Computer Systems II

Li Lu

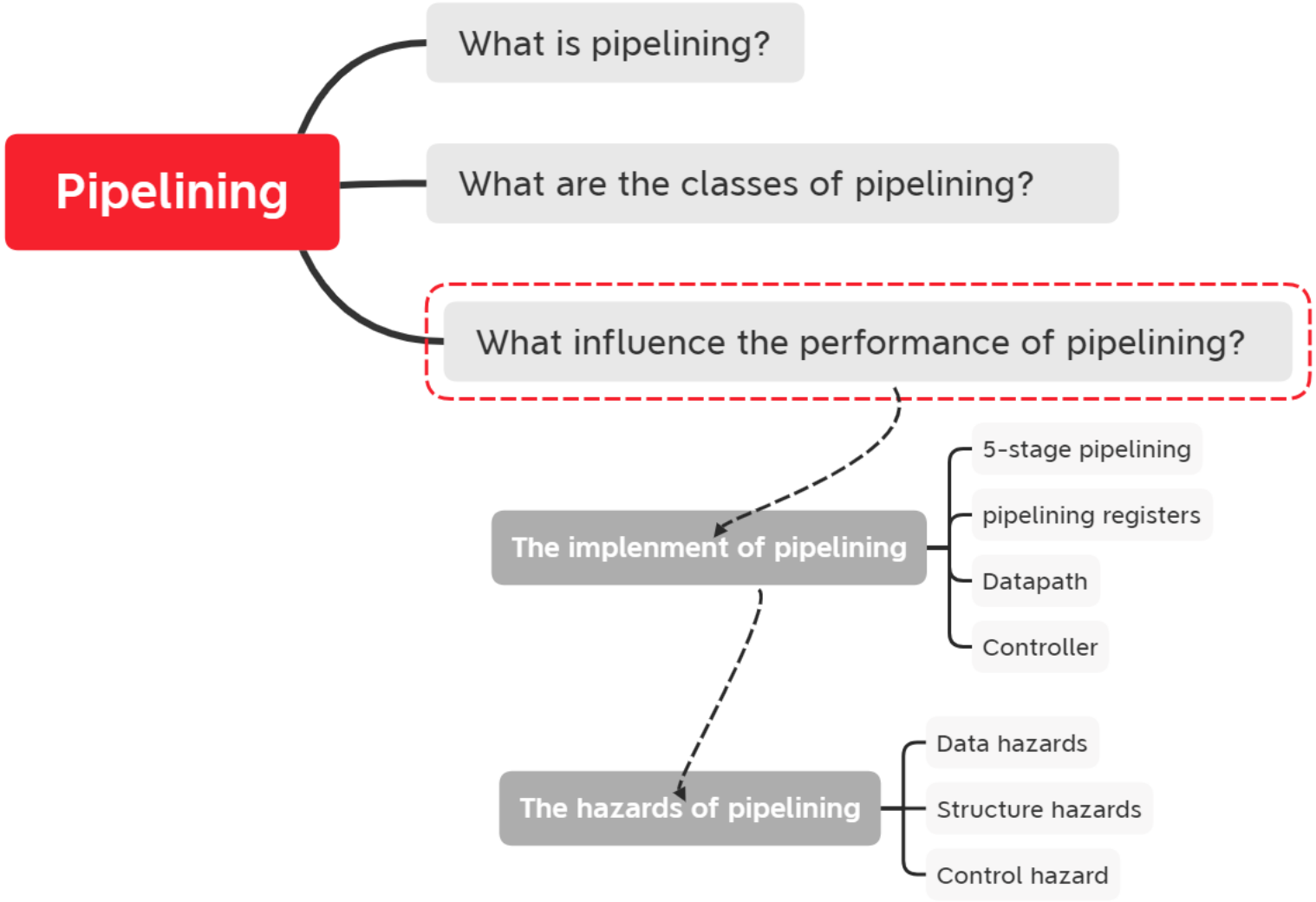
Room 319, Yifu Business and Management Building

