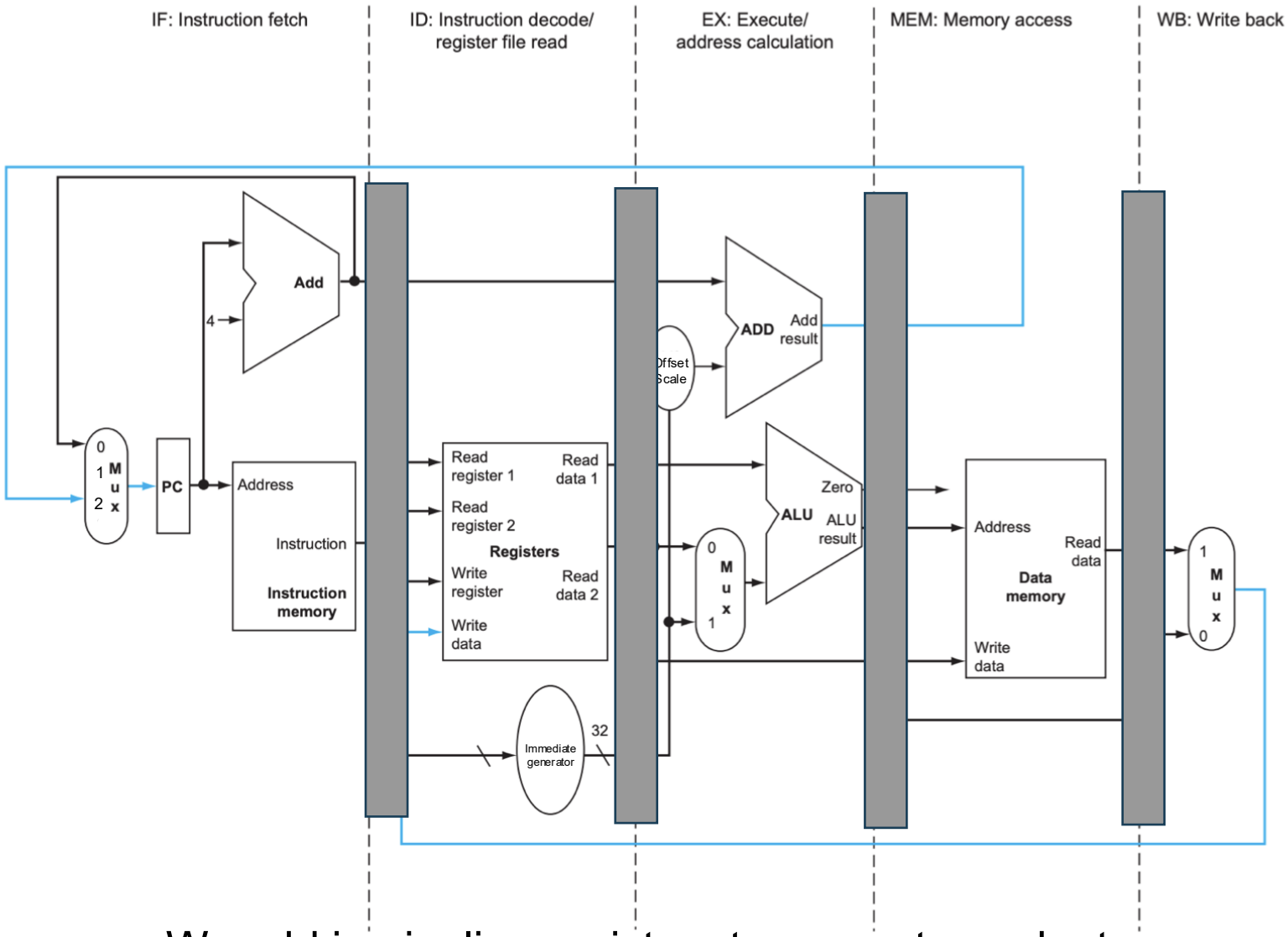


- It takes one clock cycle to execute one instruction
- If we need to execute 100 instructions, it will take 100 clock cycles
- If we set our clock period to 820ps, it will take
  - 820ps to execute one instruction
  - 82,000ps to execute 100 instructions



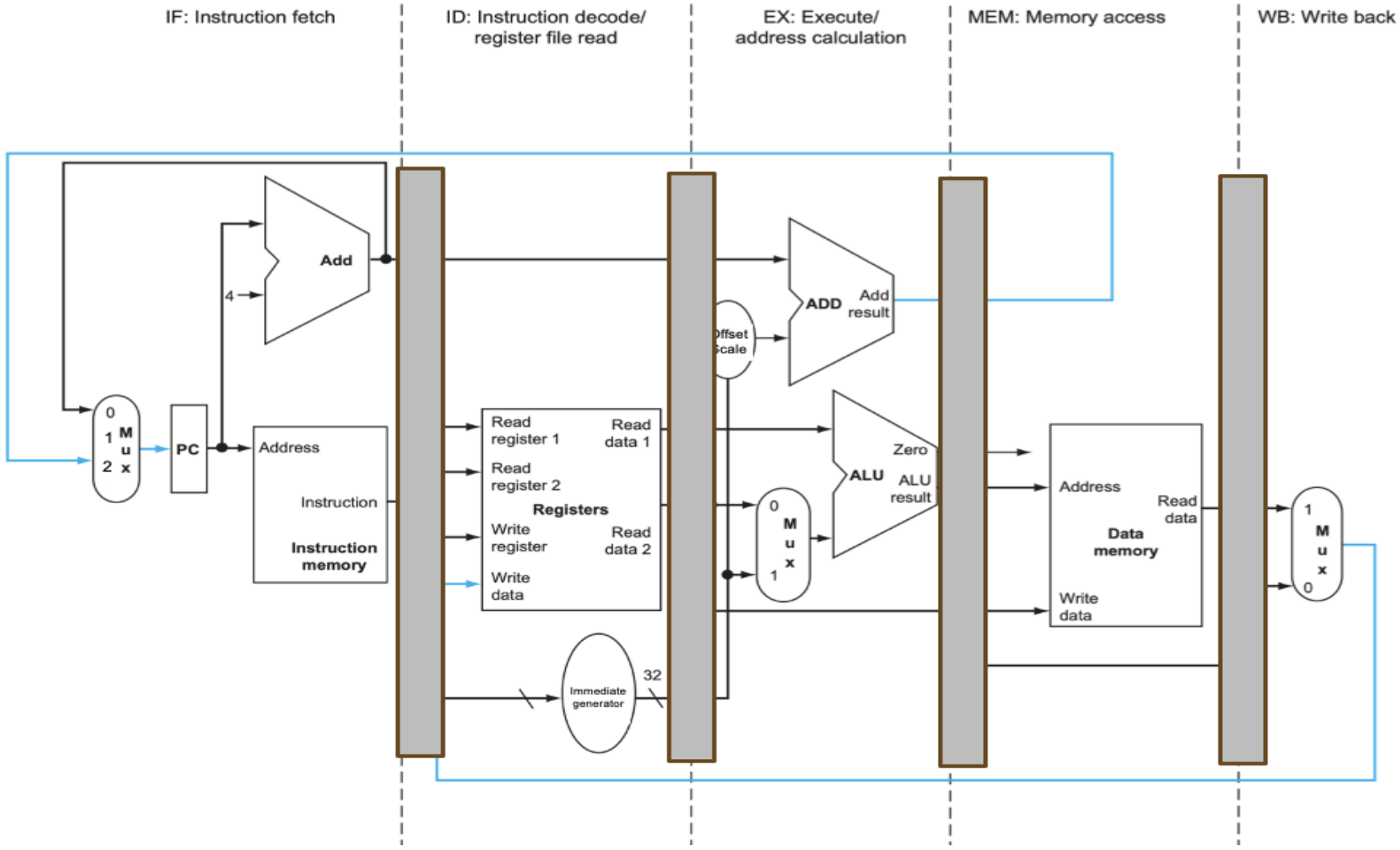
Pipelining the RISC-V CPU! ☺

5-Stage Pipeline



We add in pipeline registers to separate each stage

Each pipeline register name is based on the stages it separates



Register Names

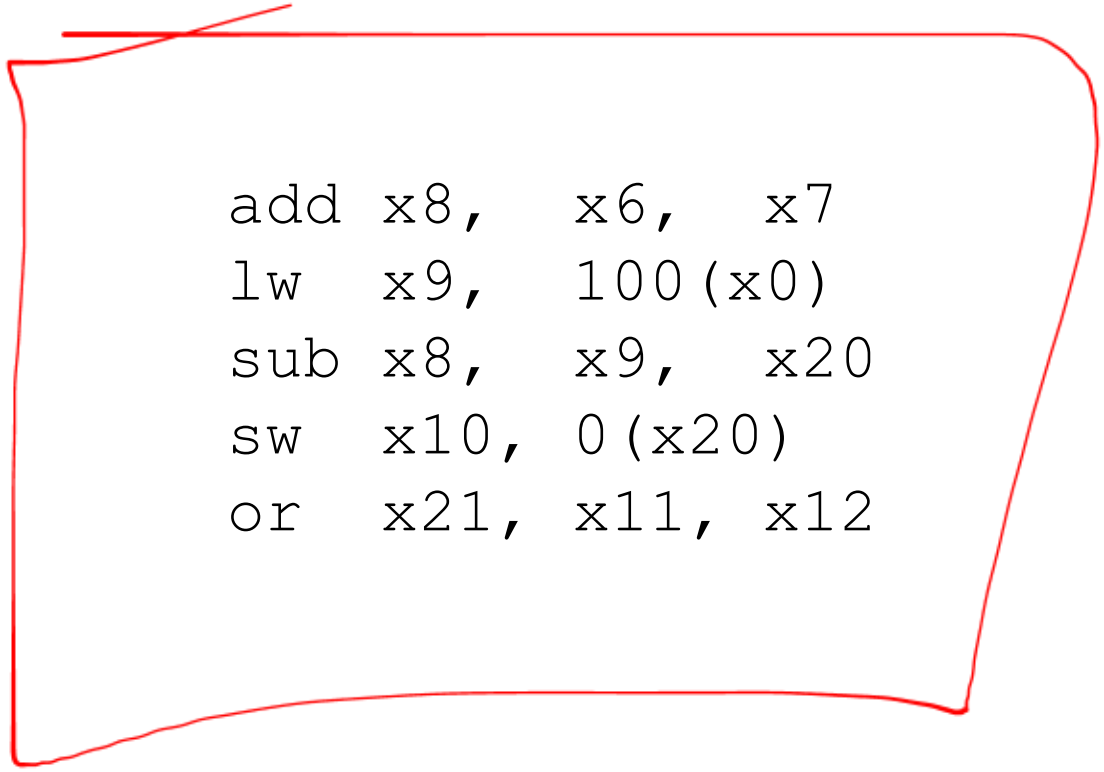
IF/ID

ID/EX

EX/MEM

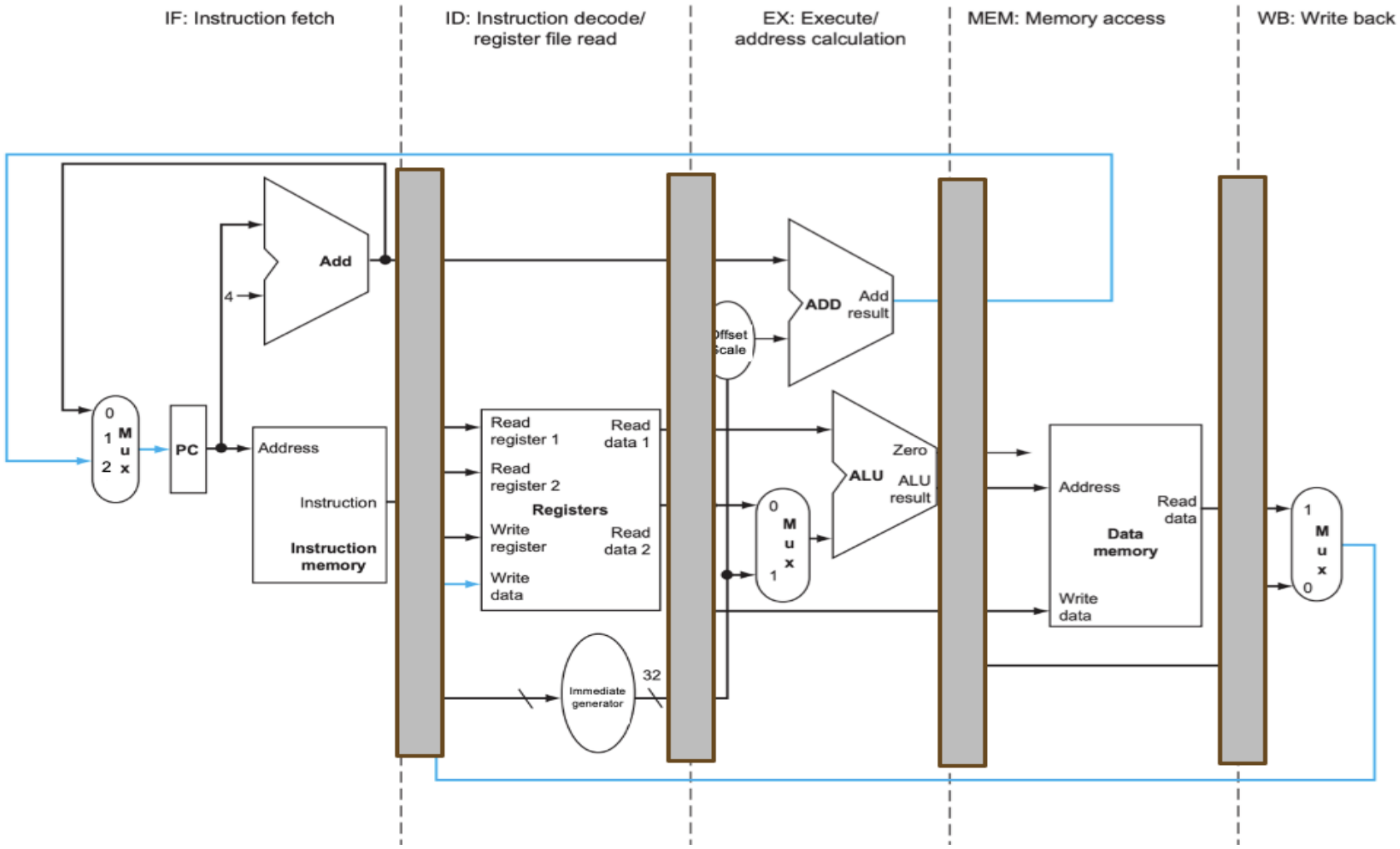
MEM/WB

Executing Instructions on Our Pipelined Datapath



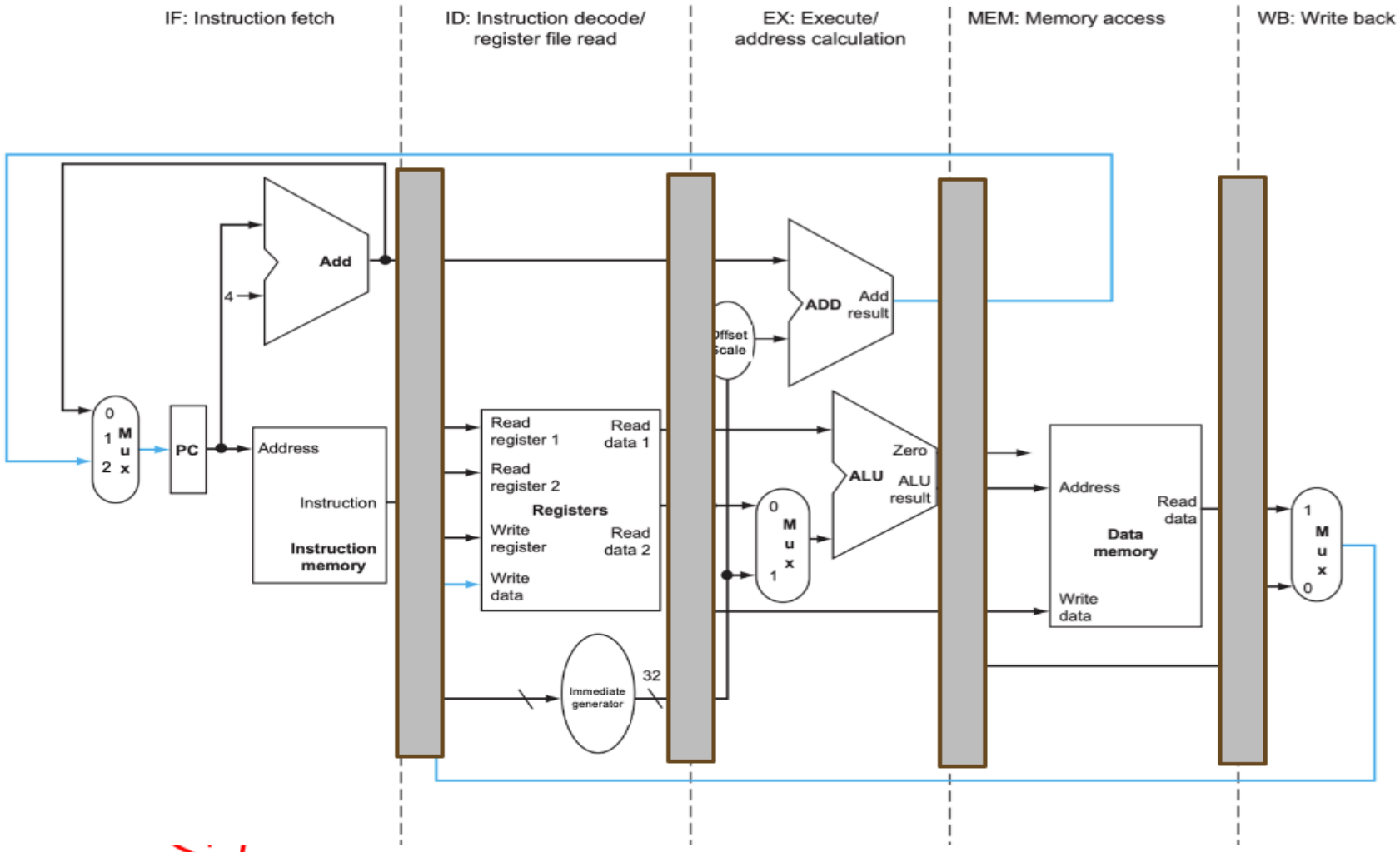
```
add x8,   x6,   x7
lw  x9,   100(x0)
sub x8,   x9,   x20
sw  x10,  0(x20)
or  x21,  x11,  x12
```

