

COMP311: *COMPUTER ORGANIZATION!*

Lecture 21: Pipeline Hazards

tinyurl.com/comp311-sp26

Logistics Update!

- Last lab (5) will be posted today!
 - *Optional lab 3.1 to update our API*
- Quiz Topics (Last Quiz on Wednesday!)
 - *Advanced RISC-V Programming*
 - *Tracing and Writing Functions with RISC-V*
 - *Reading and Writing to Memory (Look at WA6)*
 - *Pipeline Hazards (Whatever we get through today!)*

Load

Store

lw dst, add✓
4(x6)

sw data, addr✓

$$1. \quad x_6 = 20$$

$$mem[0x1058] = 20$$

RISC-V WARM UP

$$2. \quad t_5 \leftarrow i \times 4$$

$$t_0 = 8$$

$$3. \quad \underline{addi} \ t_1, t_0, 4$$

SOME REMINDERS



WE HAVE COVERED
A LOT!

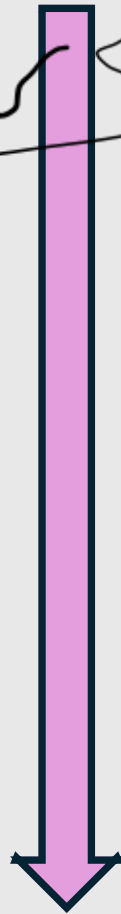
Compilation vs. Disassembly

Compilation

Disassembly

Compilers

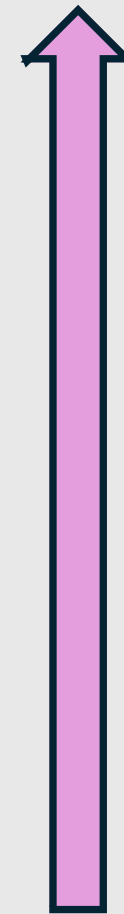
Comp
\$20



```
// High-level (C)
int add3(int a, int b)
{
    return a + b + 3;
}
```

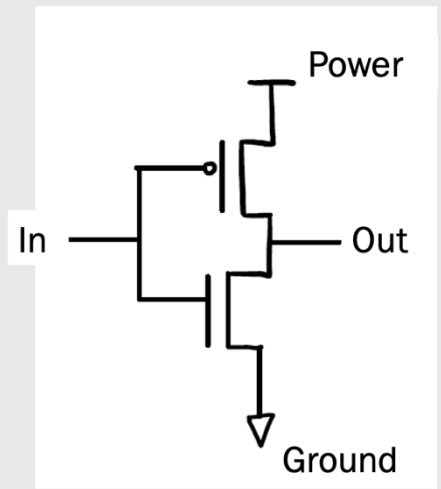
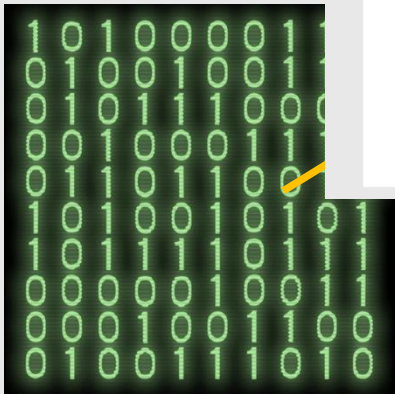
```
add    a0, a0, a1    # a0 = a0 + a1
addi   a0, a0, 3     # a0 = a0 + 3
ret                               # return
```

```
0000000 01011 01010000 01010 0110011
0000000000011 01010 000 01010 0010011
0000000000000 00001000 00000 1100111
```

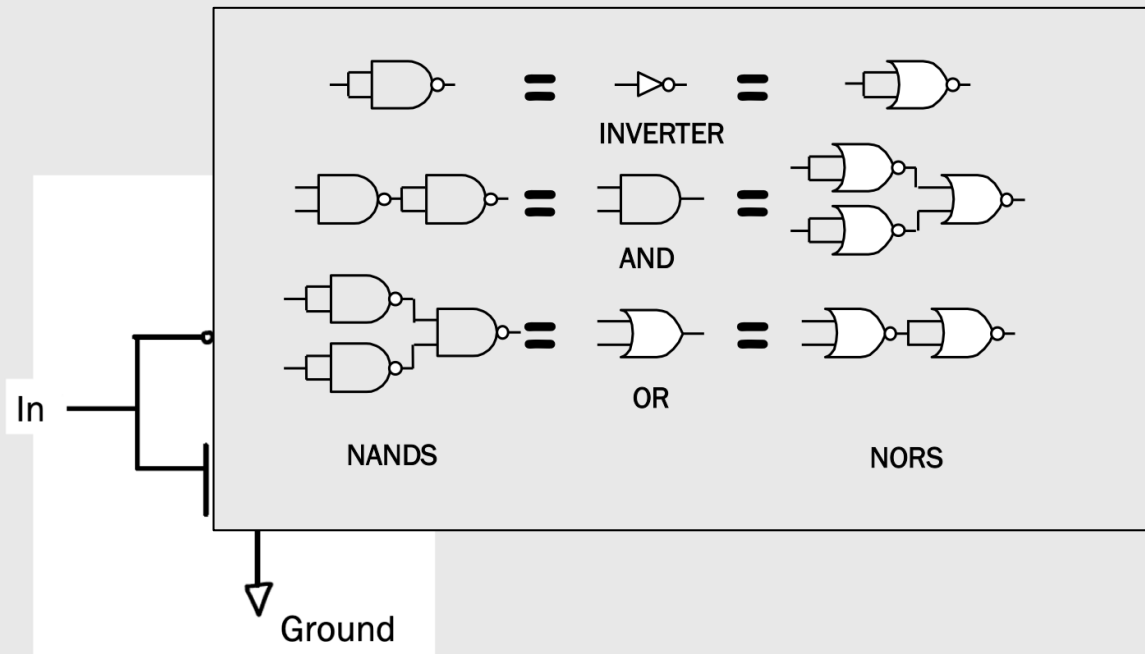


1	0	1	0	0	0	0	1	1	1
0	1	0	0	1	0	0	1	1	0
0	1	0	1	1	1	0	0	1	1
0	0	1	0	0	0	1	1	1	1
0	1	1	0	1	1	0	1	0	1
1	0	1	0	0	1	0	1	0	1
1	0	1	1	1	1	0	1	1	1
0	0	0	0	0	1	0	0	1	1
0	0	0	0	1	0	0	1	1	0
0	1	0	0	1	1	1	0	1	0





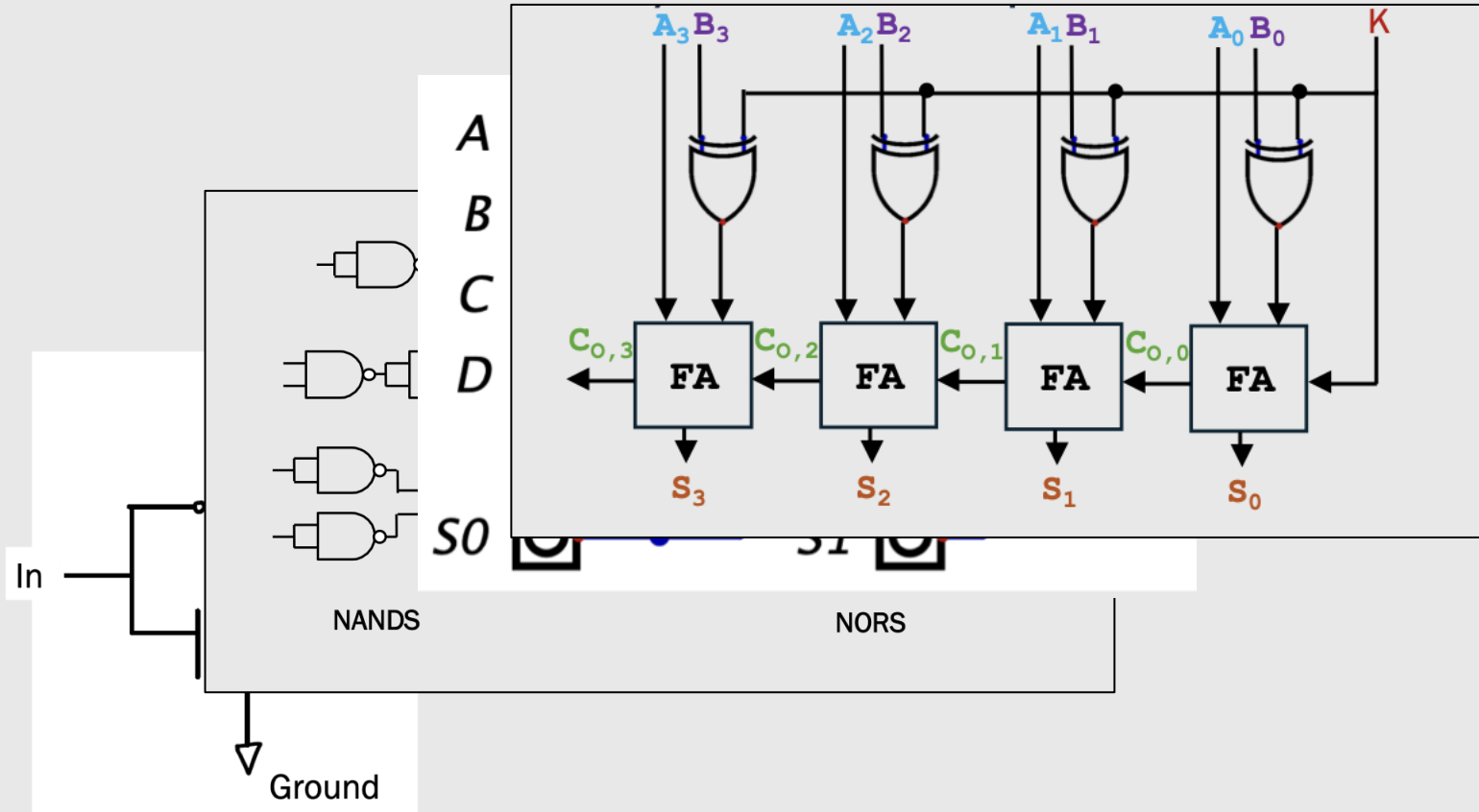
10100001
00101110
00100010
01100101
10110010
10111101
00000010
00001001
01001101

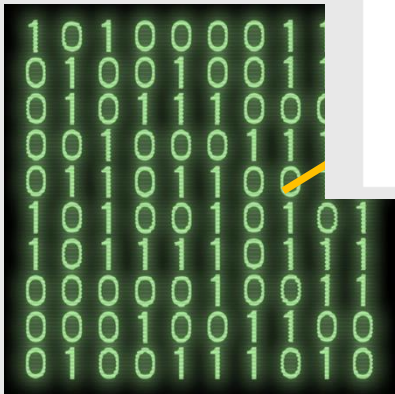
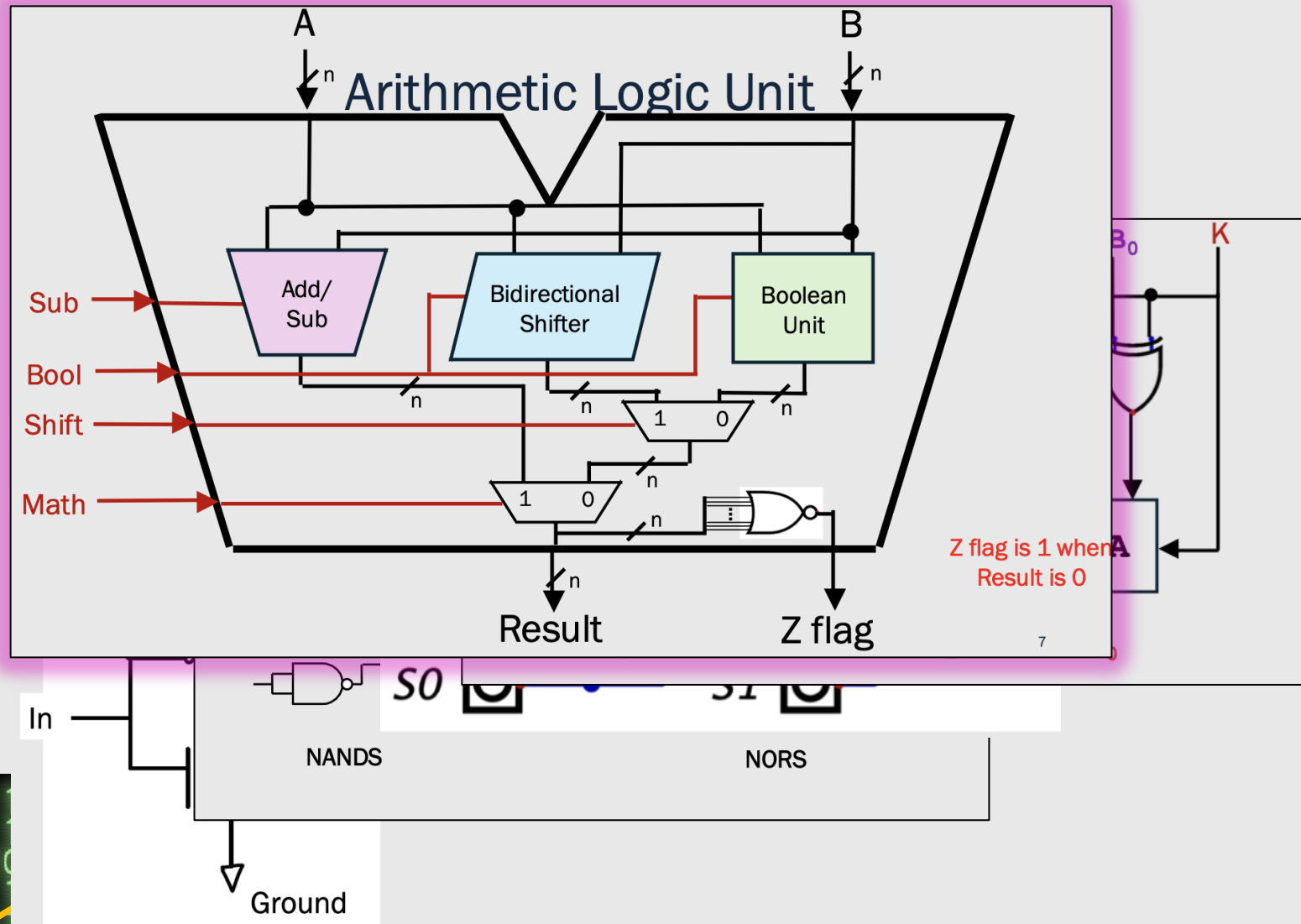


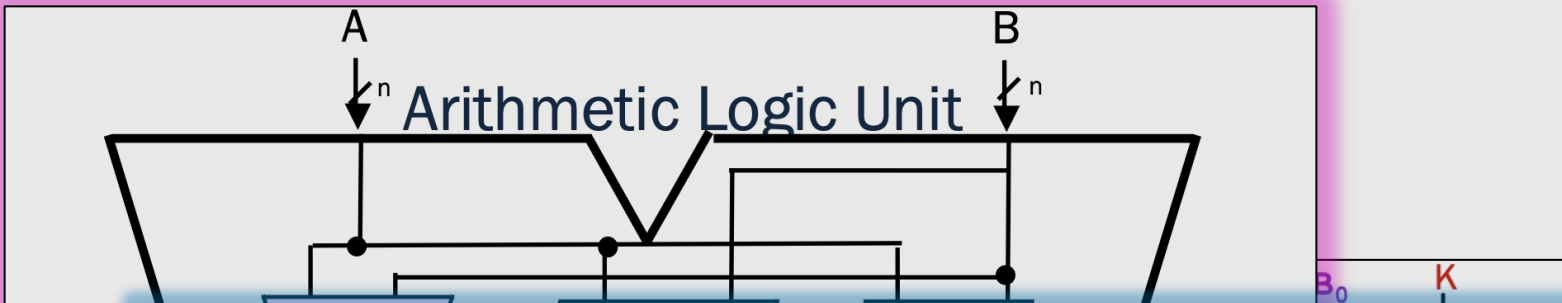

```

1 0 1 0 0 0 0 1 1
0 0 1 0 1 1 1 0 0
0 0 1 0 1 1 1 0 0
0 0 1 0 1 1 1 0 0
0 0 1 0 1 1 1 0 0
1 0 1 1 1 0 0 1 0
1 0 1 1 1 0 0 1 0
1 0 1 1 1 0 0 1 0
0 0 0 0 0 0 1 1 0
0 0 0 0 1 0 0 1 0
0 1 0 0 1 1 1 0 0

```

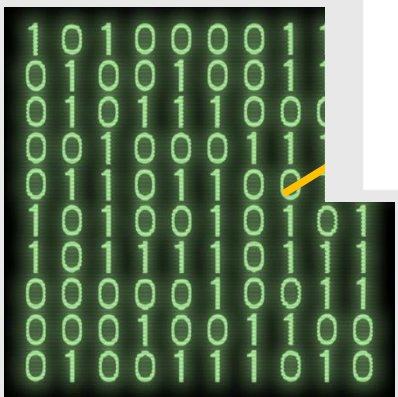
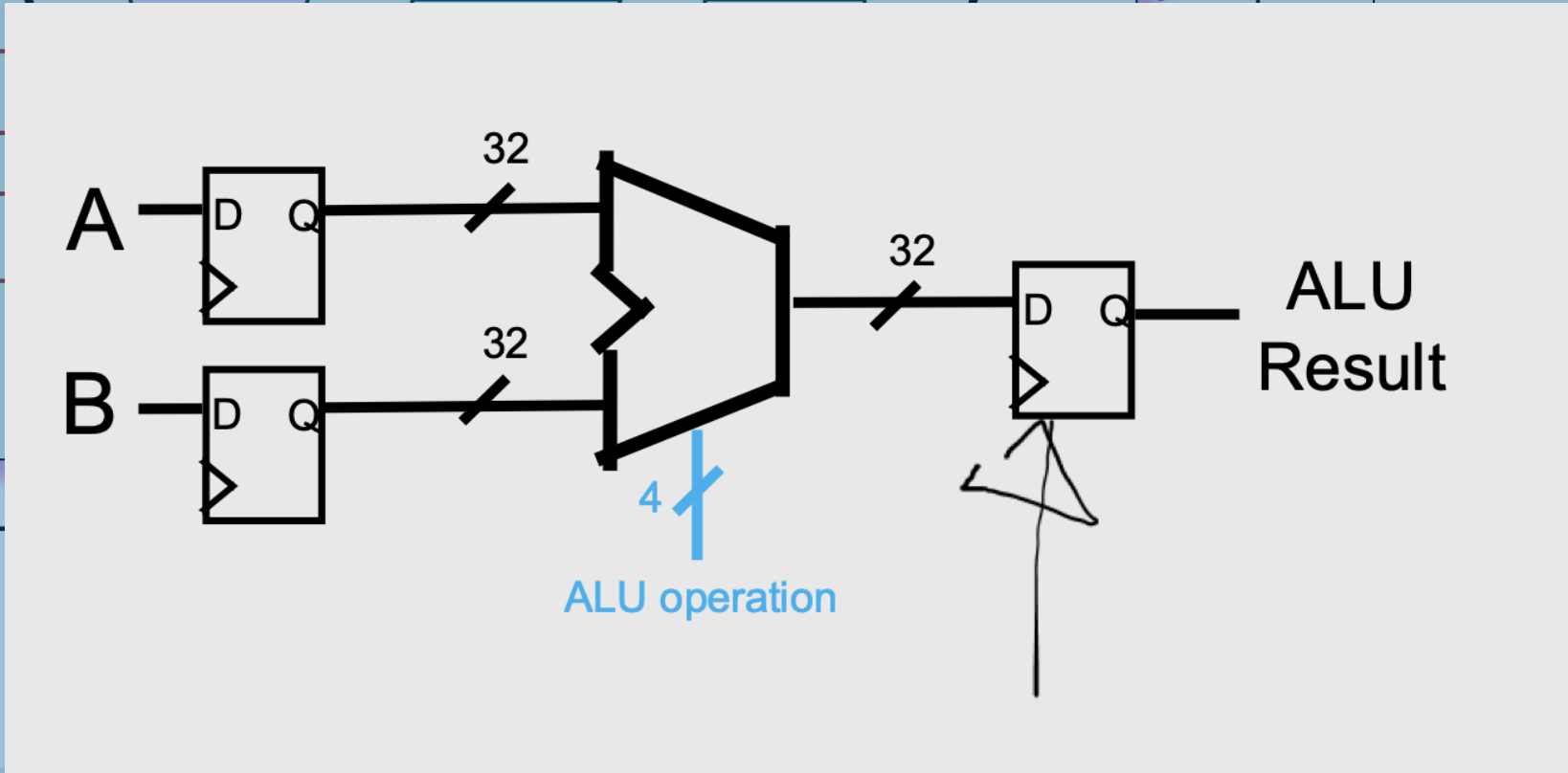


