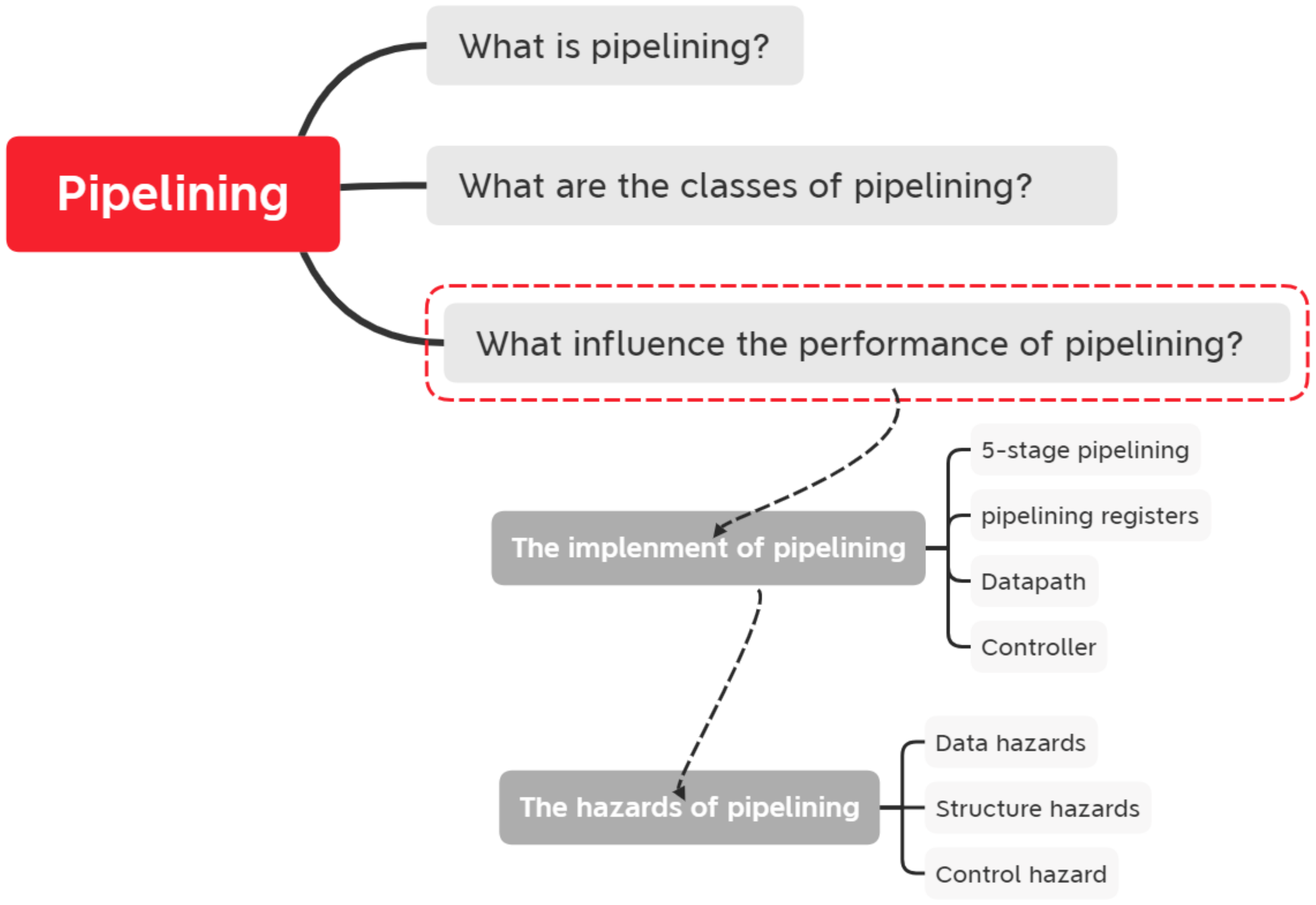
Computer Systems II

Li Lu
Room 605, CaoGuangbiao Building
li.lu@zju.edu.cn
https://person.zju.edu.cn/lynnluli

§ 3.1 An Overview of Pipelining



Pipelining
What is pipelining?
What are the classes of pipelining?
What influence the performance of pipelining?
The implenment of pipelining
5-stage pipelining
pipelining registers
Datapath
Controller
The hazards of pipelining
Data hazards
Structure hazards
Control hazard

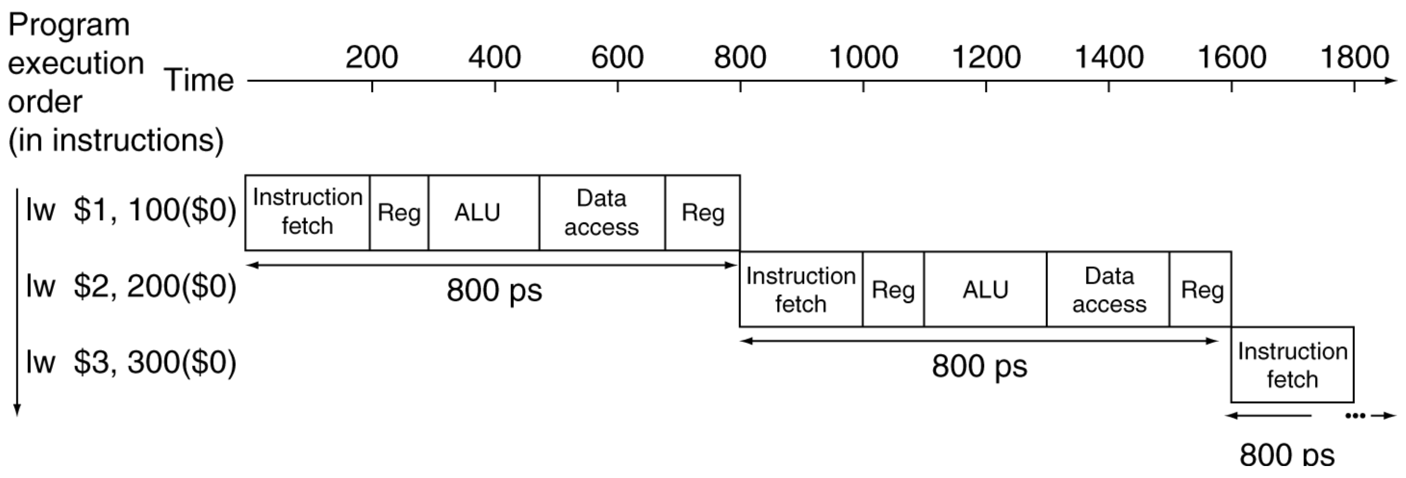
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

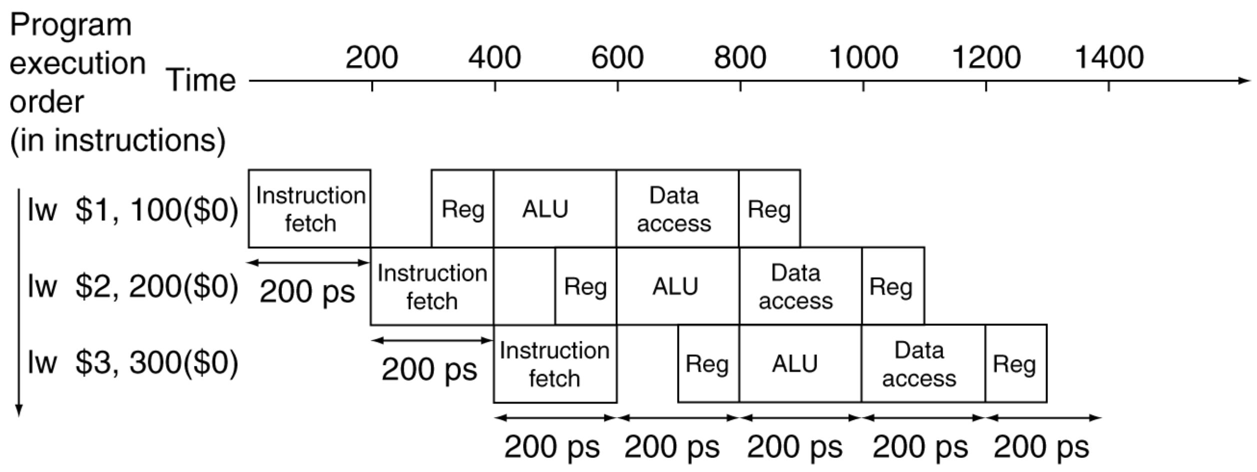
Inst	Inst fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-type	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)



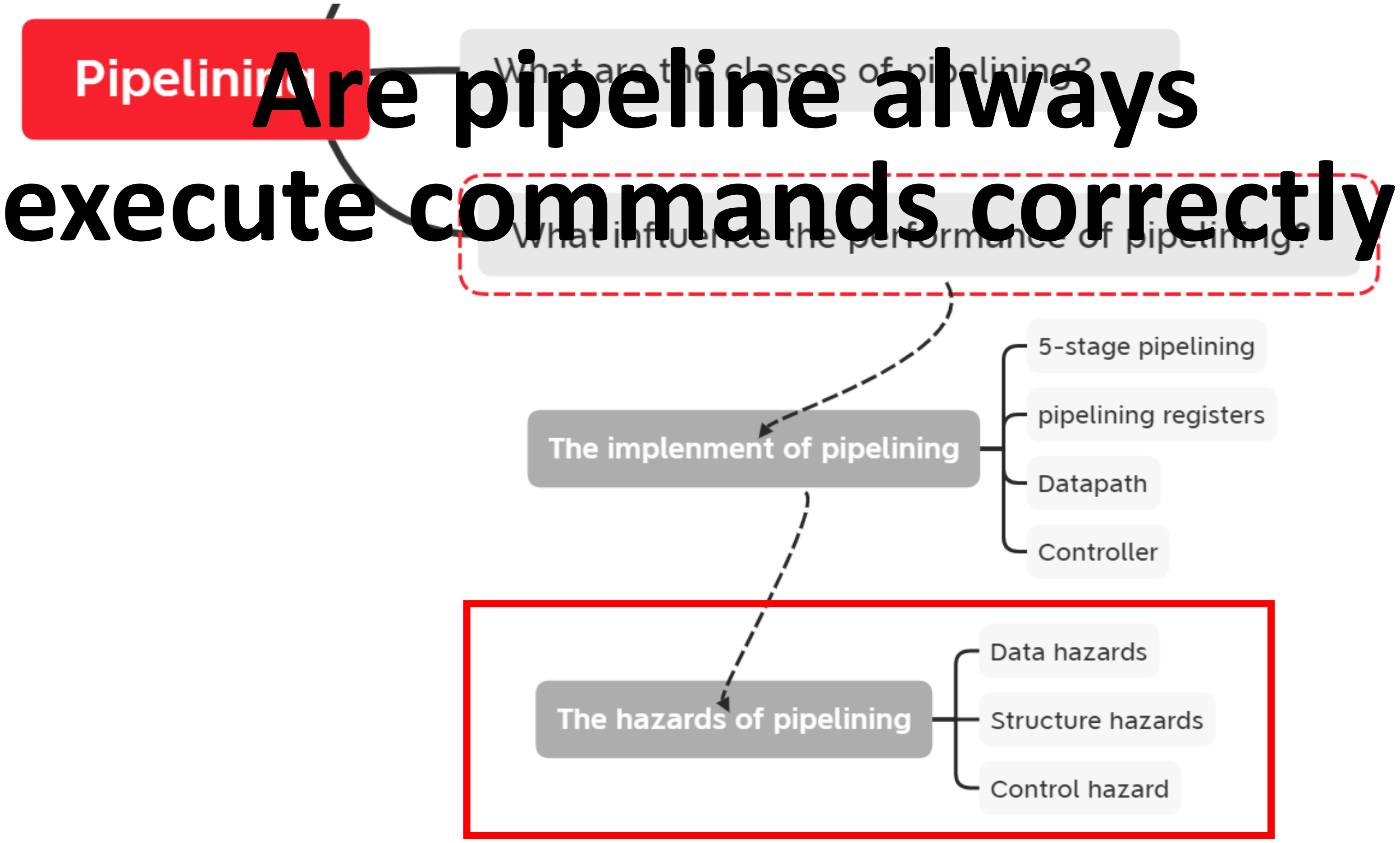
Pipelined (T_c= 200ps)



How Pipelining Improves Performance?

Decreasing the execution time of an individual instruction ×

Increasing instruction throughput √


Pipelining
What are the classes of pipelining?
What influence the performance of pipelining?
The implenment of pipelining
5-stage pipelining
pipelining registers
Datapath
Controller
The hazards of pipelining
Data hazards
Structure hazards
Control hazard

Are pipeline always execute commands correctly?

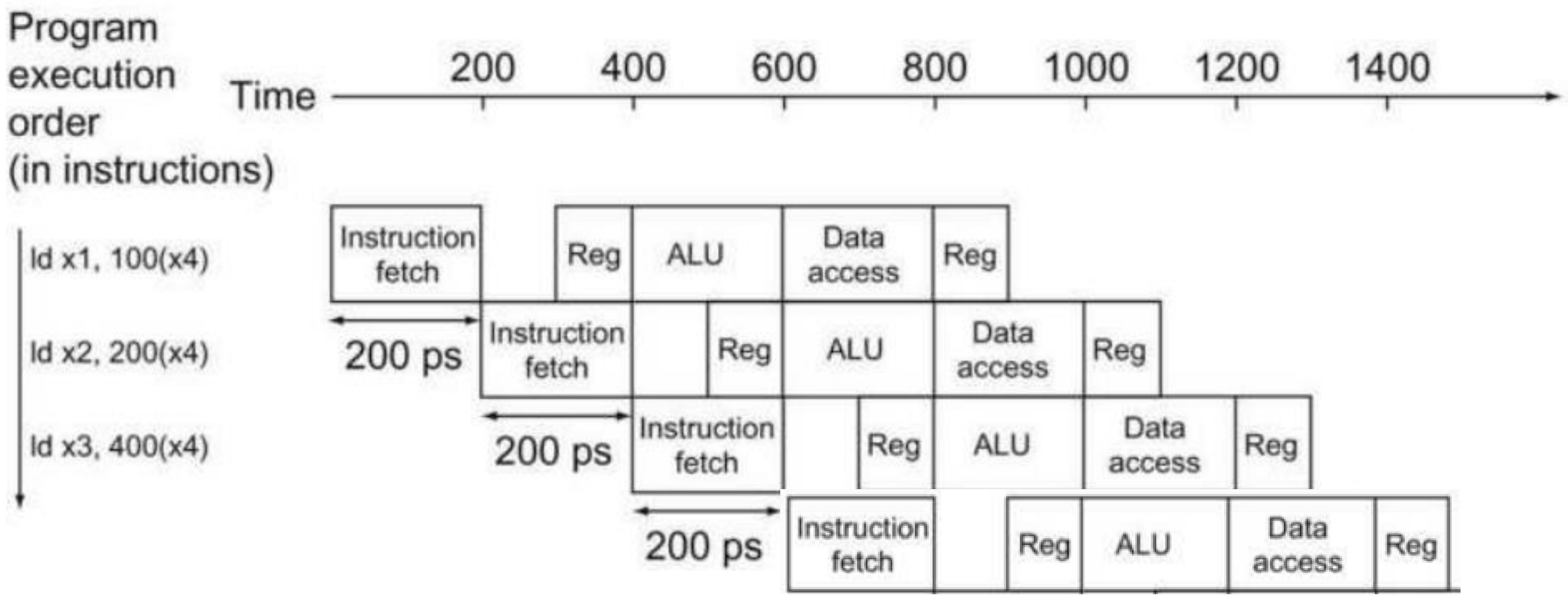
Pipeline Hazards

• Structural Hazard	A required resource is busy
• Data Hazards	• Data dependency between instructions • Need to wait for previous instruction to complete its data read/write
• Control Hazards	Flow of execution depends on previous instruction