Cycle 0




```
add x8, x6, x7
```

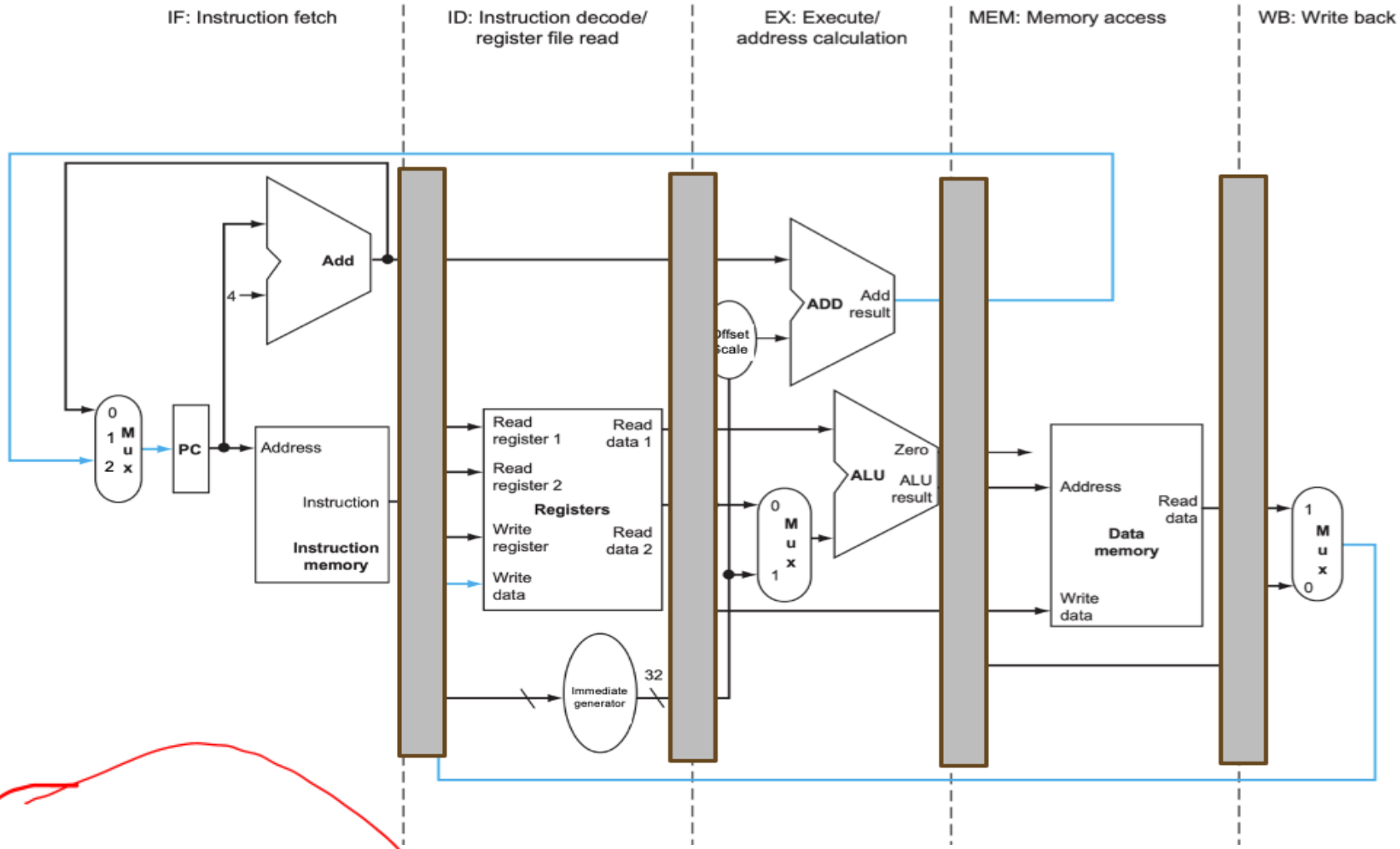
Cycle 1




```
lw x9, 100(x0)
```

```
add x8, x6, x7
```

Cycle 2

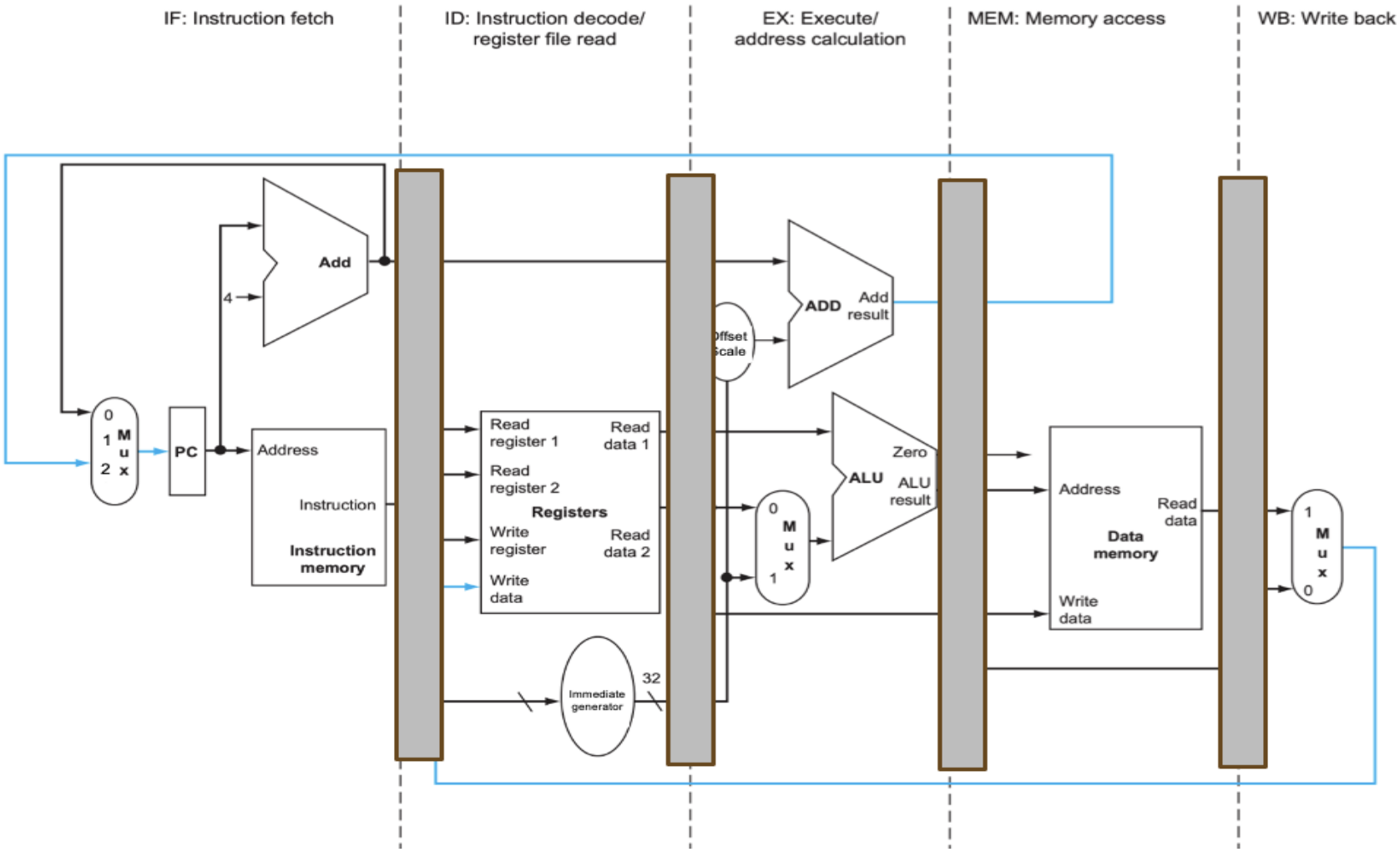



```
sub x8, x9, x20
```

```
lw x9, 100(x0)
```

```
add x8, x6, x7
```

Cycle 3

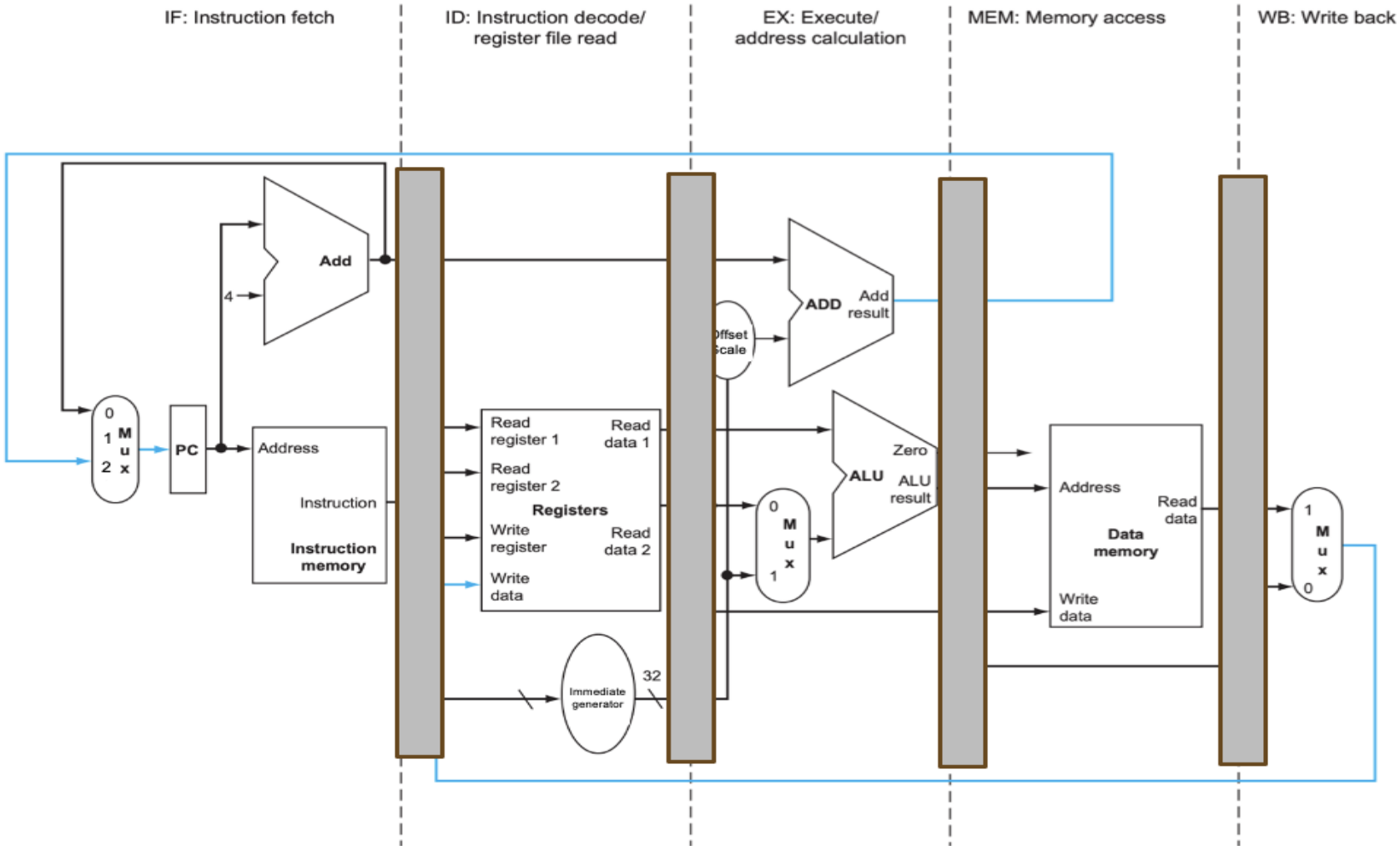


sw x10, 0(x20)