Yuquan Campus

li.lu@zju.edu.cn

https://person.zju.edu.cn/lynnluli

§ 3.1 An Overview of Pipelining

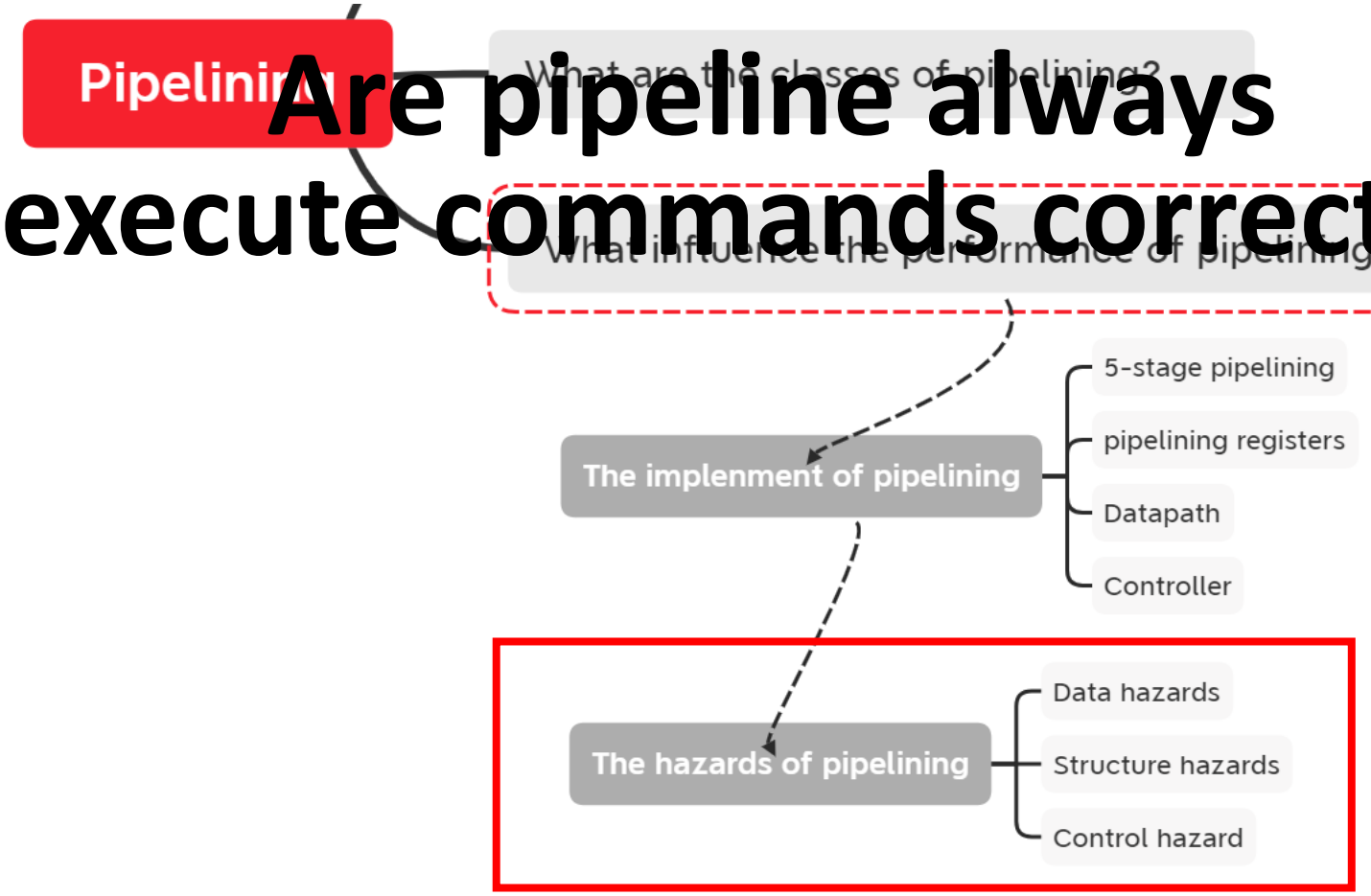


How Pipelining Improves Performance?

Decreasing the execution time of an individual instruction ×

Increasing instruction throughput √

Are pipeline always execute commands correctly?



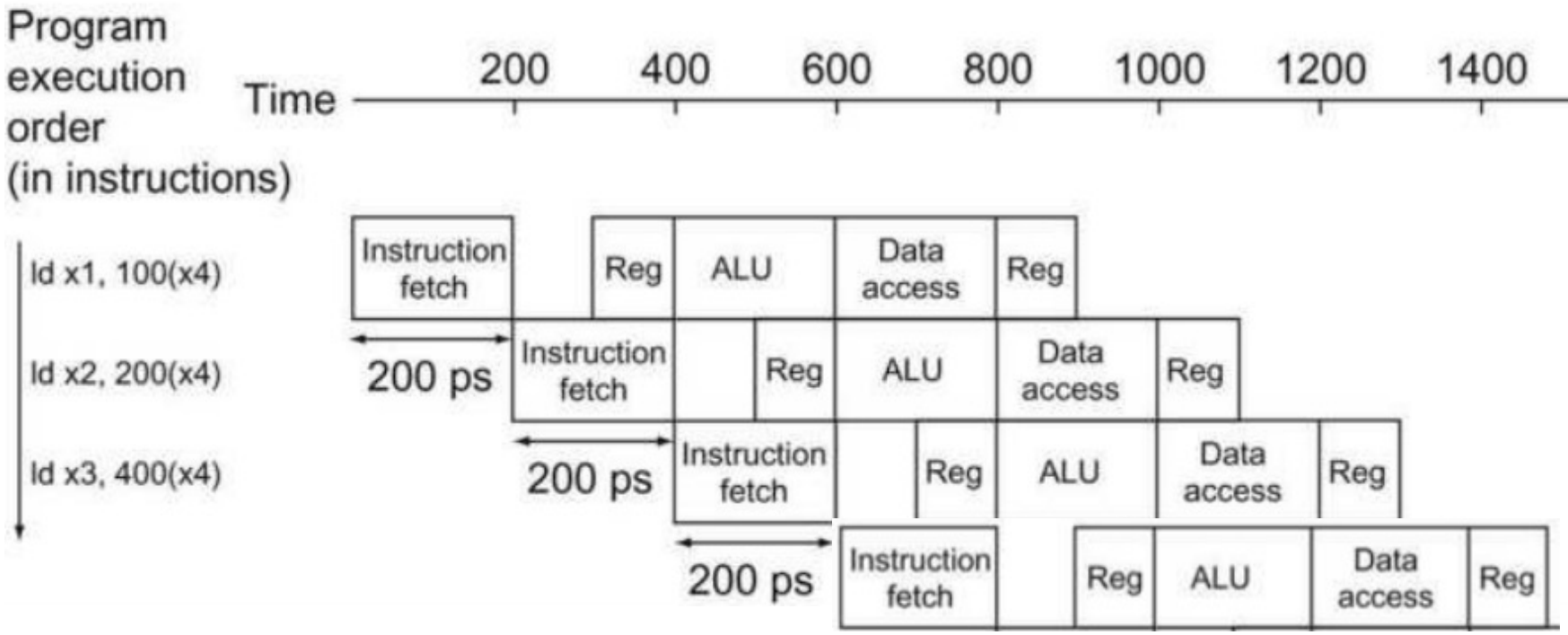
Pipeline Hazards

- 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



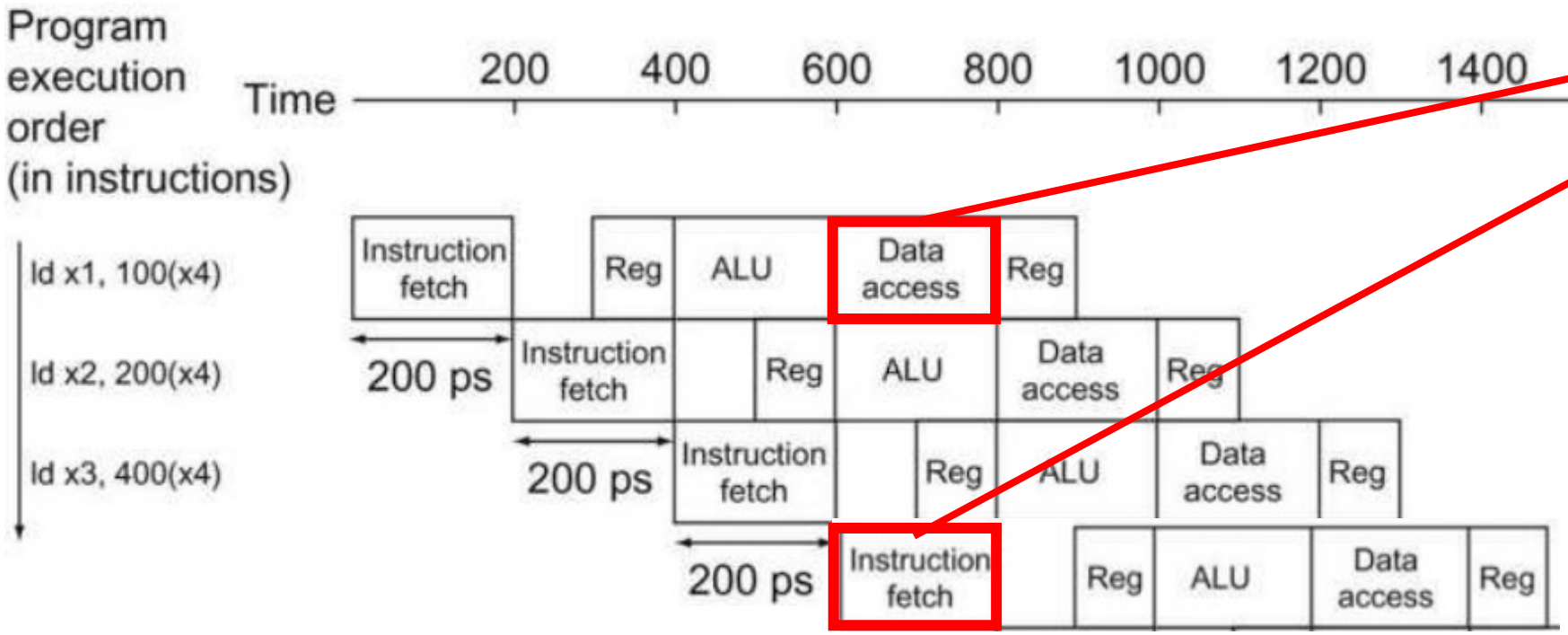
Question: Can the four instructions execute correctly?