Structural Hazard

A required resource is busy

Example: Consider the situation while the pipeline only has a single memory



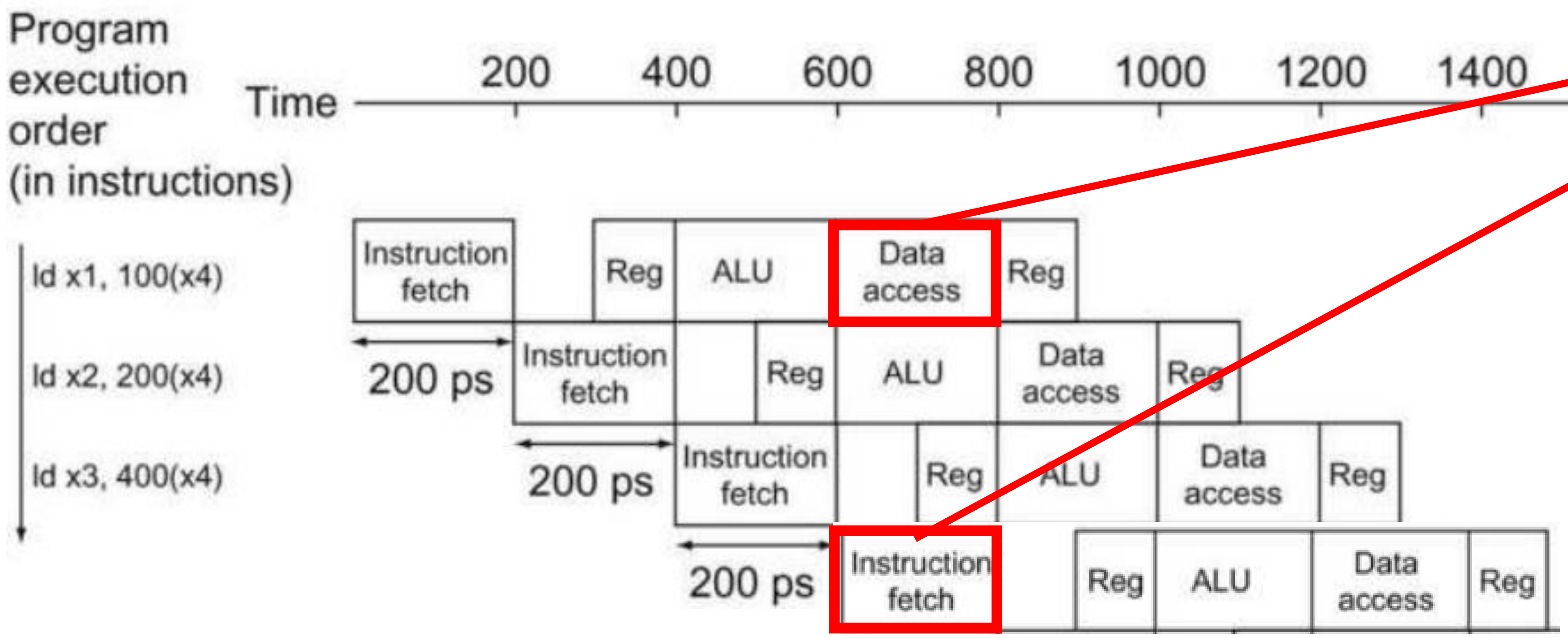
Question: Can the four instructions execute correctly?

Structural Hazard

A required resource is busy

Solution:
Use Instruction and data memory simultaneously

Example: Consider the situation while the pipeline only has a single memory



How to Deal with Structural Hazard?

Problem: Two or more instructions in the pipeline compete for access to a single physical resource

- Solution 1: Instructions take it in turns to use resource, some instructions have to stall

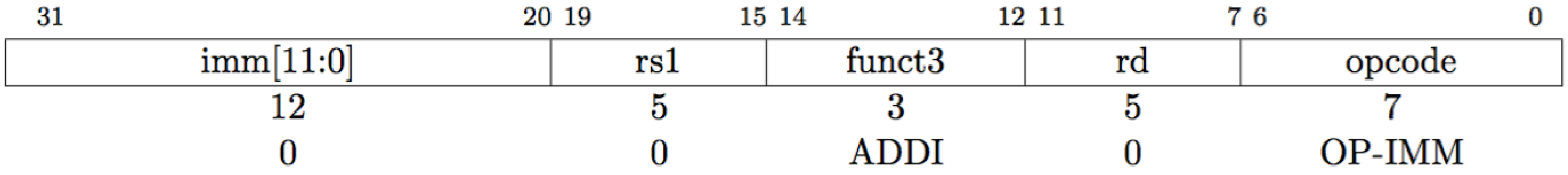
- Solution 2: Add more hardware to machine

Can always solve a structural hazard by adding more hardware

How to Stall ?

NOP instruction

ADDI x0, x0, 0



31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	ADDI	0	OP-IMM

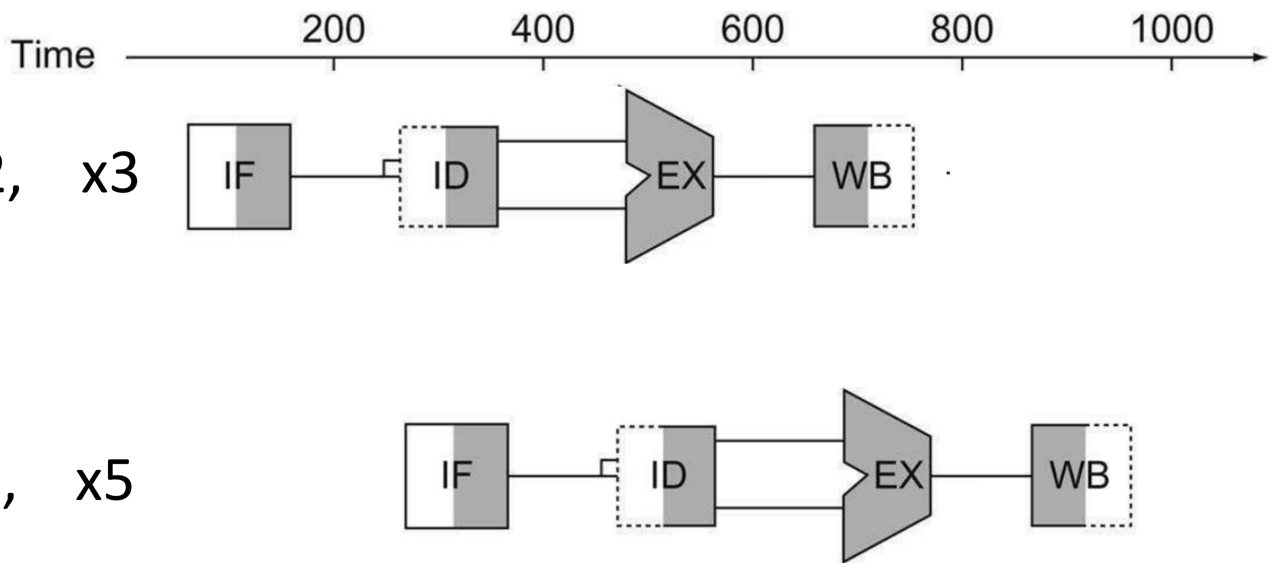
Data Hazard

- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write

Problem: Instruction depends on result from previous

add x1, x2, x3

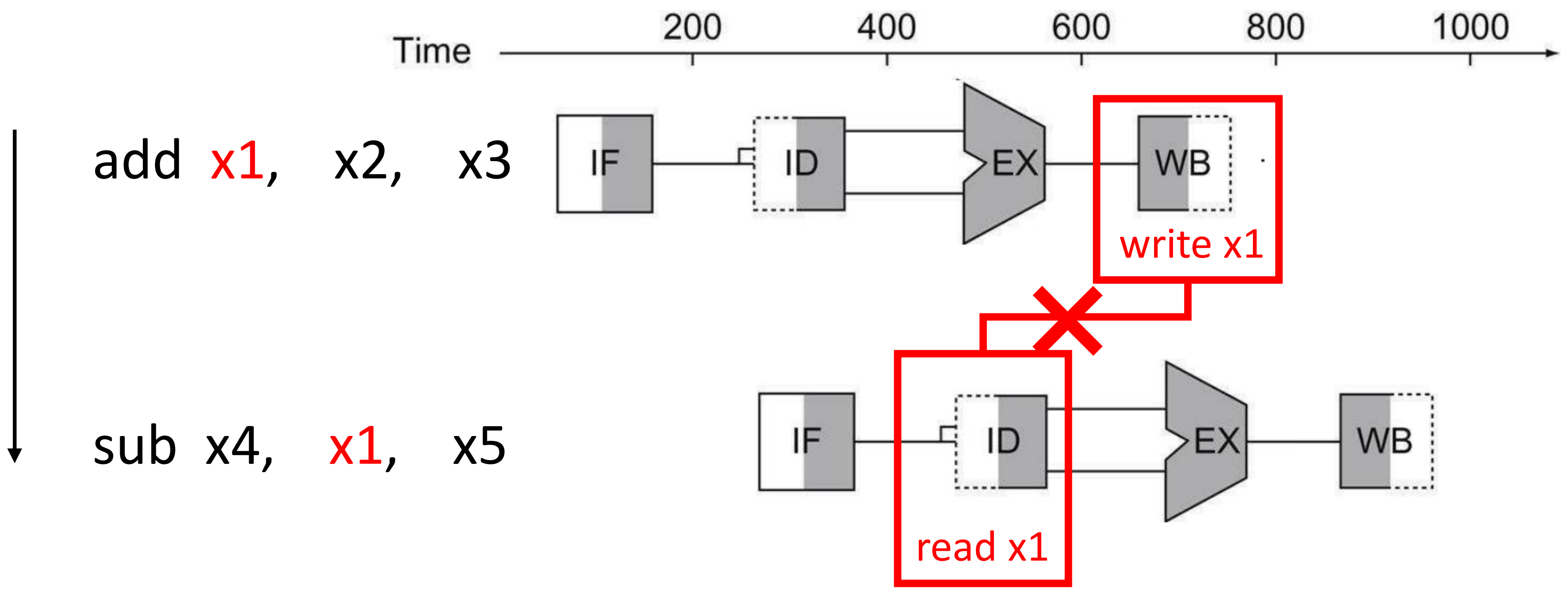
sub x4, x1, x5



Data Hazard

- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write

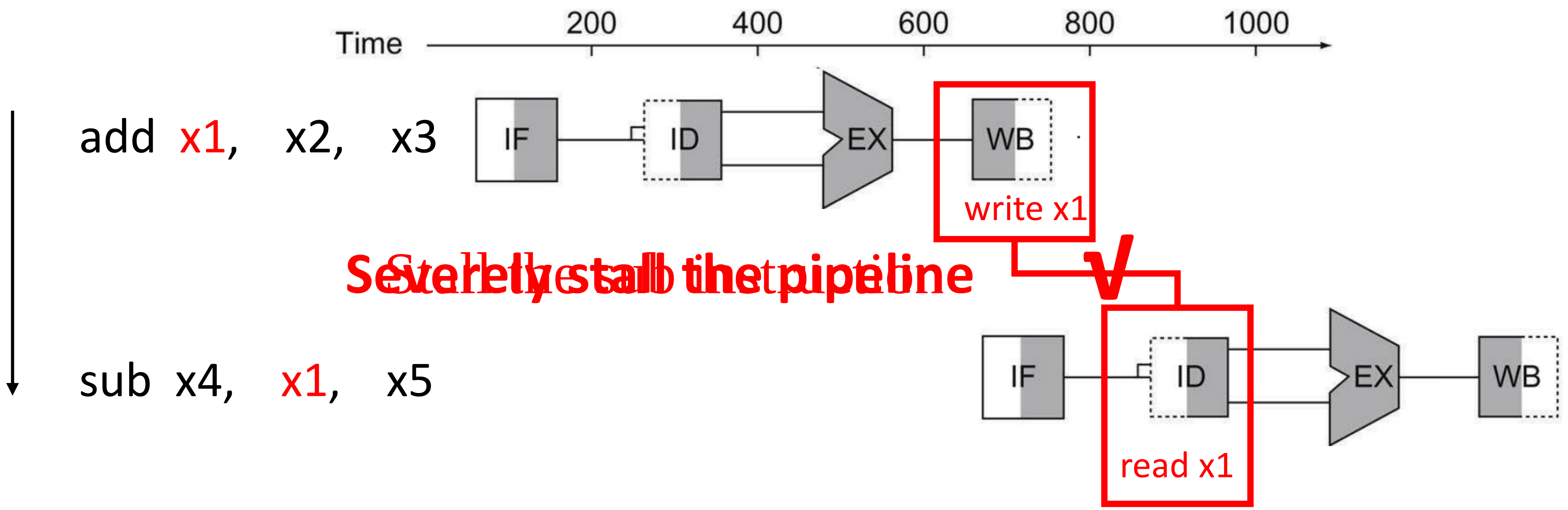
Problem: Instruction depends on result from previous



Data Hazard

- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write

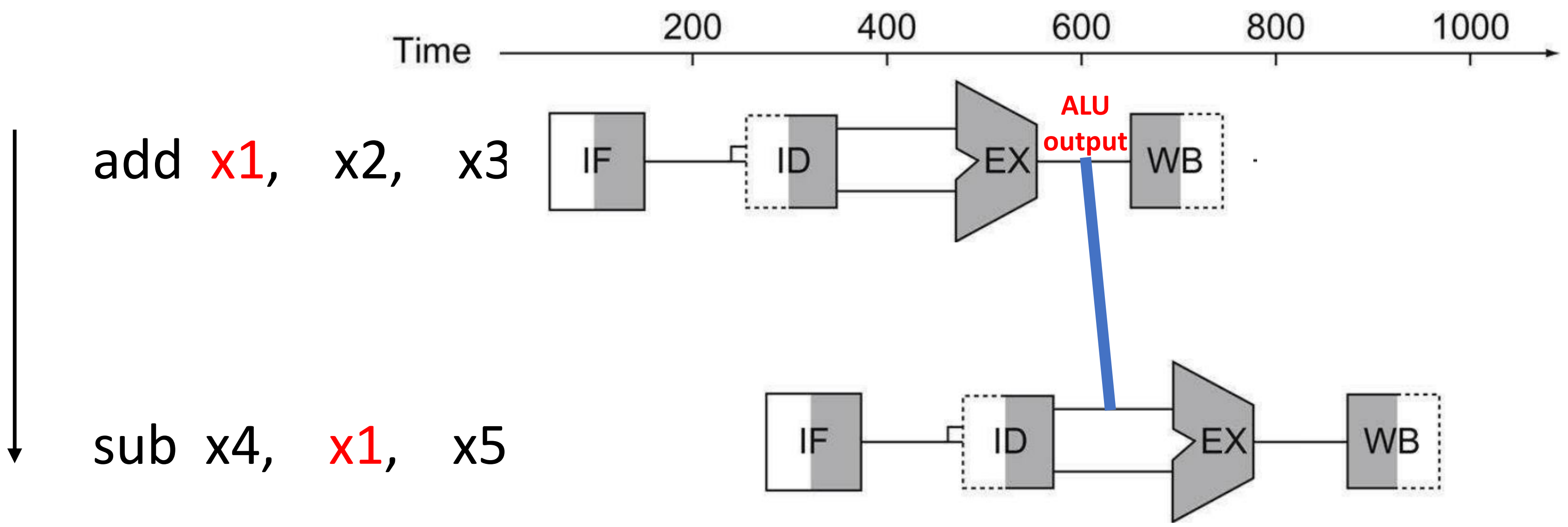
Problem: Instruction depends on result from previous