sub x8, x9, x20

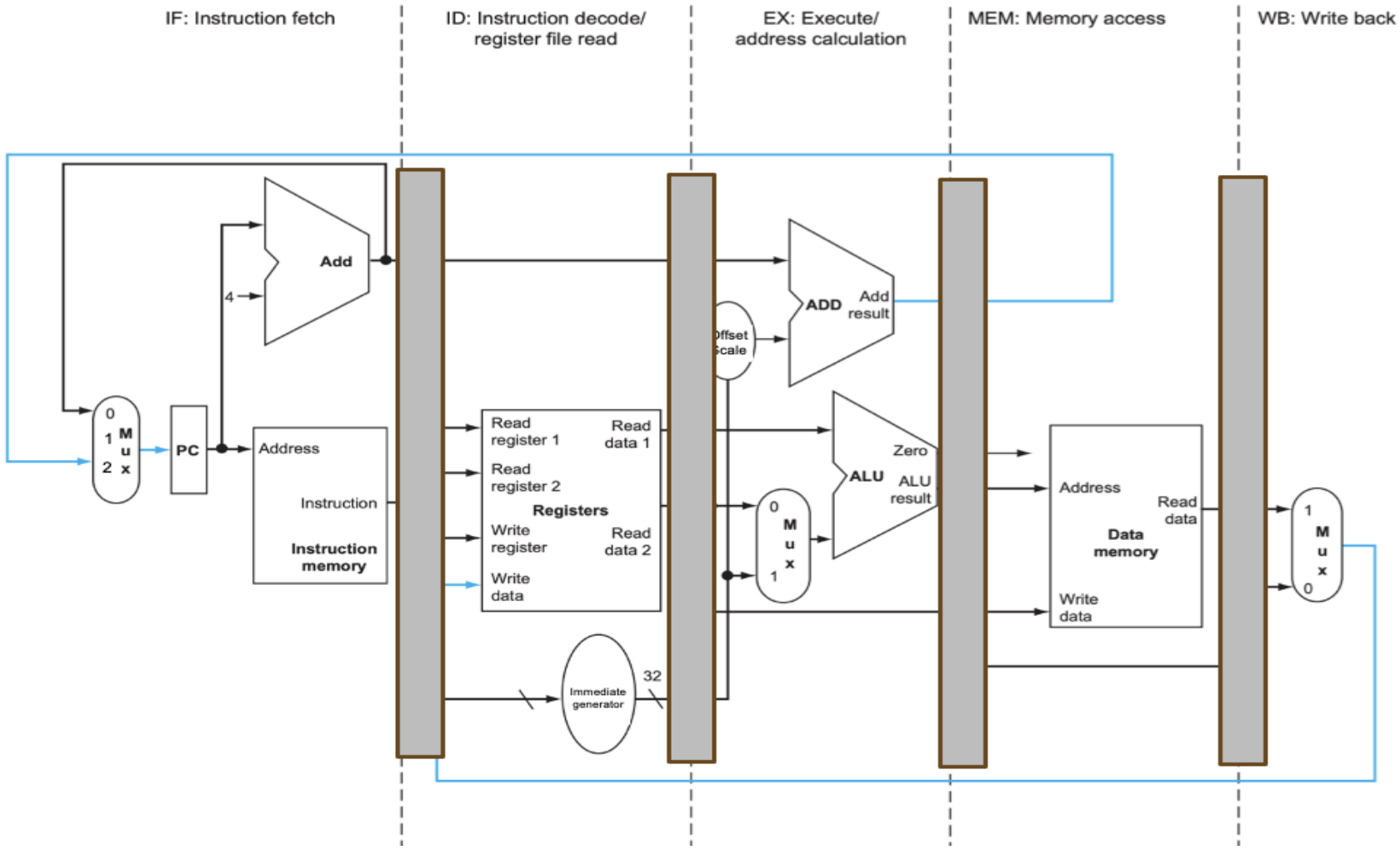
lw x9, 100(x0) add x8, x6, x7

Cycle 4



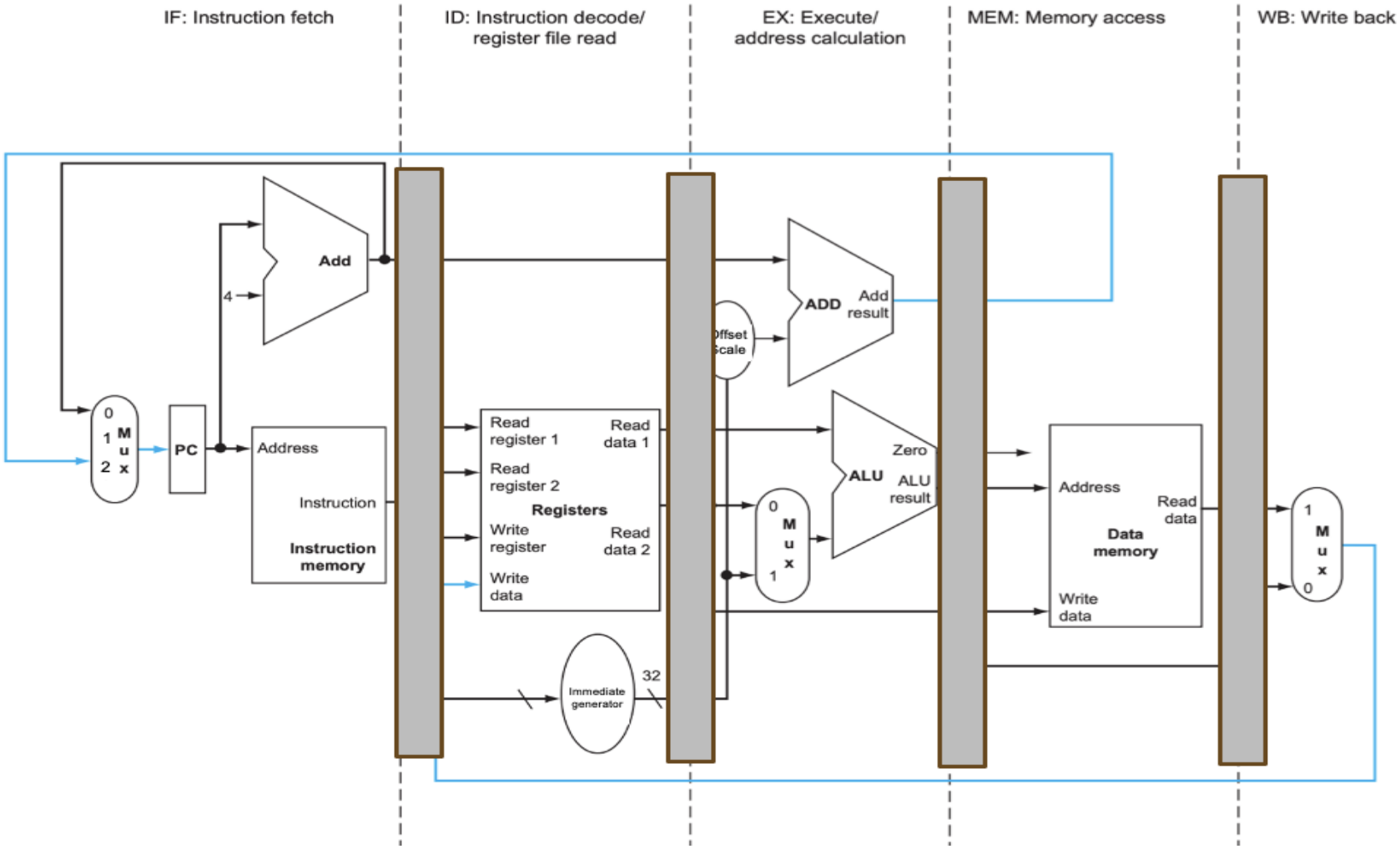
or x21, x11, x12 sw x10, 0(x20) sub x8, x9, x20 lw x9, 100(x0) add x8, x6, x7

Cycle 5



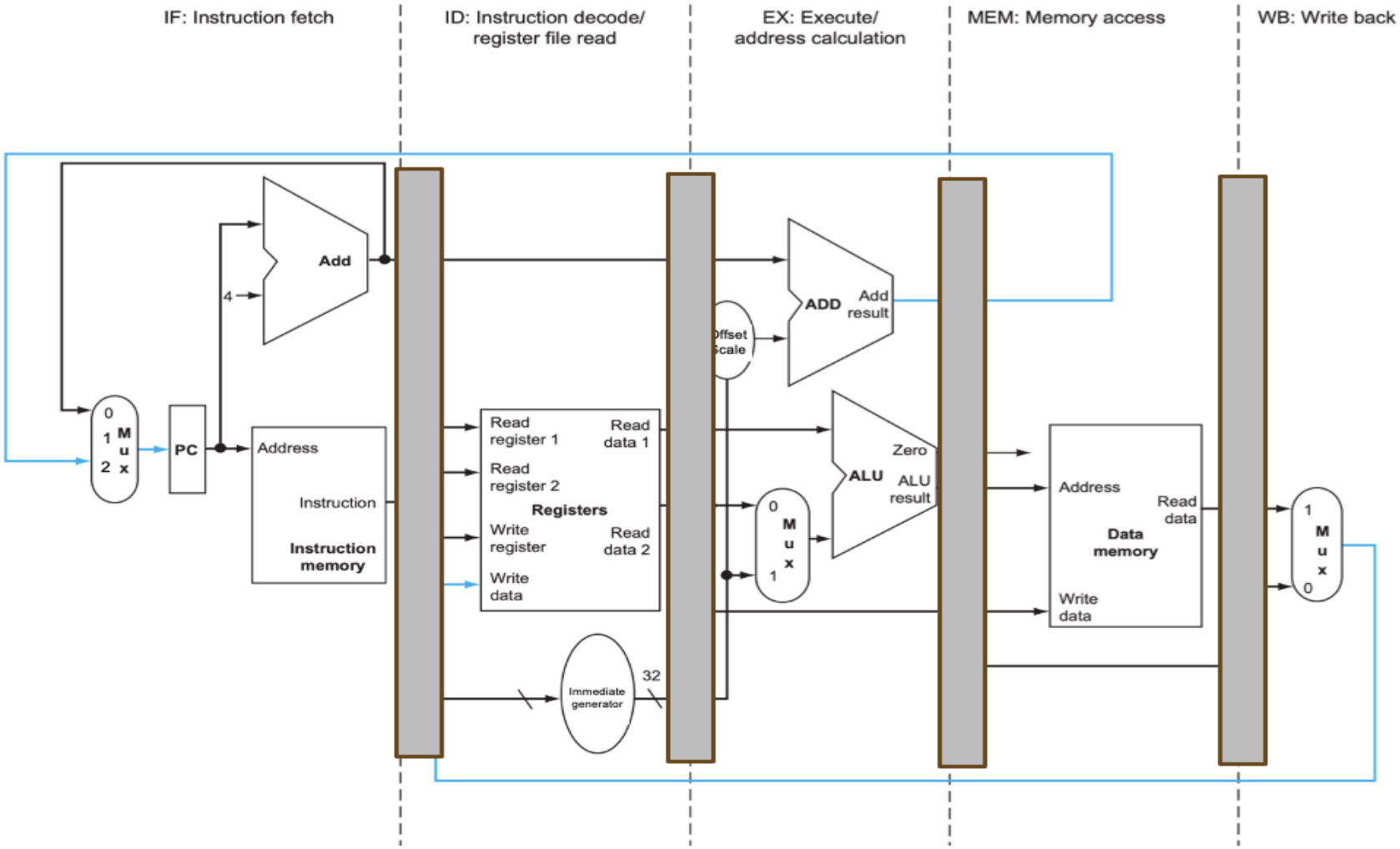
or x21, x11, x12 sw x10, 0(x20) `sub x8, x9, x20` `lw x9, 100(x0)`

Cycle 6



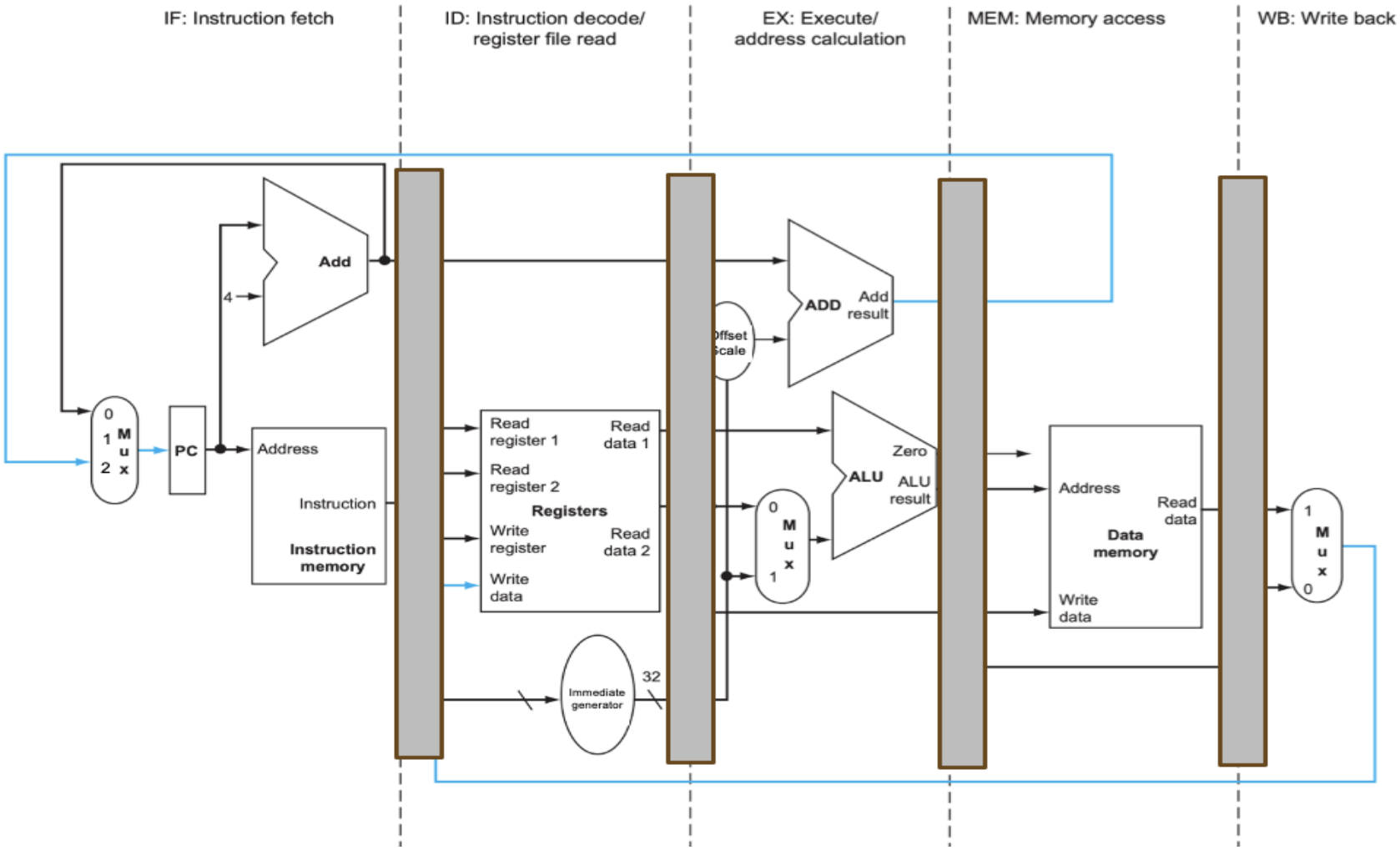
or x21, x11, x12 sw x10, 0(x20) `sub x8, x9, x20`

Cycle 7



or x21, x11, x12 sw x10, 0(x20)

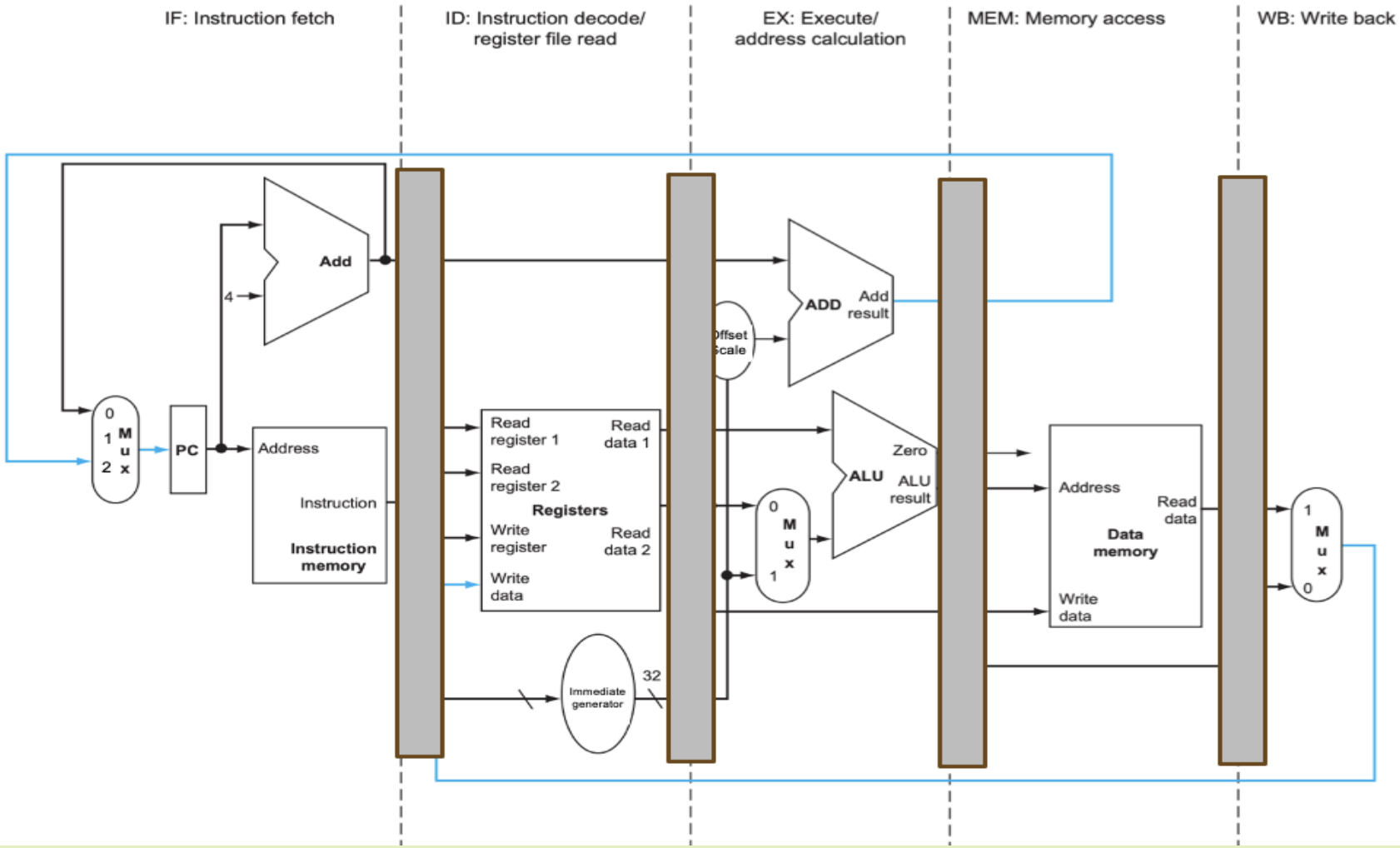
Cycle 8



or x21, x11, x12

Determine the Min Clock Period

The clock-to-q delay of a pipeline register is 20ps



200ps 100ps 200ps 200ps 100ps

Latency



- What is the latency of executing a single instruction on
  - The single cycle datapath with a clock period of 820ps?
  - The 5-stage pipelined datapath with a clock period of 220ps?