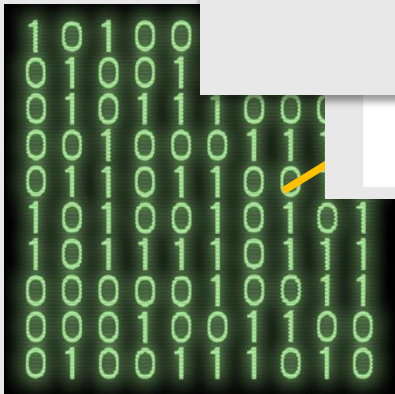
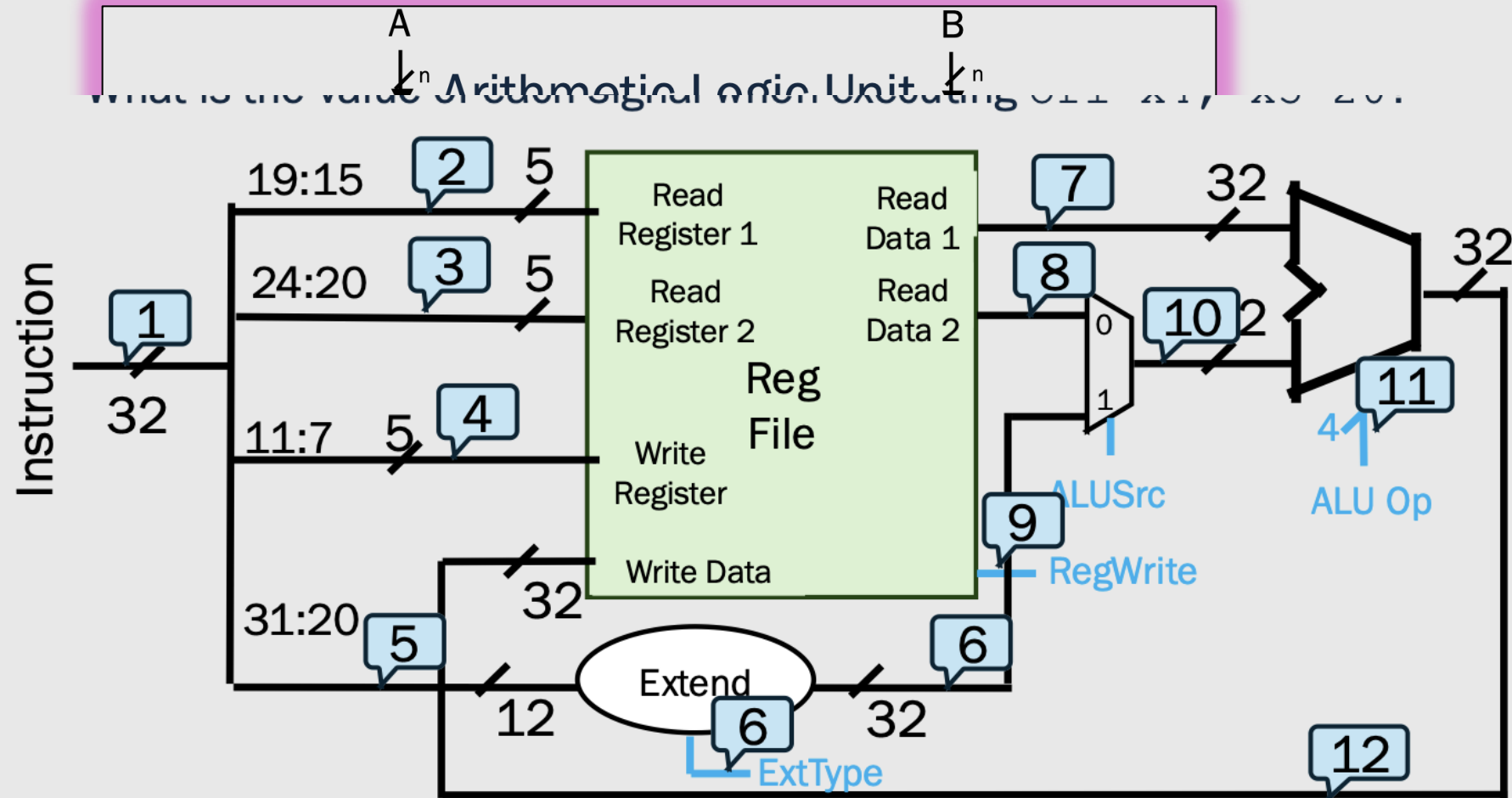




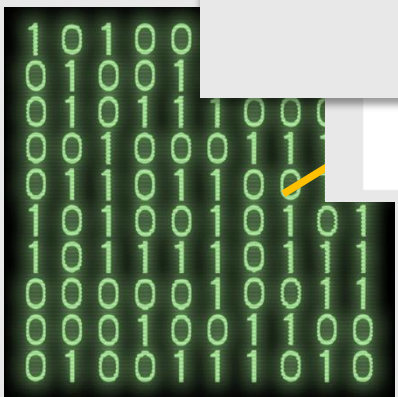
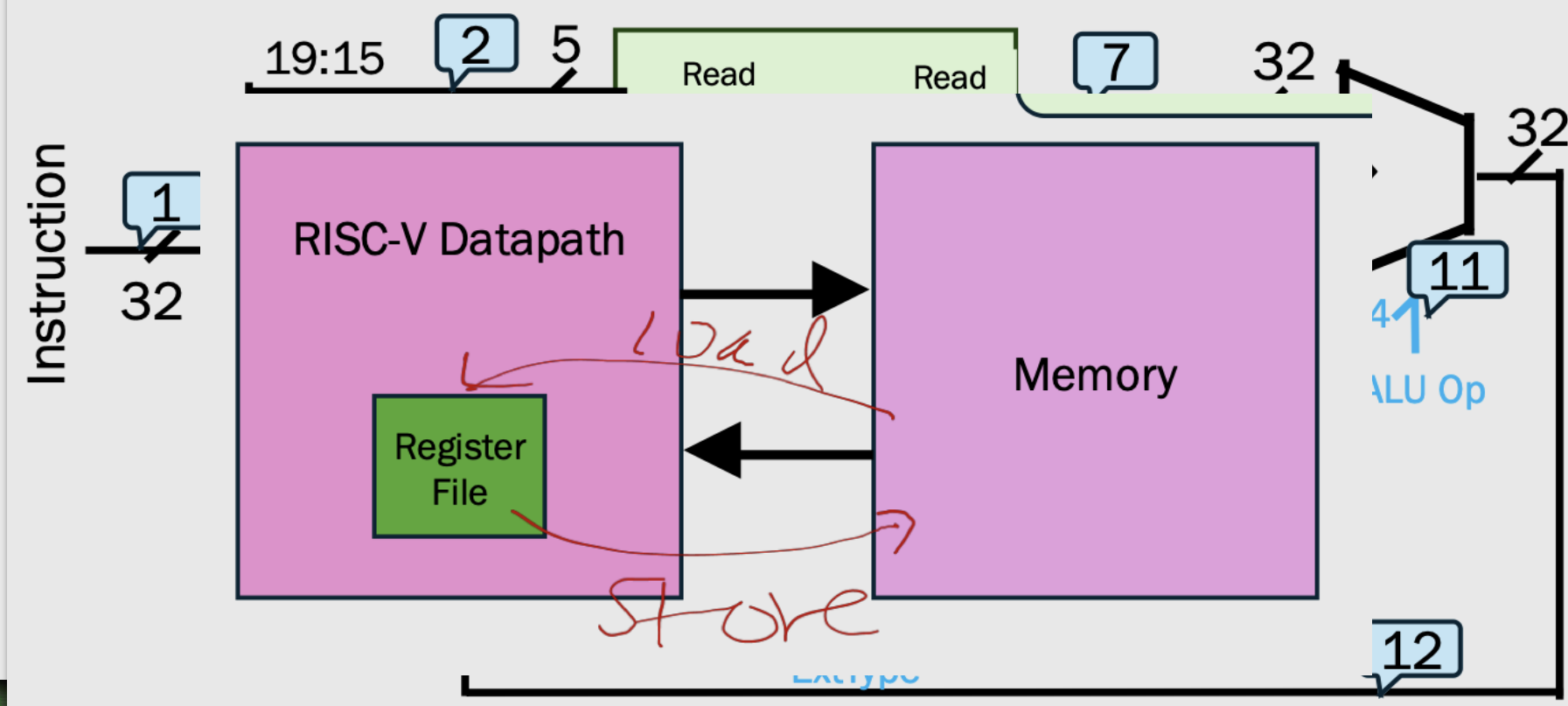
- Sub
- Bool
- Shift
- Math

In





What is the value of the Arithmetic Logic Unit (ALU) output?

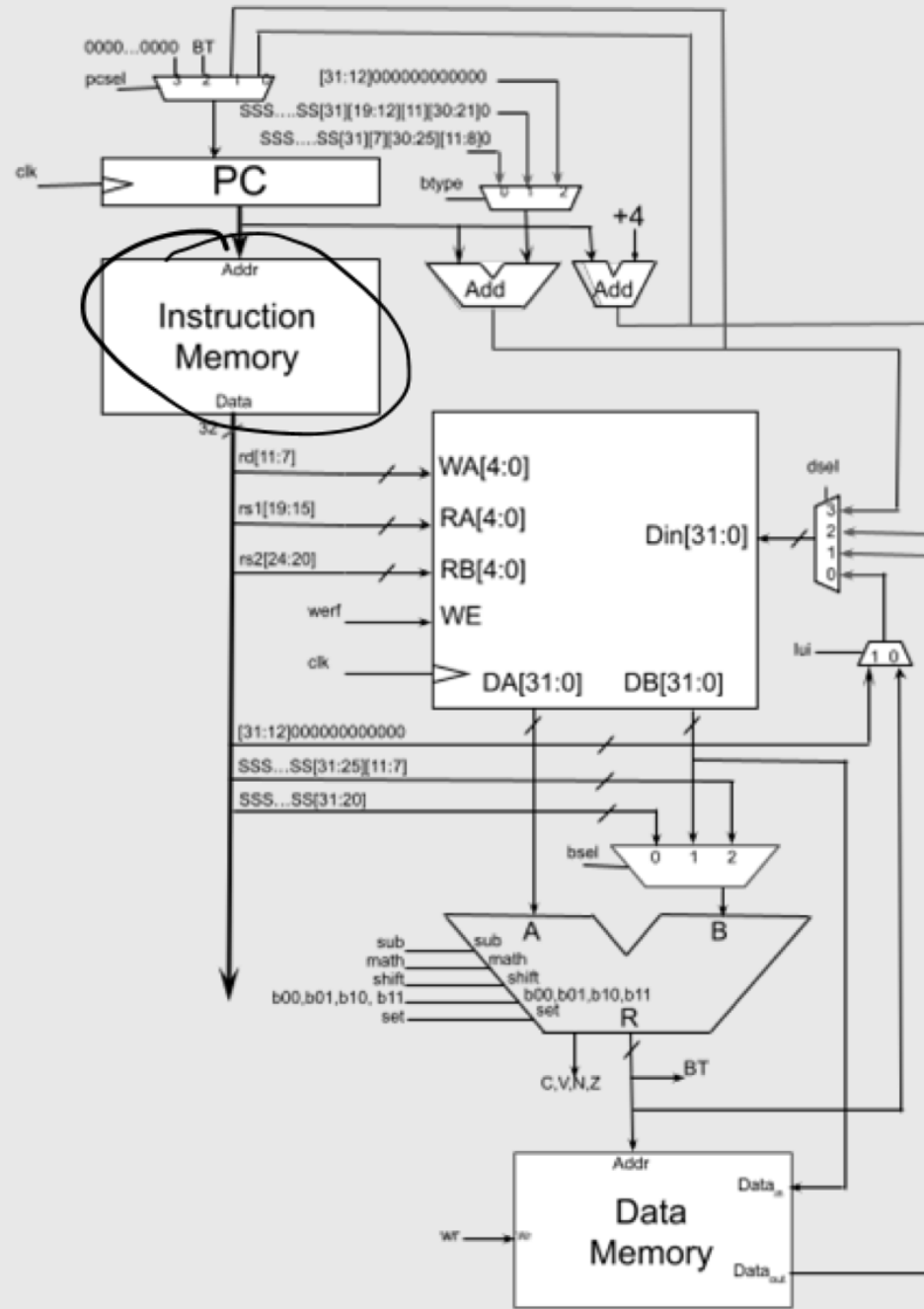


Instruction

1
32

19:

RI



32

11
4
\LU Op

12



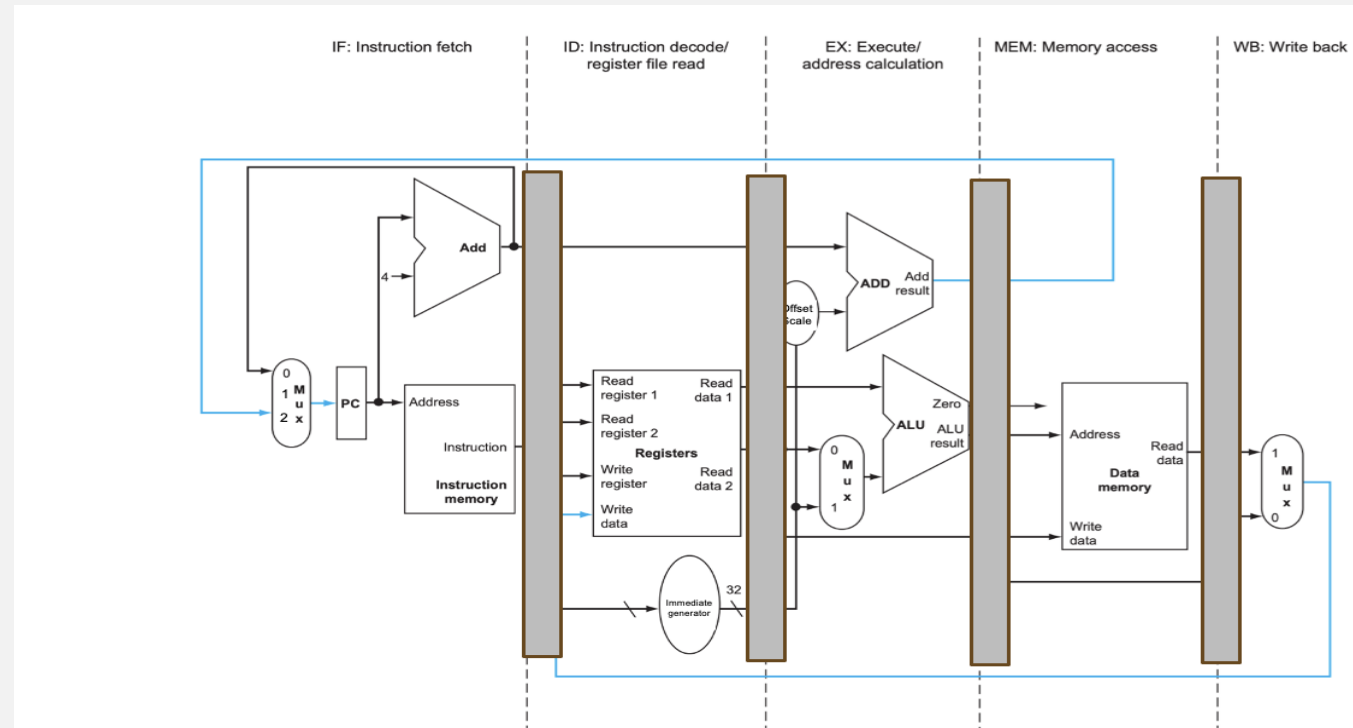
PIPELINE HAZARDS



Data Hazard Tracing

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

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

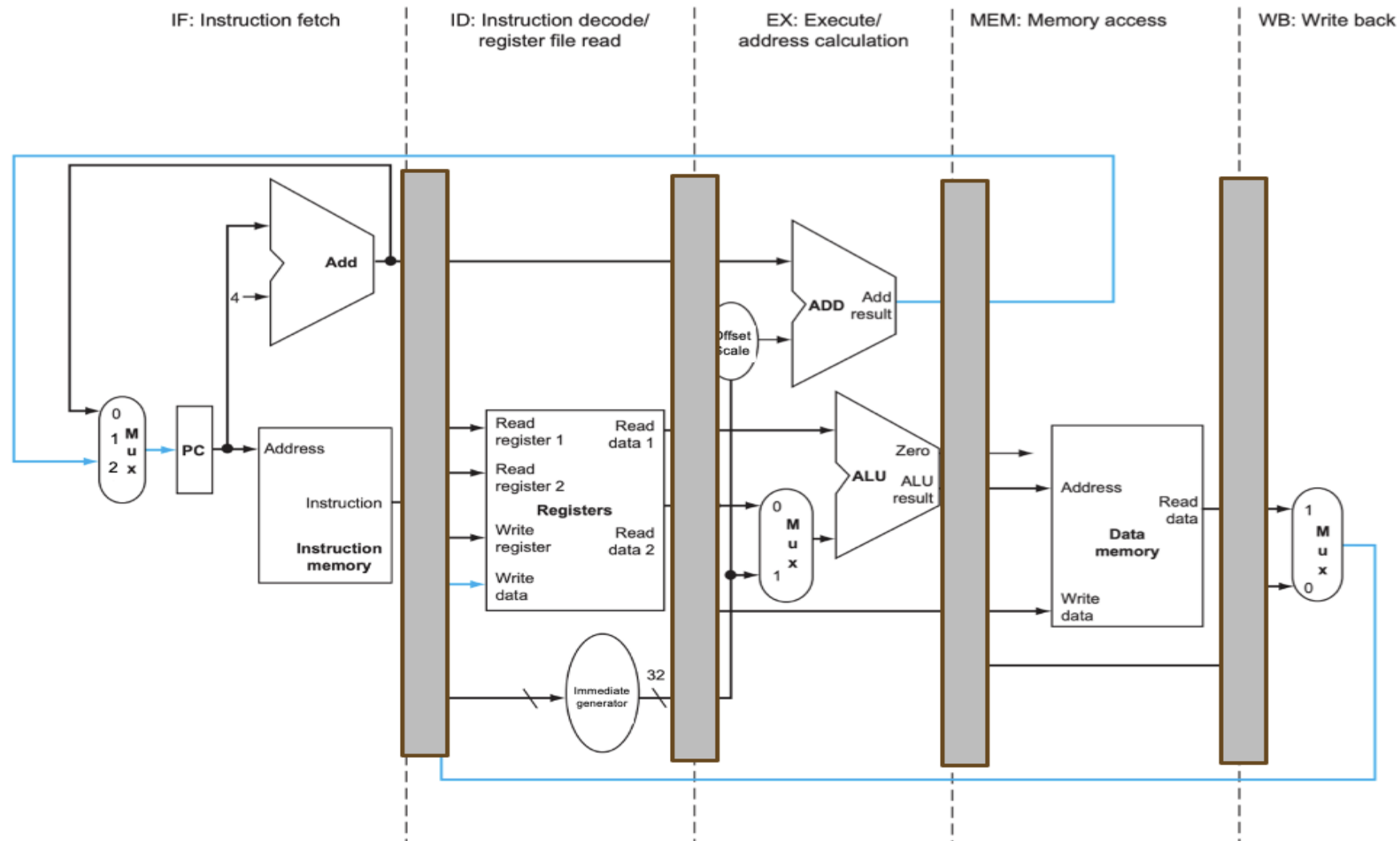


Back to 5 stages! ☺ But first a quick note about pipelining our datapath + pipeline registers