Data Hazard

- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write

Solution "forwarding": Adding extra hardware to retrieve the missing item early from the internal resources



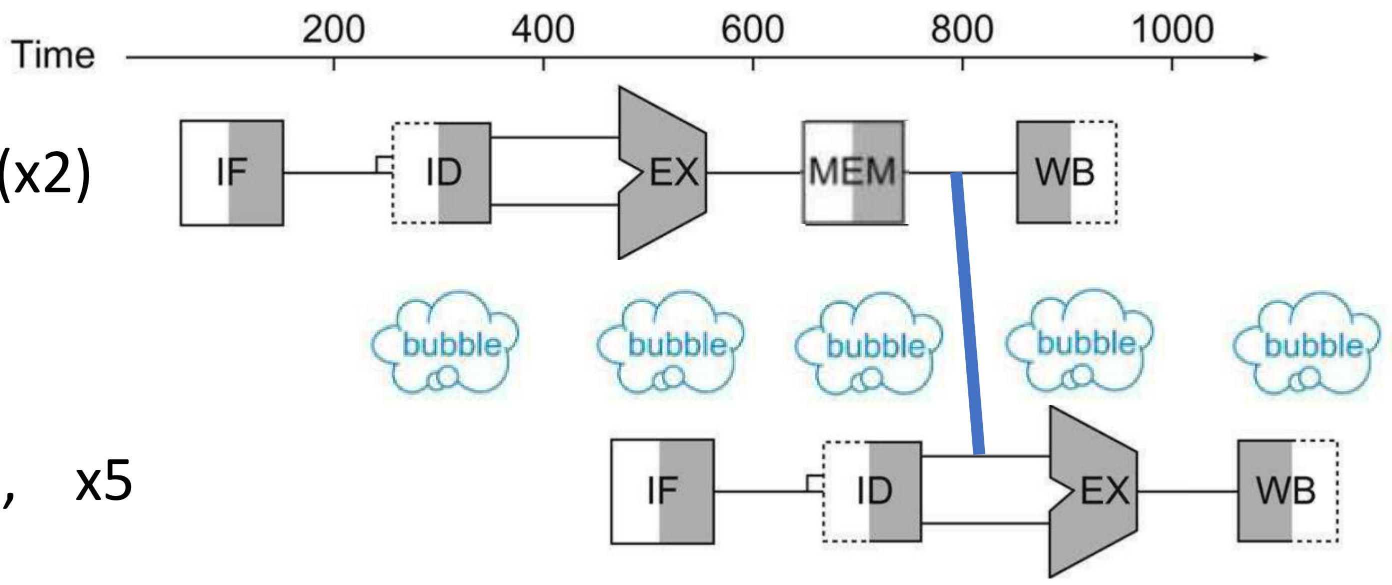
Data Hazard

- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write

Solution "forwarding": Could not avoid all pipeline stalls

load x1, 0(x2)

sub x4, x1, x5



Data Hazard

- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write

Example: Reordering code to avoid pipeline stalls

Consider the following code segment in C:

```
a = b + e;
c = b + f;
```

- Assuming all variables are in memory
- and are addressable as offsets from x31

The generated RISC-V code:

```
ld  x1, 0(x31) // Load b
ld  x2, 8(x31) // Load e
add   x3, x1, x2 // b + e
sd  x3, 24(x31) // Store a
ld  x4, 16(x31) // Load f
add   x5, x1, x4 // b + f
sd  x5, 32(x31) // Store c
```

Question: Find the data hazard and reorder RISC-V code

Data Hazard

- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write

Example: Reordering code to avoid pipeline stalls

Consider the following code segment in C:

```
a = b + e;
c = b + f;
```

- Assuming all variables are in memory
- and are addressable as offsets from x31

The generated RISC-V code:

```
ld  x1, 0(x31) // Load b
ld  x2, 8(x31) // Load e
add   x3, x1, x2 // b + e
sd  x3, 24(x31) // Store a
ld  x4, 16(x31) // Load f
add   x5, x1, x4 // b + f
sd  x5, 32(x31) // Store c
```

Data Hazard

- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write

Example: Reordering code to avoid pipeline stalls

Consider the following code segment in C:

```
a = b + e;
c = b + f;
```

- Assuming all variables are in memory
- and are addressable as offsets from x31

The generated RISC-V code:

```
ld  x1, 0(x31) // Load b
ld  x2, 8(x31) // Load e
add   x3, x1, x2 // b + e
sd  x3, 24(x31) // Store a
ld  x4, 16(x31) // Load f
add   x5, x1, x4 // b + f
sd  x5, 32(x31) // Store c
```

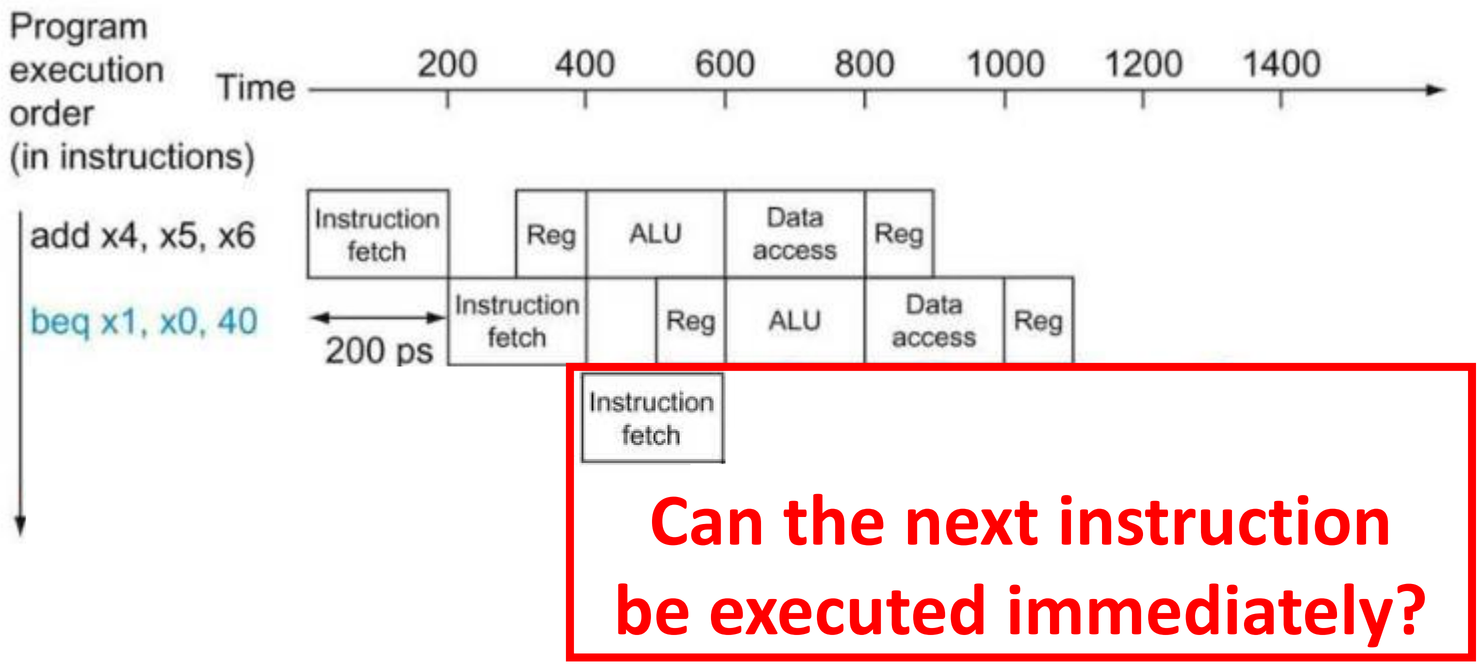
eliminates both hazards

```
ld  x1, 0(x31)
ld  x2, 8(x31)
ld  x4, 16(x31)
add   x3, x1, x2
sd  x3, 24(x31)
add   x5, x1, x4
sd  x5, 32(x31)
```

Control Hazard

Flow of execution depends on previous instruction

Problem: The conditional branch instruction

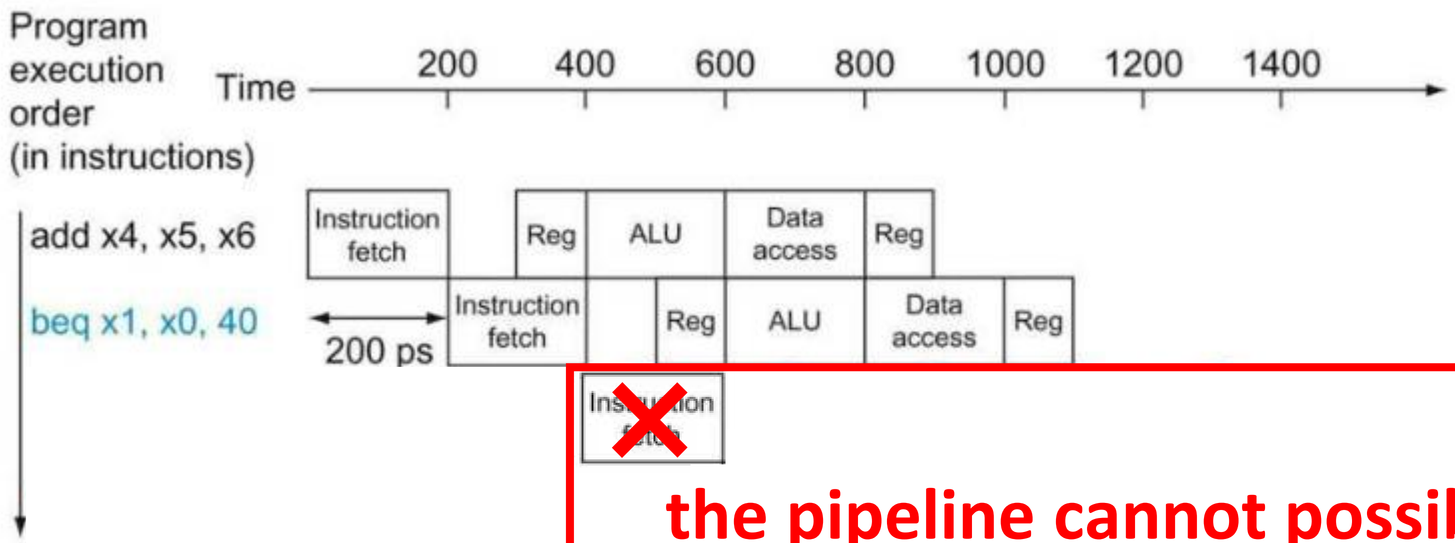


Can the next instruction be executed immediately?