Speedup



- What is the speedup of executing the following set of instructions on a single-cycle vs pipelined datapath assuming
  - single-cycle clock period = 820 ps
  - pipelined clock period = 220 ps

$$speedup = \frac{single\ cyle\ execution\ time}{pipelined\ execution\ time}$$

```
add x8,   x6,   x7
lw  x9,   100(x0)
sub x8,   x9,   x20
sw  x10,  0(x20)
or  x21,  x11,  x12
```



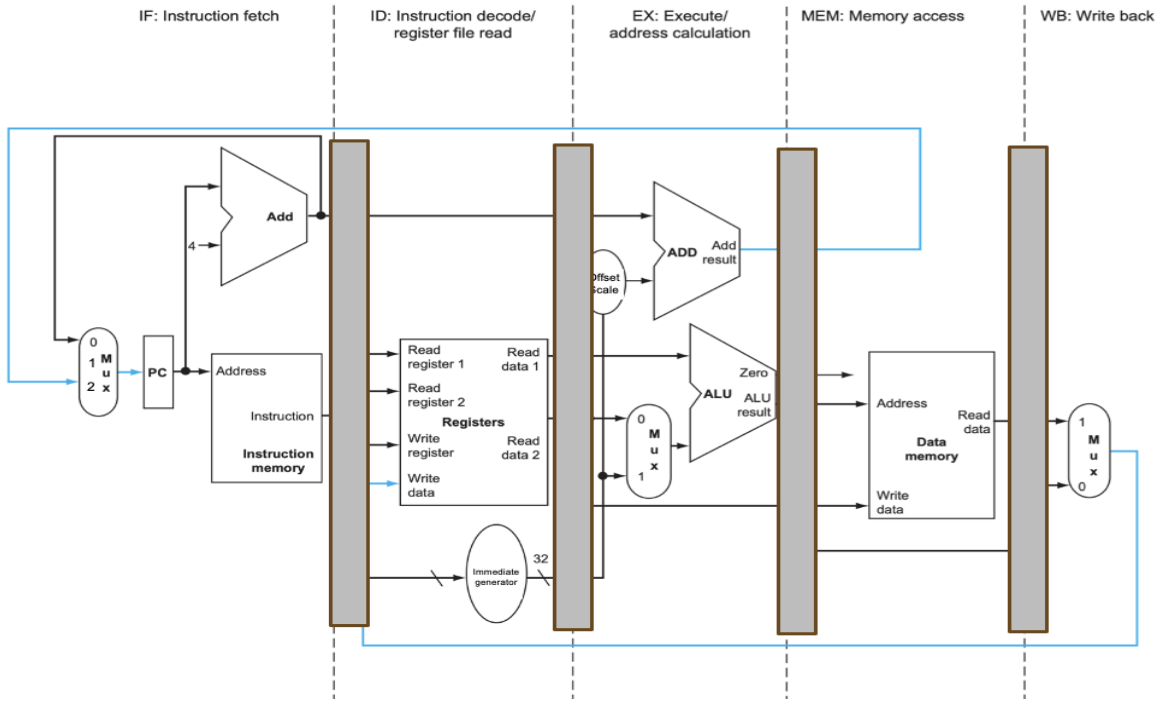
Pipeline Hazards

Data Hazard Tracing

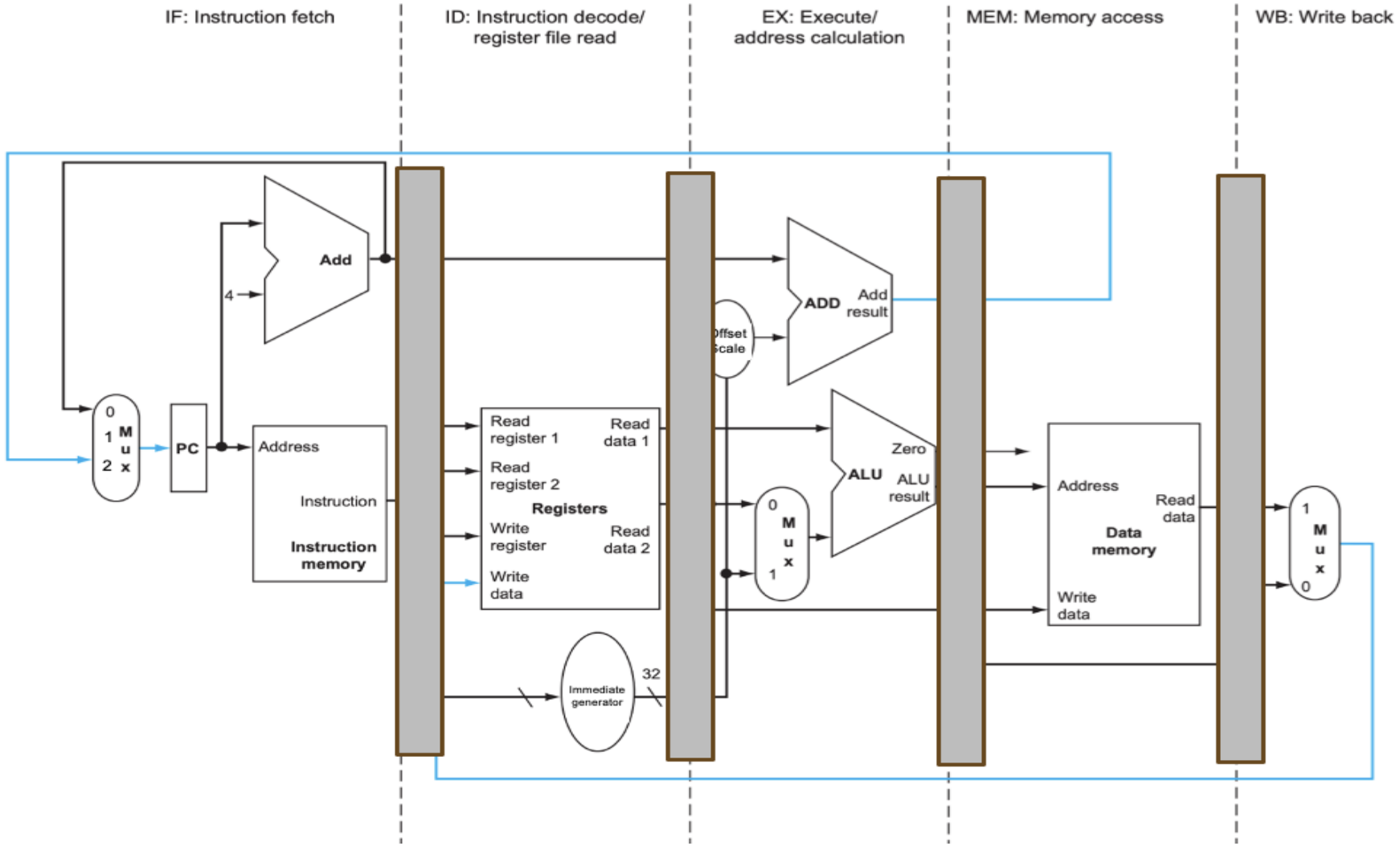
Let's say we want to execute the following two instructions sequentially