Recall: Pipeline Register

- A register placed between each stage of our pipelined datapath
- Holds data being passed from one stage to the next
- Controls the timing of instruction flow through the pipeline
- Ensures that each stage is operating in isolation

Each pipeline register name is based on the stages it separates



Register Names

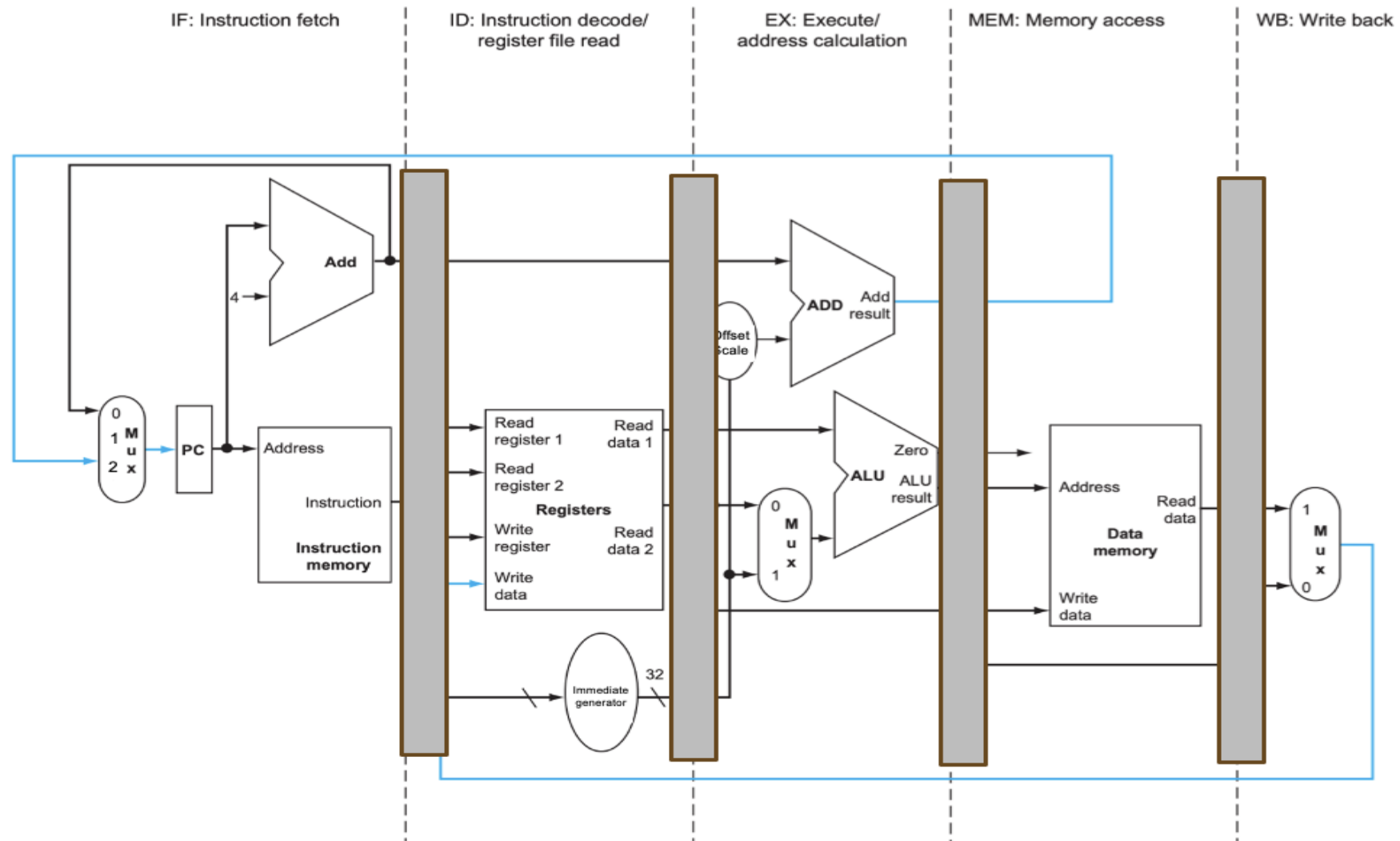
IF/ID

ID/EX

EX/MEM

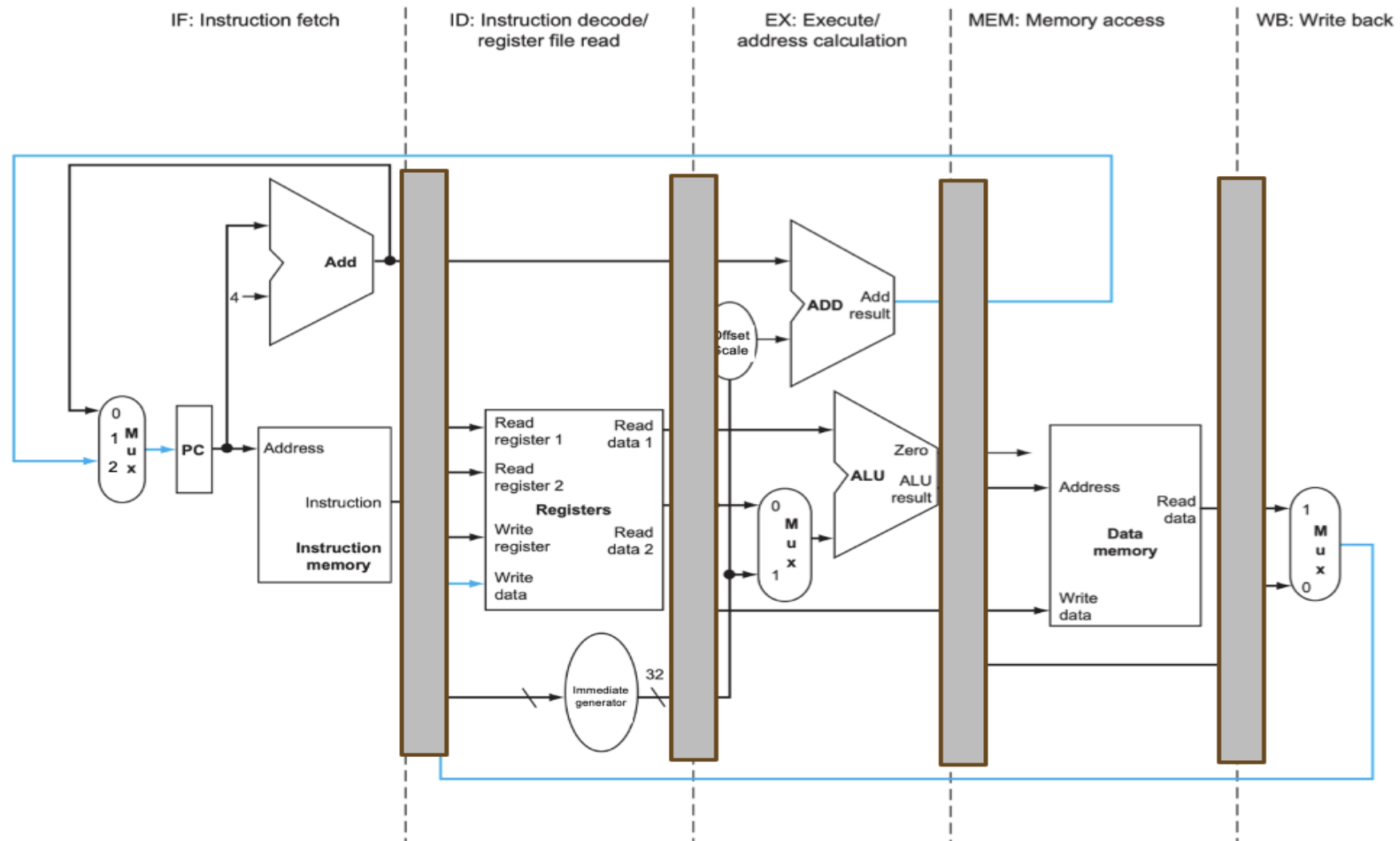
MEM/WB

Cycle 0



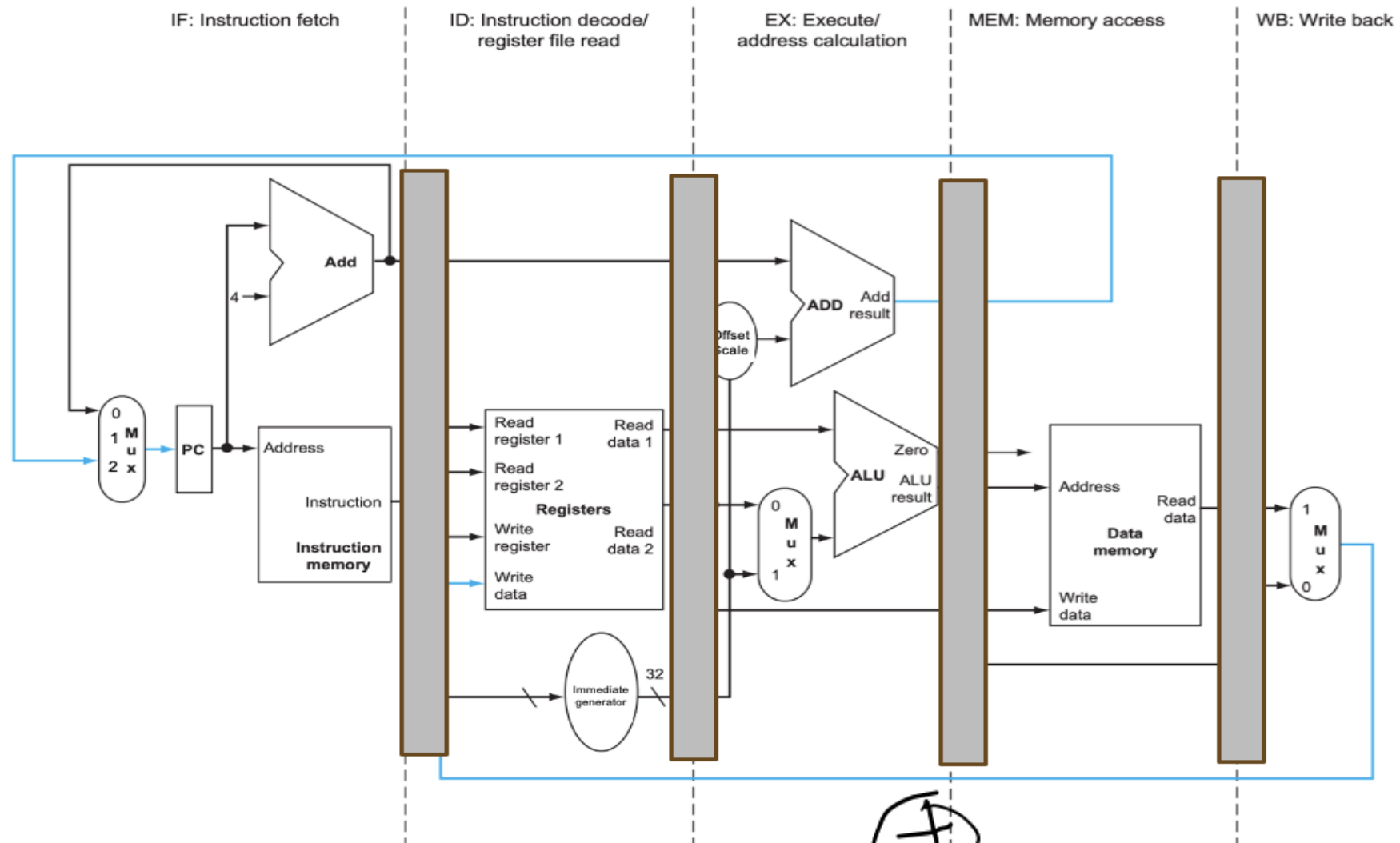
add x5, x3, x4

Cycle 1



sub x7, x5 x2 add x5, x3, x4

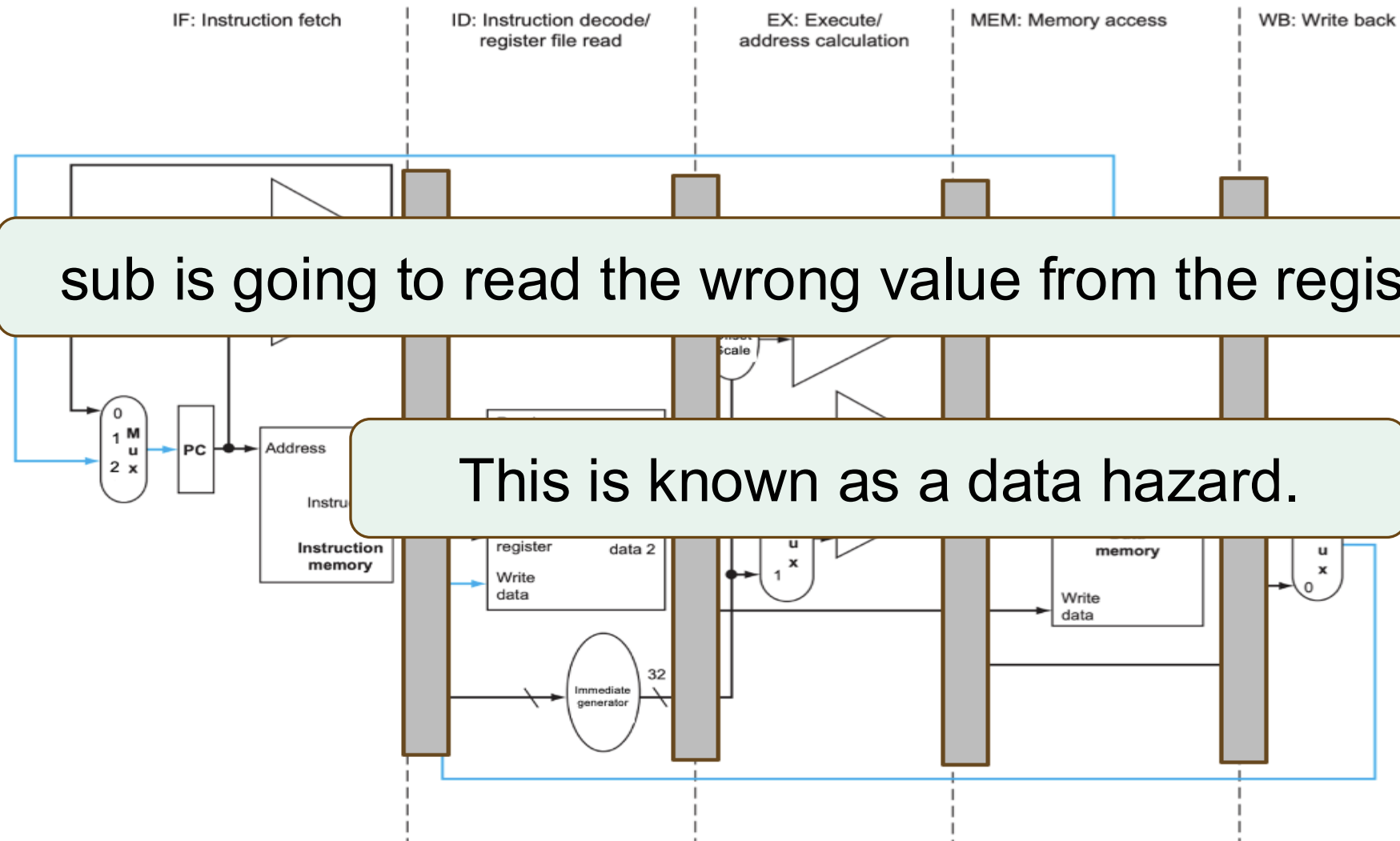
Cycle 2



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

Cycle 2

RAW read after write



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

Pipeline Control Hazards

Pipelining **HAZARDS** are situations where the next instruction cannot execute in the next clock cycle.

Consider the instruction sequence shown:

Control Hazard: the pipeline does not know which instruction to fetch next because it is waiting for the result of a branch or jump instruction to be computed.

```
...  
loop: add  t0,t0,t0  
      addi t1,t1,-1  
      blt  t1,x0,loop  
      srai t0,t0,8  
      sub  t1,t0,t1
```

Time (in clock cycles) →						
	i	i+1	i+2	i+3	i+4	i+5
Fetch	add t0,t0,t0	addi t1,t1,-1	blt t1,x0,loop	srai t0,t0,8	sub t1,t0,t1	add t0,t0,t0
Decode		add t0,t0,t0	addi t1,t1,-1	blt t1,x0,loop	srai t0,t0,8	???
Execute			add t0,t0,t0	addi t1,t1,-1	blt t1,x0,loop	???

Let's consider 3 stages again for a moment...

When the branch instruction reaches the execute stage the next 2 instructions have already been fetched!

Branch Fixes

Delay slot: an instruction slot that comes after a branch or a jump and the processor is guaranteed to execute that instruction before the branch takes effect

Problem: Two instructions following a branch are fetched before the branch decision is made (to take or not to take)

Solutions:

1. Program around it. Define the ISA such that the branch does not take effect until after instructions in the “DELAY SLOTS” complete. **This is how MIPS pipelines work.** It leads to odd looking code in tight (short) loops. You could always put “NOPs” in the delay slots.