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

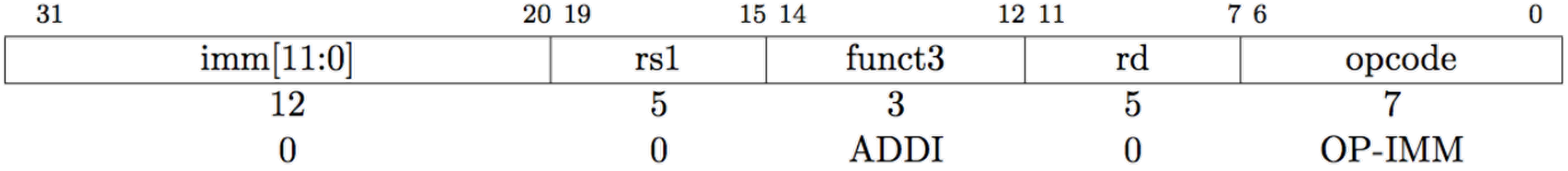
- Solution 2: Add more hardware to machine

Can always solve a structural hazard by adding more hardware

How to Stall ?

NOP instruction

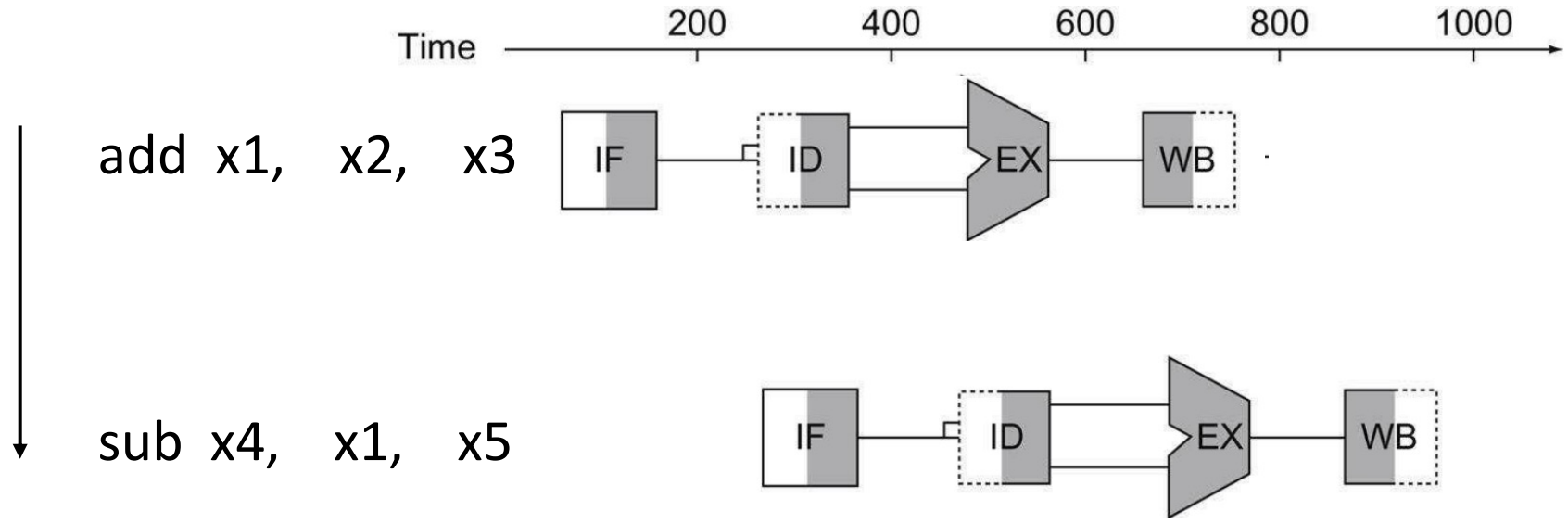
ADDI x0, x0, 0



Data Hazard

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

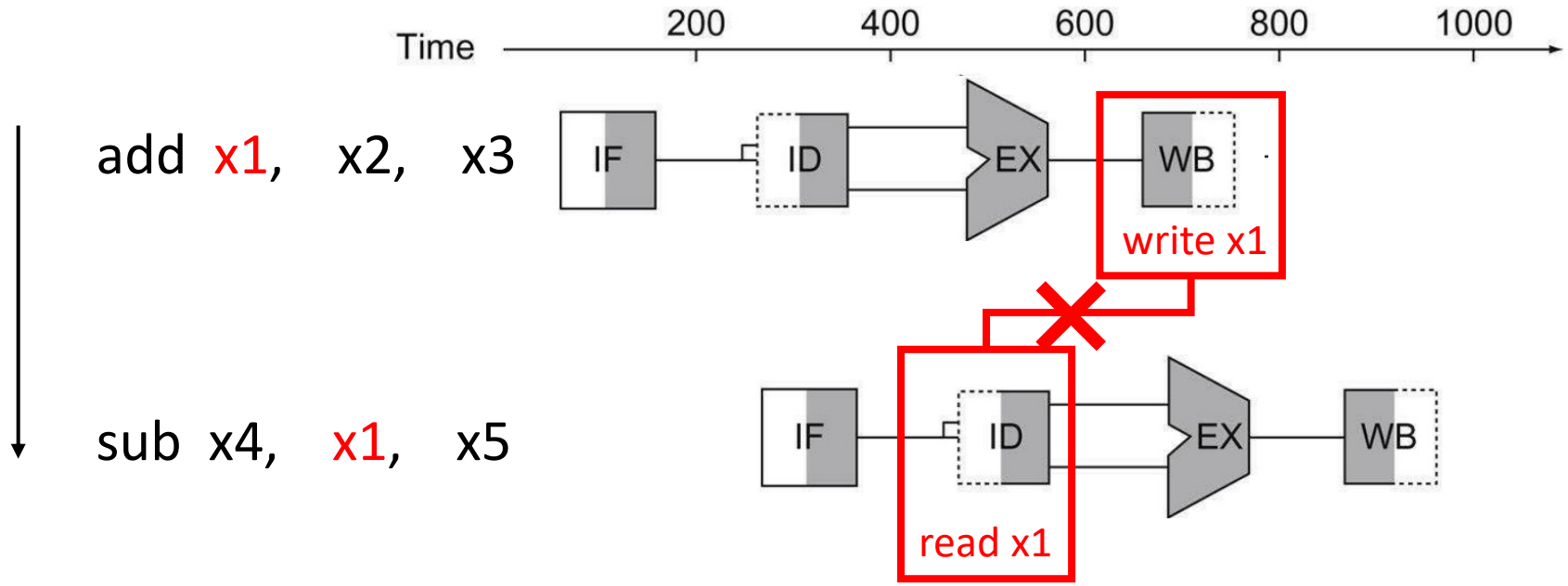
Problem: Instruction depends on result from previous