Control Hazard

Flow of execution depends on previous instruction

Problem: The conditional branch instruction

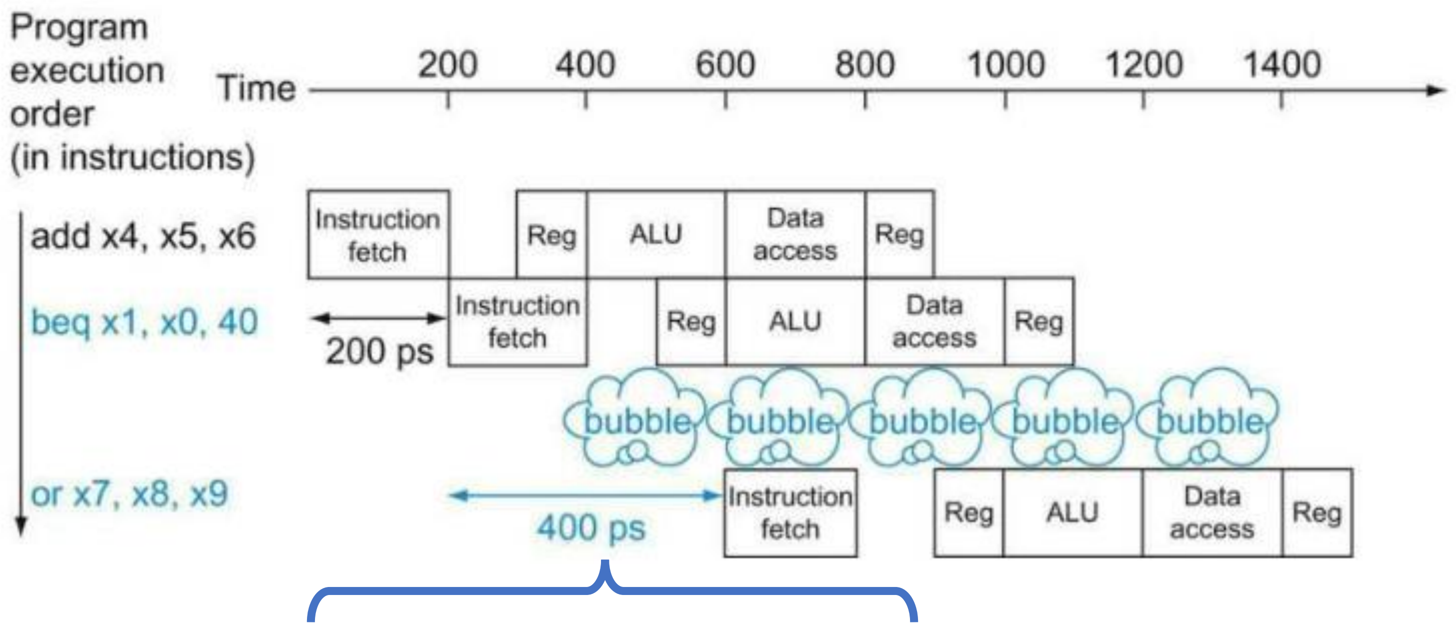


the pipeline cannot possibly know what the next instruction should be

Control Hazard

Flow of execution depends on previous instruction

Solution 1: Stall



- test a register
- calculate the branch address
- update the PC

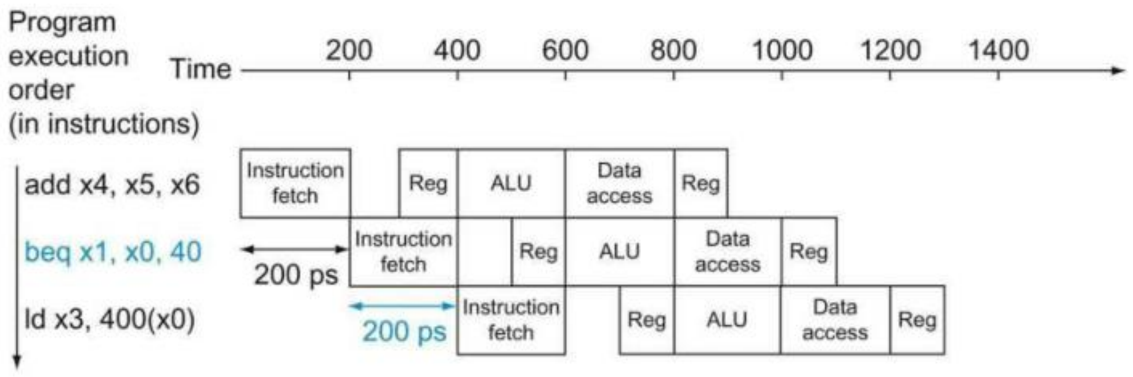
Control Hazard

Flow of execution depends on previous instruction

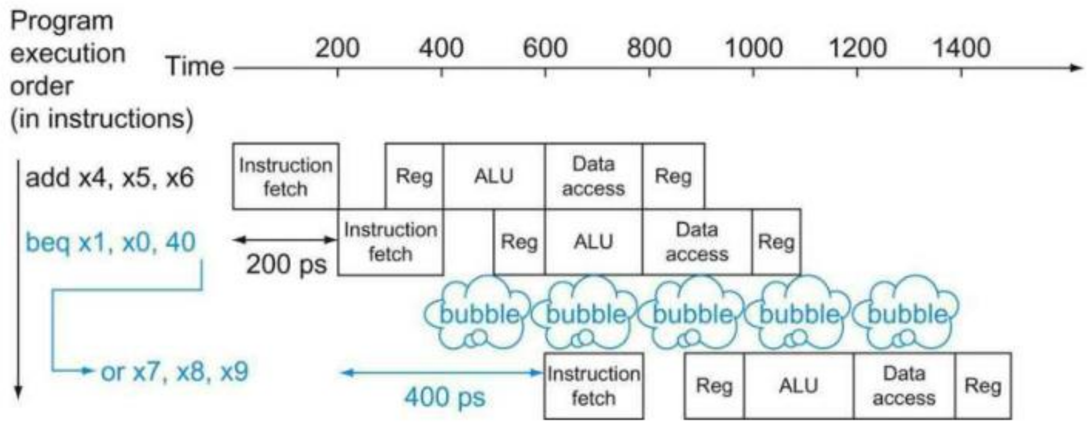
Solution 2: Prediction (simple version)

Predicts **always** that conditional branches will be untaken

branch is not taken



branch is taken



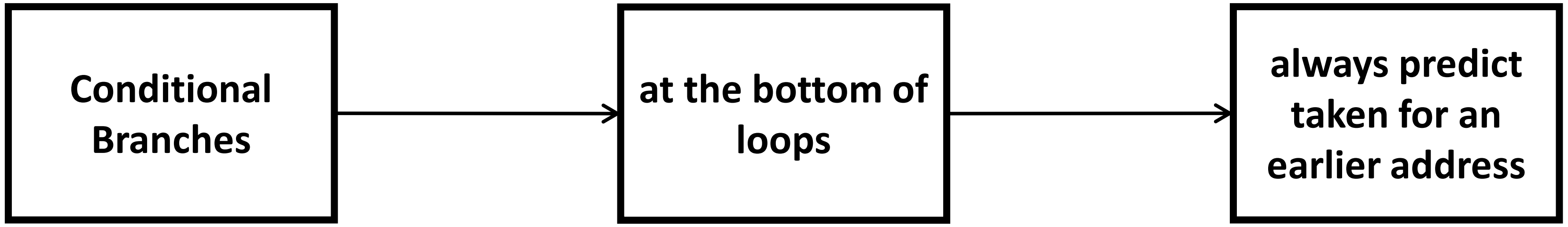
Control Hazard

Flow of execution depends on previous instruction

Solution 2: Prediction (sophisticated version)

Predicts that **some** conditional branches will be untaken

Example: While the Conditional Branches at the bottom of loops



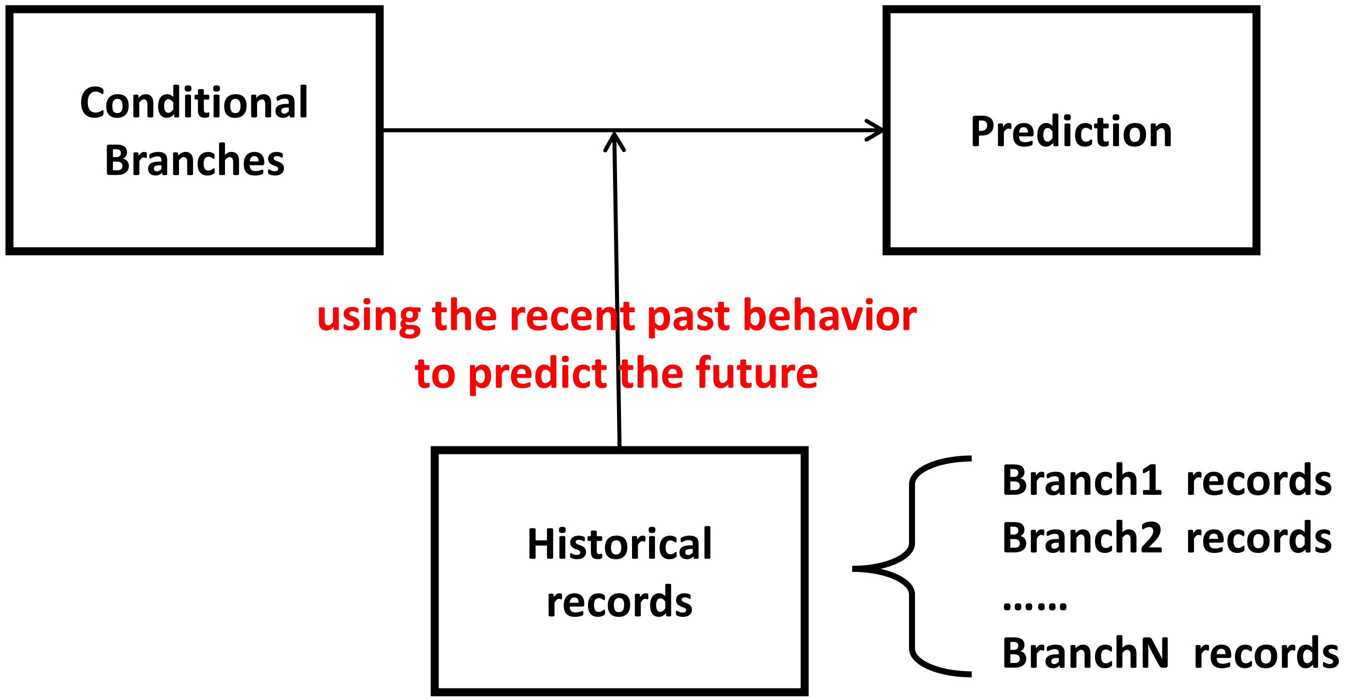
they are likely to be taken to the top of the loops

Control Hazard

Flow of execution depends on previous instruction

Solution 2: Prediction (dynamic prediction)

Predicts that **some** conditional branches will be untaken

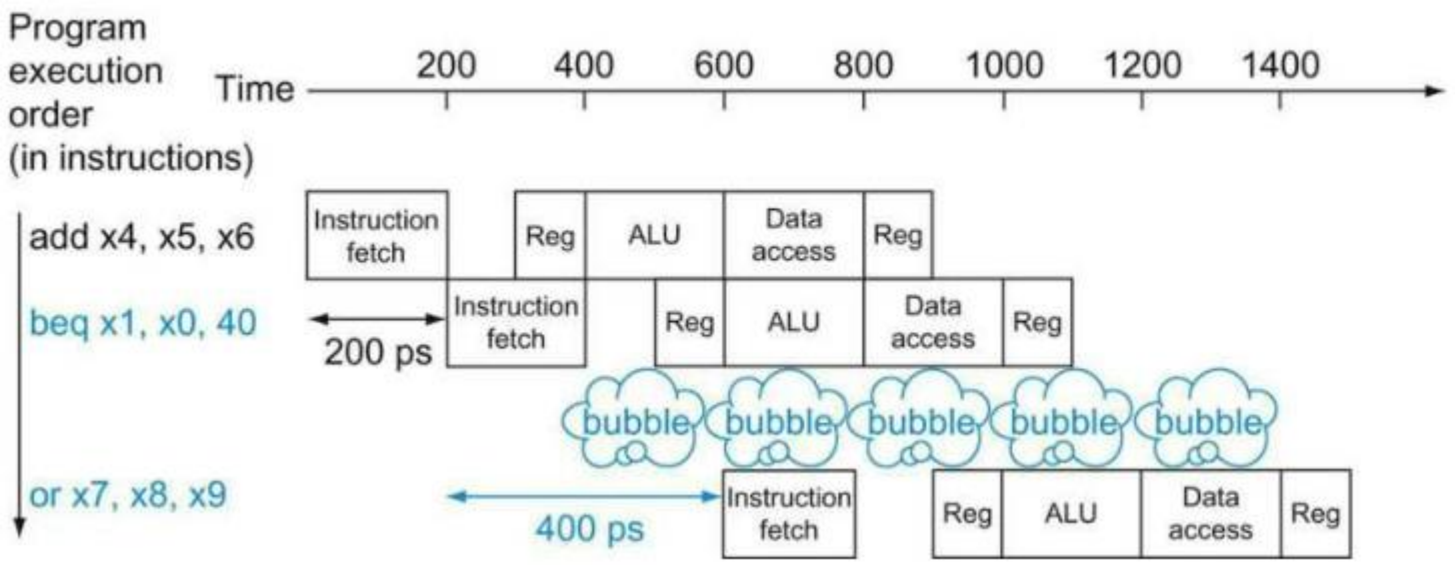


Control Hazard

Flow of execution depends on previous instruction

Solution 3: Delayed Decision

Places an instruction immediately after the delayed branch instruction that is not affected by the branch

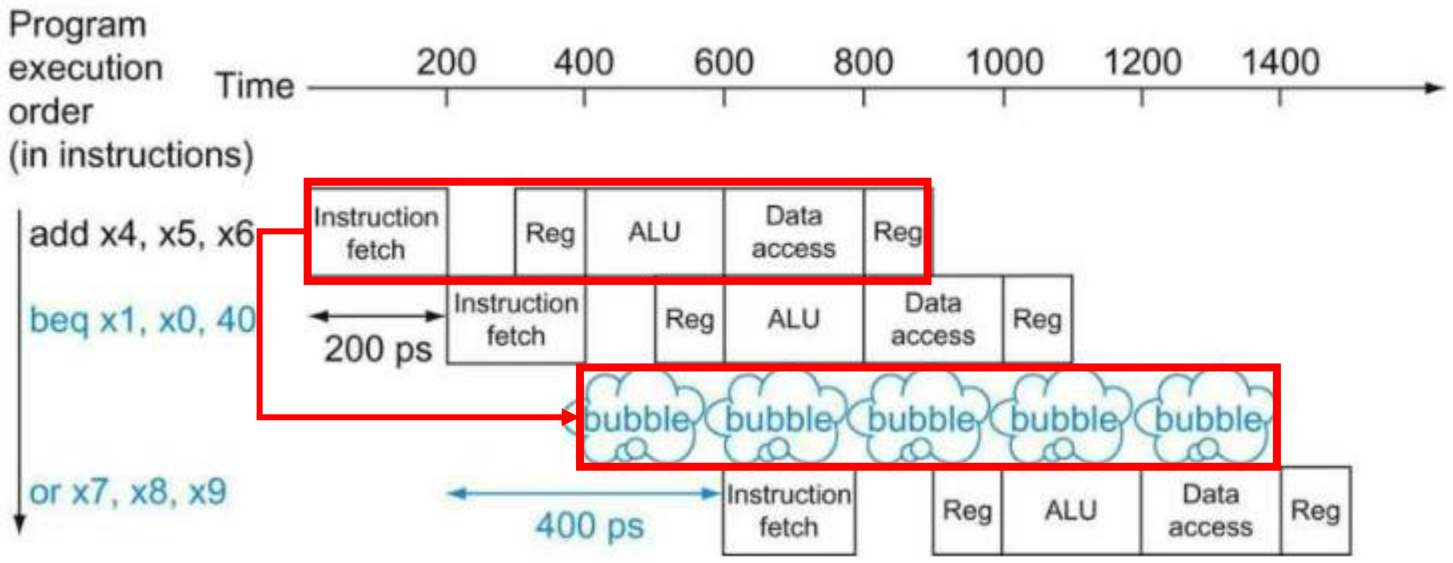


Control Hazard

Flow of execution depends on previous instruction

Solution 3: Delayed Decision

Places an instruction immediately after the delayed branch instruction that is not affected by the branch