2. Detect the branch decision as early as possible, and ANNUL instructions in the delay slots. This is what RISC-V does.

decode

Early Detect and Annul

Detect branch and jump instructions and make the branch decision in the decode stage. We then annul the following instruction by forcing it to be a NOP (addi x0, x0, 0)

```

...
loop:  add  t0,t0,t0
       addi t1,t1,-1
       bne  t1,x0,loop
       srai t0,t0,8
       sub  t1,t0,t1
    
```

It helps if the ALU is not used by branch instructions



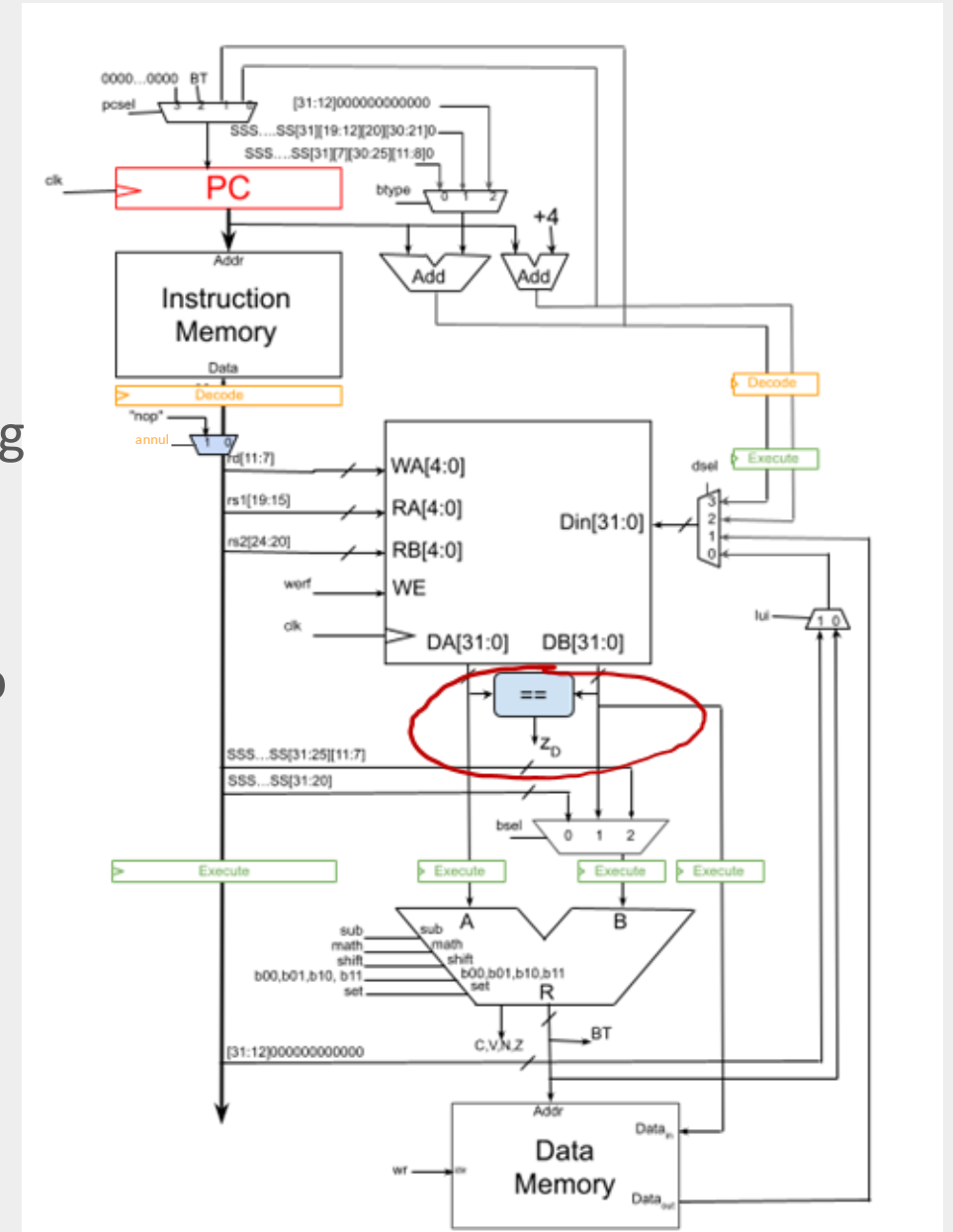
Time (in clock cycles)						
	i	i+1	i+2	i+3	i+4	i+5
Fetch	add t0,t0,t0	addi t1,t1,-1	bne t1,x0,loop	srai t0,t0,8	add t0,t0,t0	addi t1,t1,-1
Decode		add t0,t0,t0	addi t1,t1,-1	bne t1,x0,loop	srai t0,t0,8 nop	add t0,t0,t0
Execute			add t0,t0,t0	addi t1,t1,-1	bne t1,x0,loop	srai t0,t0,8 nop

We detect the branch in the decode stage then the instruction in the fetch stage can be ANNULLED and the next PC changed.



Year	Percentage of Obese People
1980	15%
1985	15%
1990	15%
1995	16%
2000	17%
2005	19%
2010	20%

1. Add H/W to test if the branch is taken during register access. Easy for beq and bne (just NOR together the bitwise XORs). Use this zero detection from the decode stage, z_D , to select the next pc
(btype = 0, pcsel = 1)
2. Add a multiplexer to replace the already fetched instruction with a NOP, and, thus ANNUL it.



The cost of taken branches

When an RISC-V branch is *taken* the branch instructions are effectively **2 cycles rather than 1** when they aren't. In a MIPS-like instruction set, one can often fill the delay slots with useful instructions, but they are executed whether or not the branch is taken.

The RISC-V approach is easier to understand, and since it does not “EXPOSE” the pipeline, it also allows for an alternative number of pipeline stages to be implemented in future designs, while conserving code compatibility.

Lastly, using RISC-V, many **conditional branches can be eliminated using the condition execution**, which pipelines beautifully!

The MIPS vs. RISC-V Approach

```
loop:
  add  $t0, $t0, $t0    # work
  addi $t1, $t1, -1
  blt  $t1, $zero, loop
  nop          # delay slot (executes even if taken!)
  srai $t0, $t0, 8
  sub  $t1, $t0, $t1
```

```
loop:
  add  t0, t0, t0
  addi t1, t1, -1
  blt  t1, x0, loop    # resolved in Decode
  srai t0, t0, 8
  sub  t1, t0, t1
```

One instruction after branch always executes. The compiler must fill this with useful work, or a NOP!

Branch is resolved early and the next instruction can be “Annulled” or turned into a NOP if mispredicted

Structural Pipeline Hazards

Structural hazard: two instructions competing for hardware unit

There's another problem with our code fragment!

The destination register of instructions are written at the end of the Execute stage. However the following instruction might use this result as a source operand.

```
...
loop:  add  t0,t0,t0
       addi t1,t1,-1
       bne  t1,x0,loop
       srai t0,t0,8
       sub t1,t0,t1
```

Time (in clock cycles)						
	i	i+1	i+2	i+3	i+4	i+5
Fetch	add t0,t0,t0	addi t1,t1,-1	bne t1,x0,loop	srai t0,t0,8	add t0,t0,t0	addi t1,t1,-1
Decode		add t0,t0,t0	addi t1,t1,-1	bne t1 ,x0,loop	srai t0,t0,8 nop	add t0,t0,t0
Execute			add t0,t0,t0	addi t1 ,t1,-1	bne t1,x0,loop	nop

The "bne" instruction needs to access the contents of t1 before it is actually written at the end of i+3

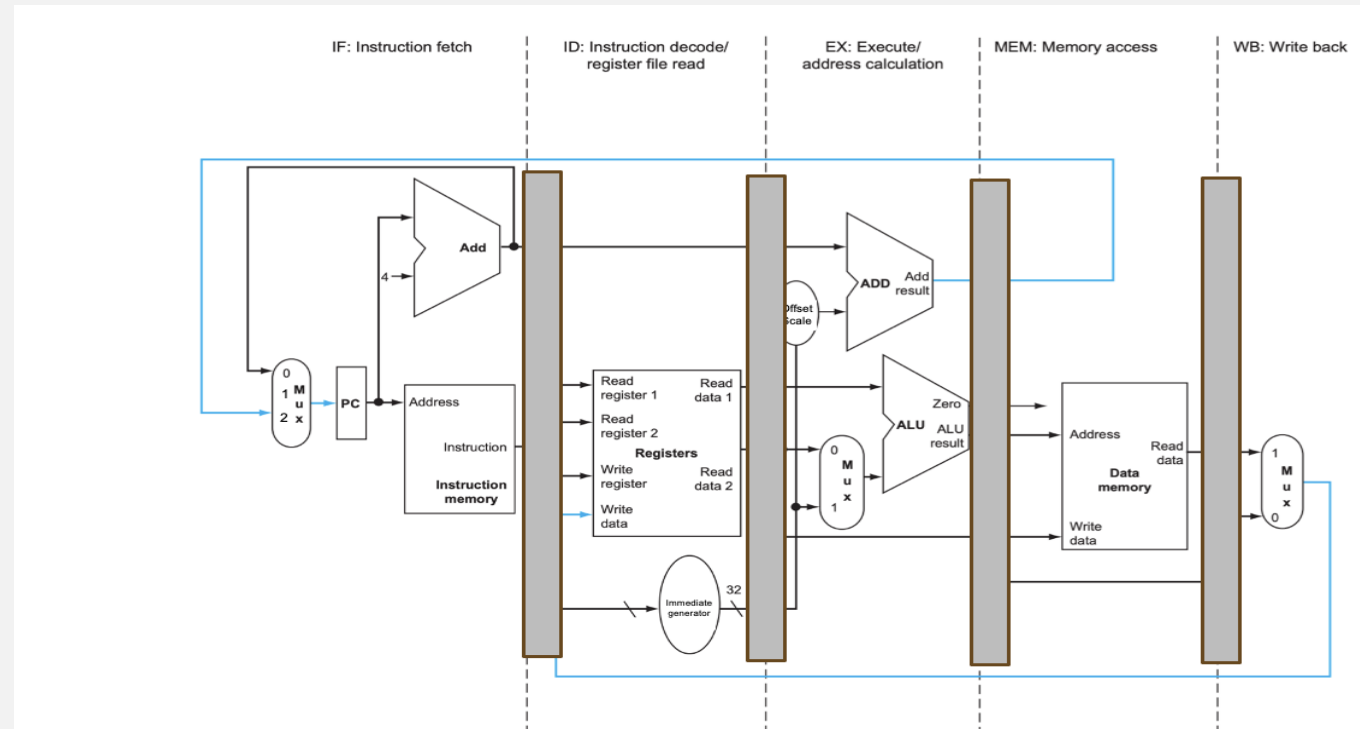


Data Hazard Tracing

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

add x5, x3, x4

sub x7, x5, x2

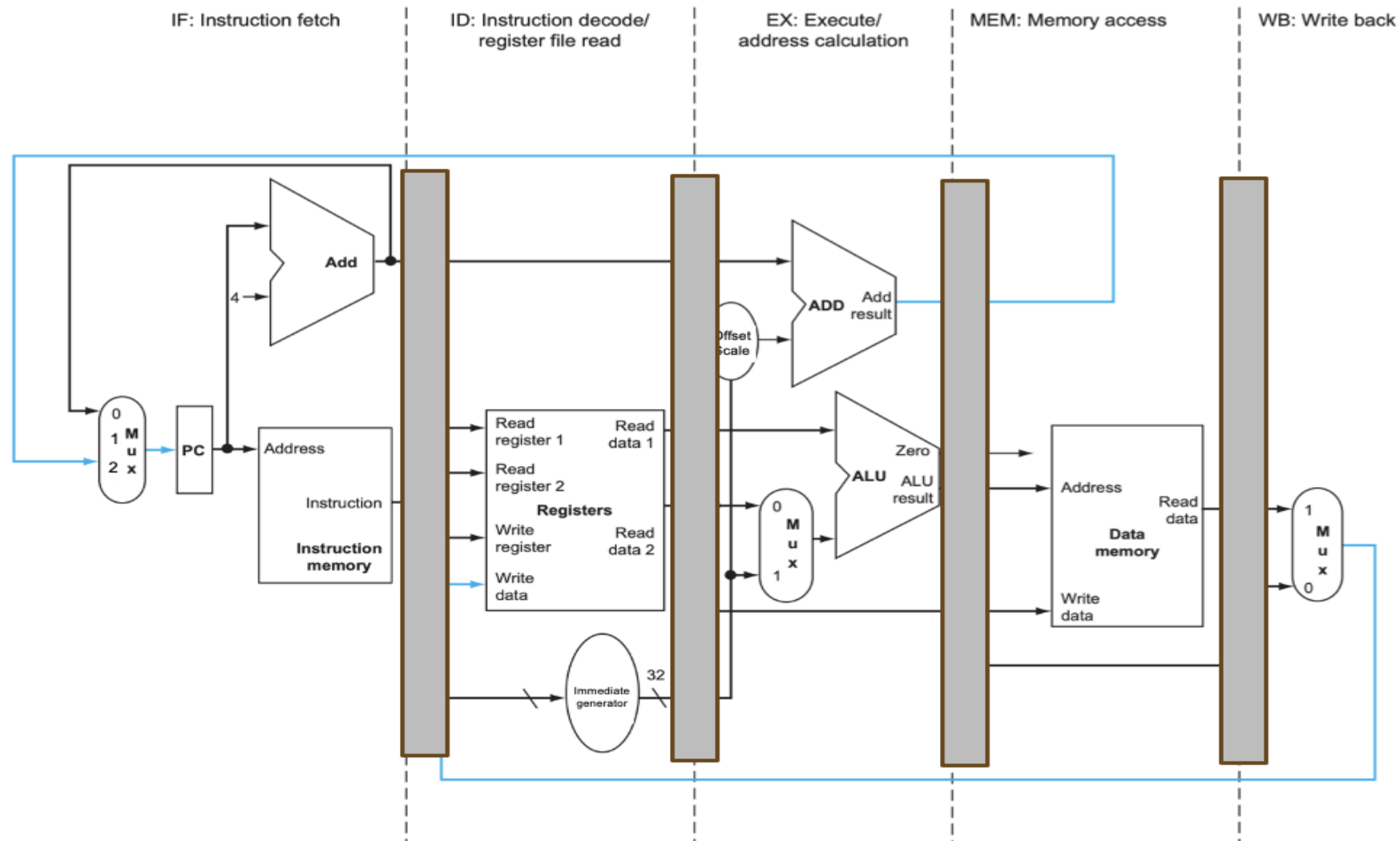


Back to 5 stages! ☺ But first a quick note about pipelining our datapath + pipeline registers

Recall: Pipeline Register

- A register placed between each stage of our pipelined datapath
- Holds data being passed from one stage to the next
- Controls the timing of instruction flow through the pipeline
- Ensures that each stage is operating in isolation