add x5, x3, x4

sub x7, x5, x2

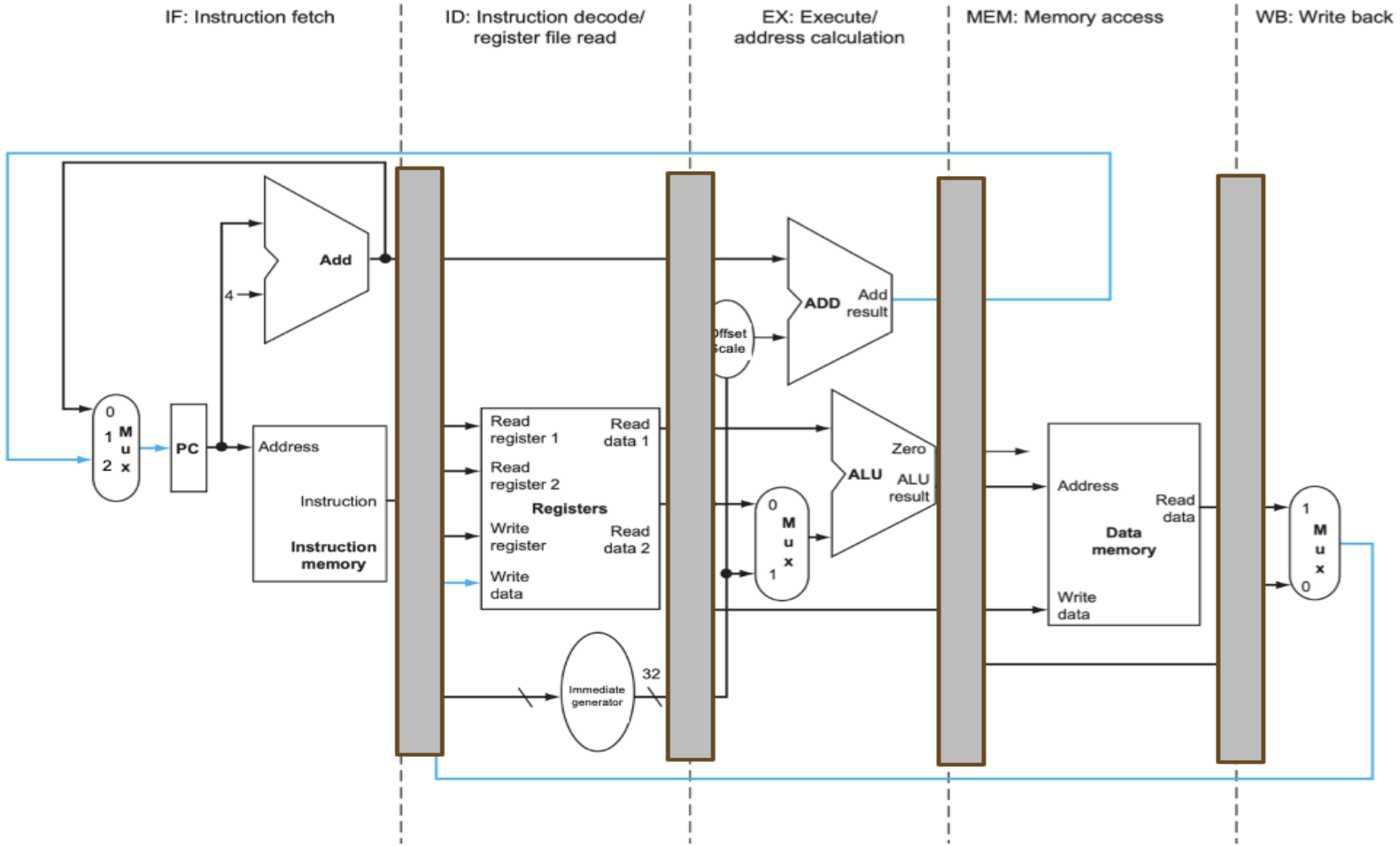


Cycle 0



add x5, x3, x4

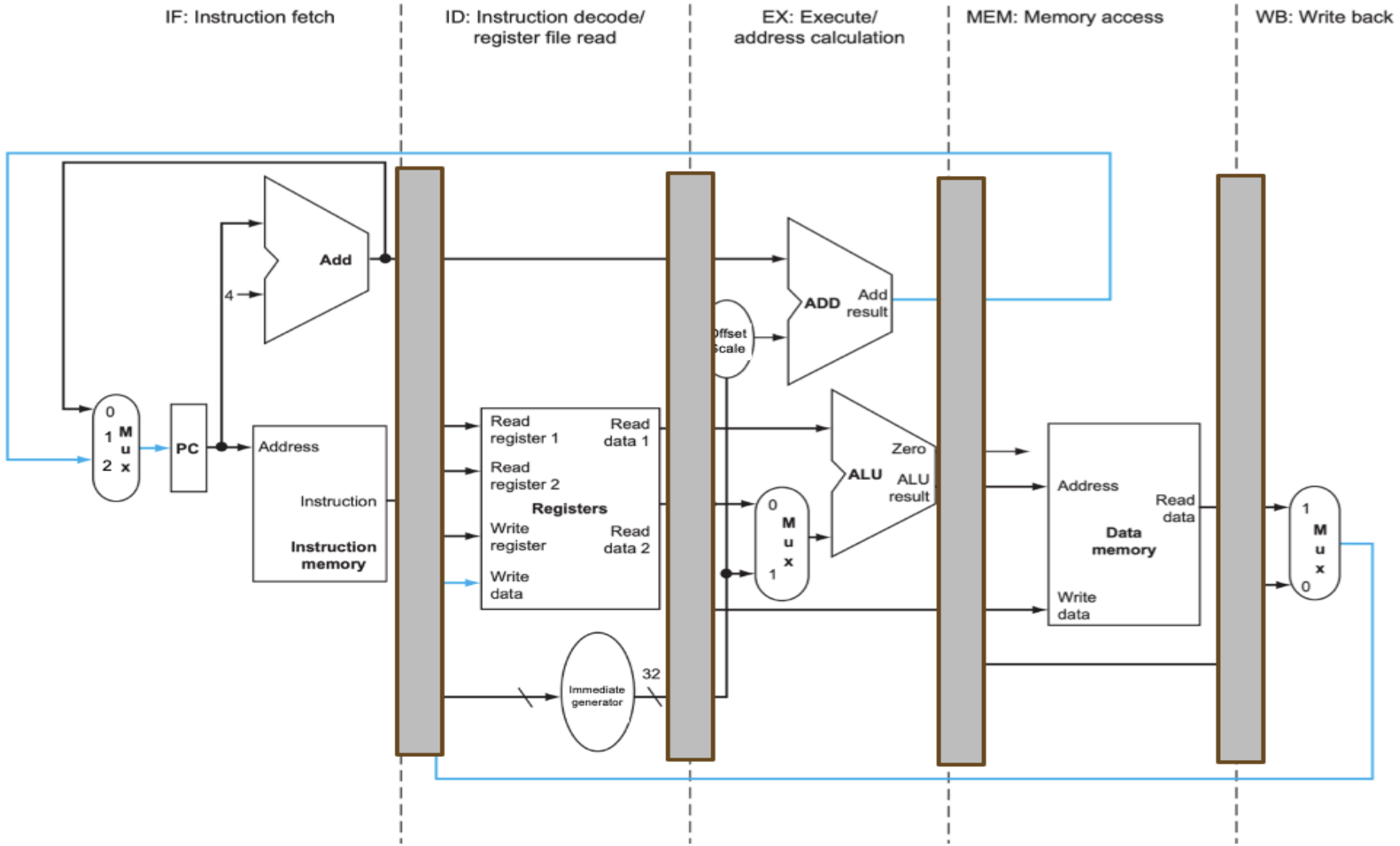
Cycle 1



sub x7, x5 x2

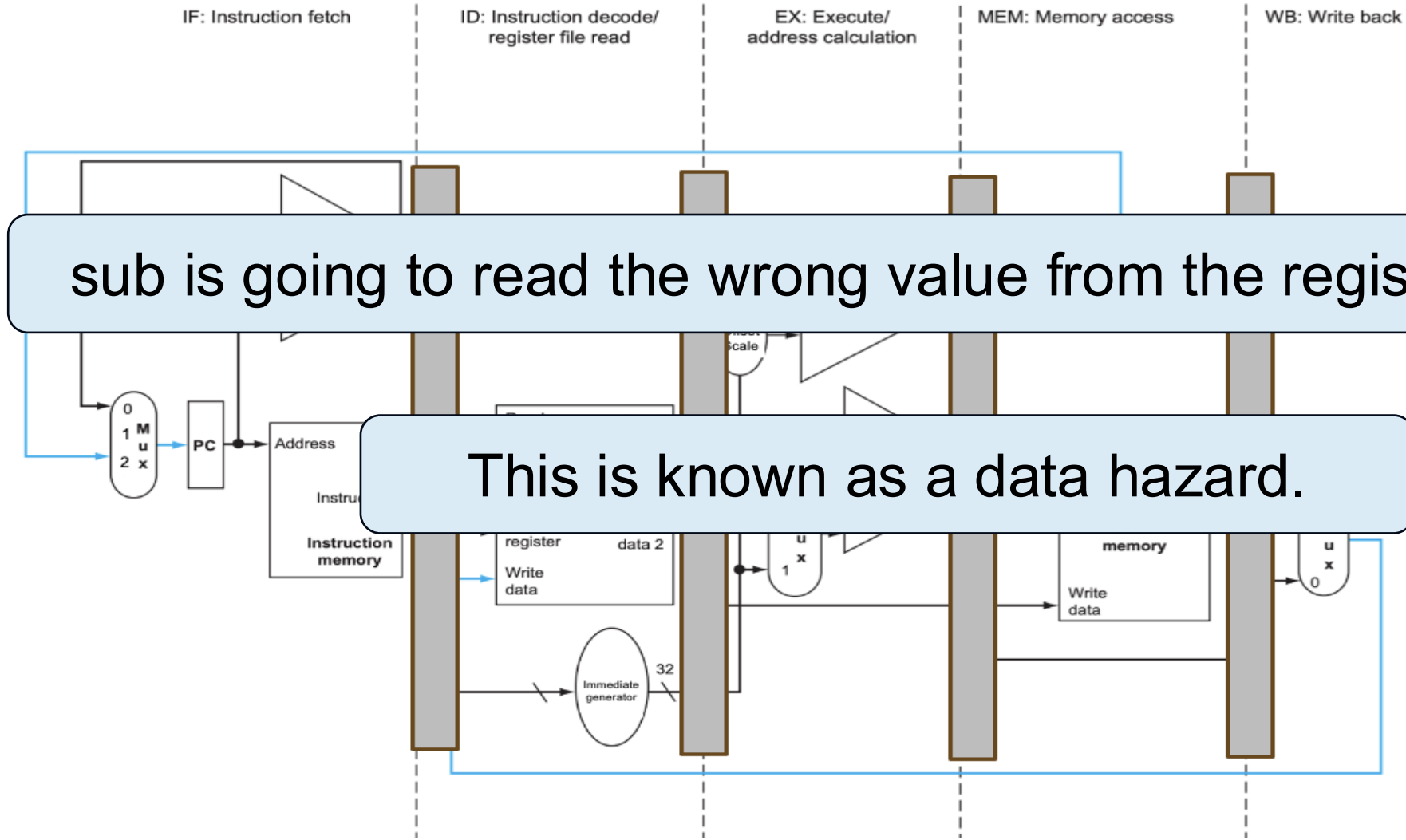
add x5, x3, x4

Cycle 2



sub x7, x5 x2 add x5, x3, x4

Cycle 2



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

How can we fix this?



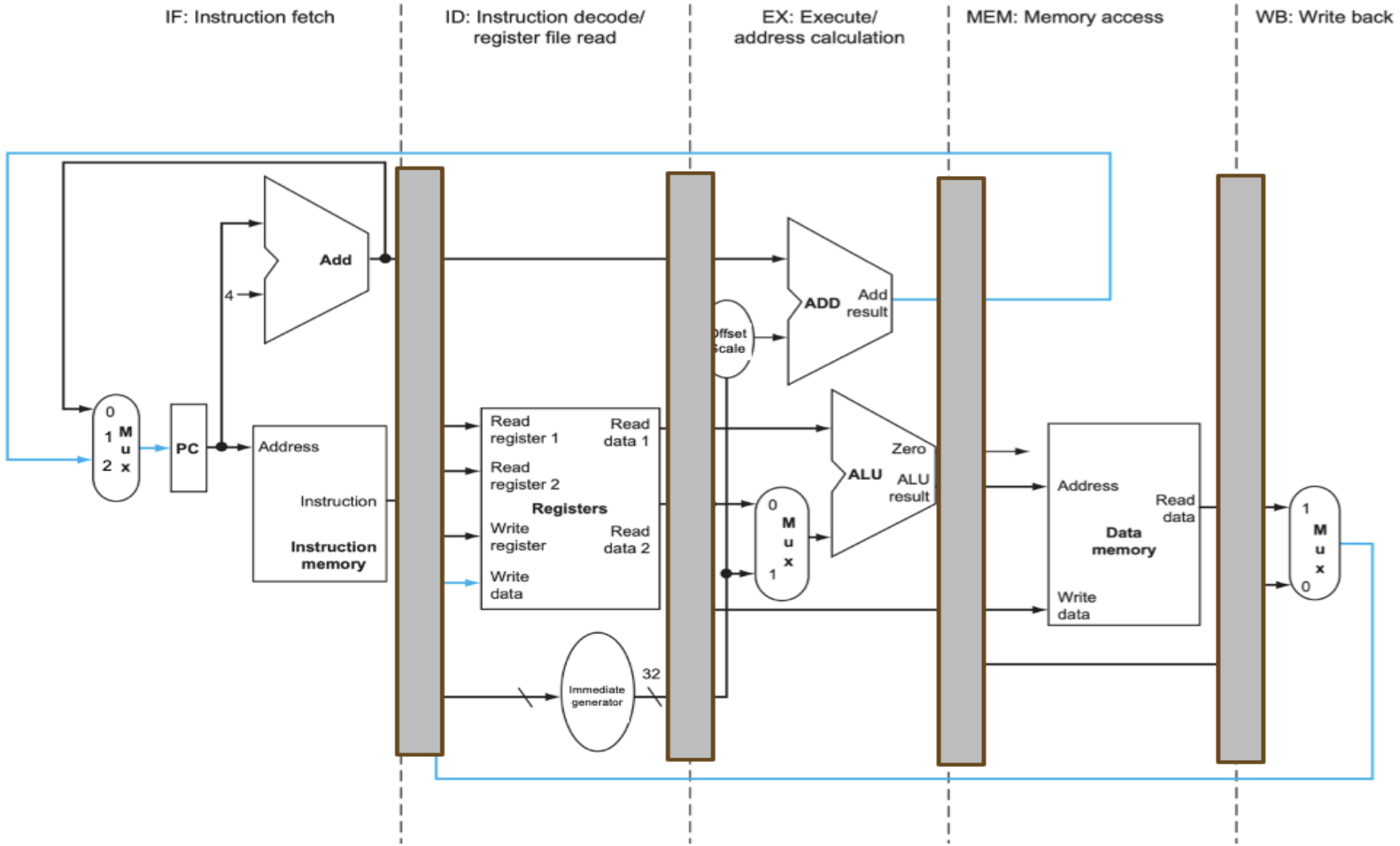
Introducing pipeline stalls!
(aka bubbles or nops)

Pipeline Stalls



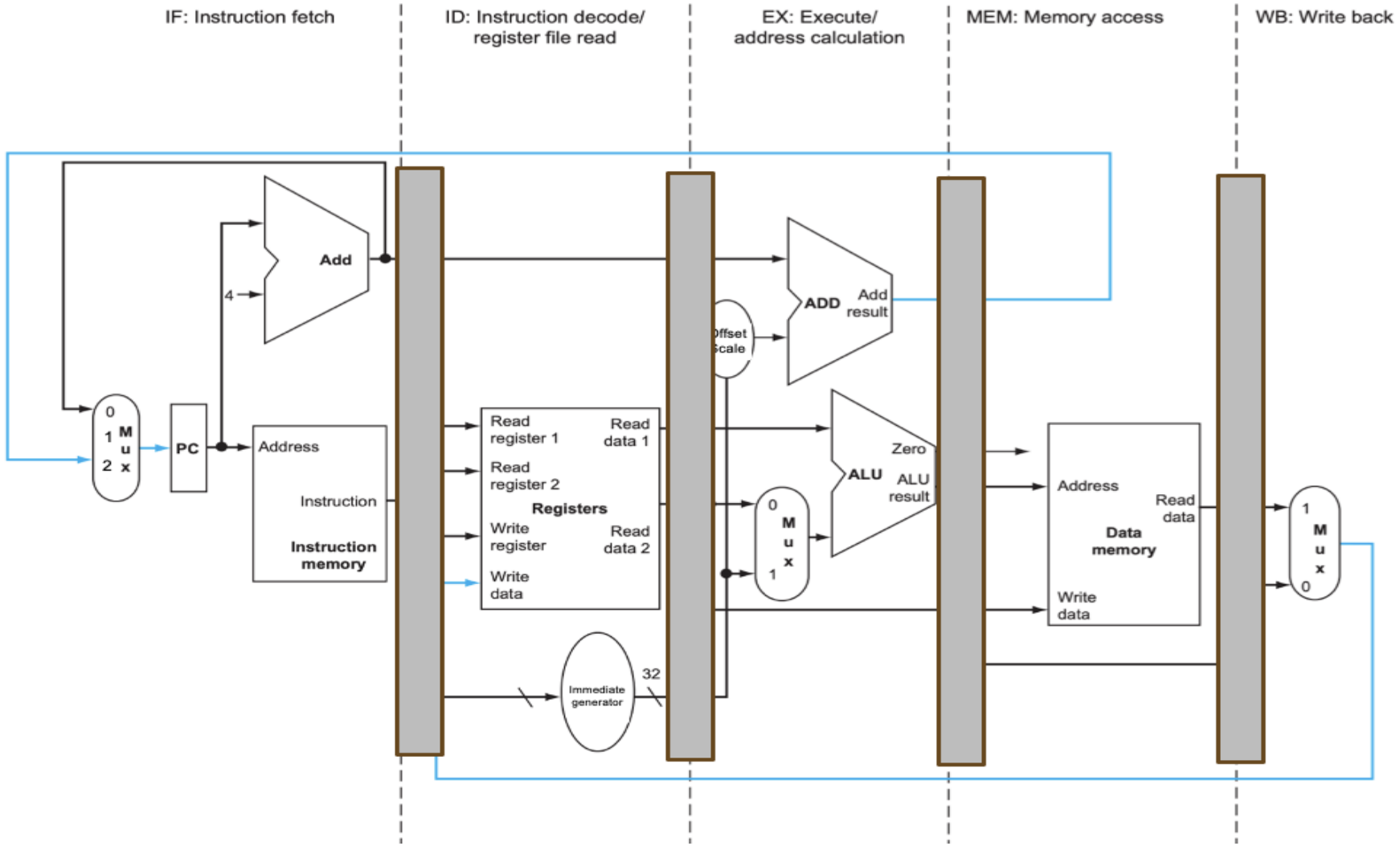
- Also known as
  - bubbles
  - nops (no operation)
- An instruction that does nothing
- Provides time for the correct value to be written to the register file

Cycle 2



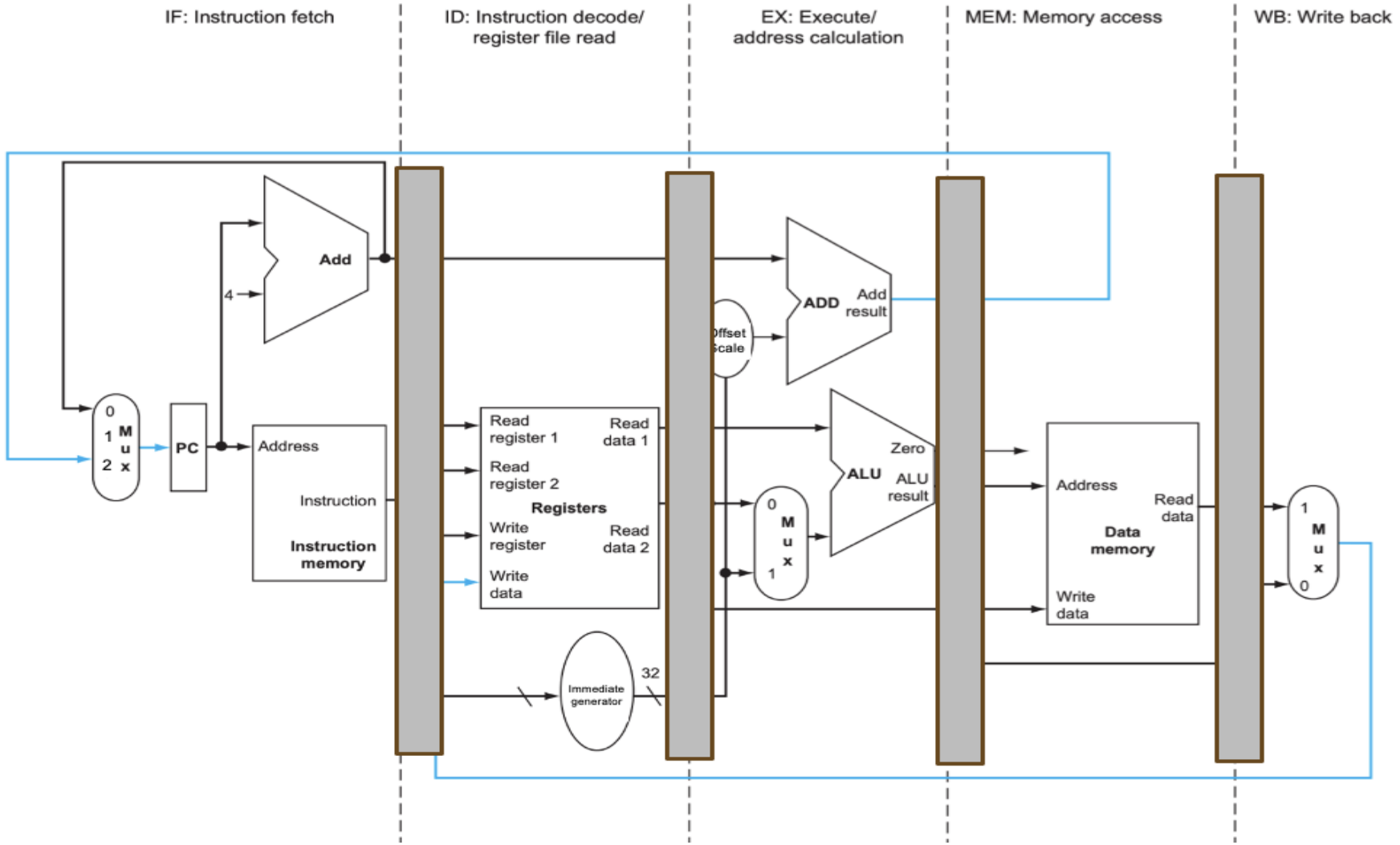
sub x7, x5, x2add x5, x3, x4

Cycle 3



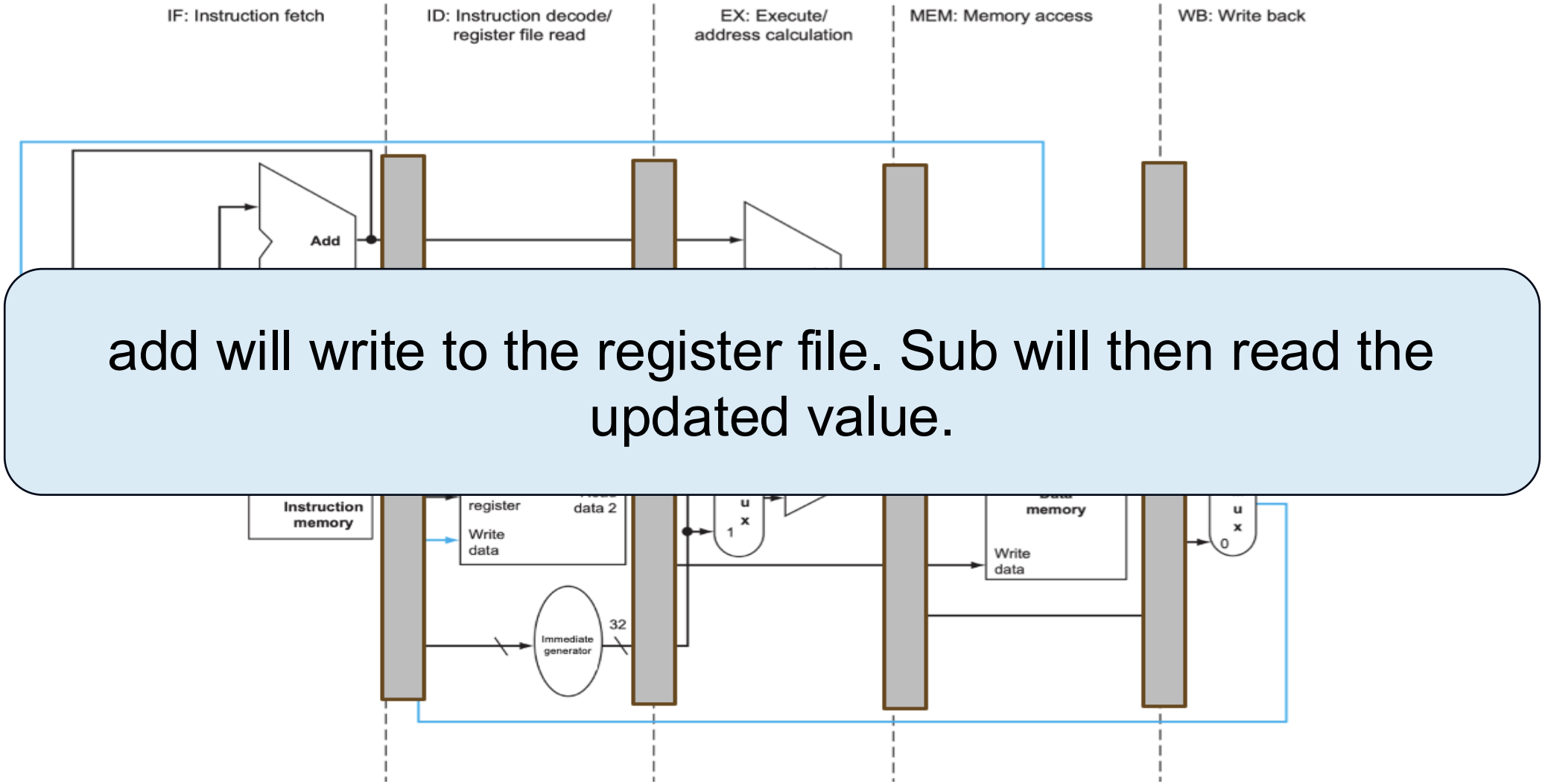
sub x7, x5, x2 ***STALL*** add x5, x3, x4