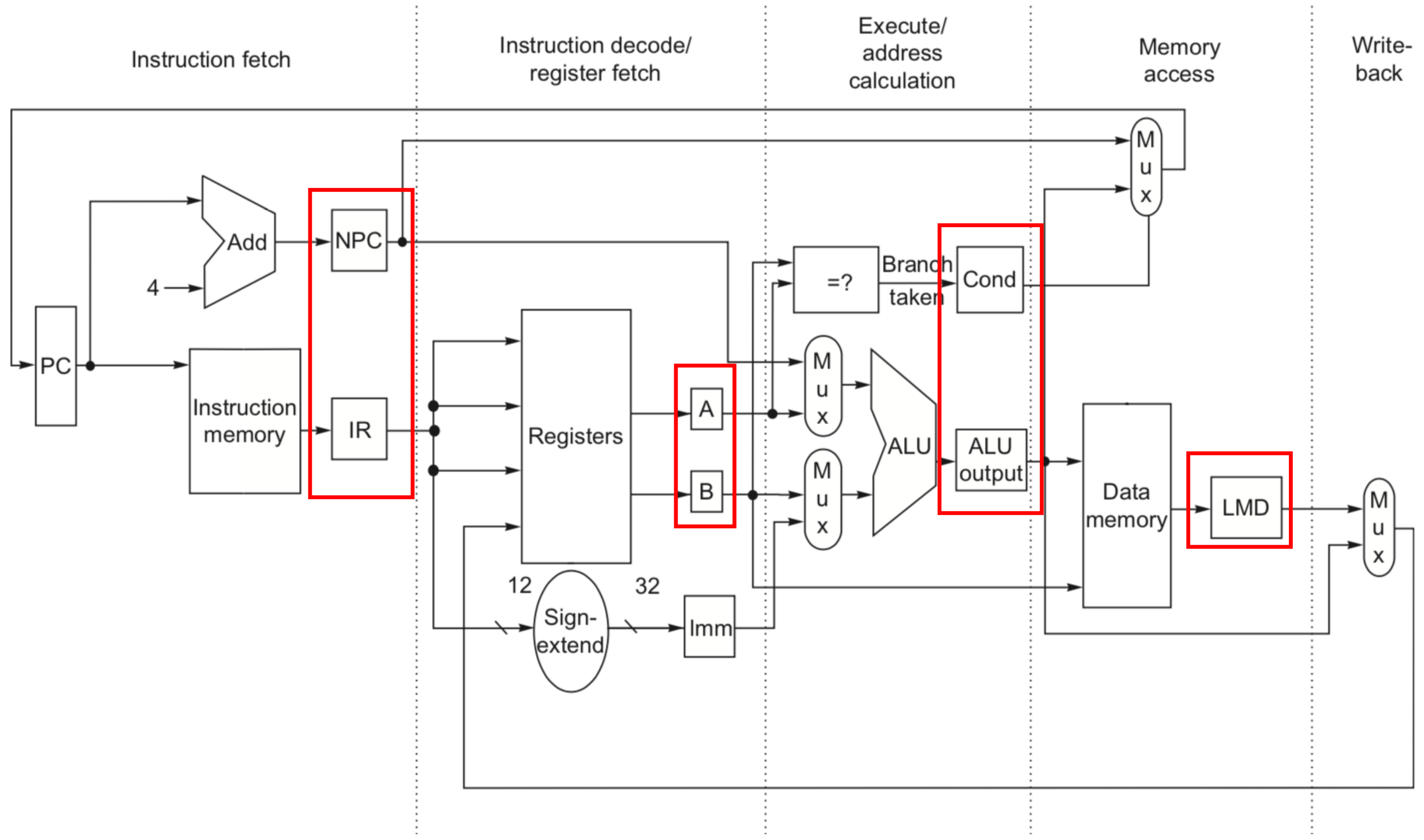


Summary of Pipelining Design

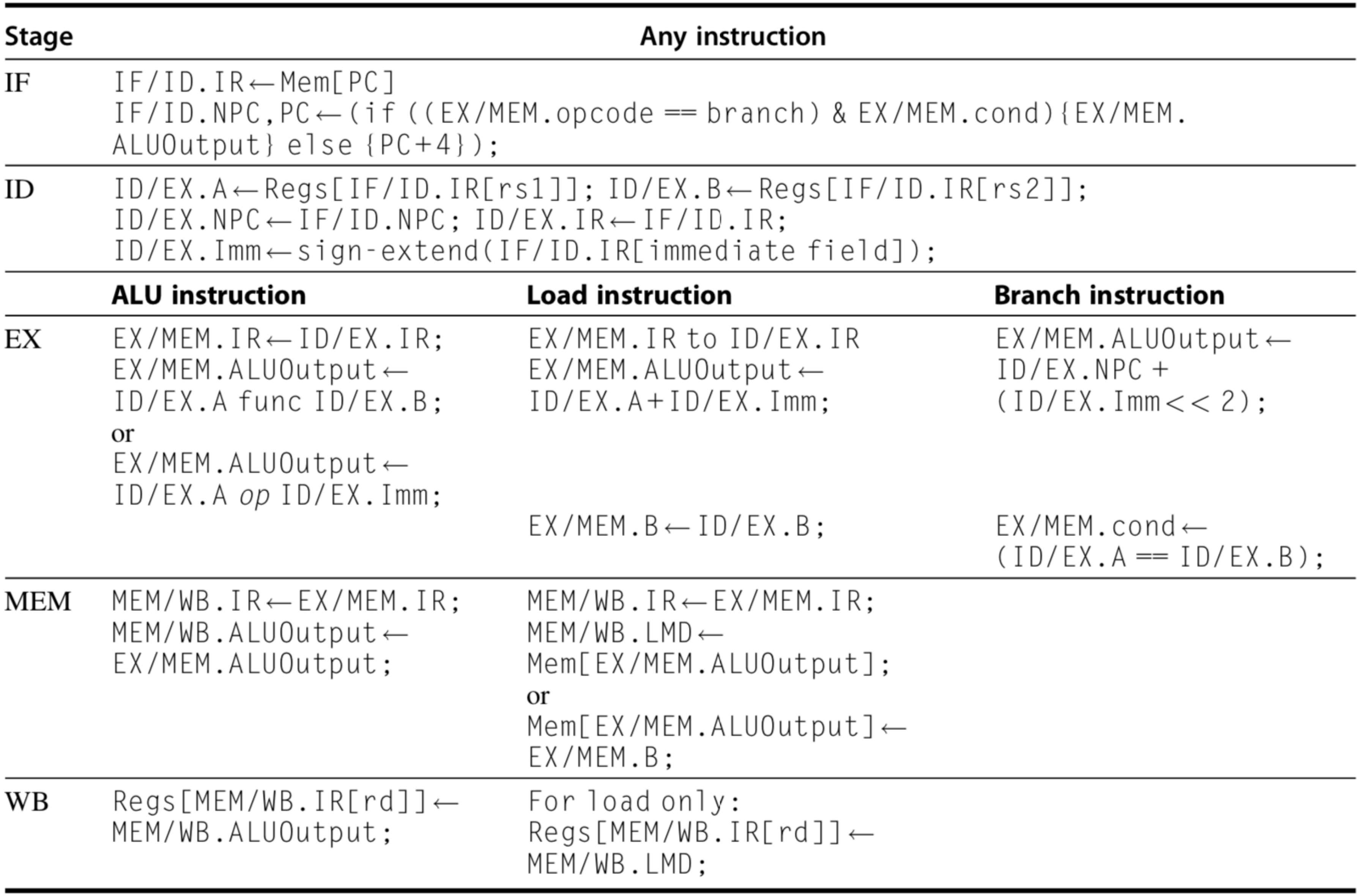
- Pipeline is a technique that exploits parallelism between the instructions in a sequential instruction stream
- Pipeline increases the number of simultaneously executing instructions and the rate at which instructions are started and completed
- Pipeline designers need to cope with structural, control, and data hazards
- Pipeline is fundamentally invisible to the programmer

Pipelining Design and Implementation

- RISC-V ISA designed for pipelining



§3.2.0 An Implementation of Pipelining



Stage	Any instruction		

IF	IF/ID.IR←Mem[PC]
	IF/ID.NPC,PC←(if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM.
	ALUOutput} else {PC+4});		

ID	ID/EX.A←Regs[IF/ID.IR[rs1]]; ID/EX.B←Regs[IF/ID.IR[rs2]];
	ID/EX.NPC←IF/ID.NPC; ID/EX.IR←IF/ID.IR;
	ID/EX.Imm←sign-extend(IF/ID.IR[immediate field]);		

	ALU instruction	**Load instruction**	**Branch instruction**

EX	EX/MEM.IR←ID/EX.IR;
	EX/MEM.ALUOutput←
	ID/EX.A func ID/EX.B;
	or
	EX/MEM.ALUOutput←
	ID/EX.A *op* ID/EX.Imm;
		EX/MEM.IR to ID/EX.IR
		EX/MEM.ALUOutput←
		ID/EX.A+ID/EX.Imm;
		
		EX/MEM.B←ID/EX.B;
			EX/MEM.ALUOutput←
			ID/EX.NPC +
			(ID/EX.Imm<< 2);
			
			EX/MEM.cond←
			(ID/EX.A == ID/EX.B);

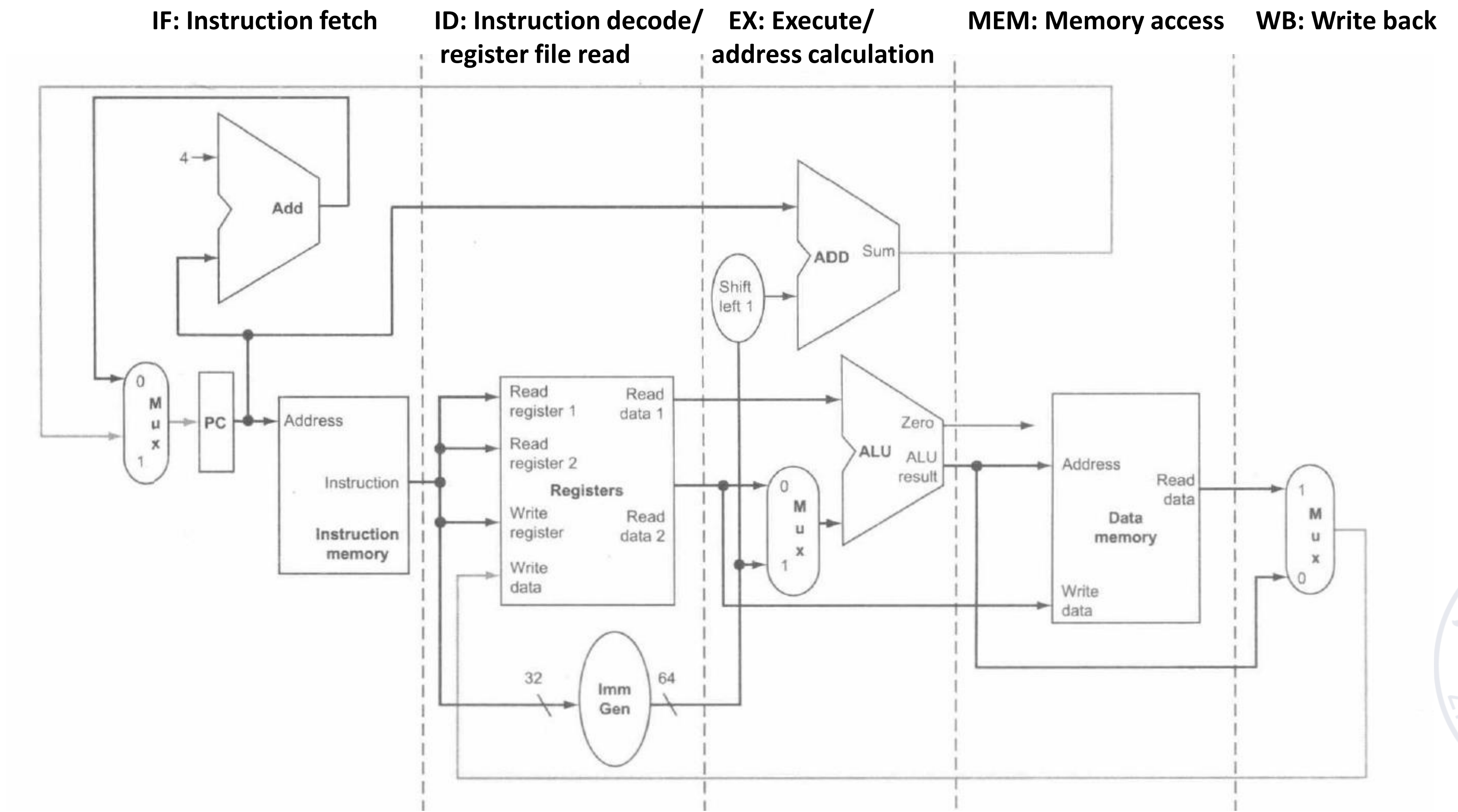
MEM	MEM/WB.IR←EX/MEM.IR;
	MEM/WB.ALUOutput←
	EX/MEM.ALUOutput;
		MEM/WB.IR←EX/MEM.IR;
		MEM/WB.LMD←
		Mem[EX/MEM.ALUOutput];
		or
		Mem[EX/MEM.ALUOutput]←
		EX/MEM.B;	

WB	Regs[MEM/WB.IR[rd]]← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rd]]← MEM/WB.LMD;	