Data Hazard

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

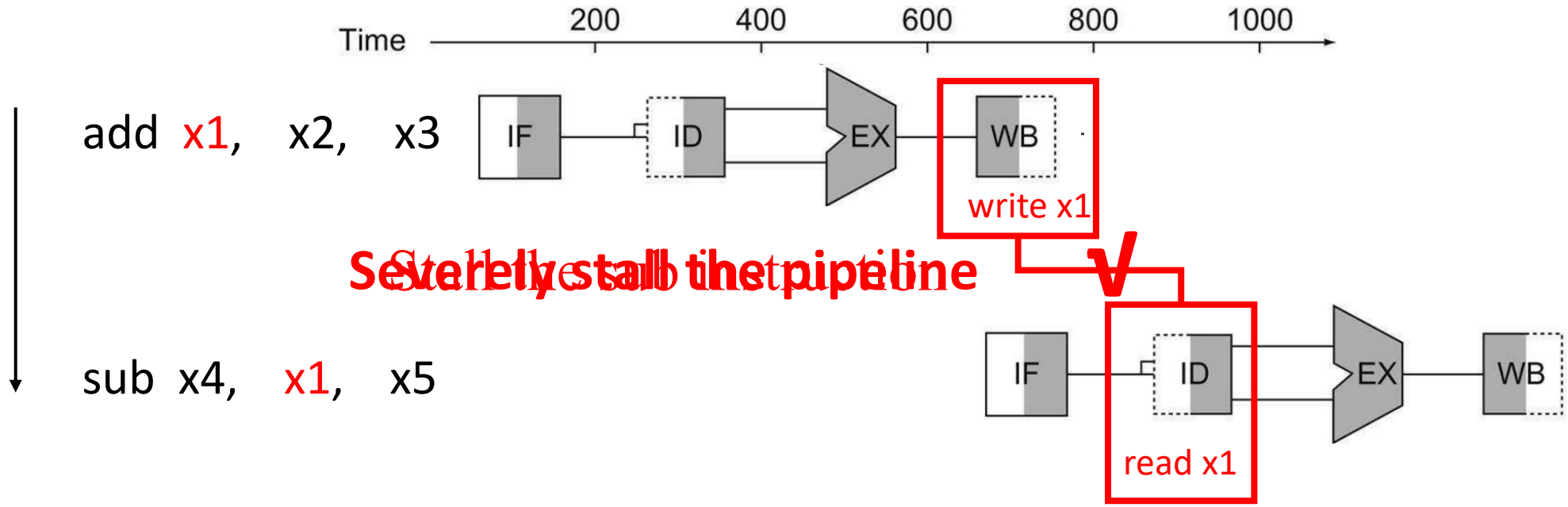
Problem: Instruction depends on result from previous



Data Hazard

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

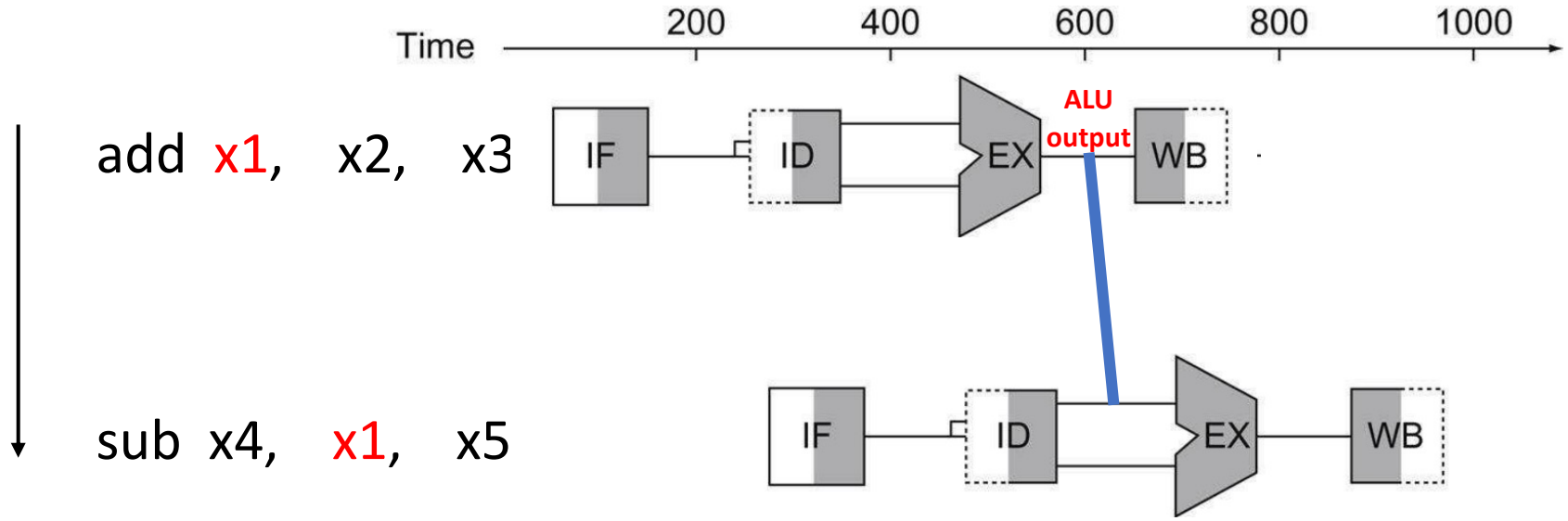
Problem: Instruction depends on result from previous