Each pipeline register name is based on the stages it separates



Register Names

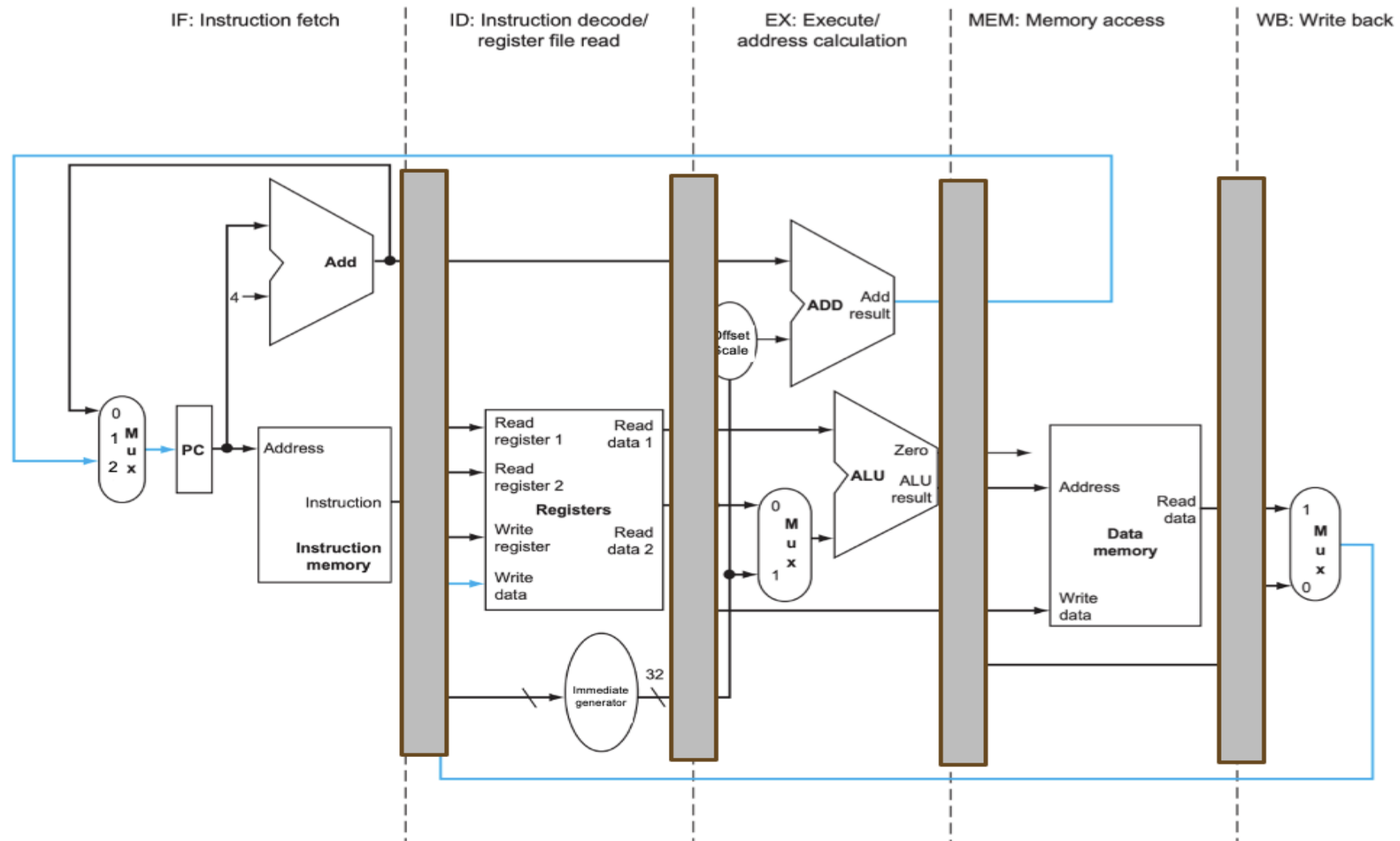
IF/ID

ID/EX

EX/MEM

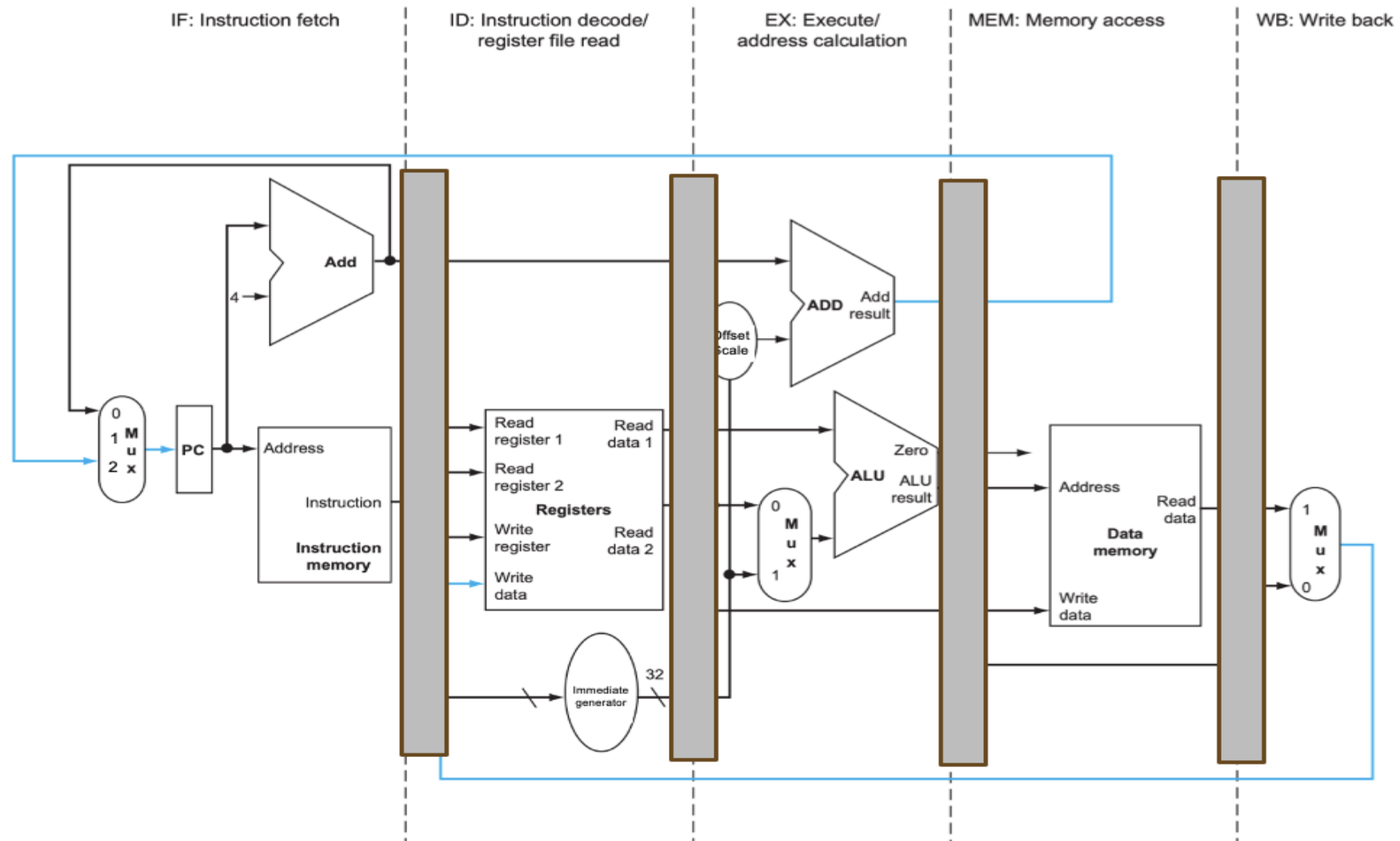
MEM/WB

Cycle 0



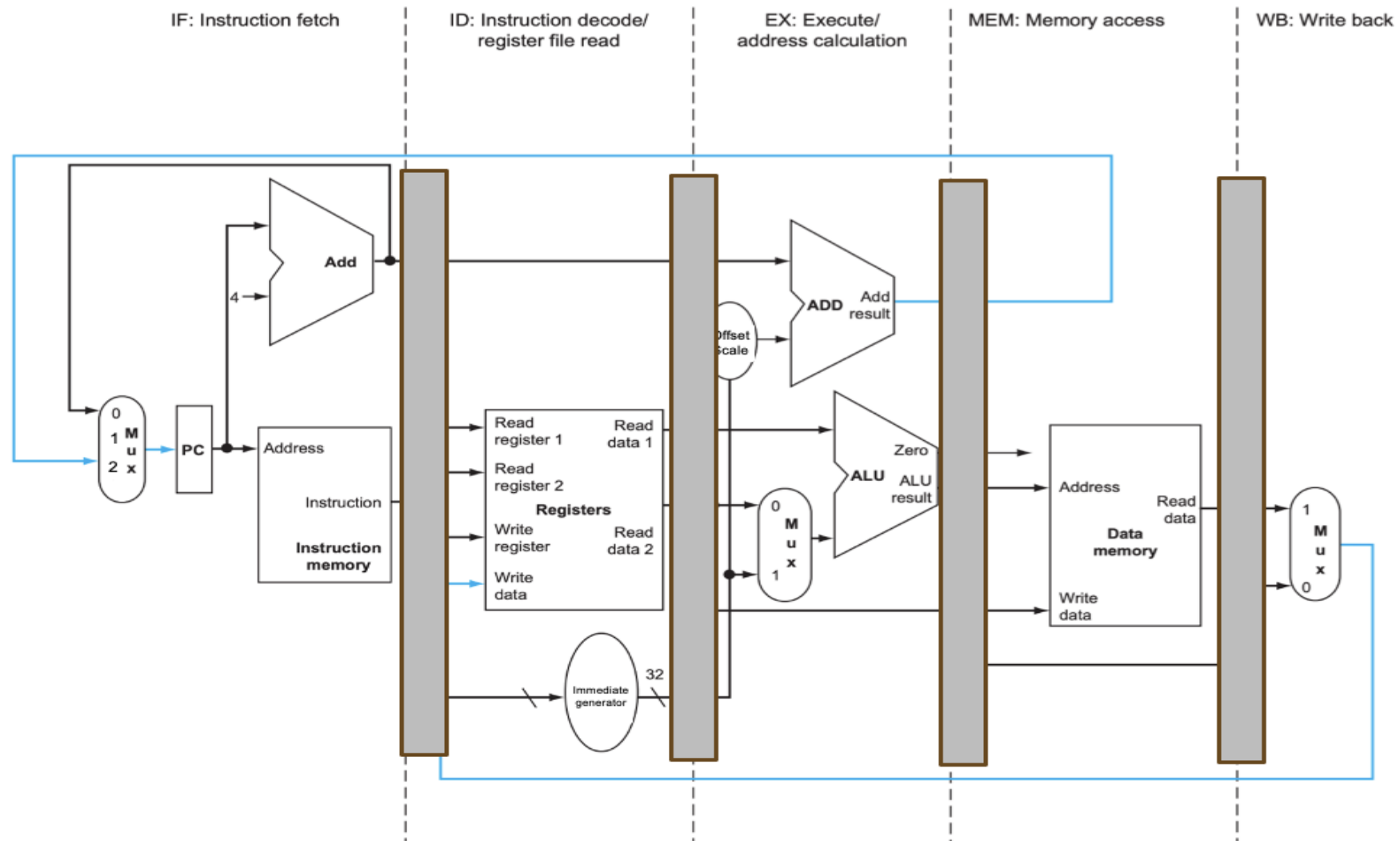
add x5, x3, x4

Cycle 1



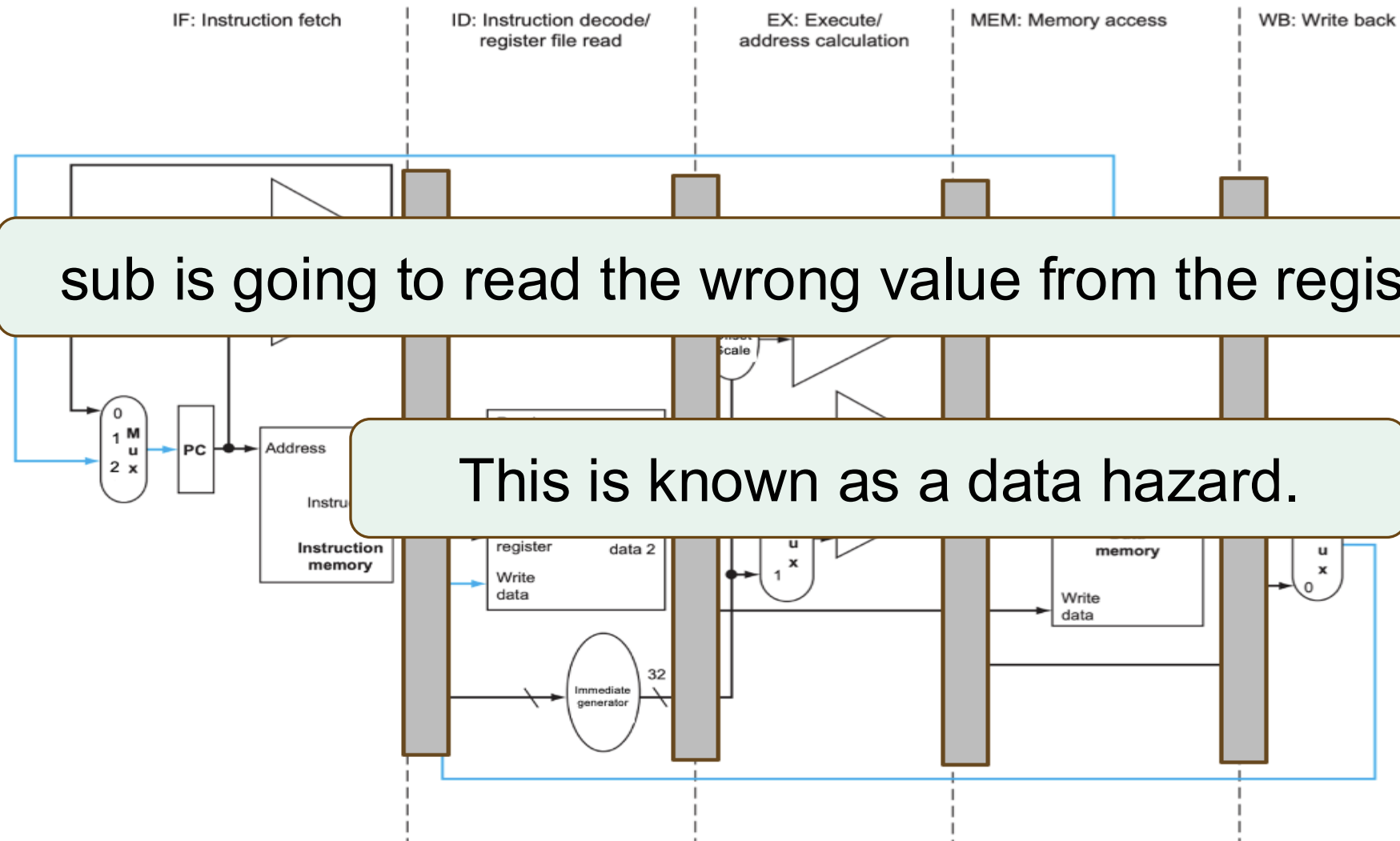
sub x7, x5, x2 add x5, x3, x4

Cycle 2



sub x7, x5 x2 add x5, x3, x4

Cycle 2



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

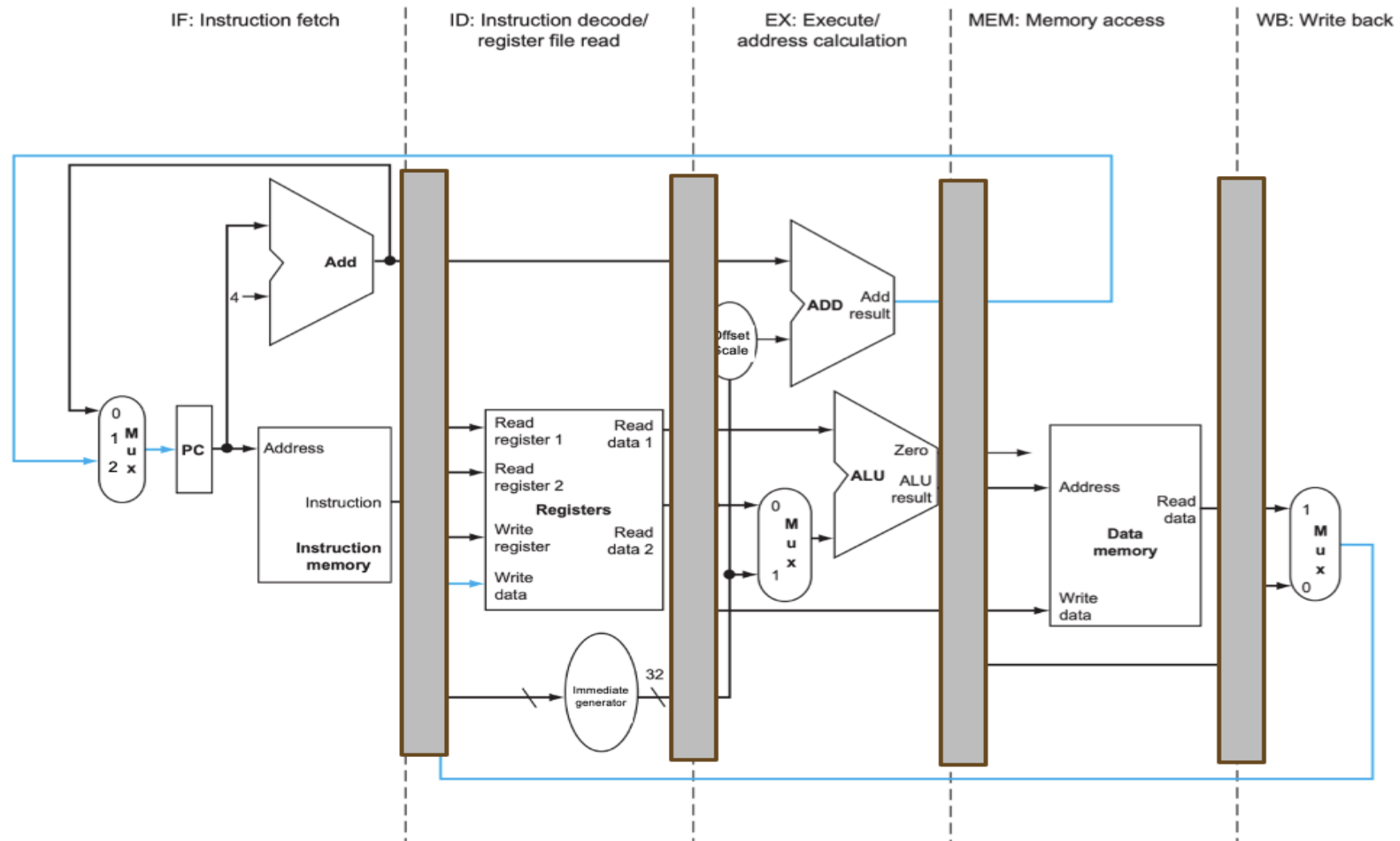
How can we fix this?

→ Introducing pipeline stalls!
(aka bubbles or nops)

Pipeline Stalls

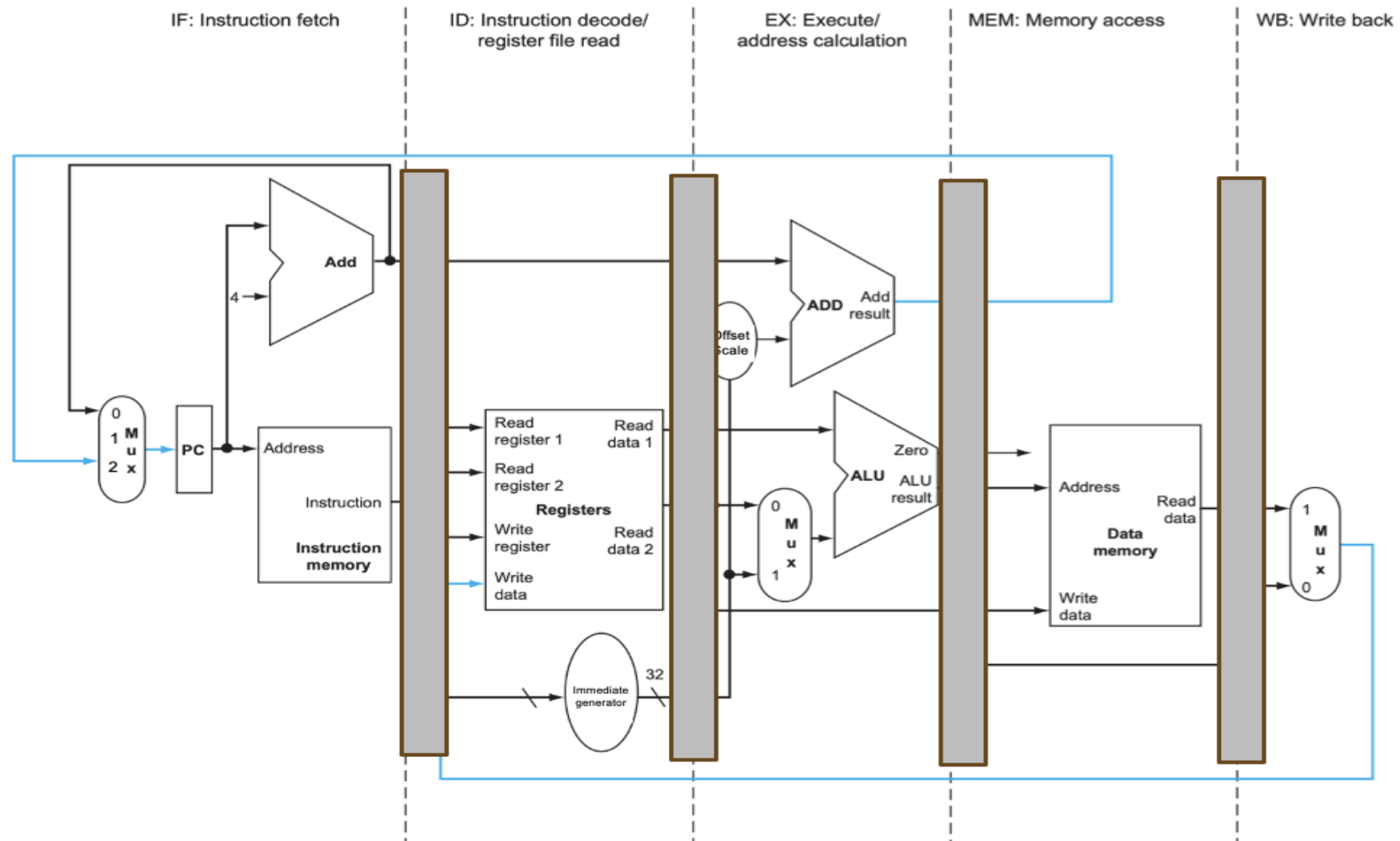
- Also known as
 - bubbles
 - nops (no operation)
- An instruction that does nothing
- Provides time for the correct value to be written to the register file