Pipelined Datapath

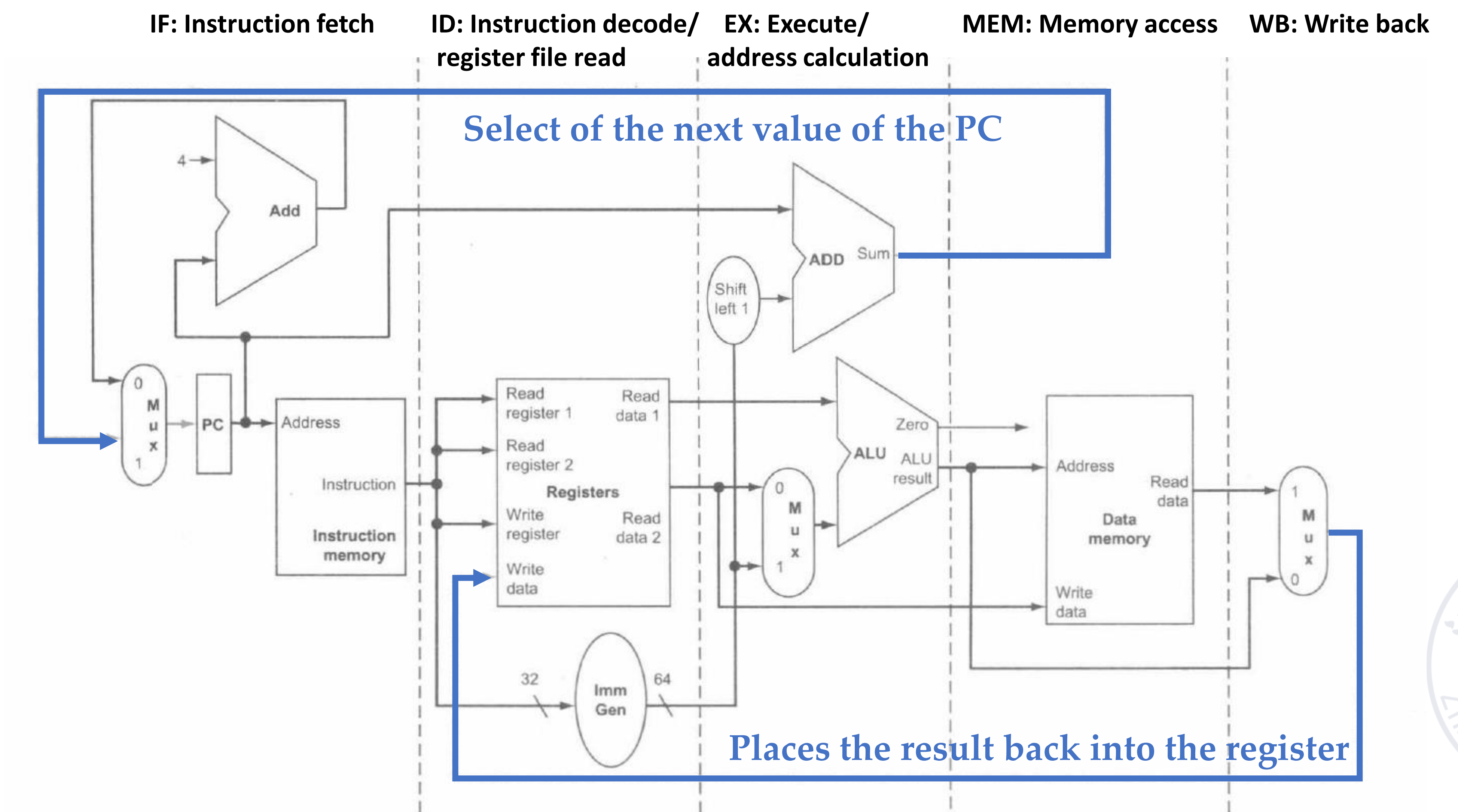
Single-Cycle Datapath

Let's find two exceptions
to this left-to-right flow of instructions



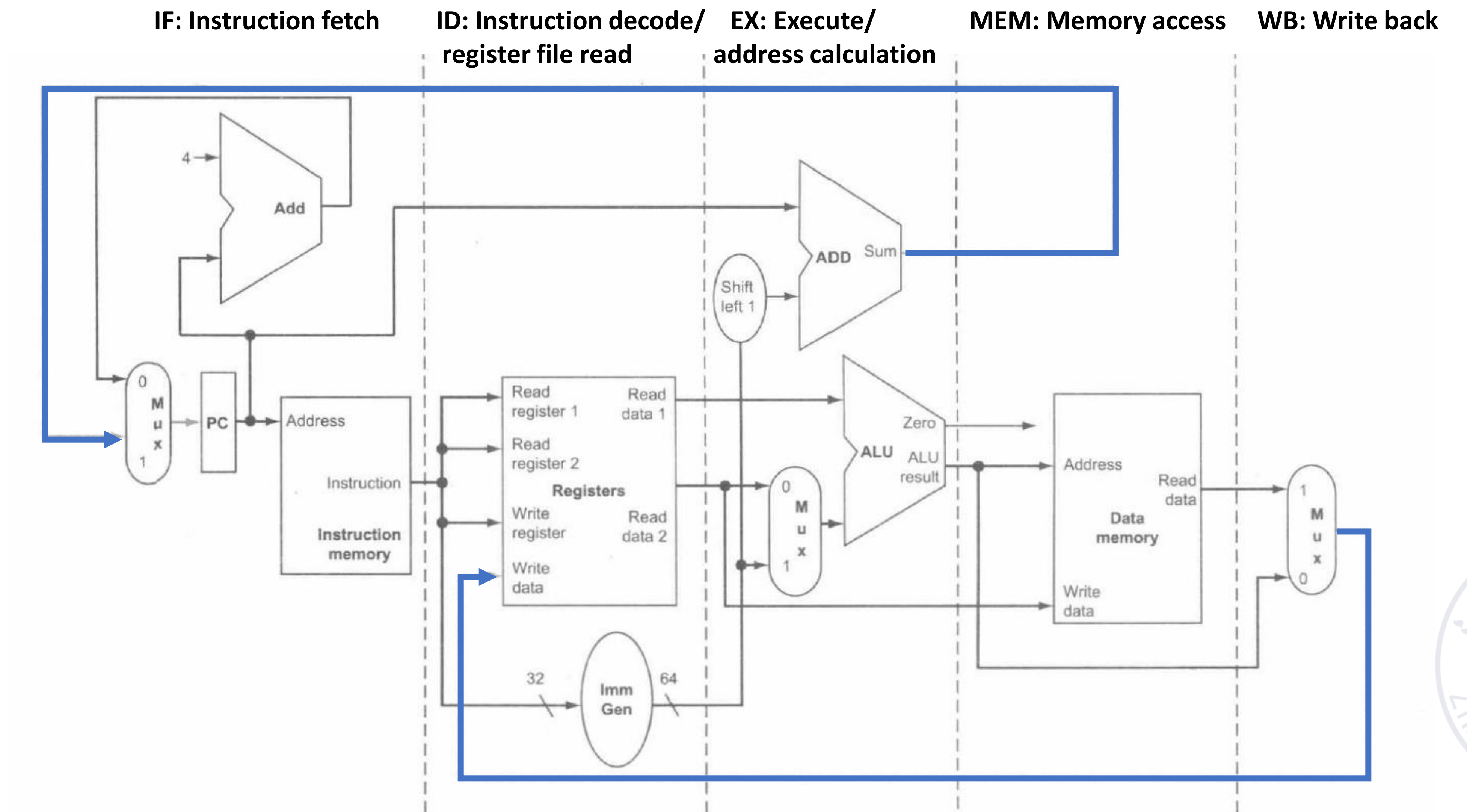
Single-Cycle Datapath

Let's find two exceptions to this left-to-right flow of instructions



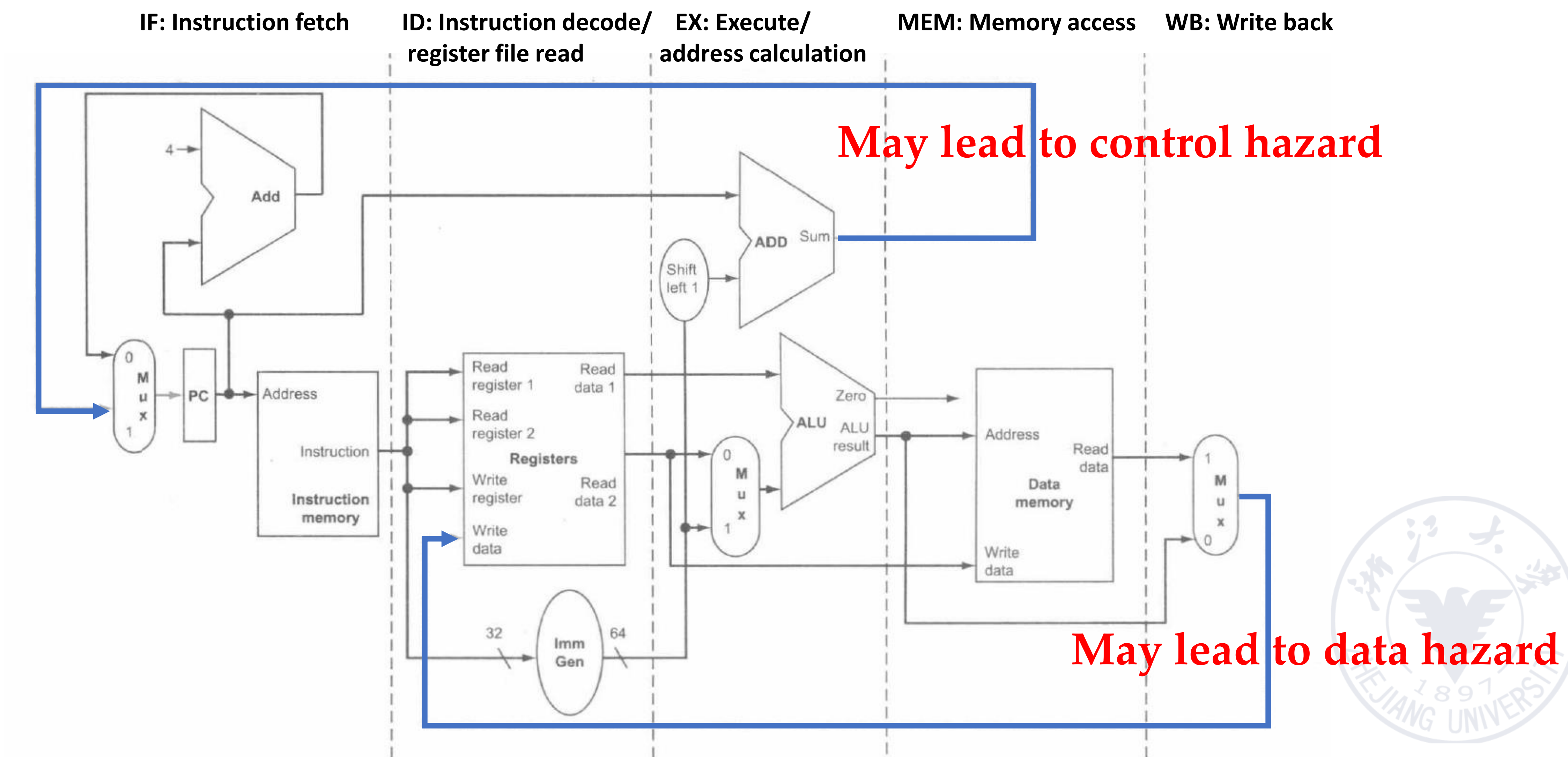
Single-Cycle Datapath

Question: The impacts of the two exceptions



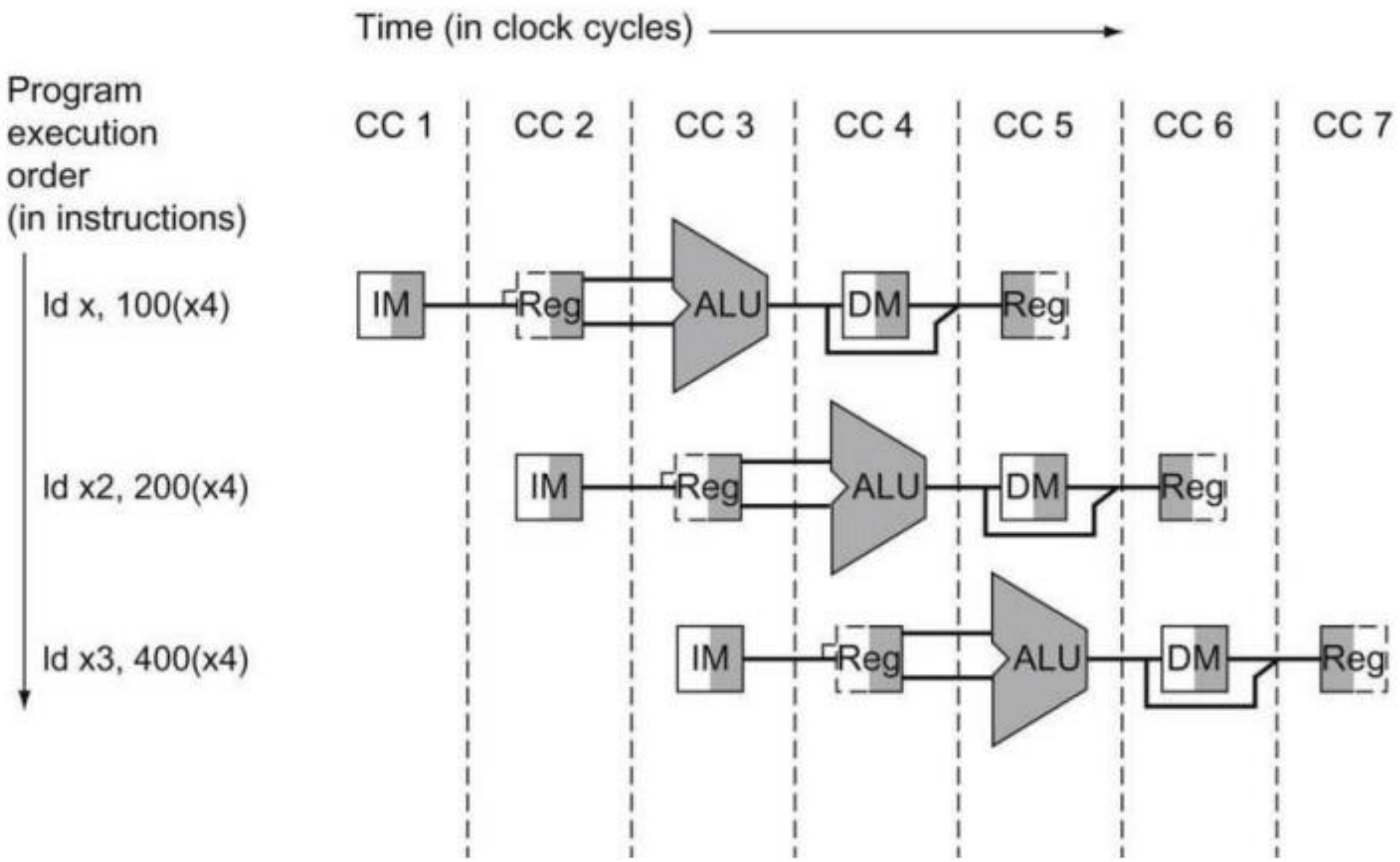
Single-Cycle Datapath

Question: The impacts of the two exceptions