Data Hazard

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

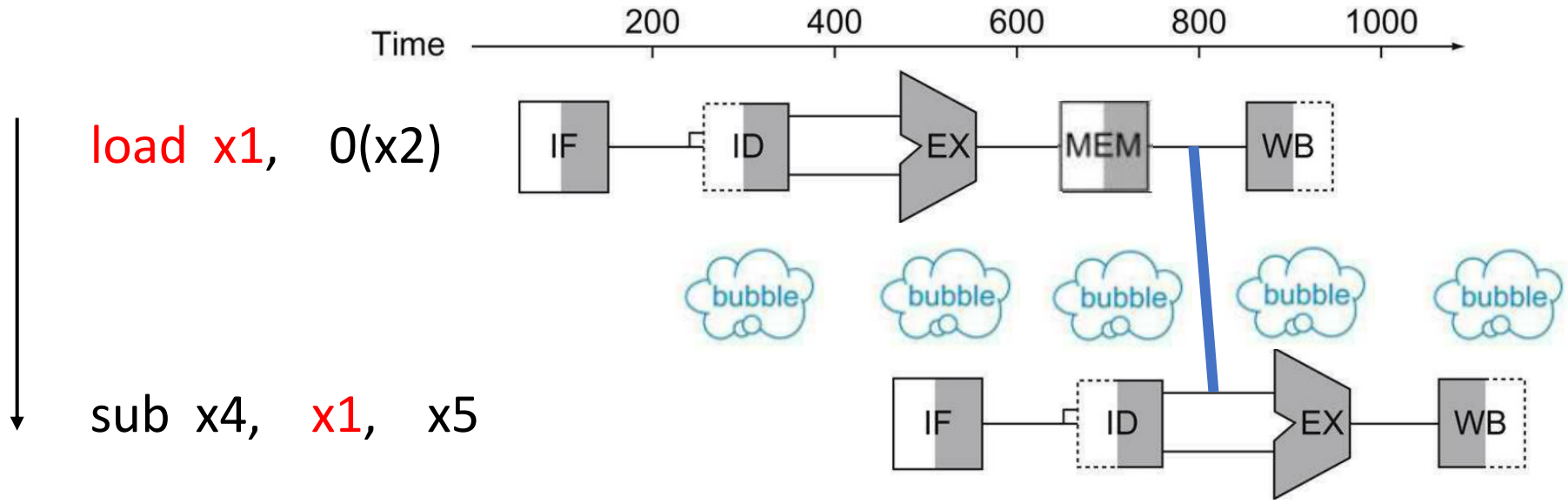
Solution "forwarding": Adding extra hardware to retrieve the missing item early from the internal resources



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



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
```

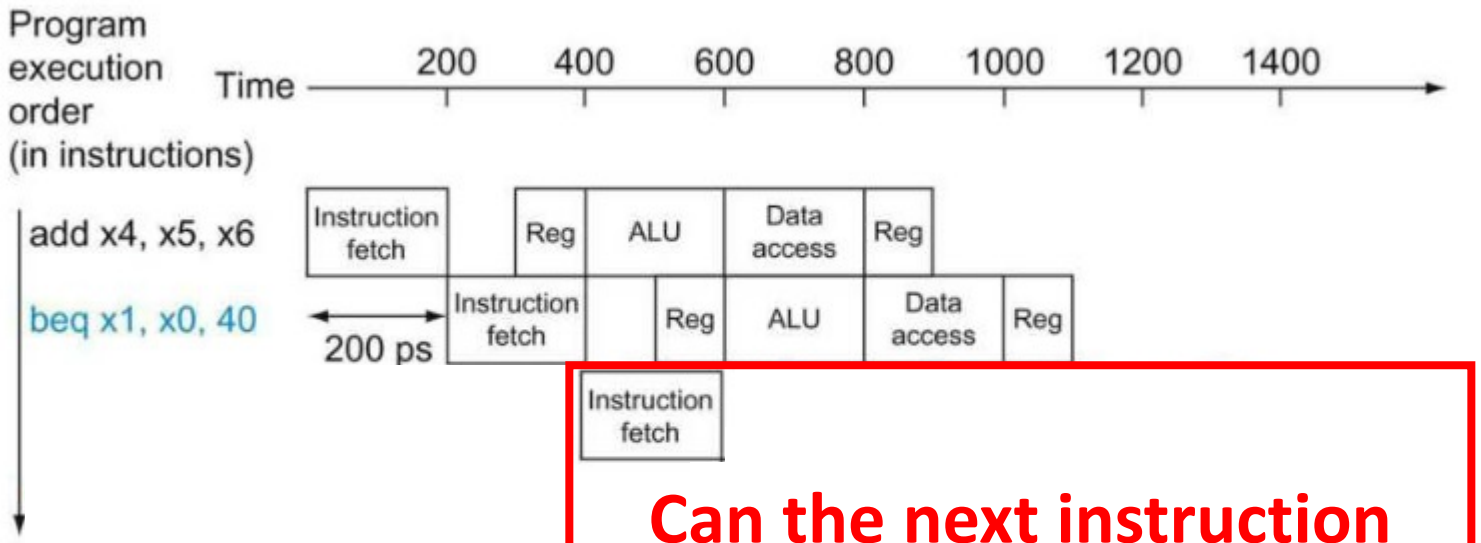
eliminates both hazards