Cycle 2



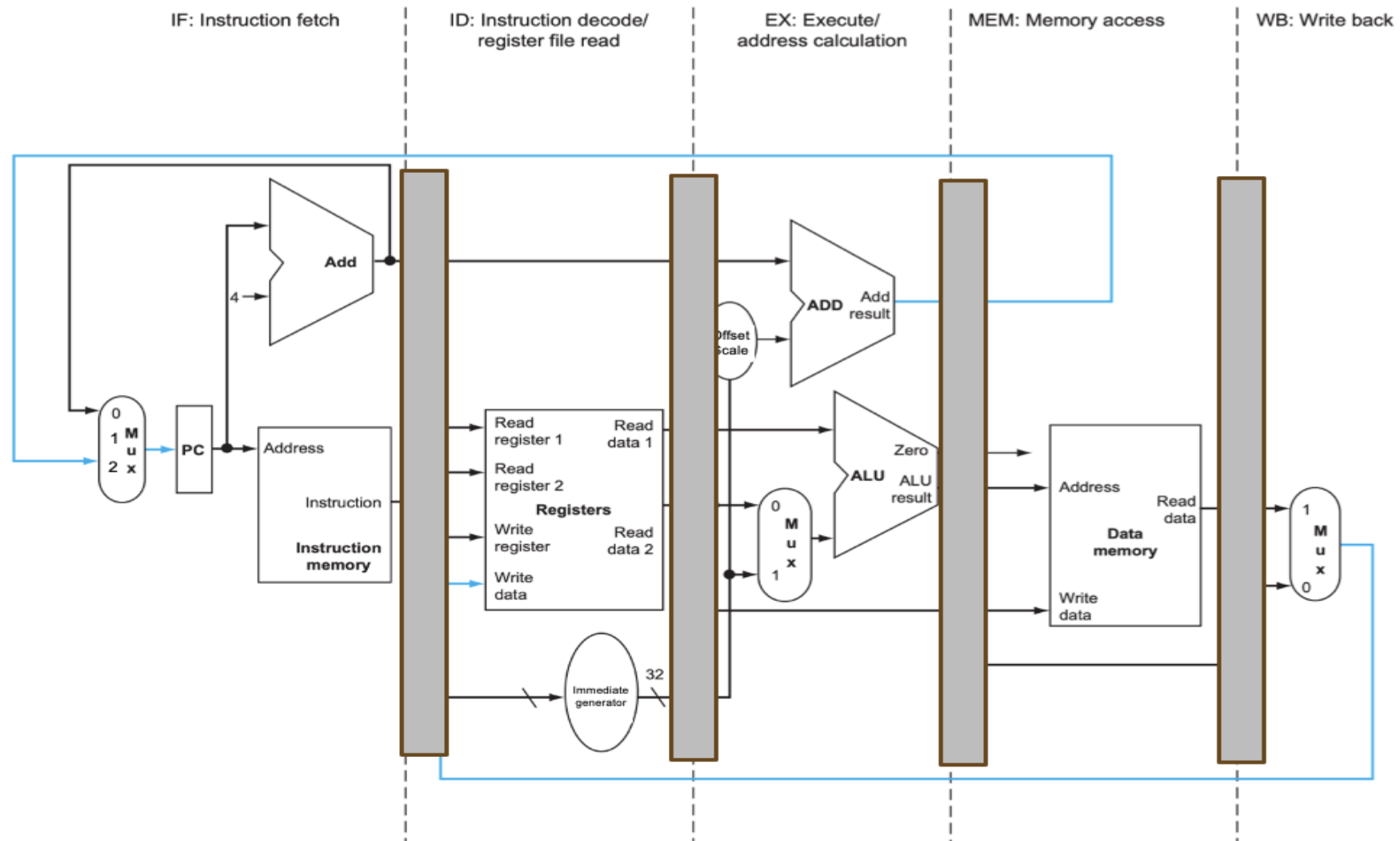
sub x7, x5, x2add x5, x3, x4

Cycle 3



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

Cycle 4



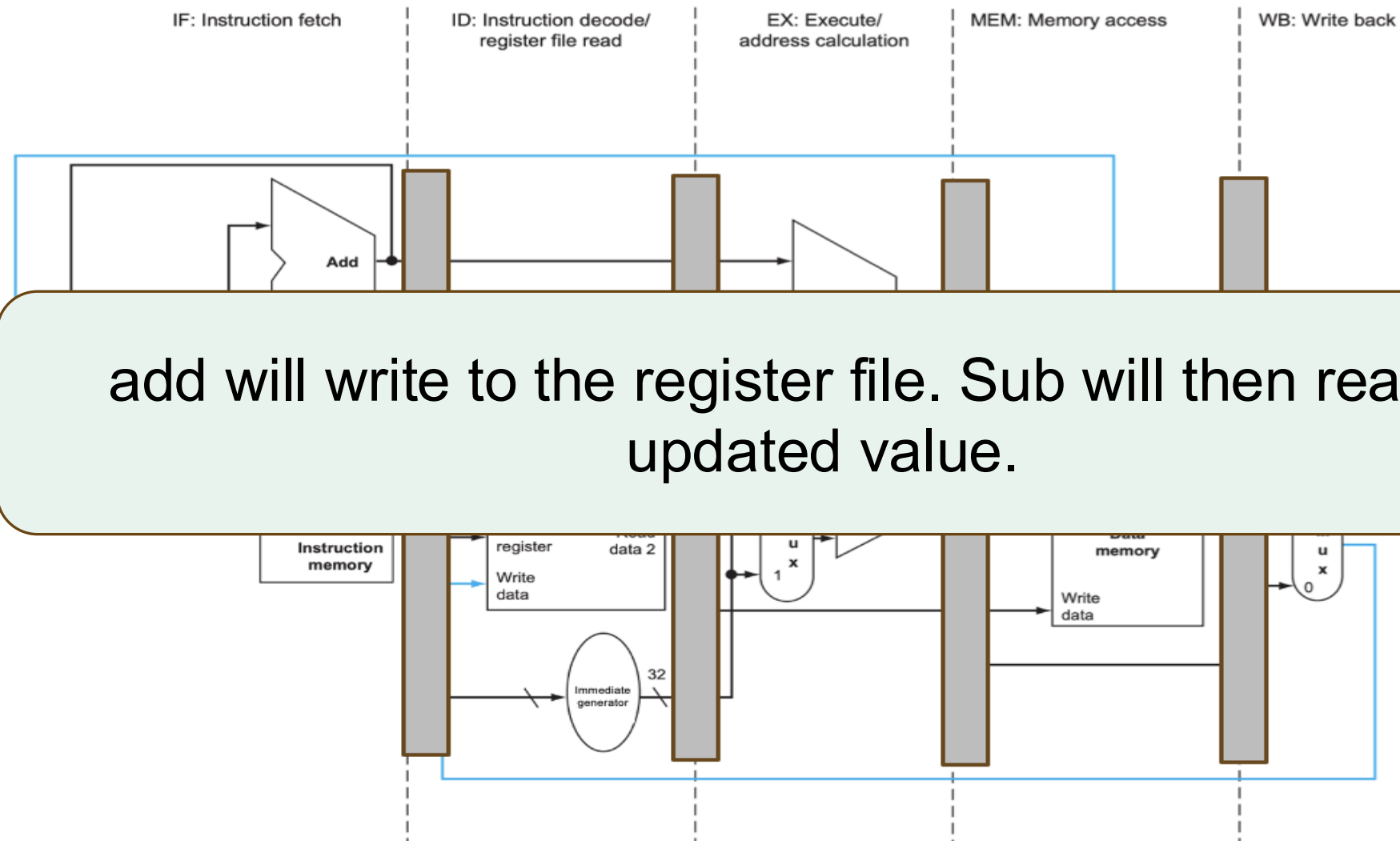
sub x7, x5, x2

STALL

STALL

add x5, x3, x4

Cycle 4



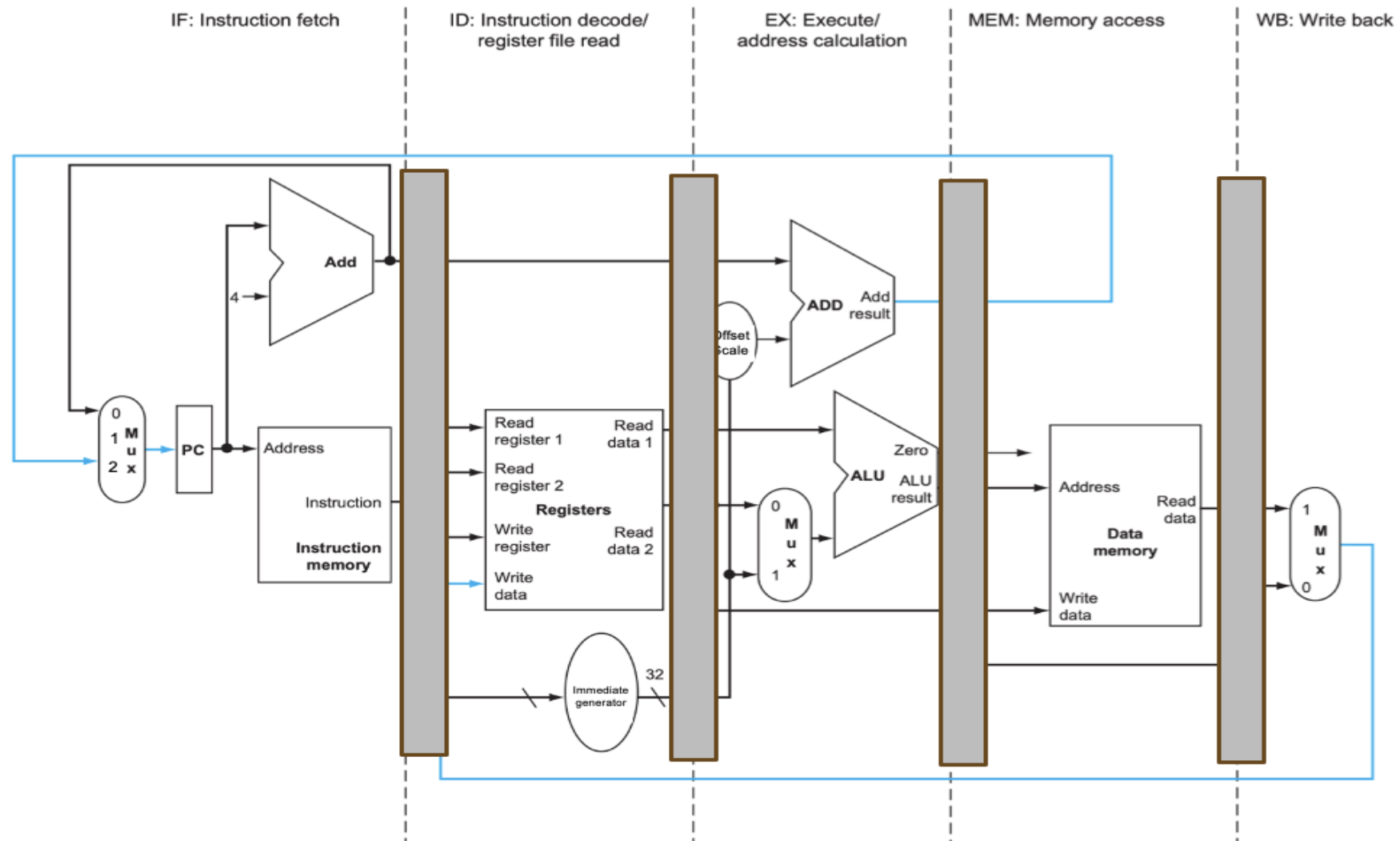
sub x7, x5, x2

STALL

STALL

add x5, x3, x4

Cycle 5

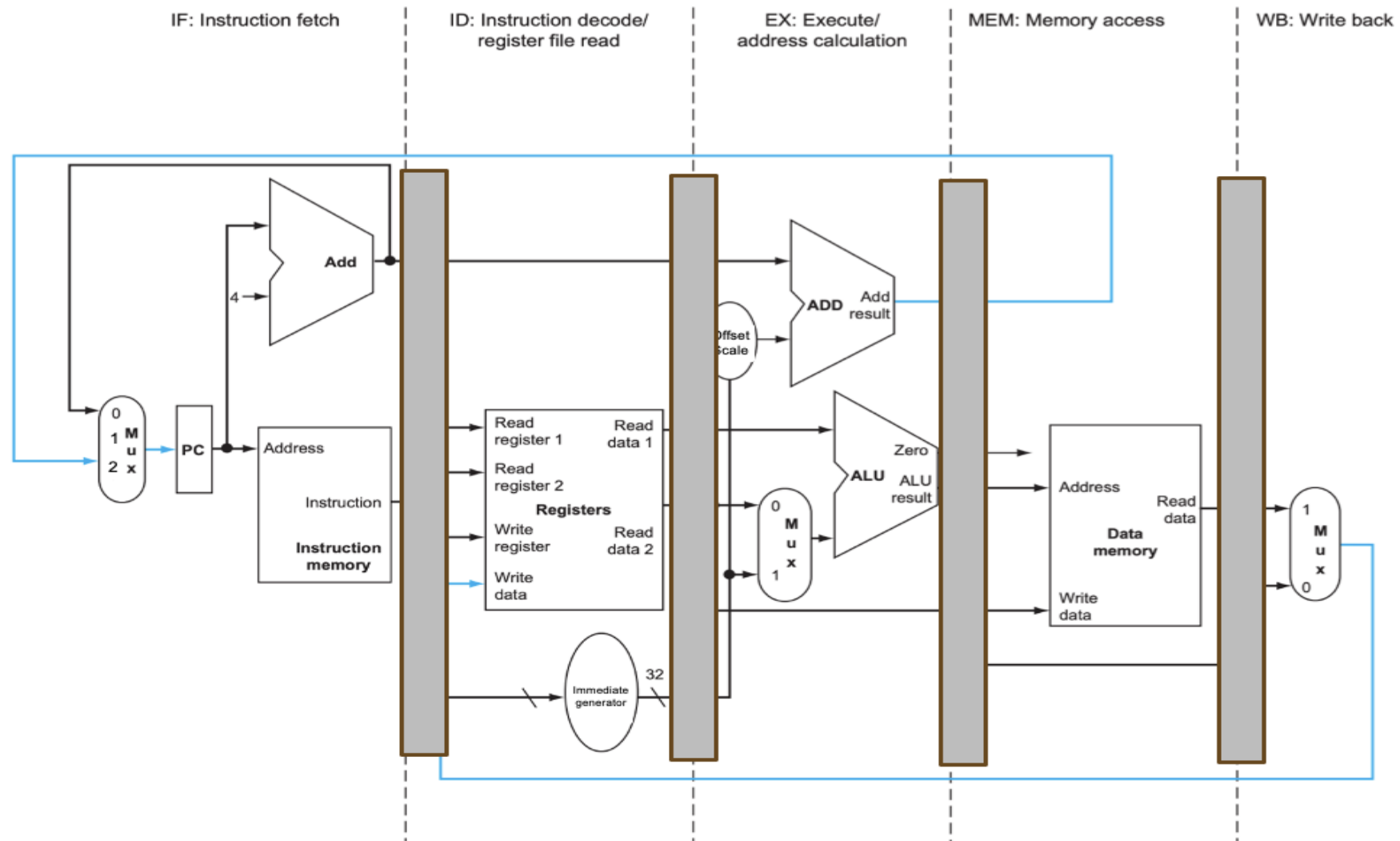


sub x7, x5, x2

STALL

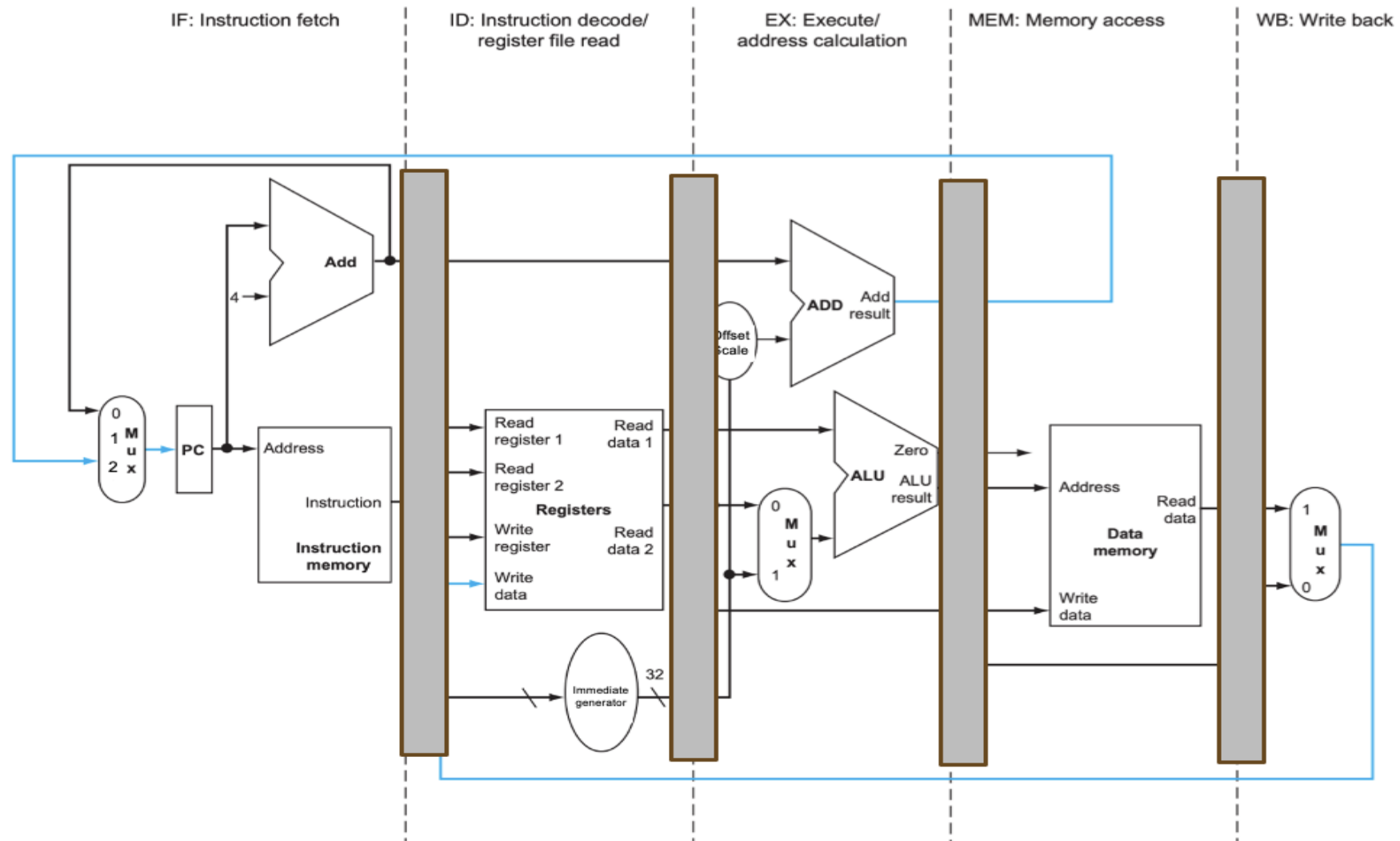
STALL

Cycle 6



sub x7, x5, x2 **STALL**

Cycle 7



sub x7, x5, x2

Pipeline Stalls

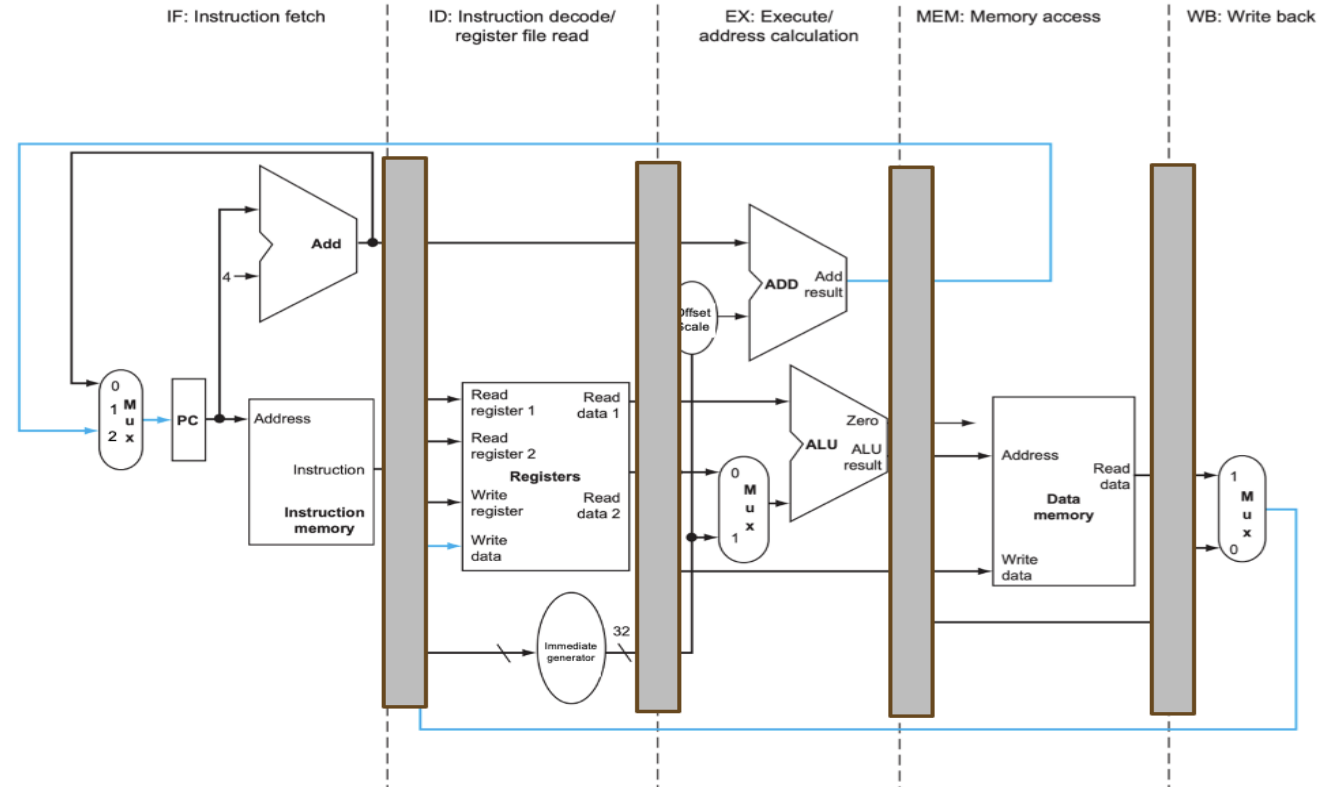
- In this example, we inserted two stalls between the instructions to ensure that sub would read the correct value from x5

```
add x5, x3, x4  
} STALL }  
} STALL }  
sub x7, x5, x2
```

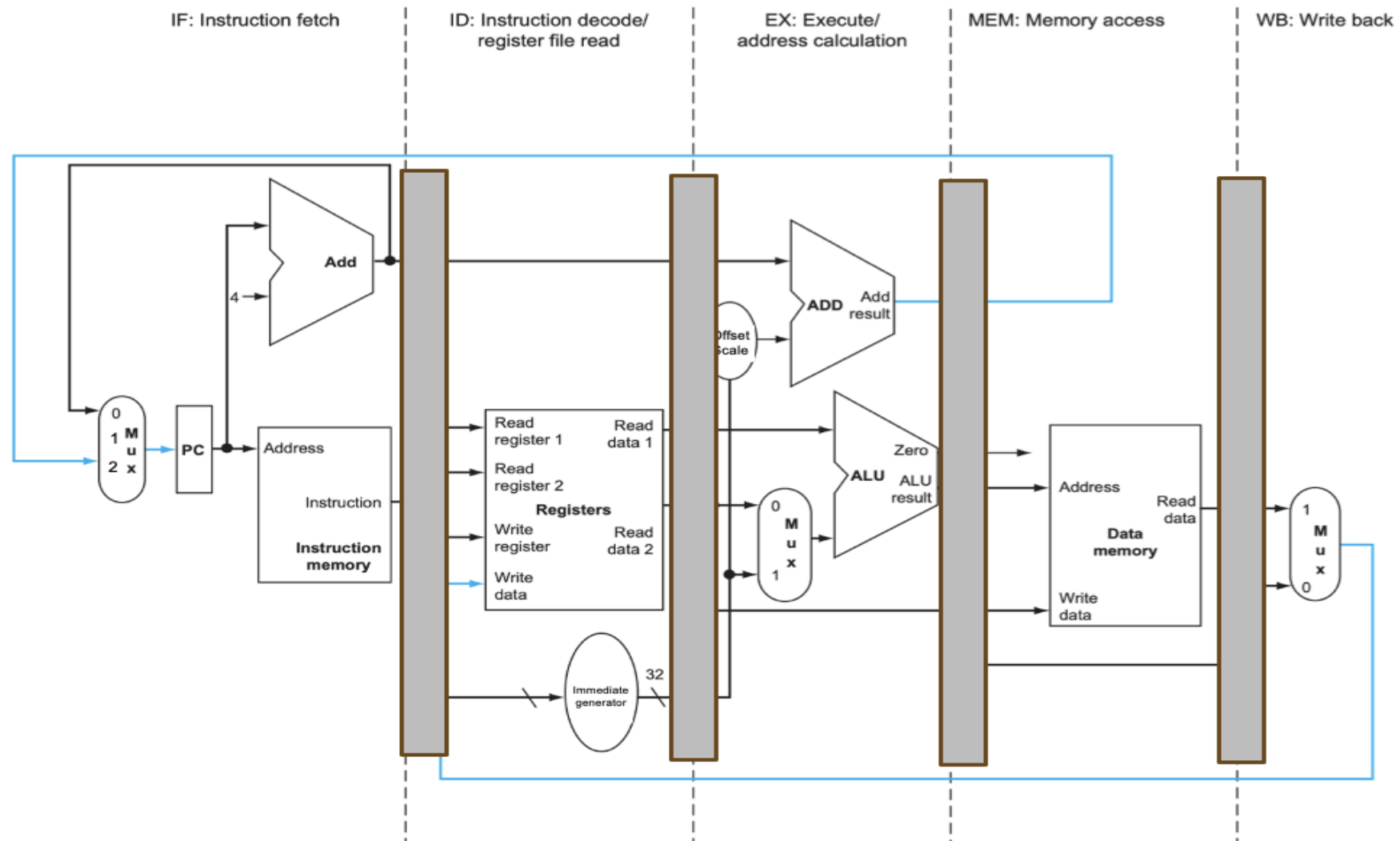
Data Hazards

What if there was an independent instruction in between?