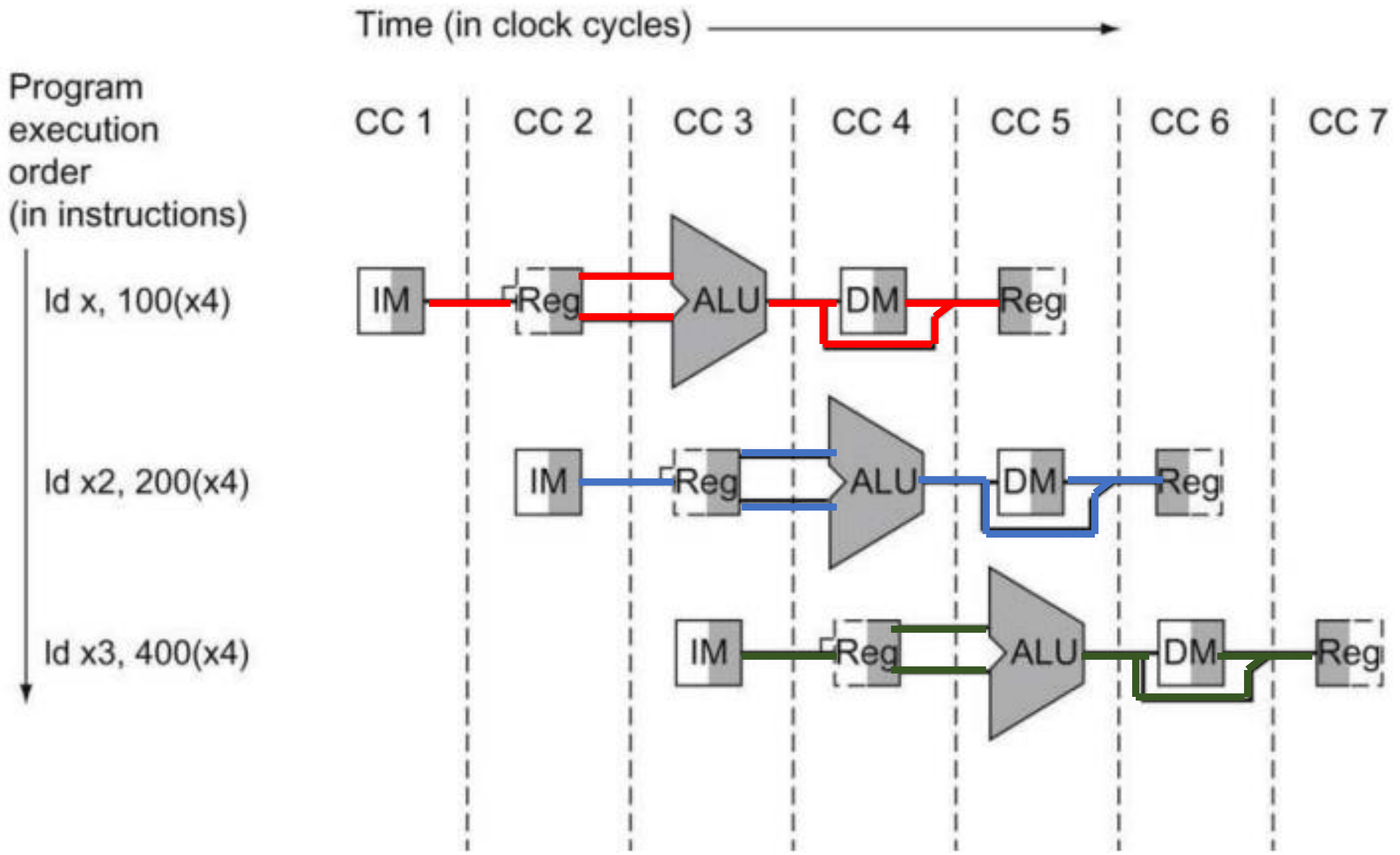


Single-Cycle Datapath to the Pipelined Version

Instructions Being Executed Using the Single-Cycle Datapath

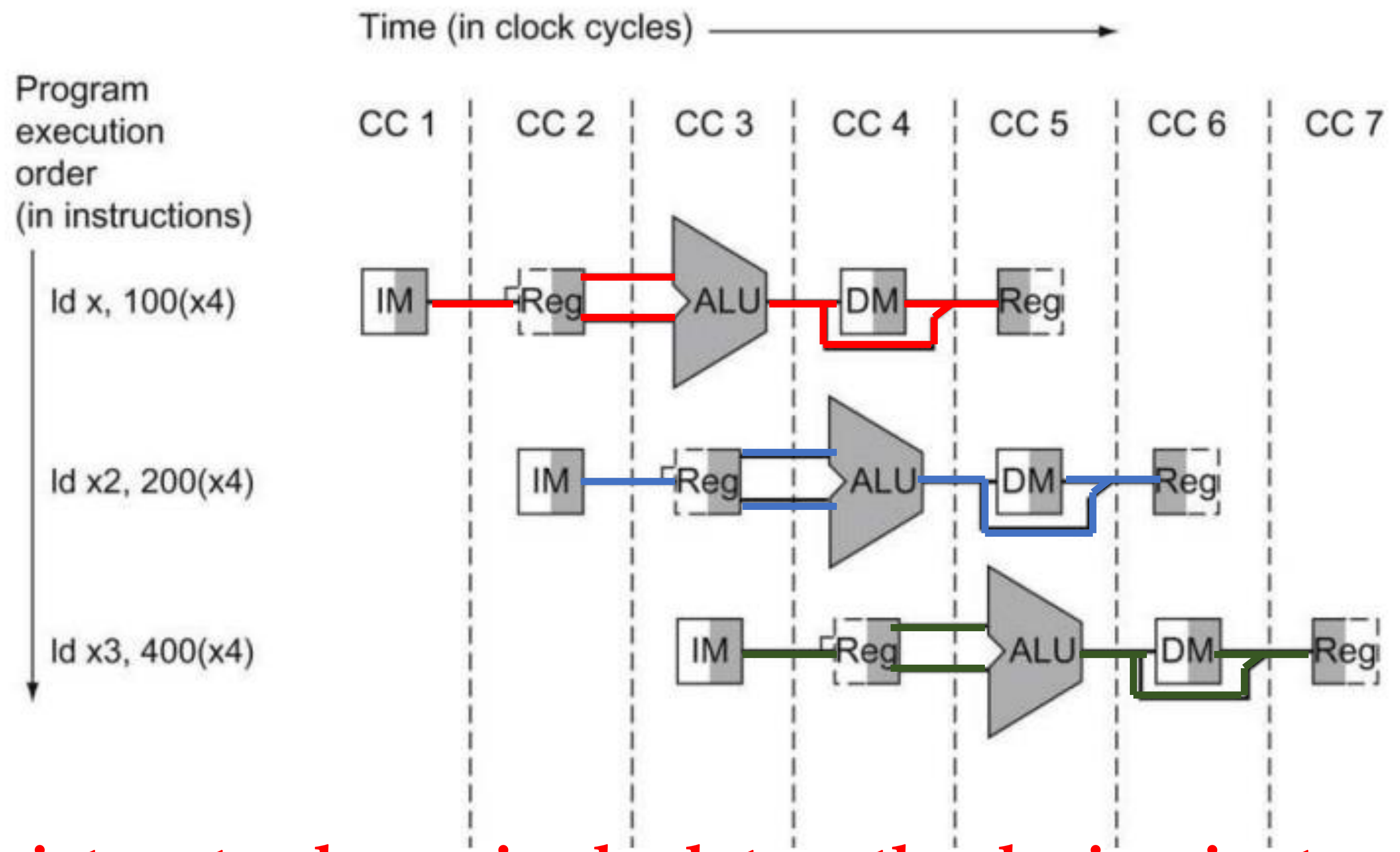


Instructions Being Executed Using the Single-Cycle Datapath



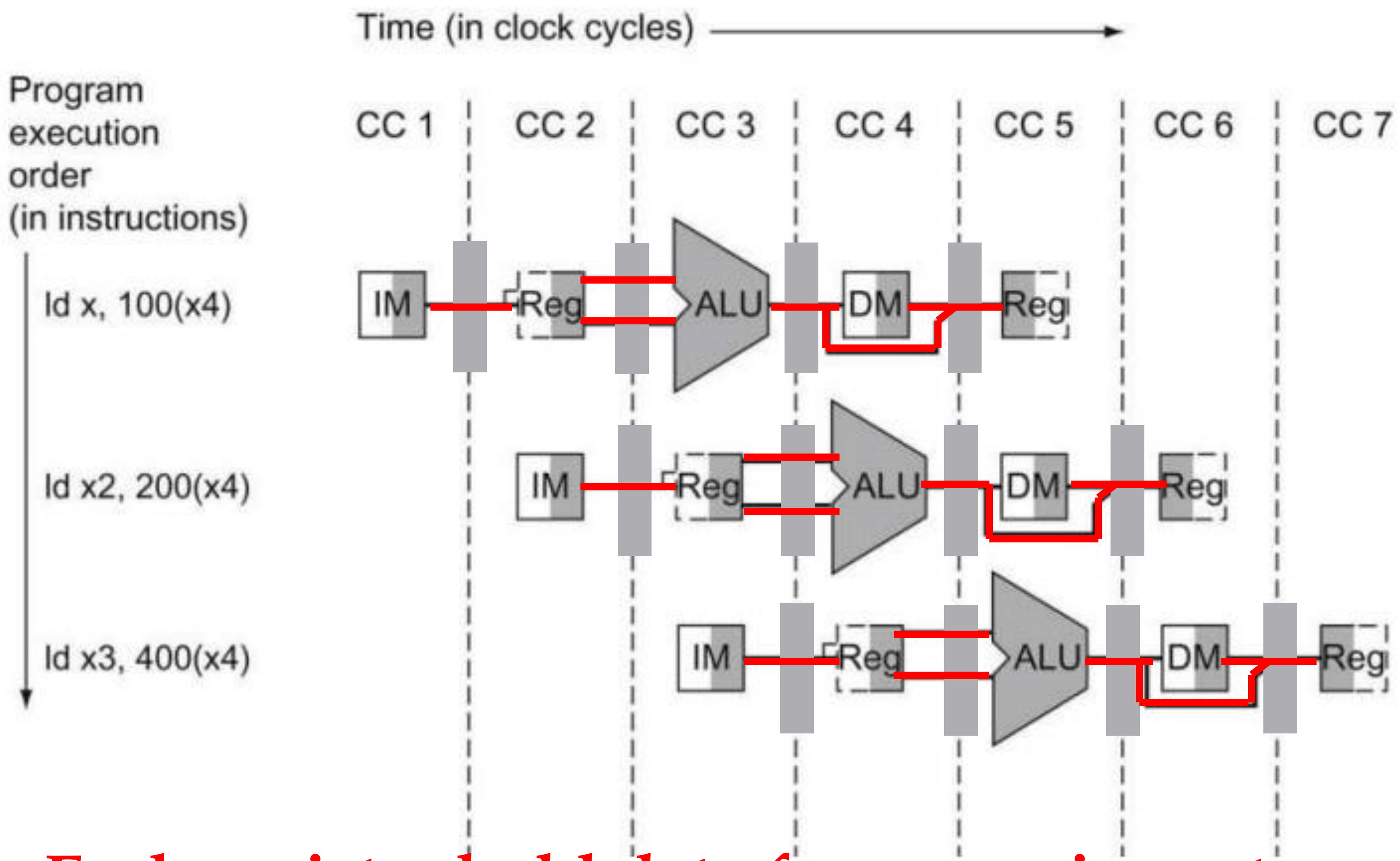
Three instructions need three datapaths

Instructions Being Executed Using the Single-Cycle Datapath



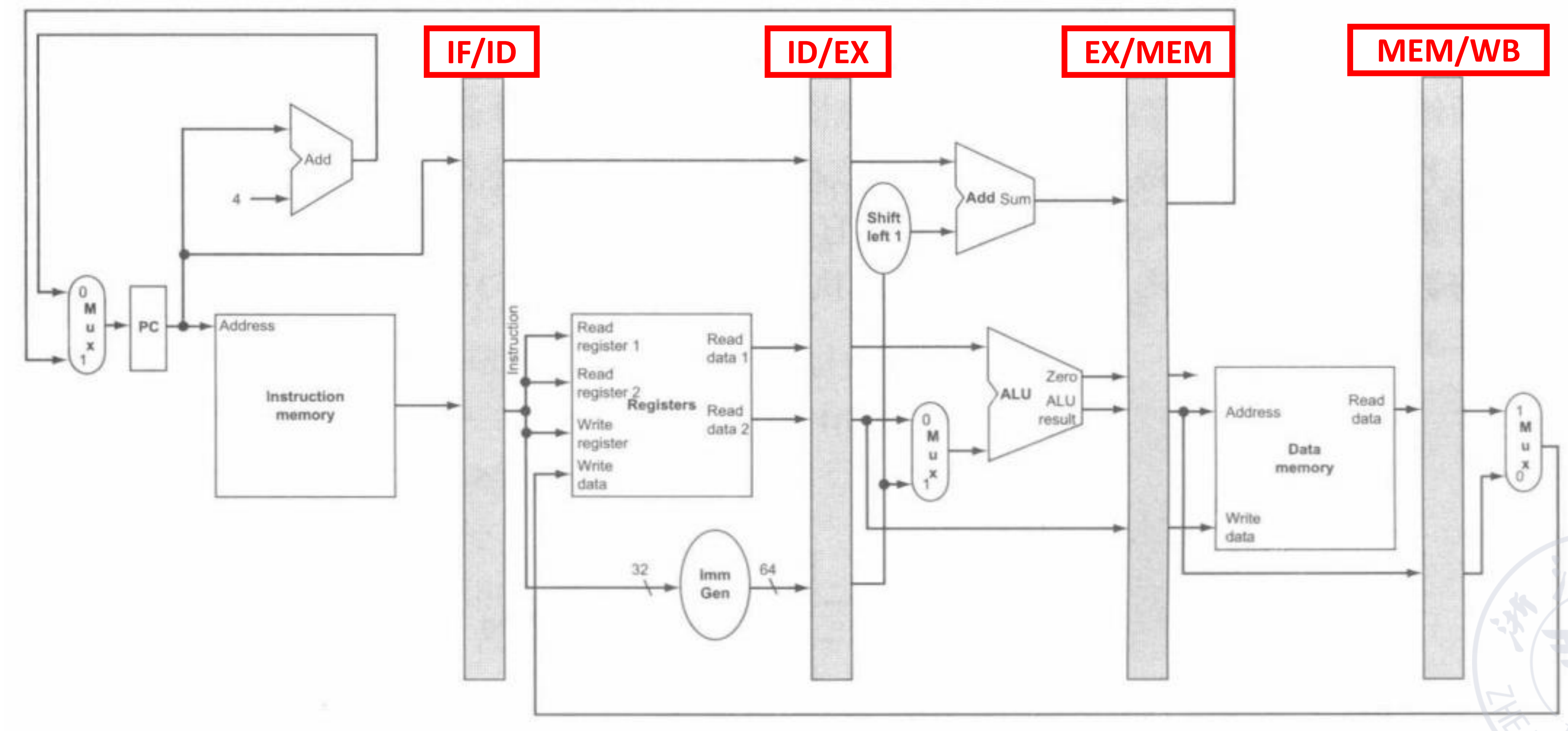
Let's add registers to share single datapaths during instruction execution