```
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

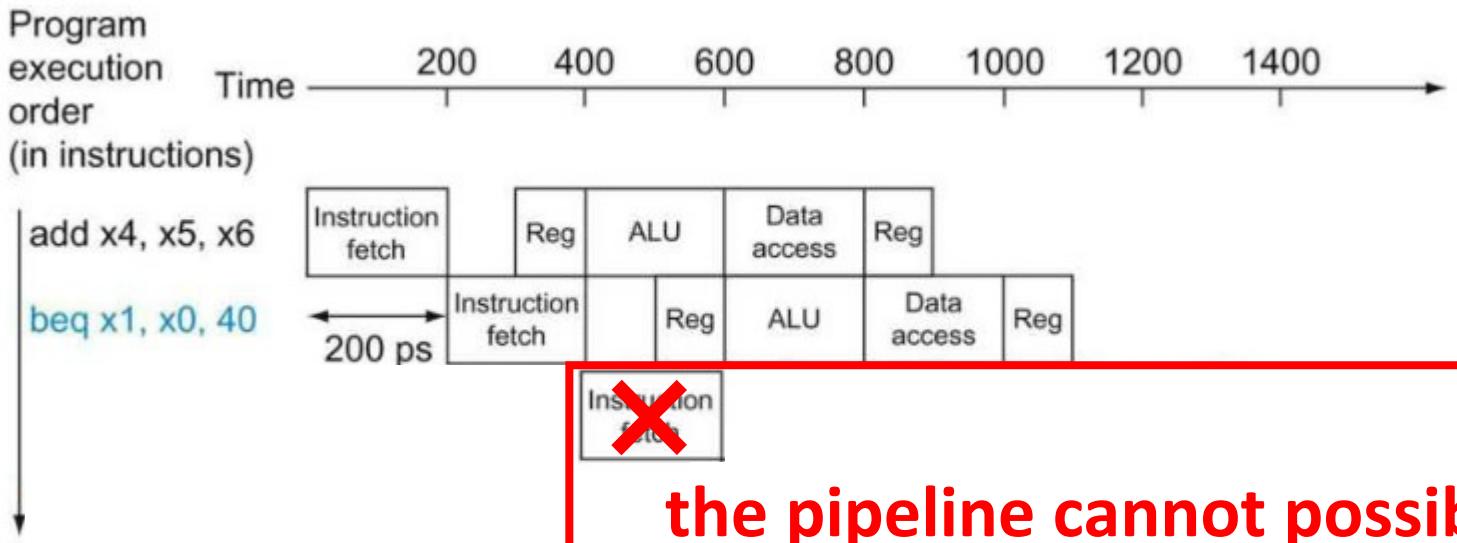


Can the next instruction be executed immediately?

Control Hazard

Flow of execution depends on previous instruction

Problem: The conditional branch instruction

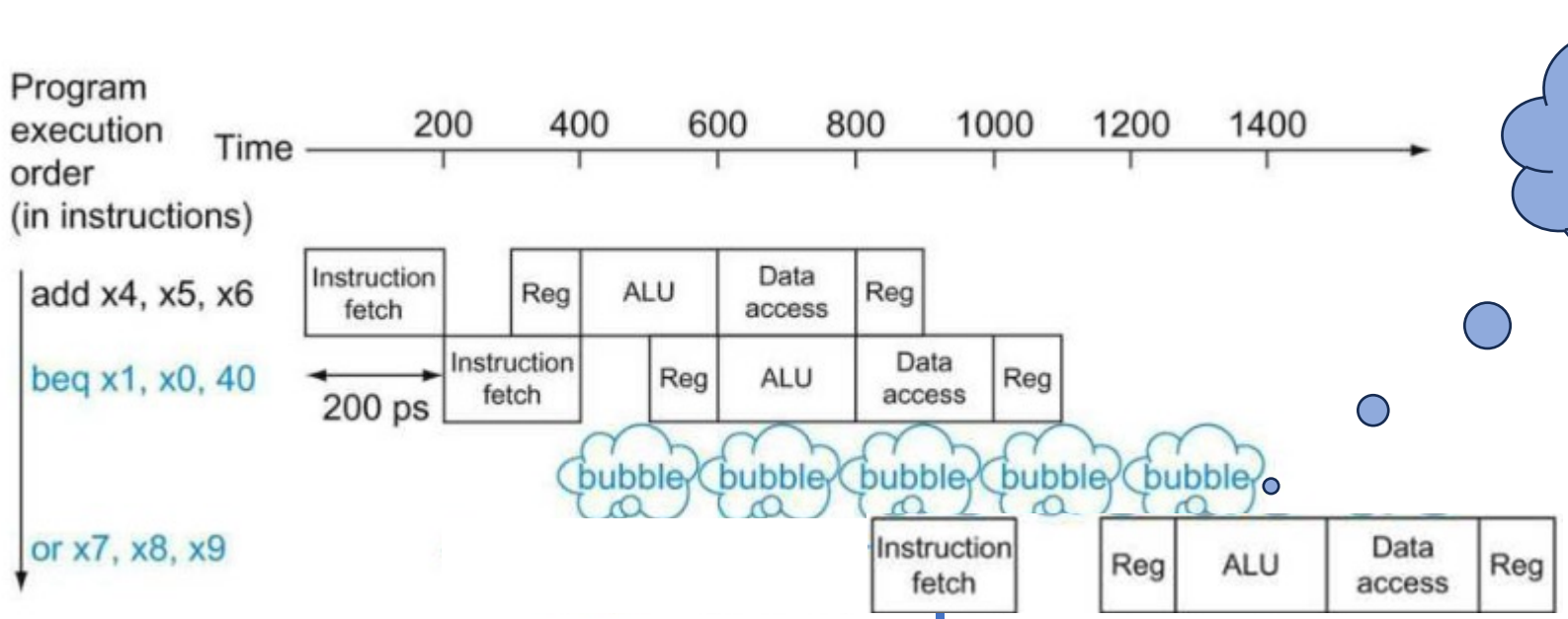


the pipeline cannot possibly know what the next instruction should be

Control Hazard

Flow of execution depends on previous instruction

Solution 1: Stall



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

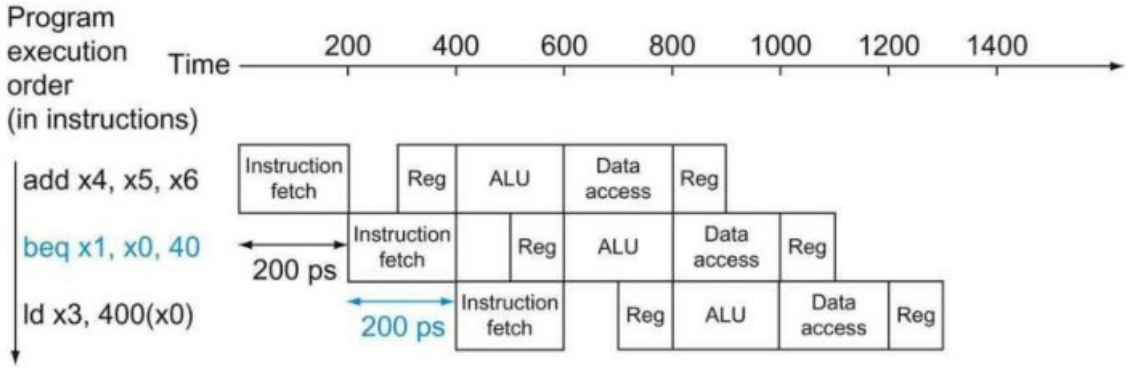
Control Hazard

Flow of execution depends on previous instruction

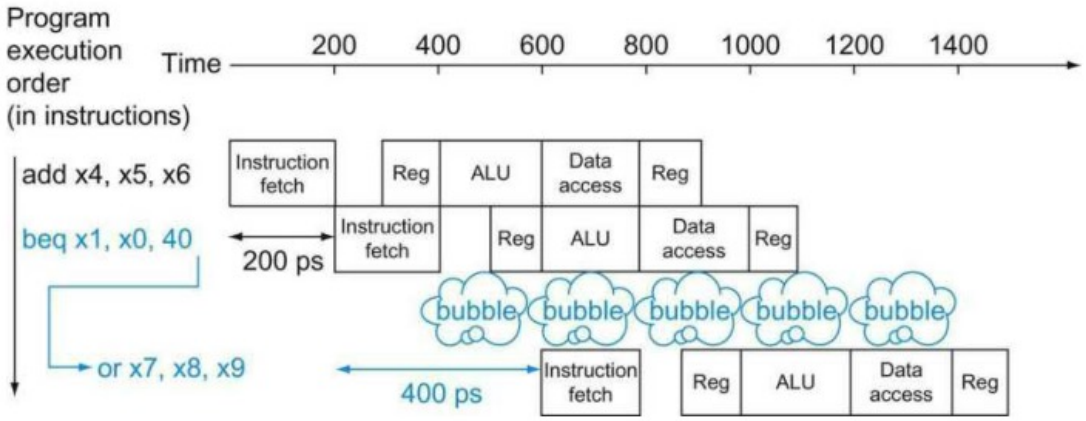
Solution 2: Prediction (simple version)