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

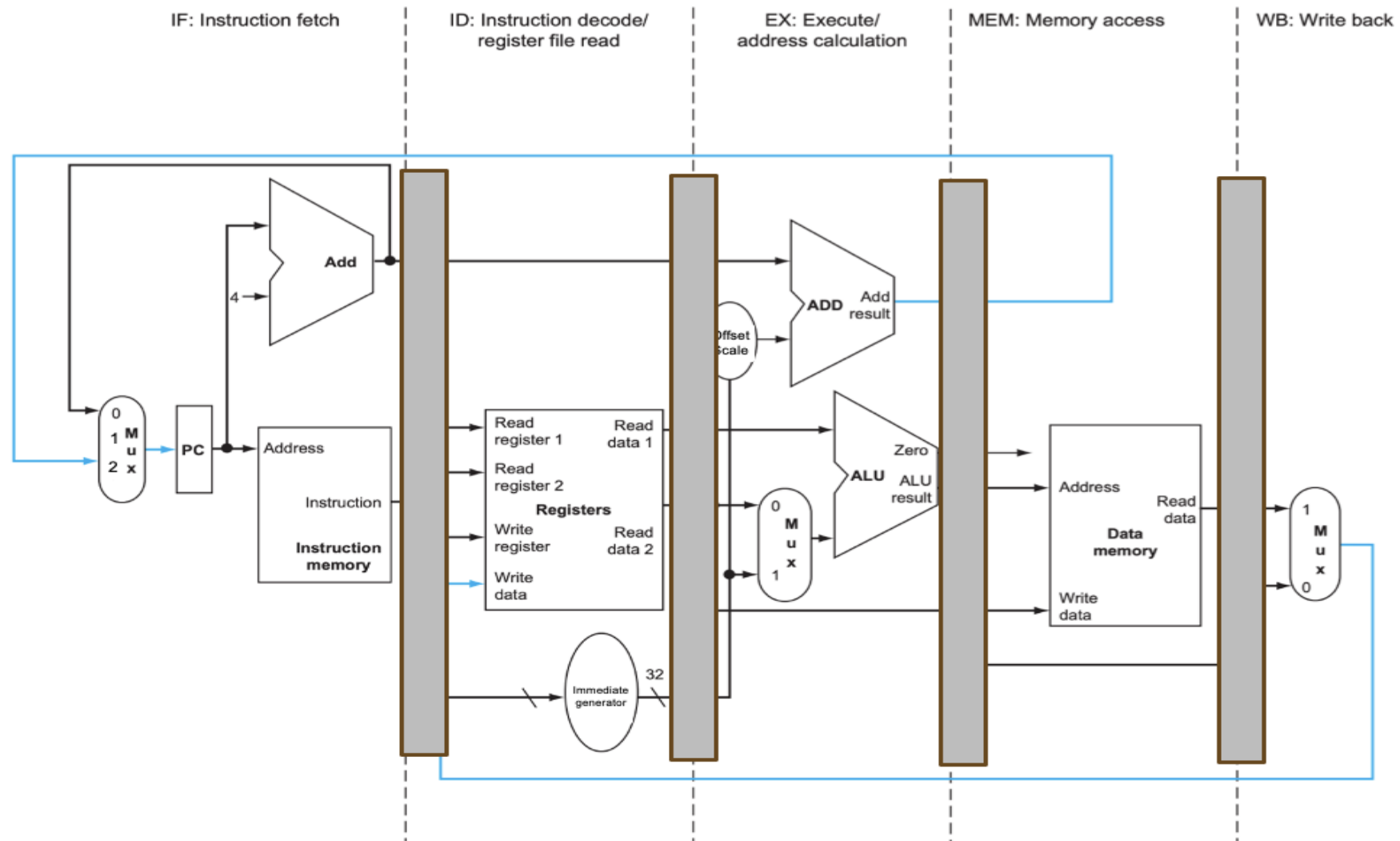


Cycle 0



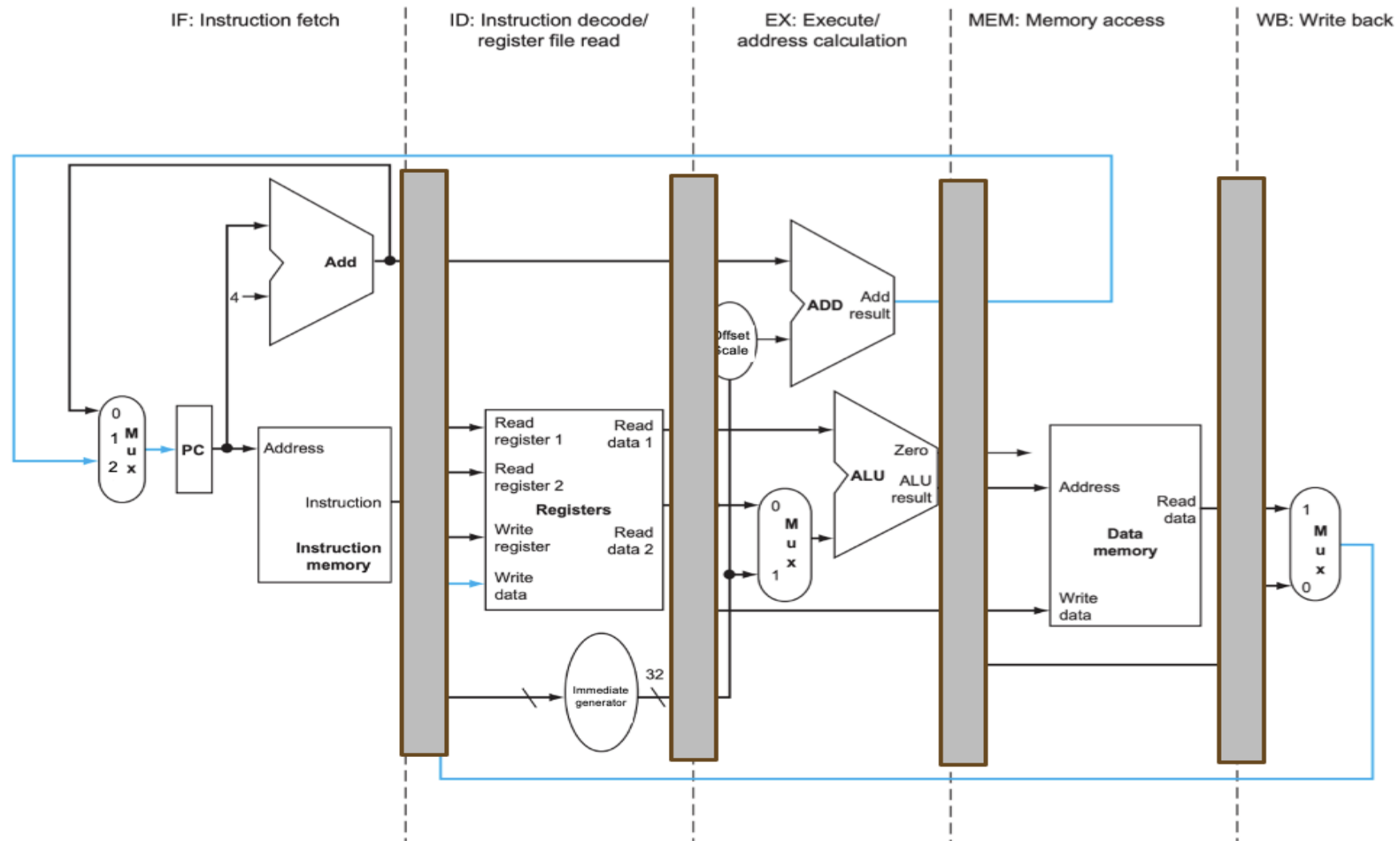
add x5, x3, x4

Cycle 1



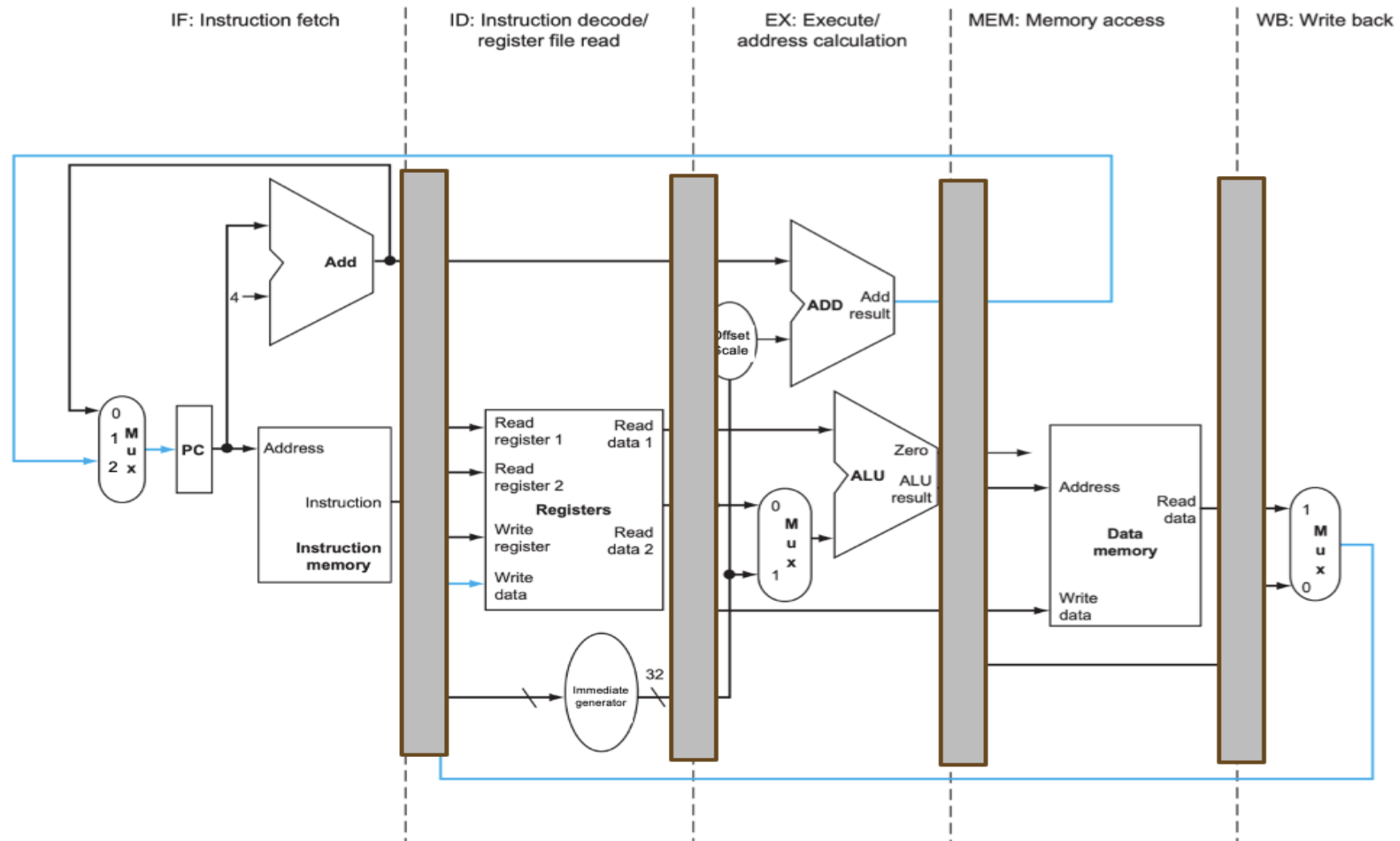
or x8, x9, x10 add x5, x3, x4

Cycle 2



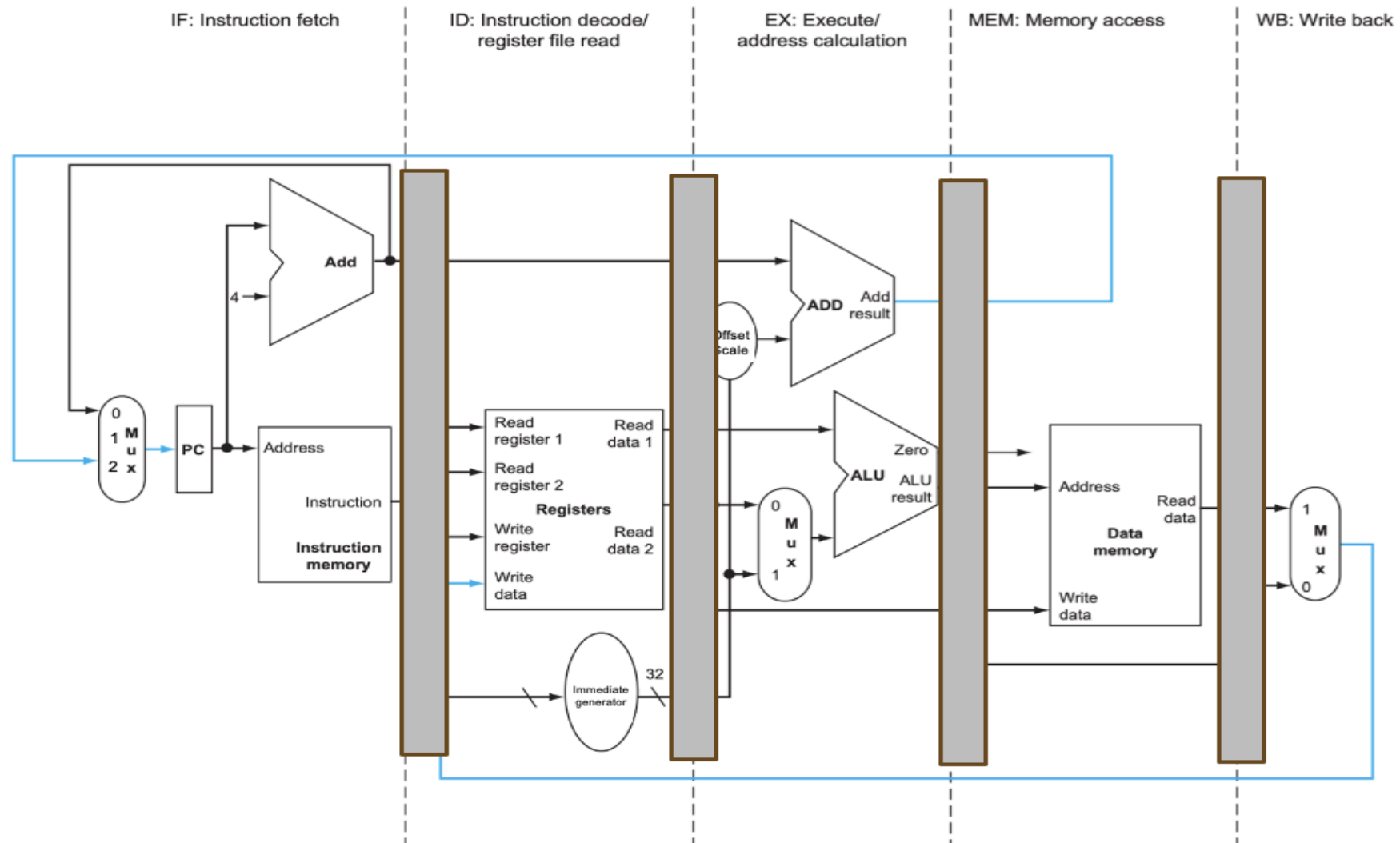
sub x7, x5, x2 or x8, x9, x10 add x5, x3, x4

Cycle 3



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

Cycle 4



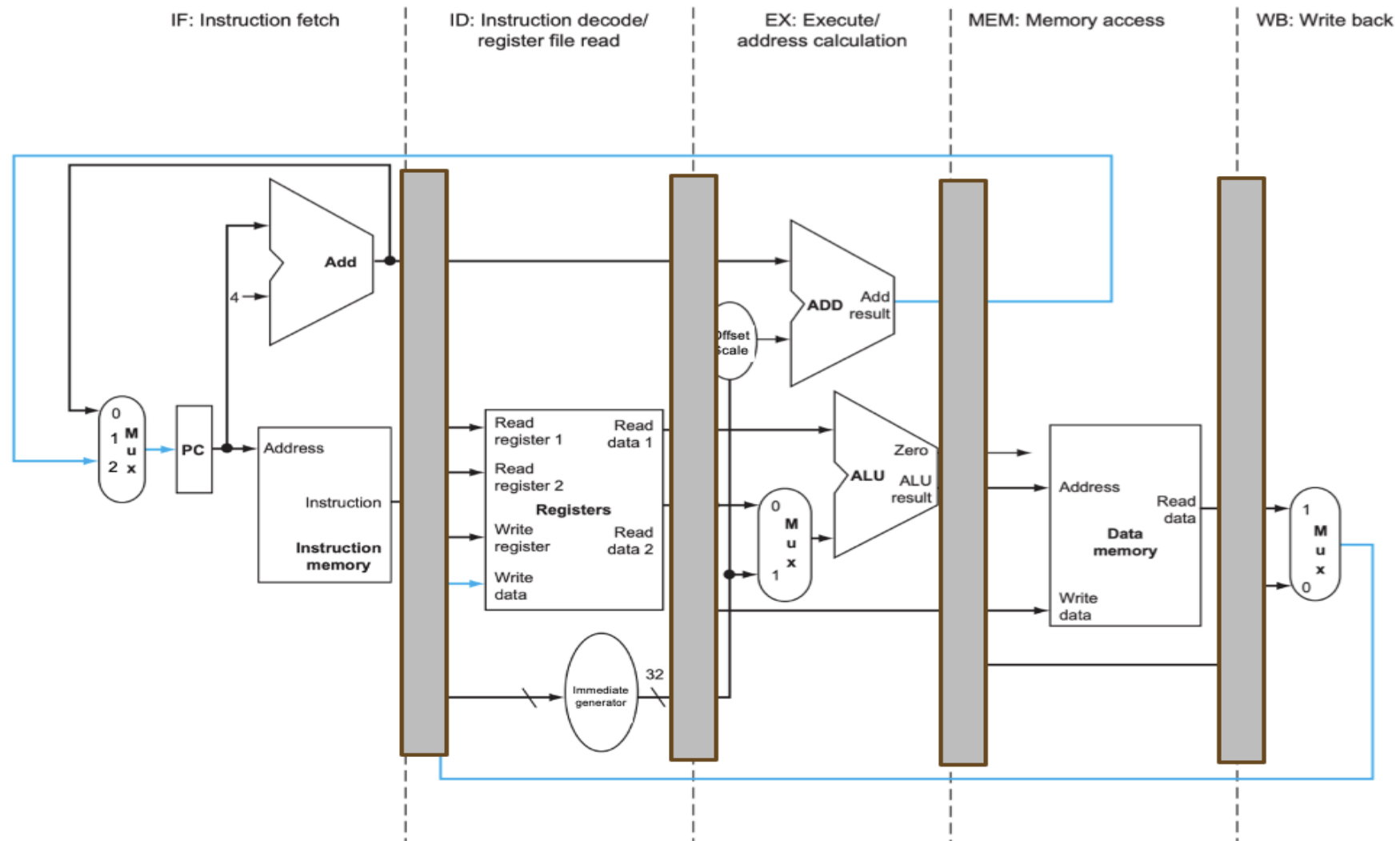
sub x7, x5, x2

STALL

or x8, x9, x10

add x5, x3, x4

Cycle 5

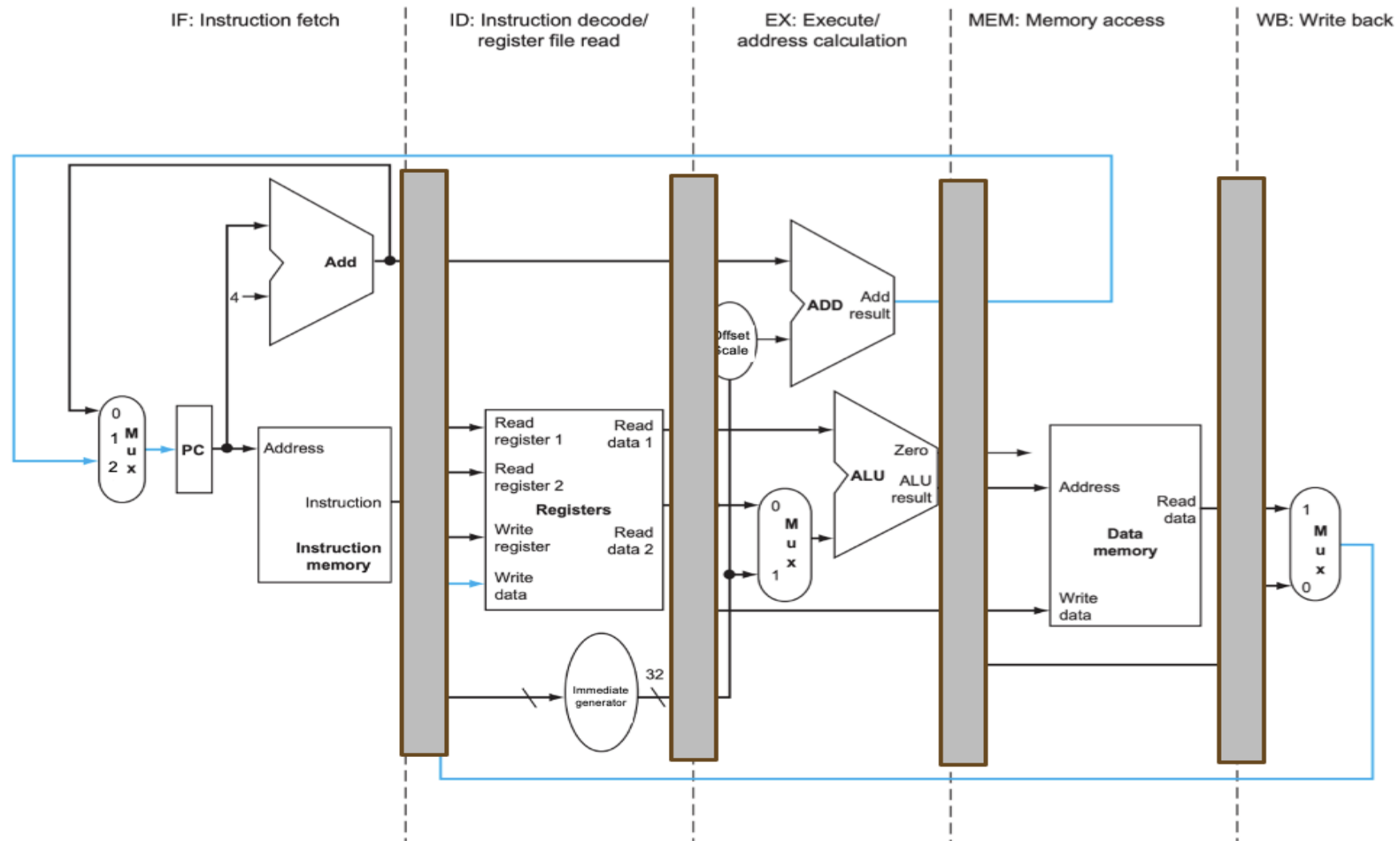


sub x7, x5, x2

STALL

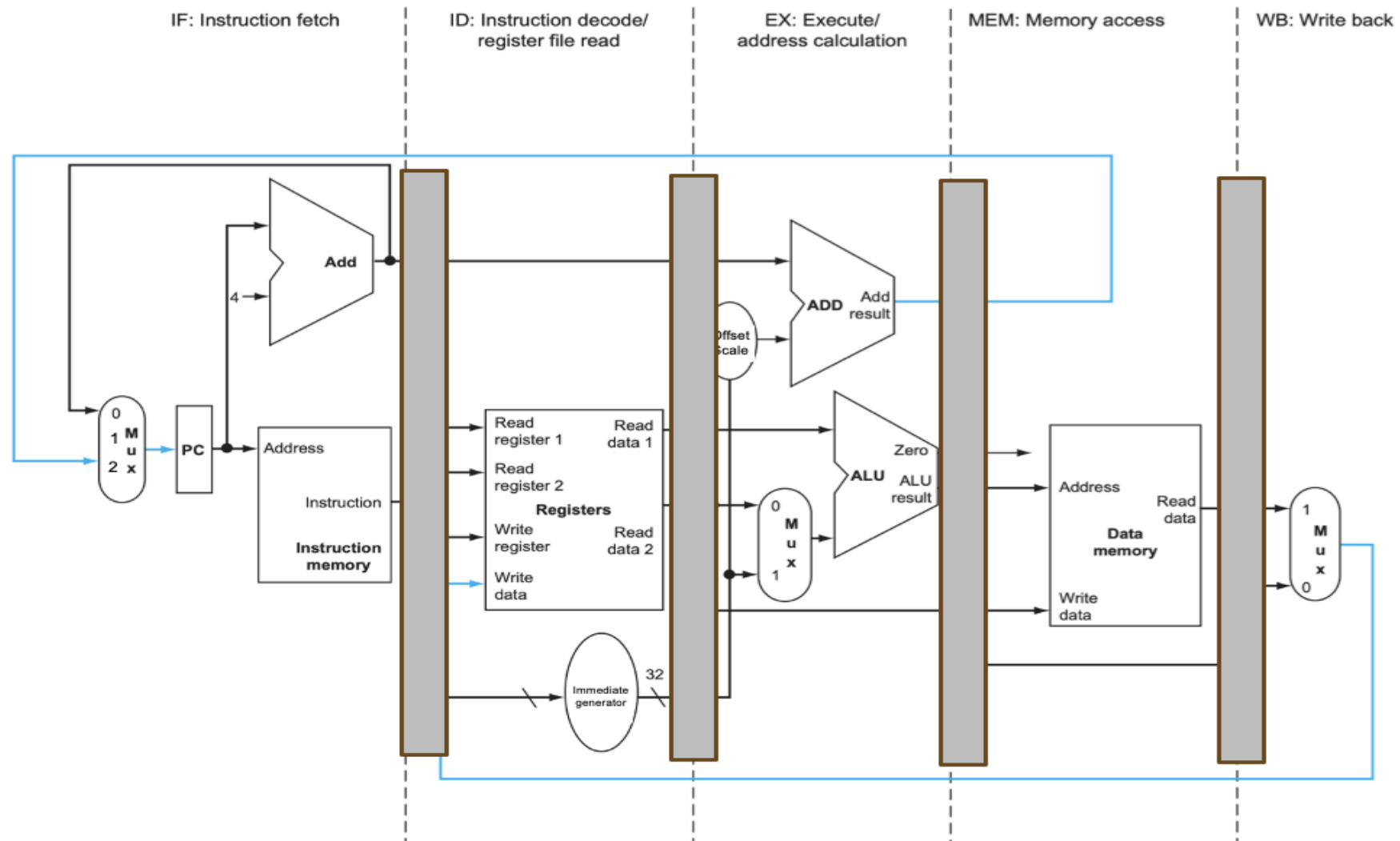
or x8, x9, x10

Cycle 6



sub x7, x5, x2 **STALL**

Cycle 7



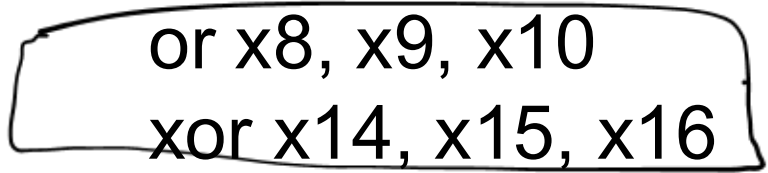
sub x7, x5, x2

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
```

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

No!

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															
sub x7, x5, x2															

no stalls,

no hazards

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?

yes

2

add x20, x21, x22

sub x28, x20, x15

and x27, x5, x18

1 → or x9, x28, x3

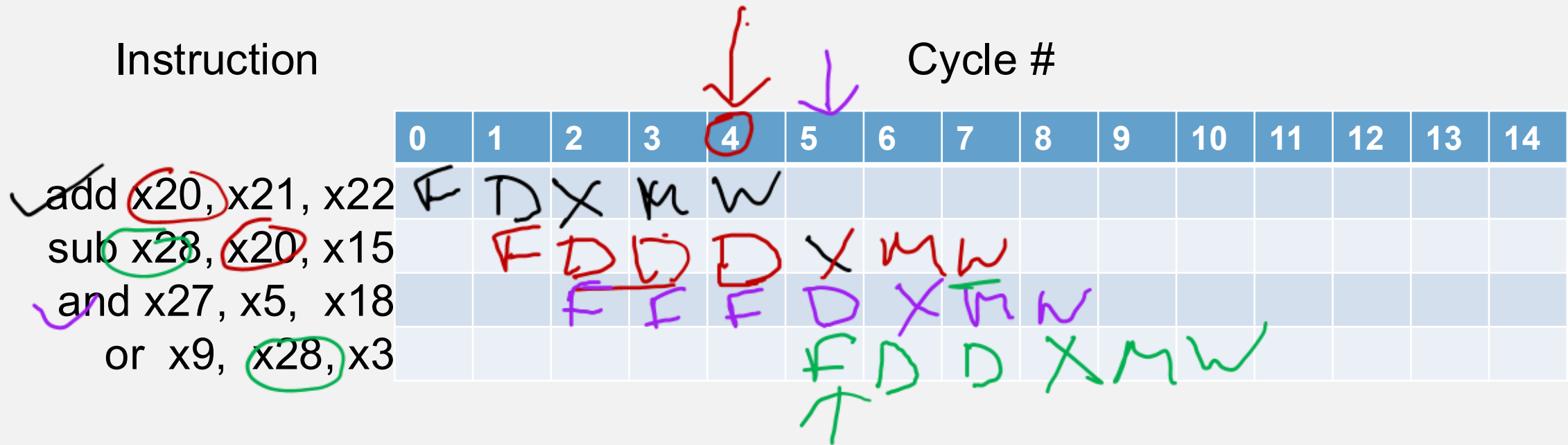
Would we need to add any stalls for this program?

Data Hazard!

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

Yes! We need to add two stalls after the **add** instruction!
And one stall after the **and** instruction!

Pipeline Diagram



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 #

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

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F	D	X	M	W										
	F	D	D	D	X	M	W							