Cycle 4



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

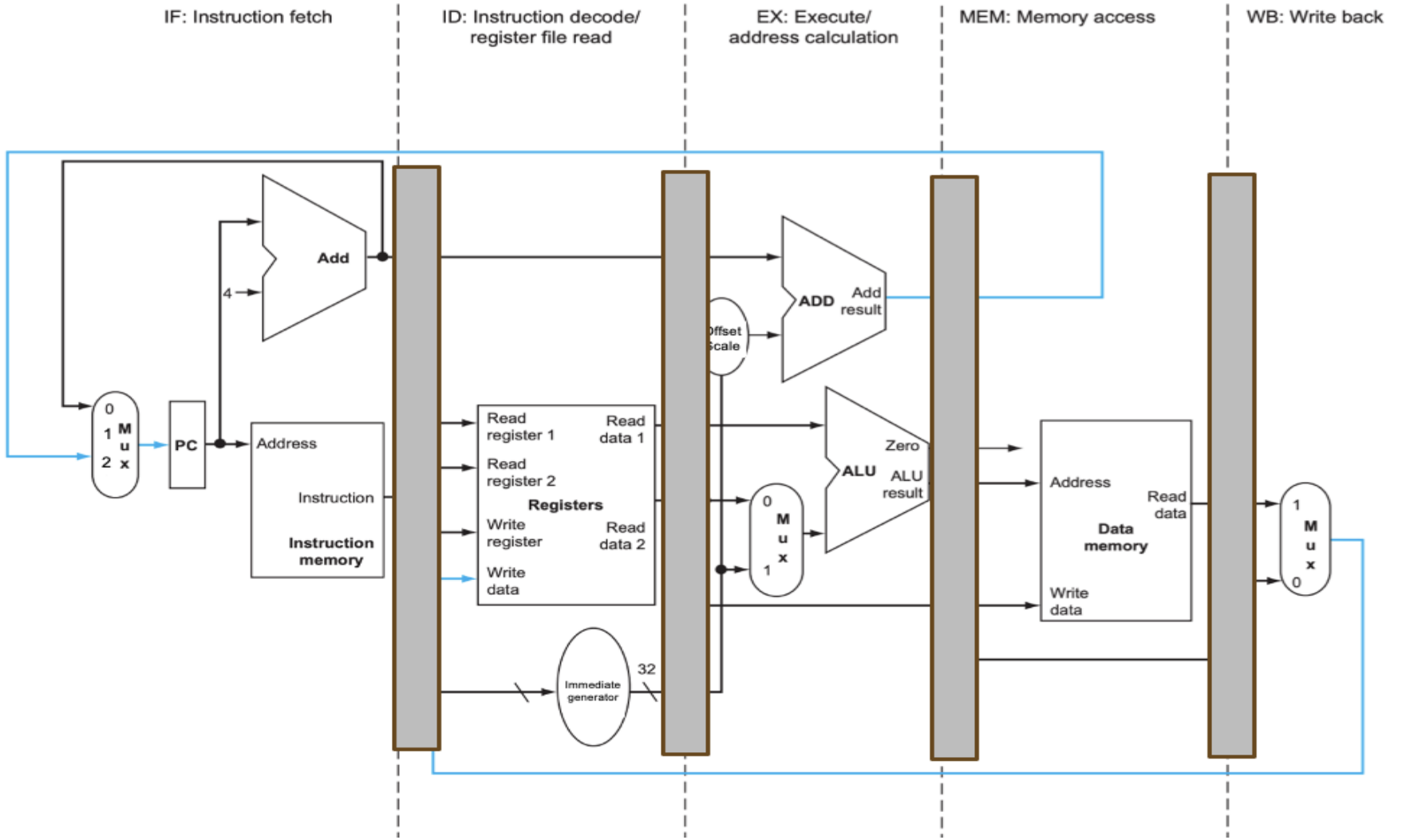
Cycle 4



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

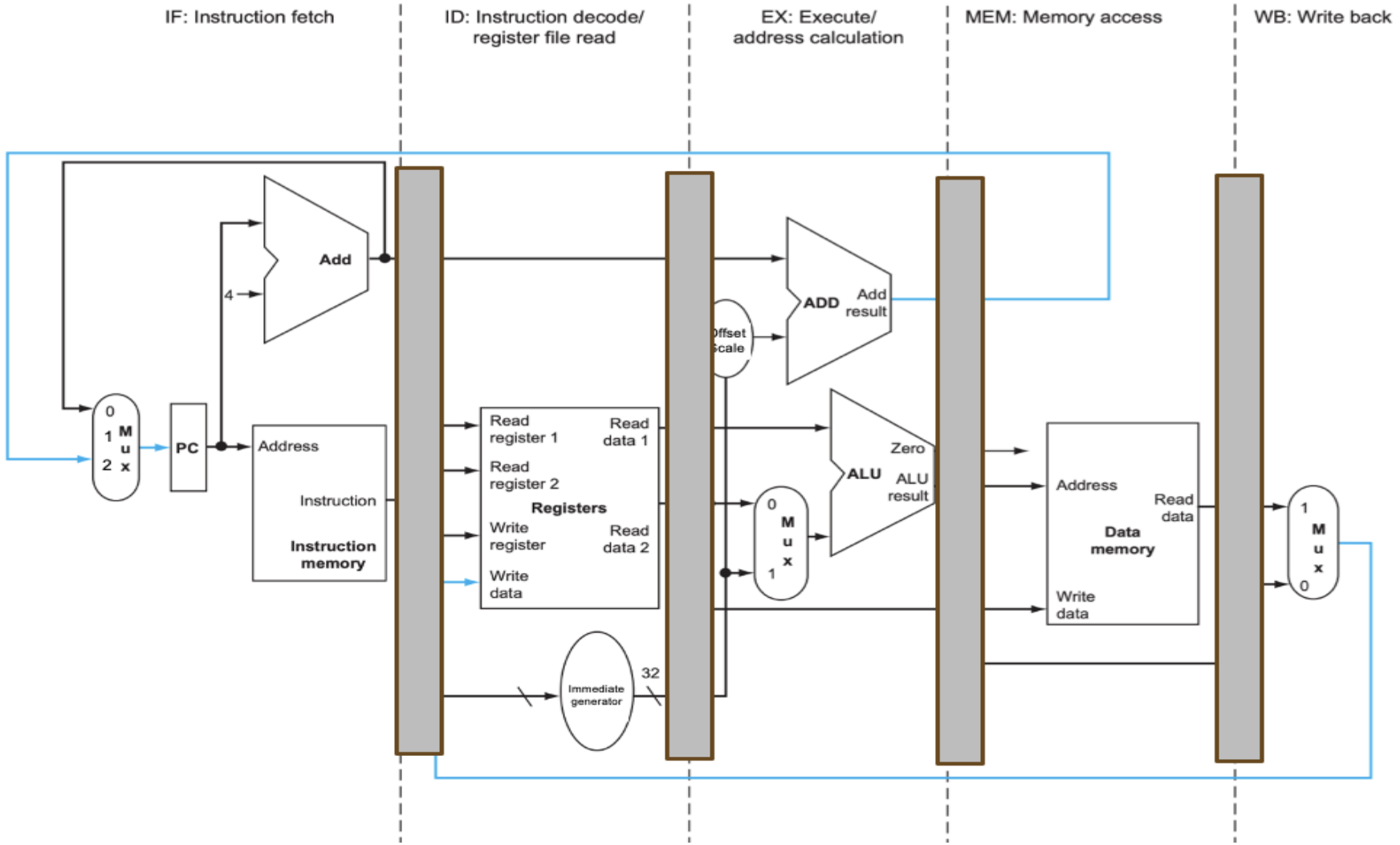
sub x7, x5, x2 ***STALL*** ***STALL*** add x5, x3, x4

Cycle 5



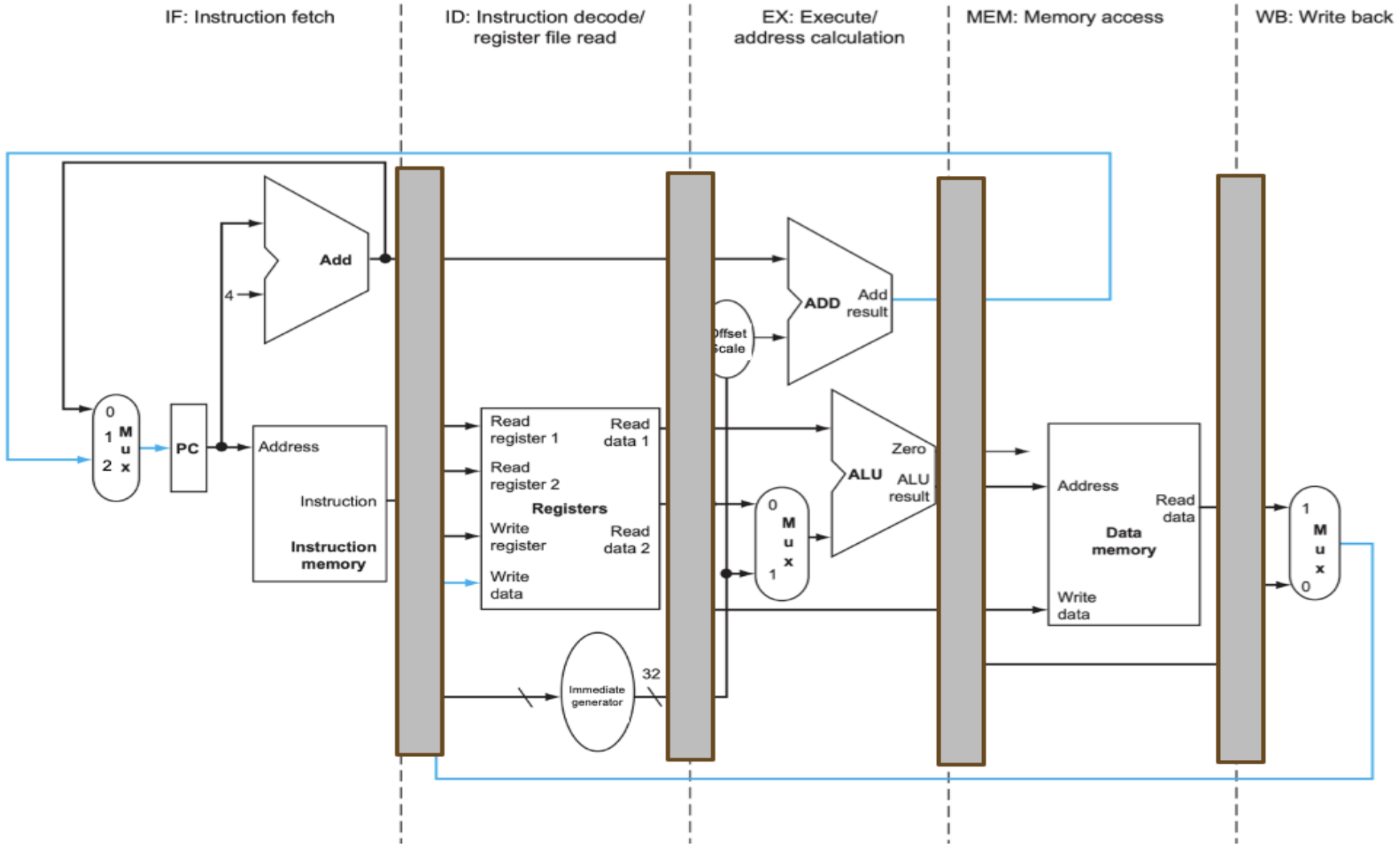
sub x7, x5, x2 ***STALL*** ***STALL***

Cycle 6



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

Cycle 7



sub x7, x5, x2

Pipeline Stalls



- In this example, we inserted two stalls between the instructions to ensure that sub would read the correct value from x5