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

branch is not taken



branch is taken



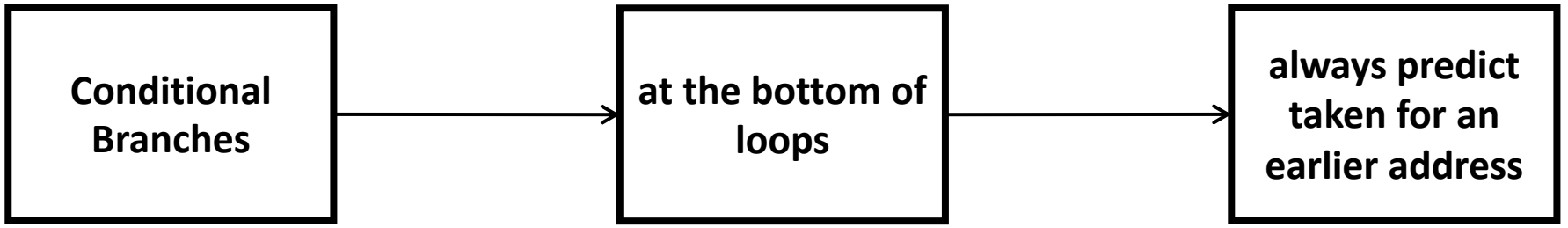
Control Hazard

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



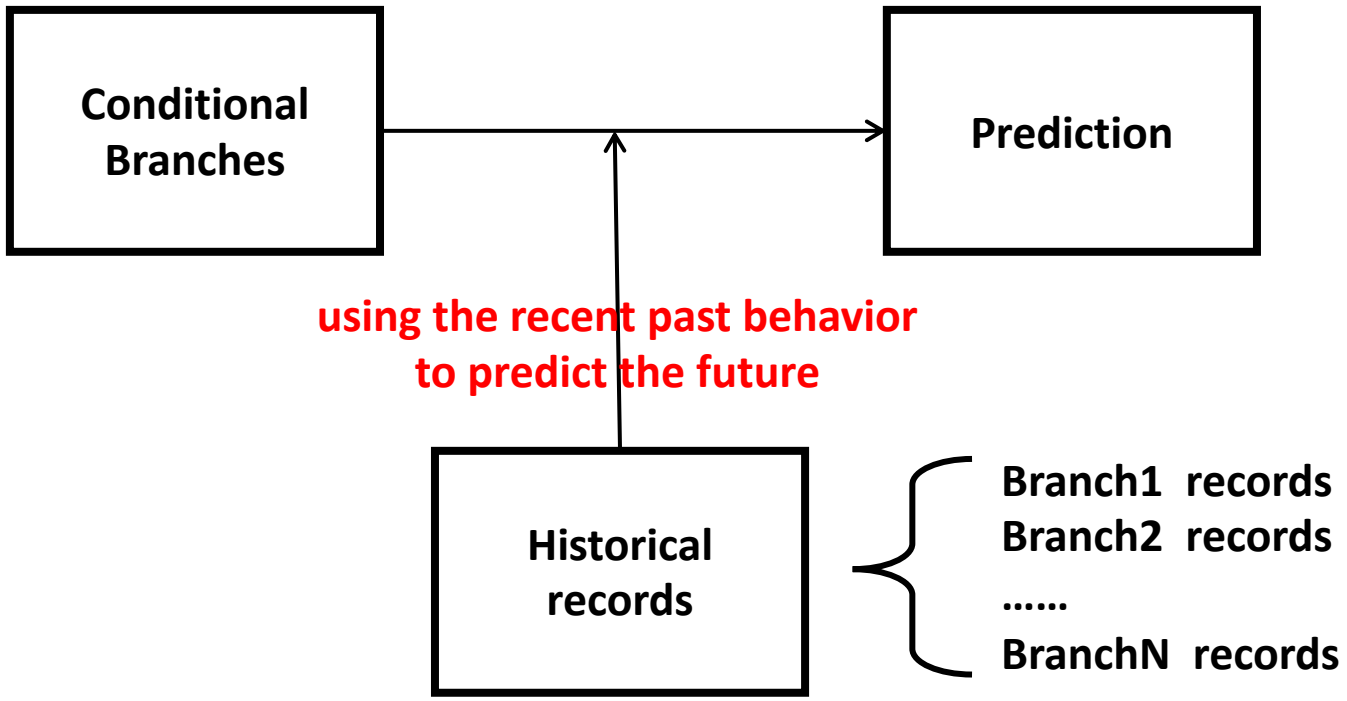
they are likely to be taken to the top of the loops

Control Hazard

Flow of execution depends on previous instruction

Solution 2: Prediction (dynamic prediction)