		F	F	F	D	X	M	W						
					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)

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

Performance Metric: CPI (Cycles per instruction)

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

	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				

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

Performance Metric: CPI (Cycles per instruction)

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

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



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F	D	X	M	W										
	F	D	D	D	X	M	W							
		F	F	F	D	X	M	W						
					F	D	D	X	M	W				

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

Ideal CPI (Cycles per instruction) = 1

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

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).

$$CPI = \frac{\# \text{ of cycles}}{\# \text{ of instructions}} = \frac{4 + \# \text{ of instructions}}{\# \text{ of instructions}} \quad \text{(this equation assumes no stalls)}$$

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

add x3, x2, x2
→ or x4, x2, x5
and x9, x4, x10
sub x11, x4, x14
add x11, x15, x18

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Pipeline Diagram with Stalls



add x3, x2, x2
or x4, x2, x5
and x9, x4, x10
sub x11, x4, x14
add x11, x15, x18

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F	D	X	M	W										
	F	D	X	M	W									
		F	D	D	D	X	M	W						
			F	F	F	D	X	M	W					
						F	D	X	M	W				

$$CPI = \frac{11}{5} = 2.2$$

$$\text{Speedup} = \frac{5 \cdot 820ps}{11 \cdot 220ps} = \frac{4100ps}{2420ps} = 1.69$$