add x5, x3, x4

STALL

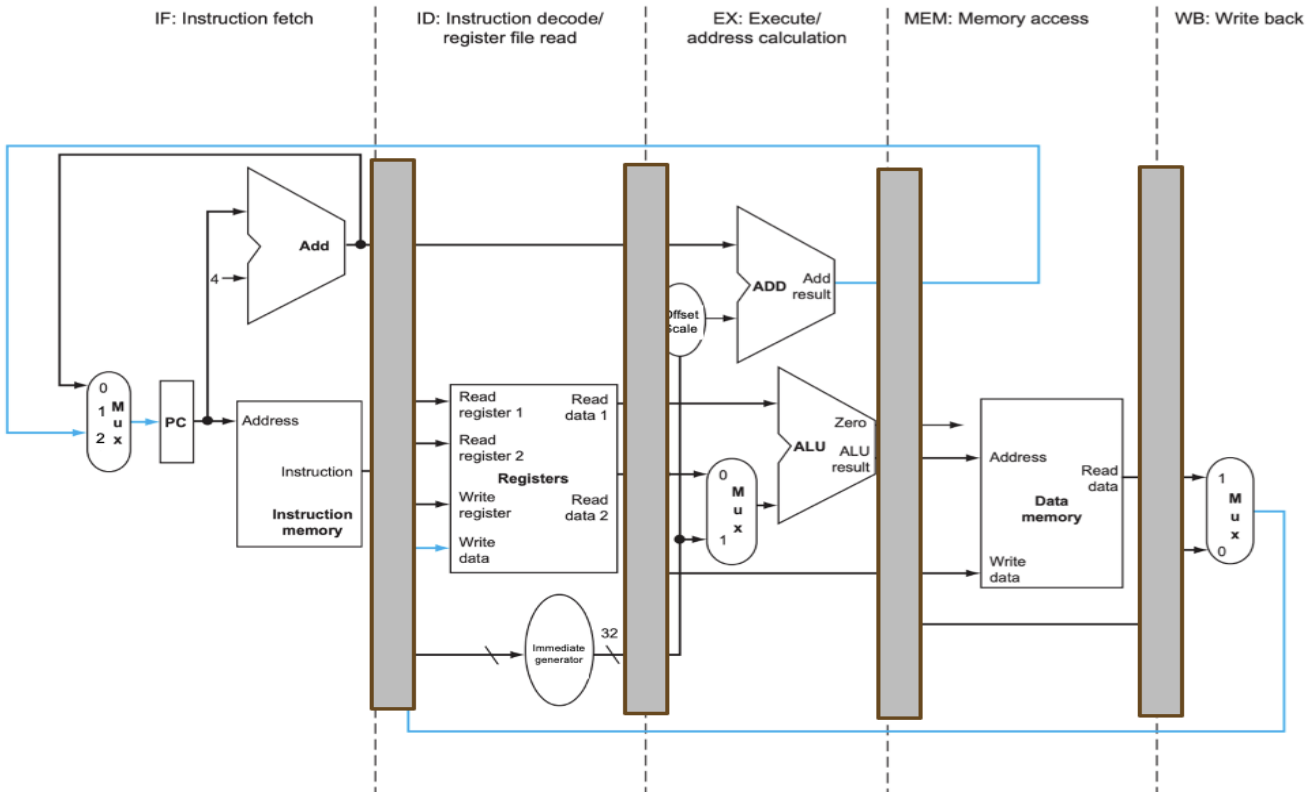
STALL

sub x7, x5, x2

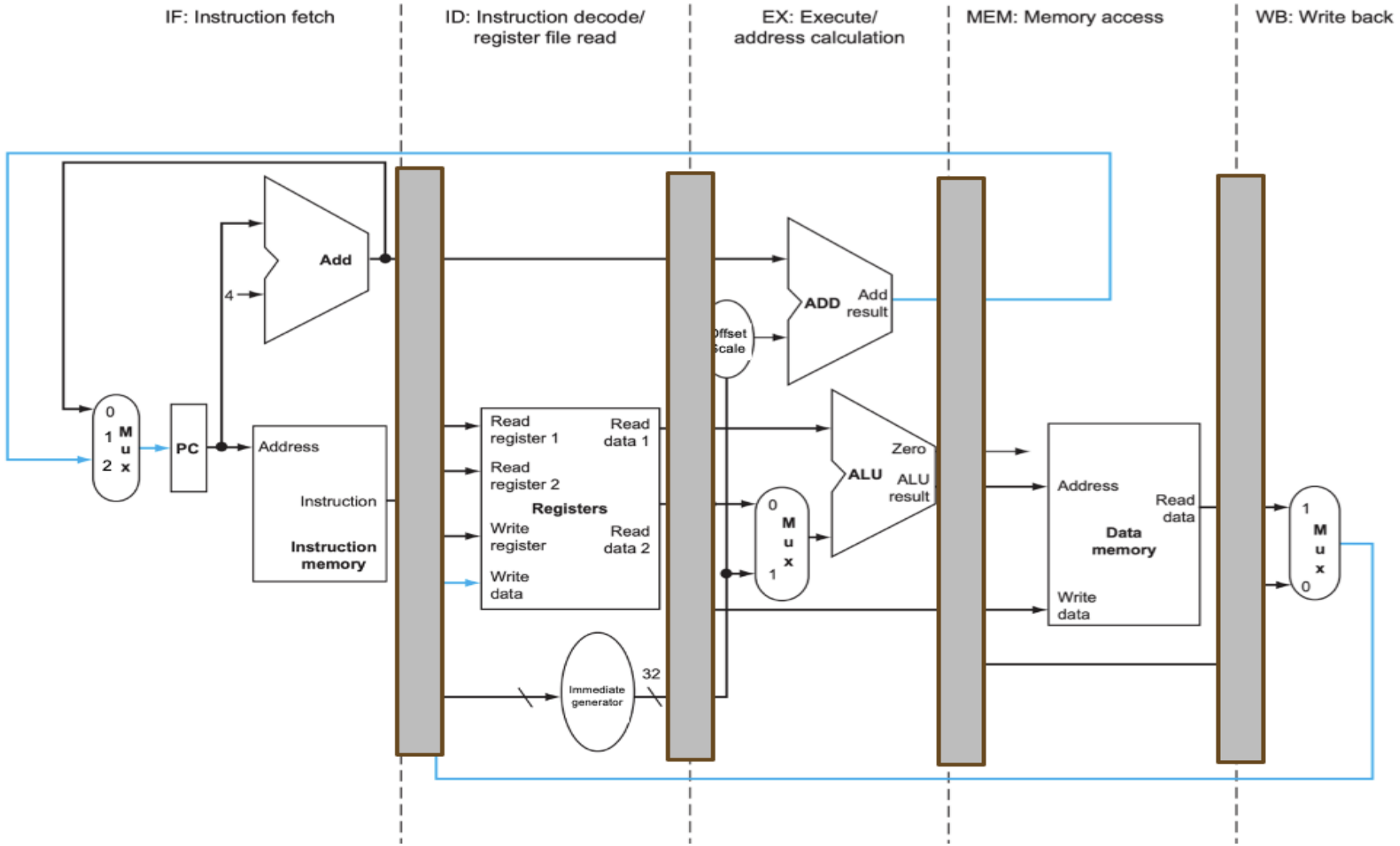
Data Hazards

What if there was an independent instruction in between?