Instructions Being Executed Using the Single-Cycle Datapath



Each register hold data from previous stage

Pipelined Version of the Datapath

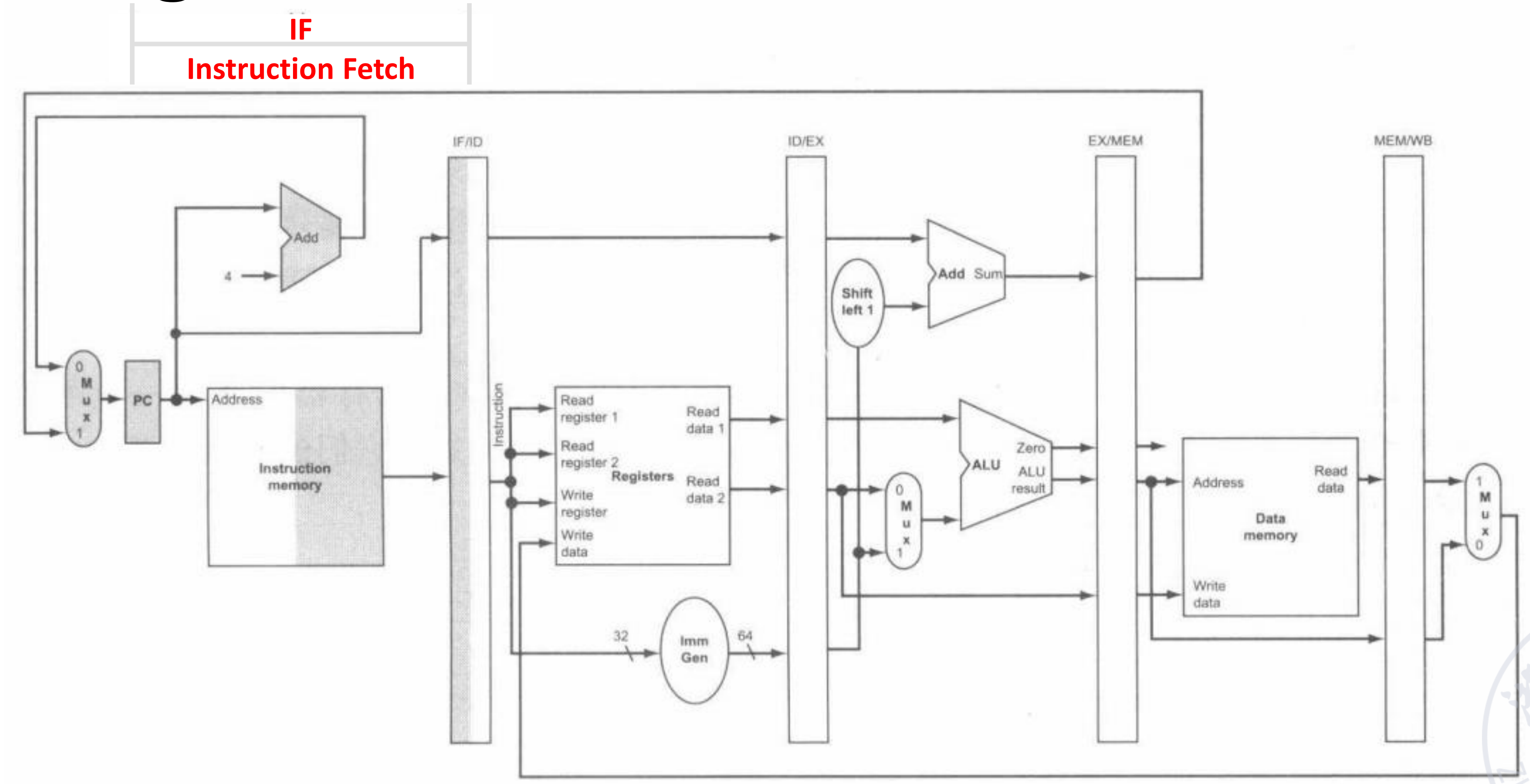


Understand How Pipelined Datapath Work

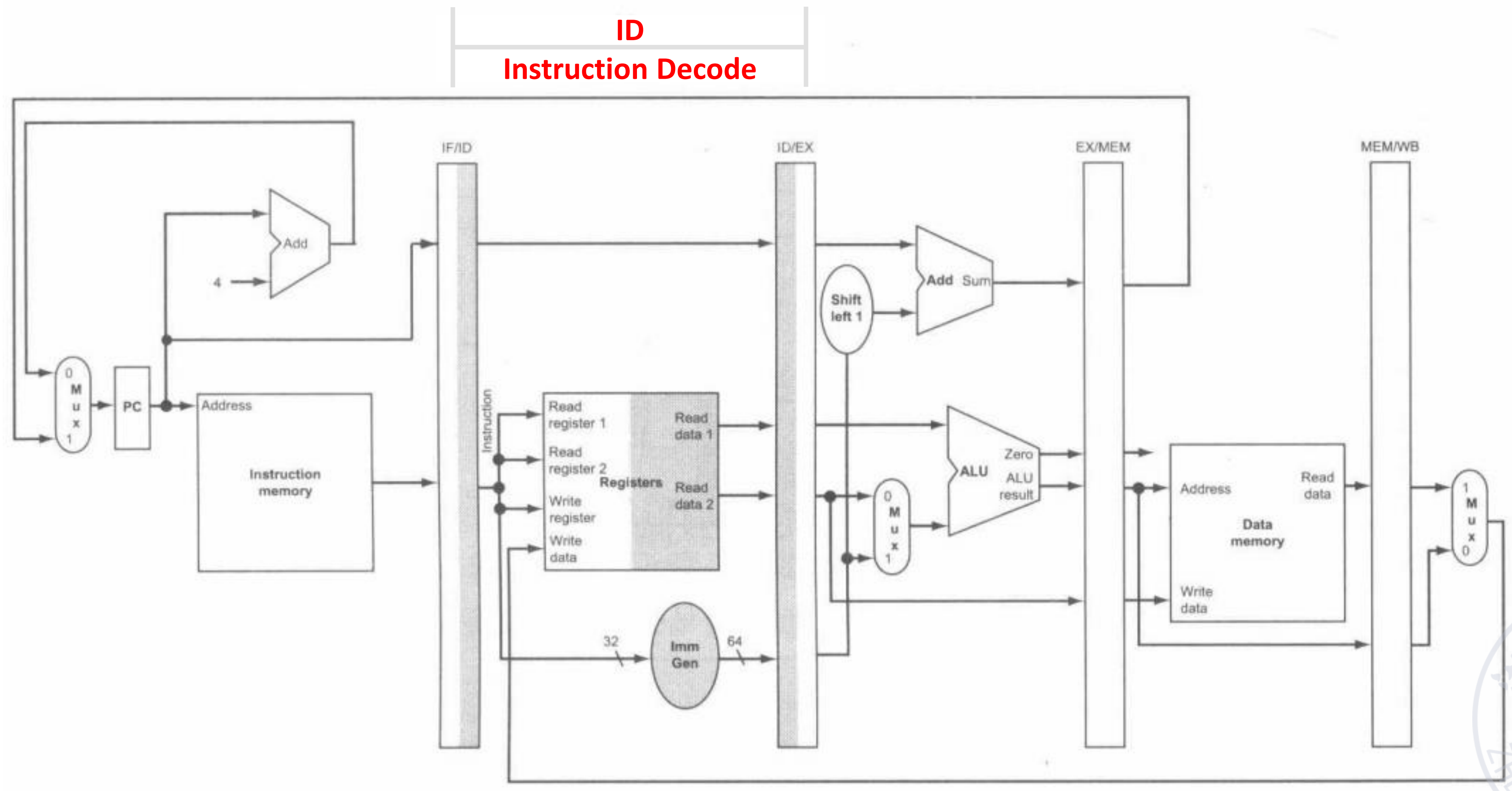
Through Load Instruction and Store Instruction

Load Instruction

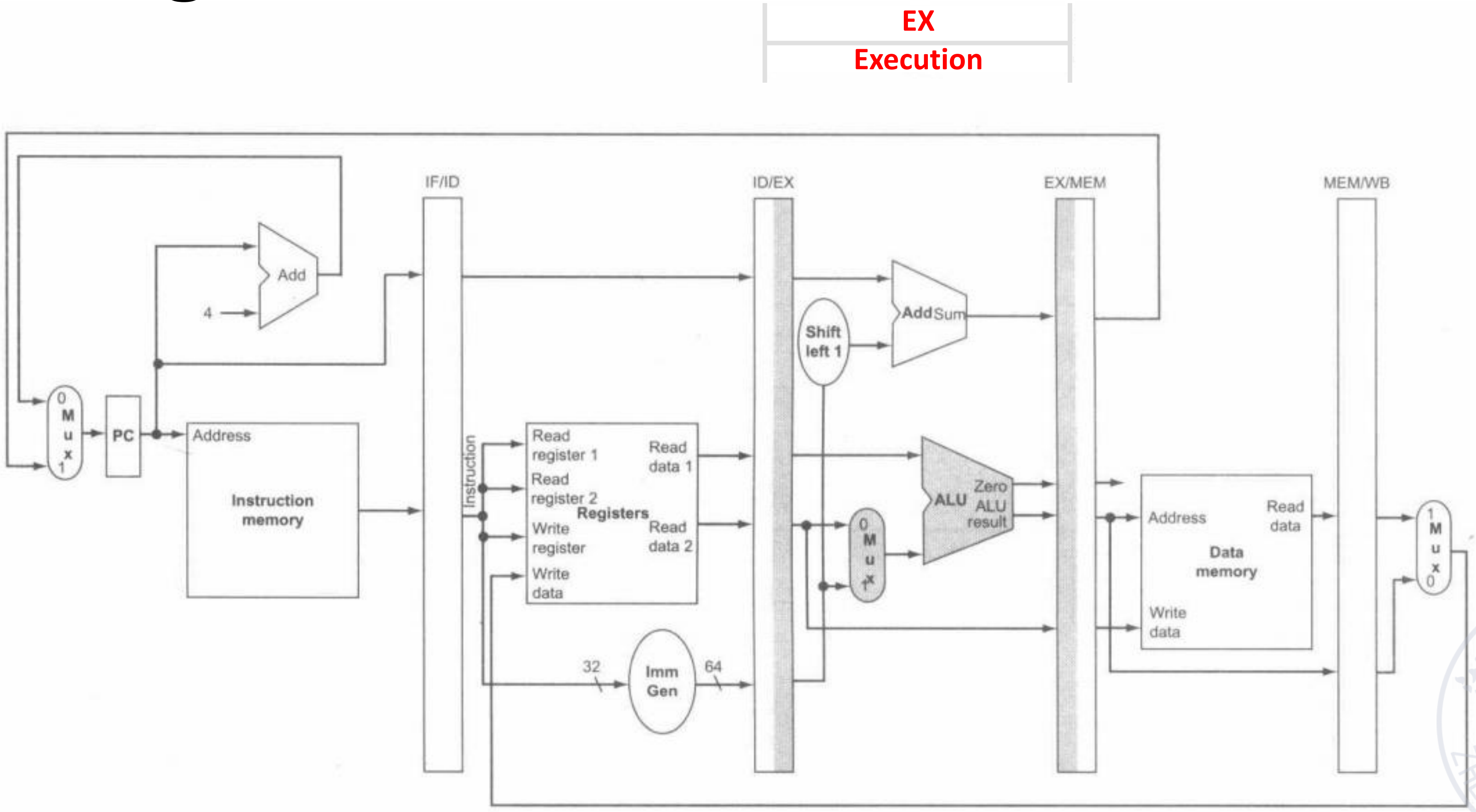
IF Stage of Load Instruction



ID Stage of Load Instruction

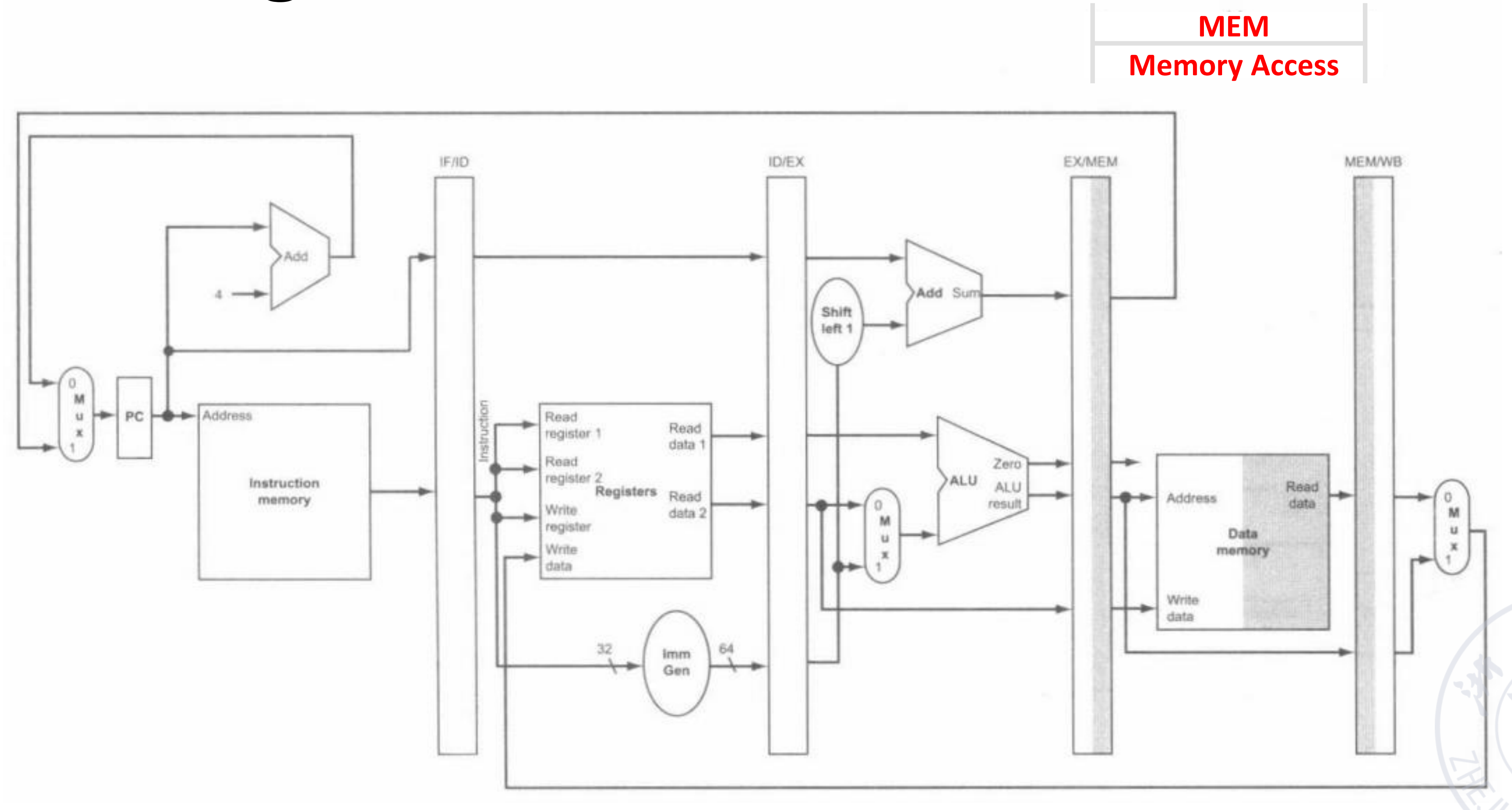


EX Stage of Load Instruction

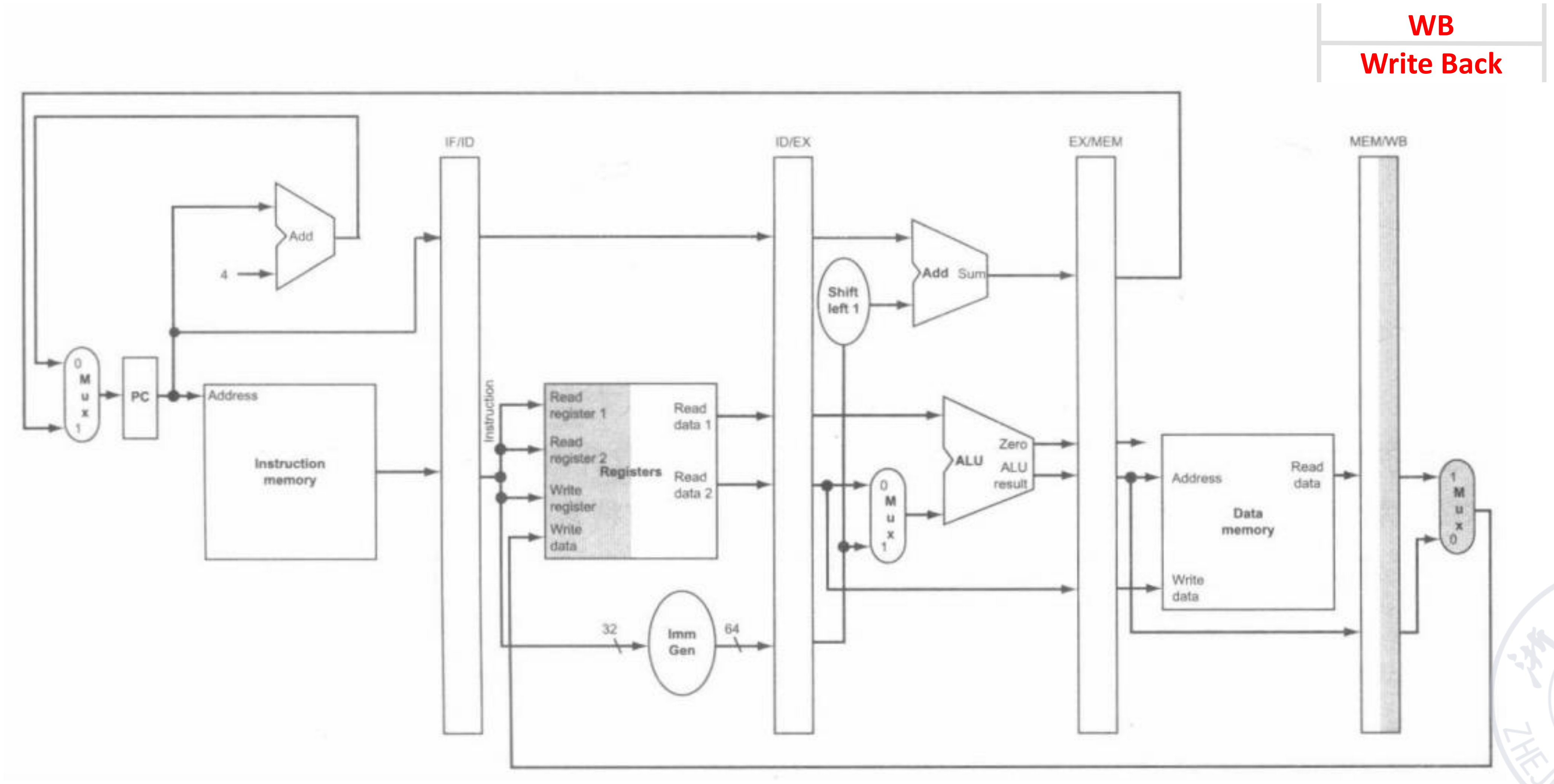


MEM Stage of Load Instruction

MEM
Memory Access



WB Stage of Load Instruction


WB
Write Back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
32
Imm Gen
64
Shift left 1
Add Sum
ALU
Zero
ALU result
Address
Read data
Data memory
Write data