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

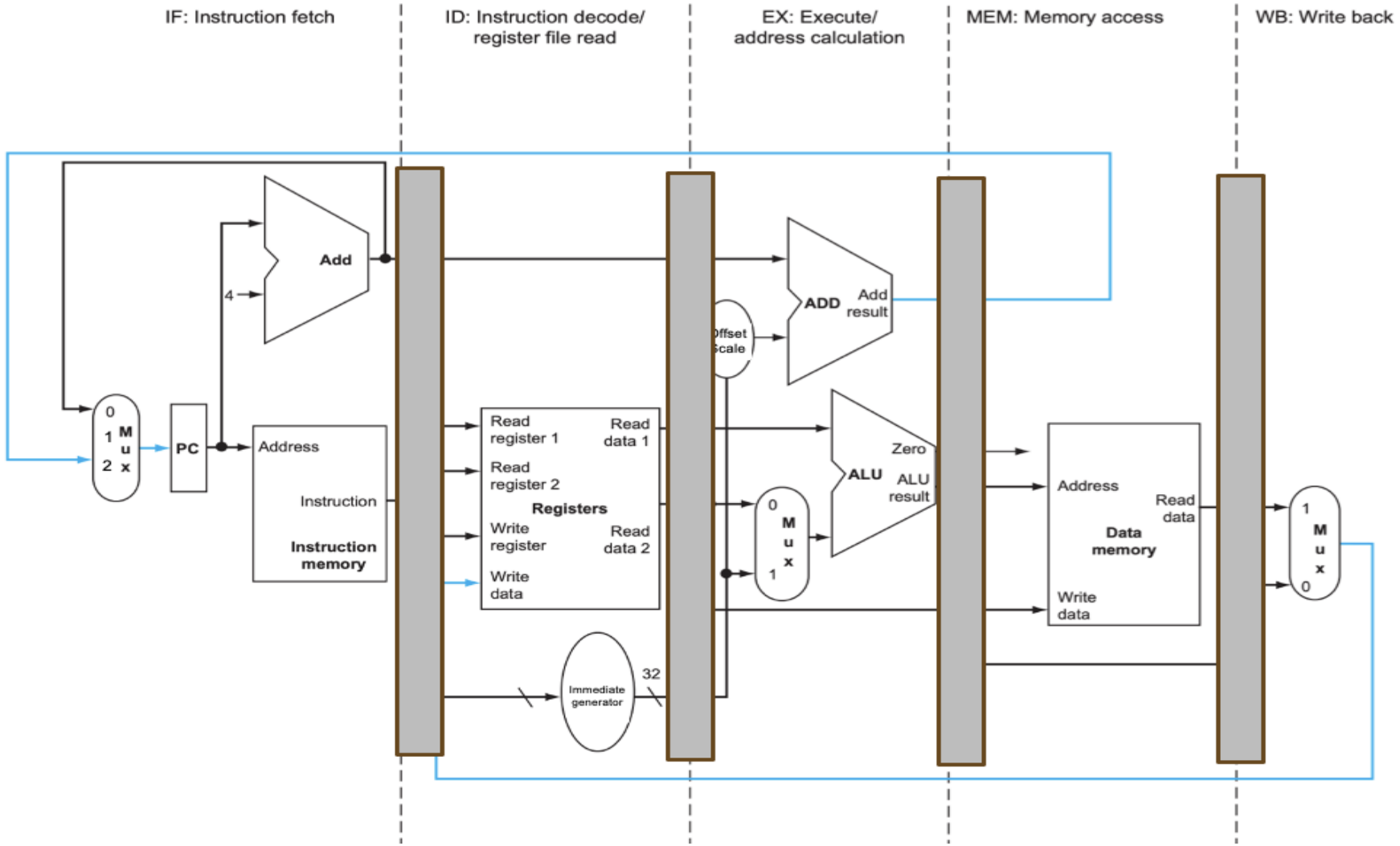


Cycle 0



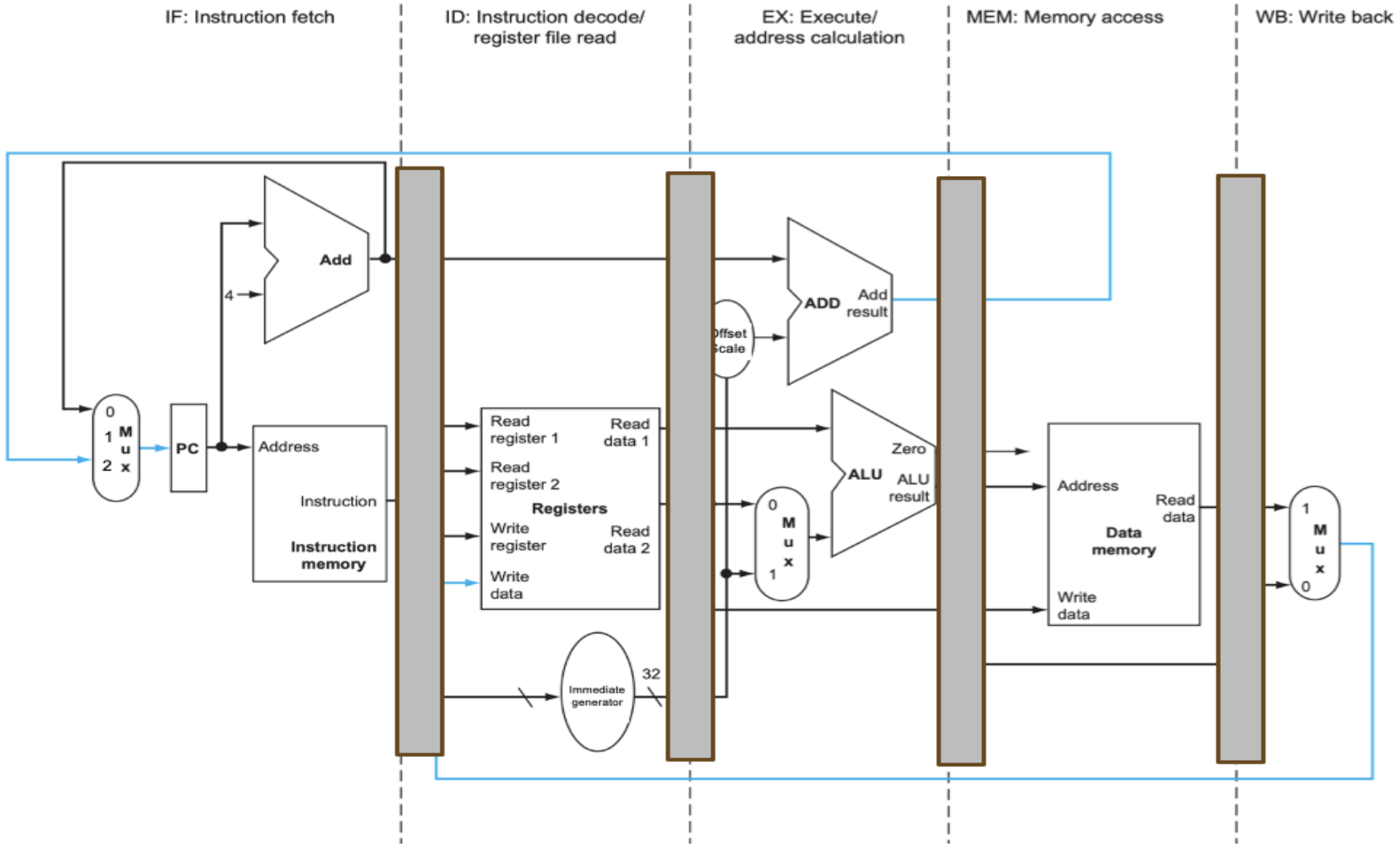
add x5, x3, x4

Cycle 1



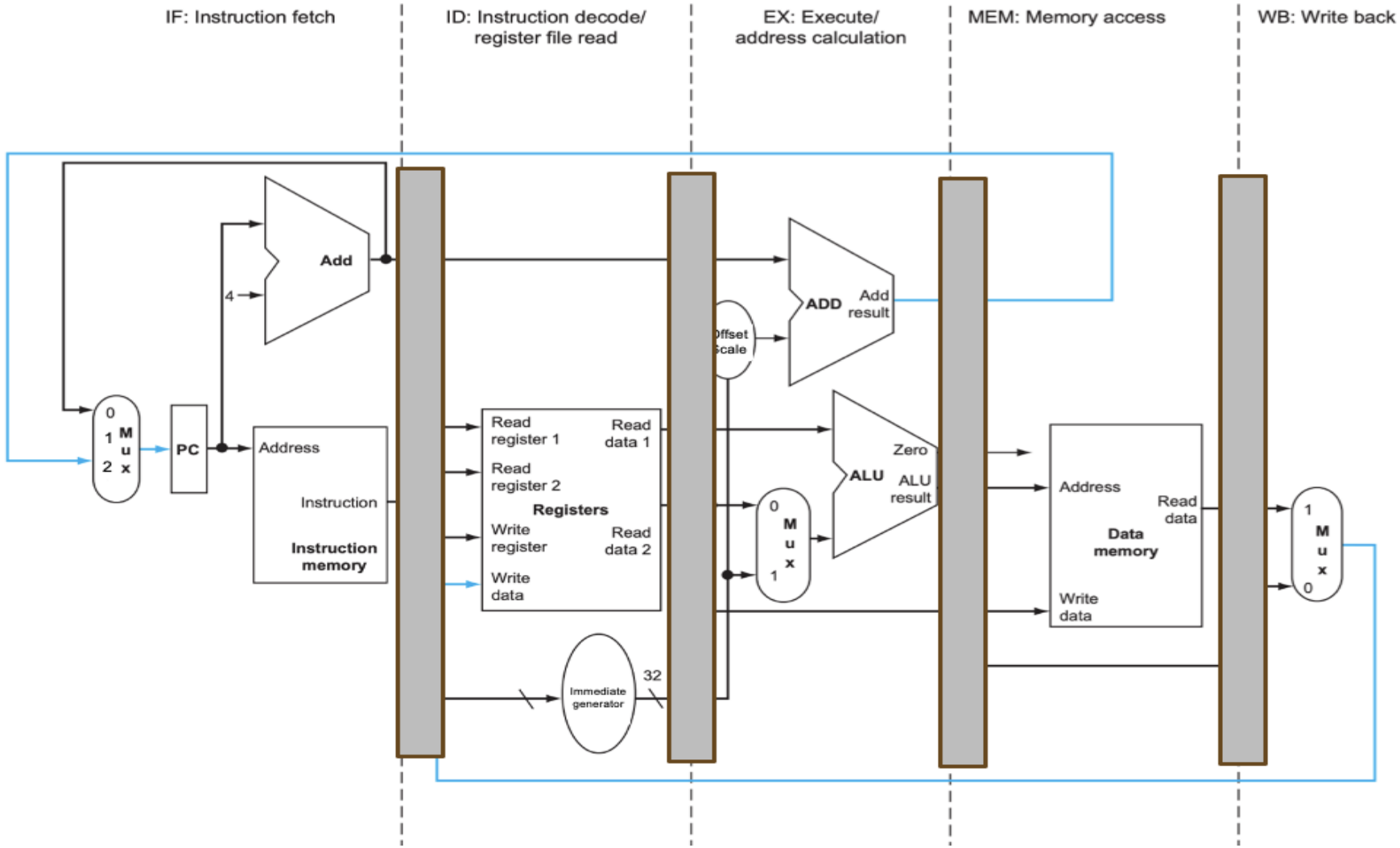
or x8, x9, x10 add x5, x3, x4

Cycle 2



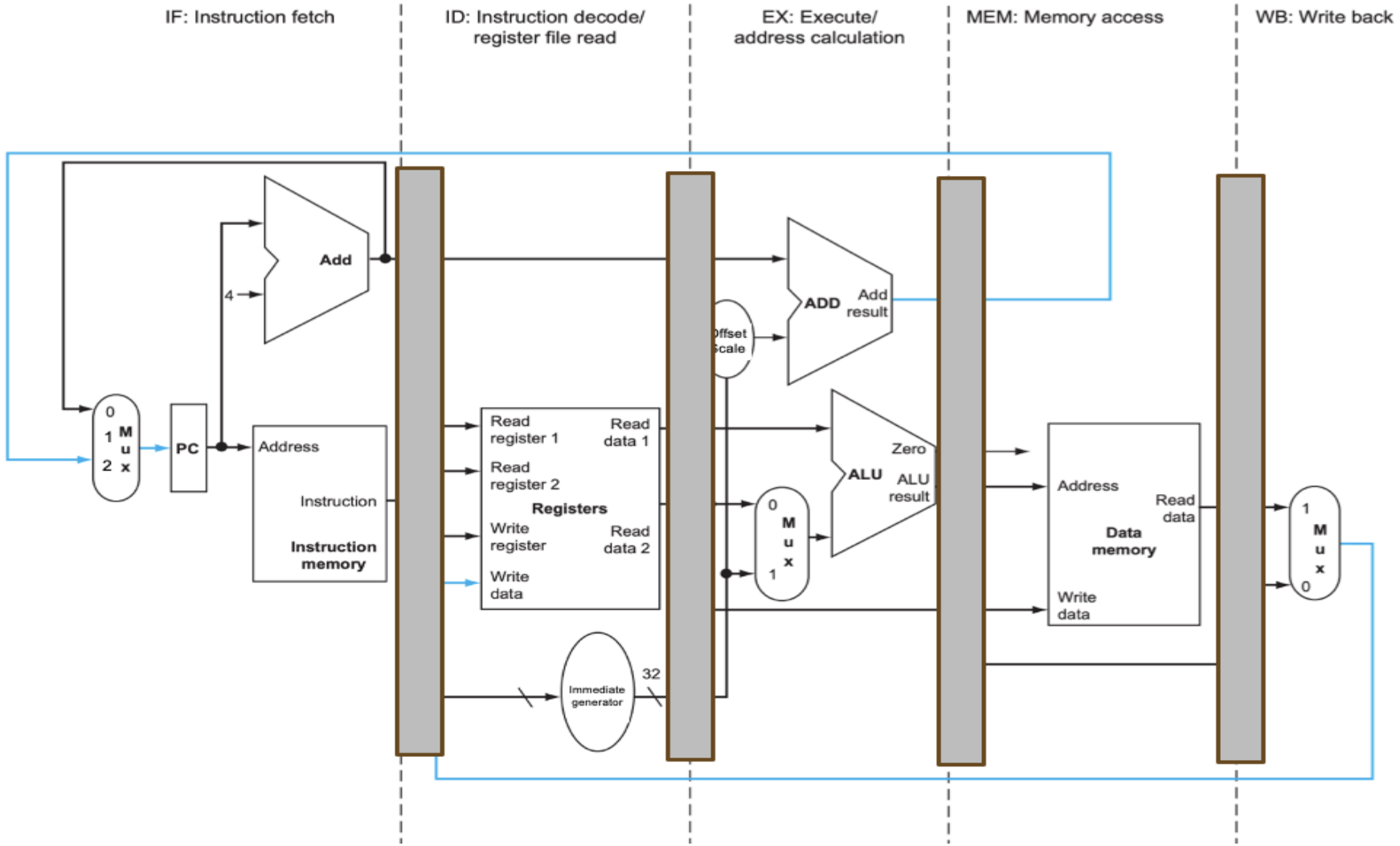
sub x7, x5, x2 or x8, x9, x10 add x5, x3, x4

Cycle 3



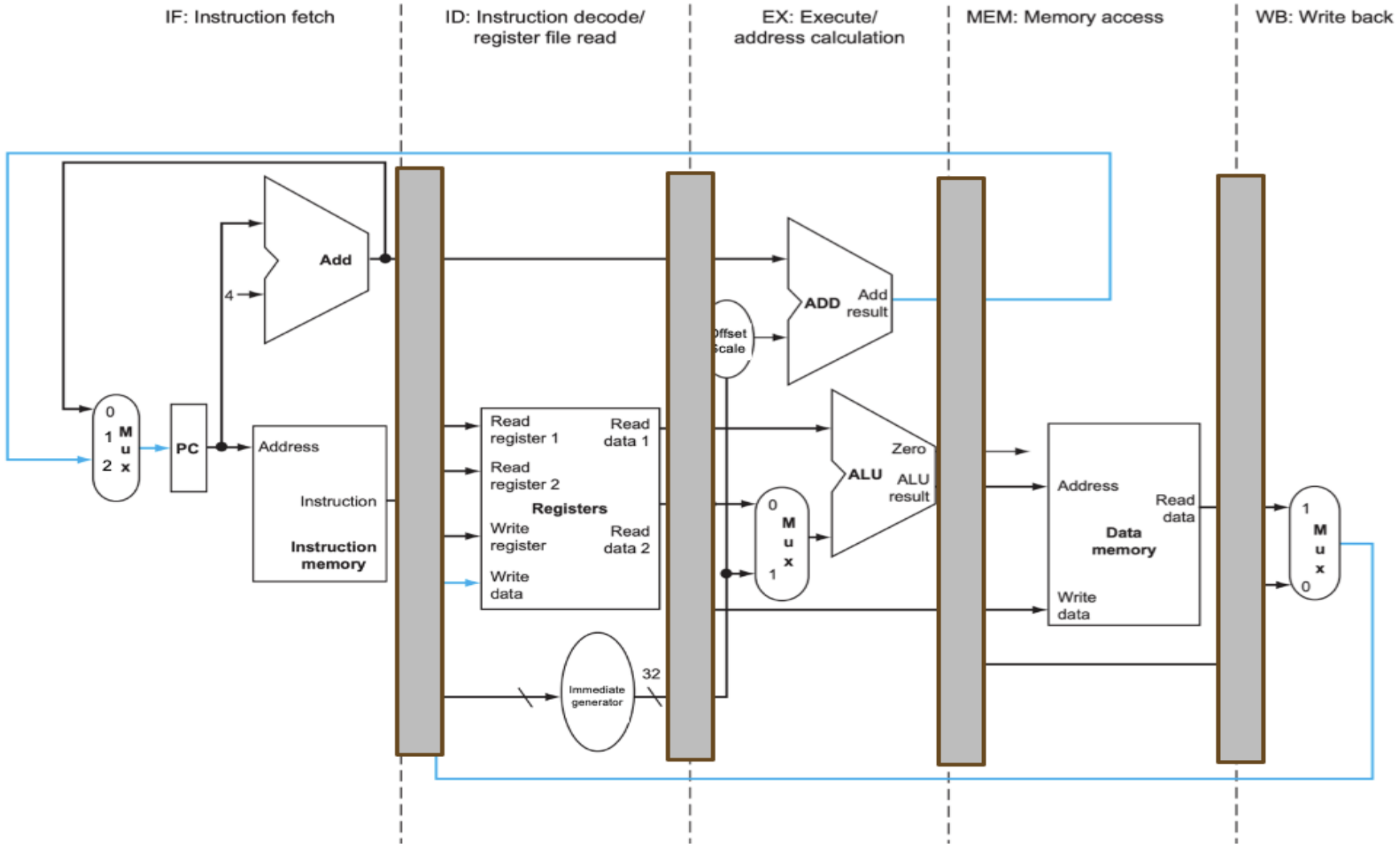
sub x7, x5, x2 or x8, x9, x10 add x5, x3, x4

Cycle 4



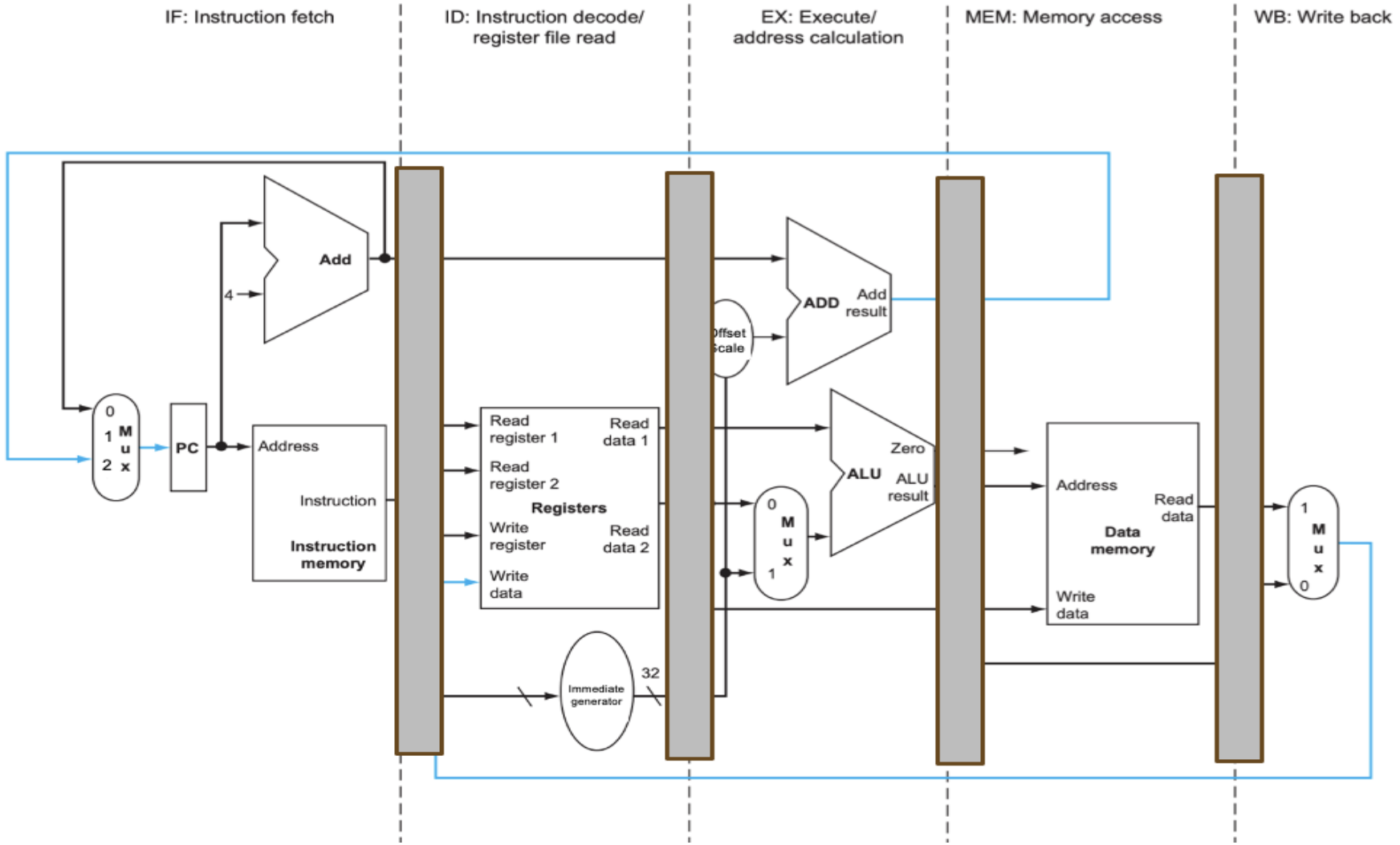
sub x7, x5, x2 ***STALL*** or x8, x9, x10 add x5, x3, x4

Cycle 5



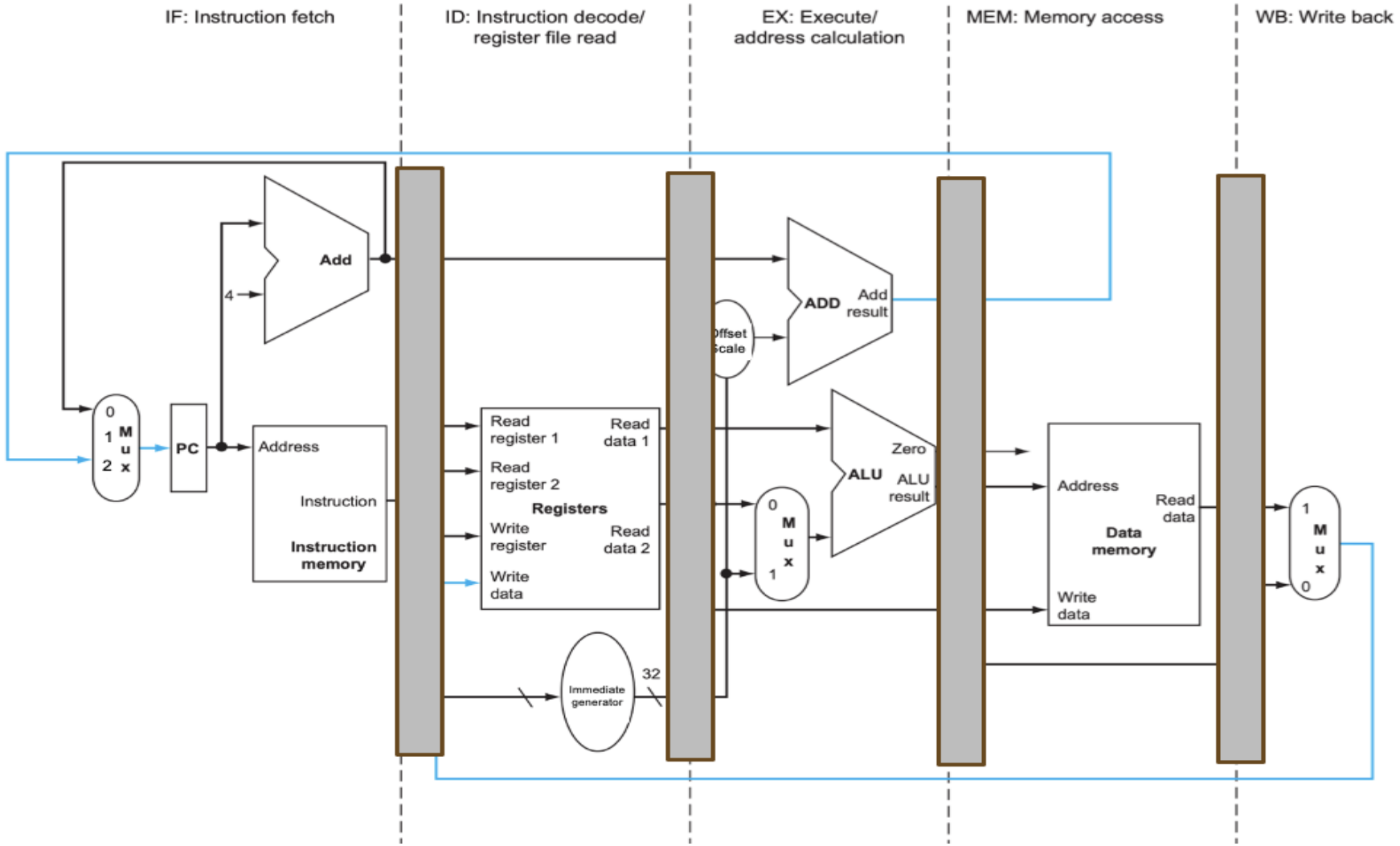
sub x7, x5, x2 ***STALL*** or x8, x9, x10

Cycle 6



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

Cycle 7



sub x7, x5, x2

Would we need to add any stalls for this program?



```
add x5, x3, x4
or x8, x9, x10
xor x14, x15, x16
sub x7, x5, x2
```

Pipeline Diagram

Provides a way for us to visualize instructions moving through the pipeline.

Instruction	Cycle #													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
add x5, x3, x4															
or x8, x9, x10															
xor x14, x15, x16															
sub x7, x5, x2															

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

Pipeline Diagram

Provides a way for us to visualize instructions moving through the pipeline.

Instruction	Cycle #													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
add x5, x3, x4	F	D	X	M	W										
or x8, x9, x10		F	D	X	M	W									
xor x14, x15, x16			F	D	X	M	W								
sub x7, x5, x2				F	D	X	M	W							

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

Would we need to add any stalls for this program?



```
add x20, x21, x22
sub x28, x20, x15
and x27, x5,  x18
or  x9,  x28, x3
```

Pipeline Diagram

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

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															
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

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)

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

	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															