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



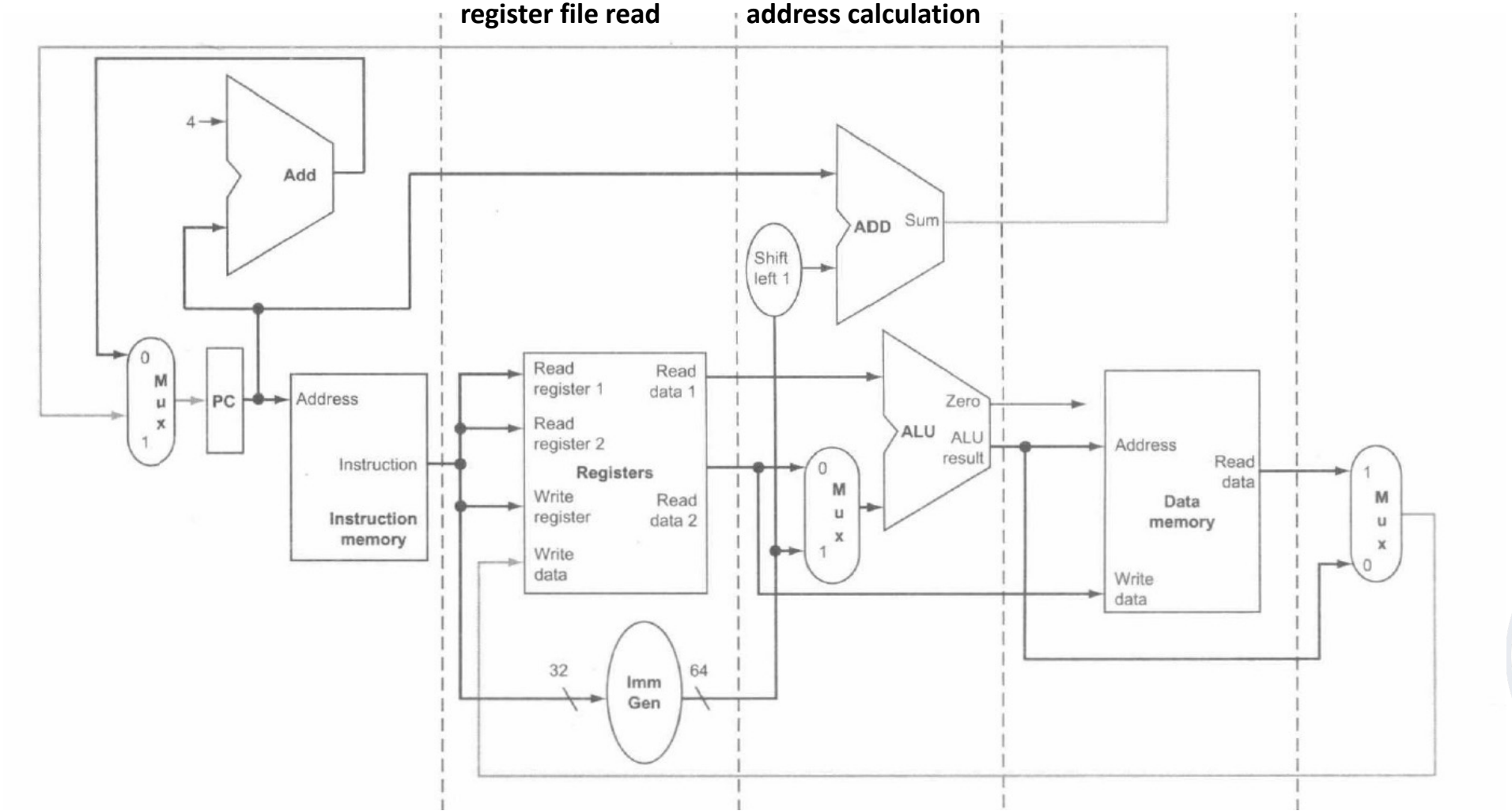
Summary of Pipelining Design

- 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

Pipelined Datapath

Single-Cycle Datapath

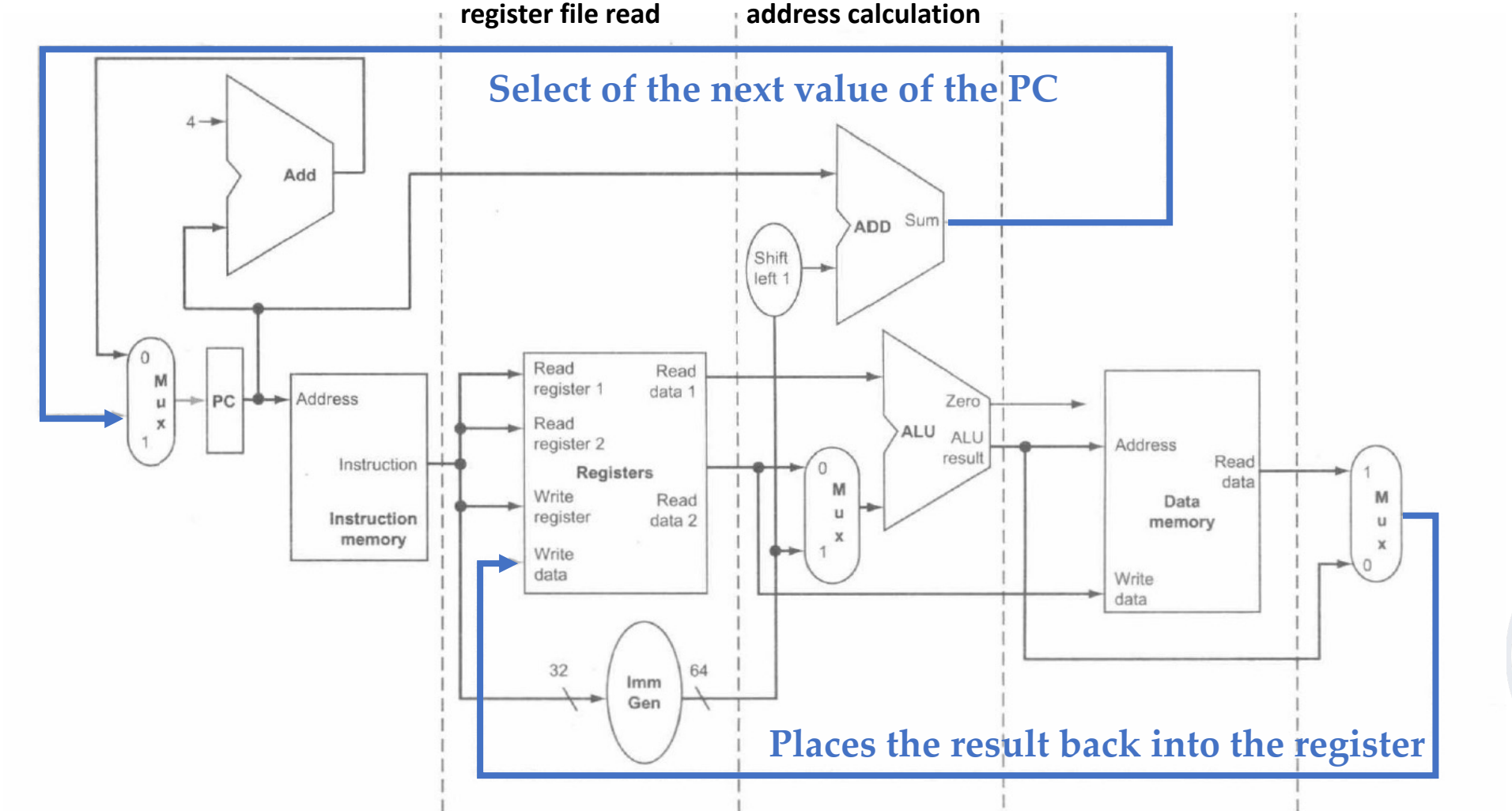
Let's find two exceptions to this left-to-right flow of instructions



Single-Cycle Datapath

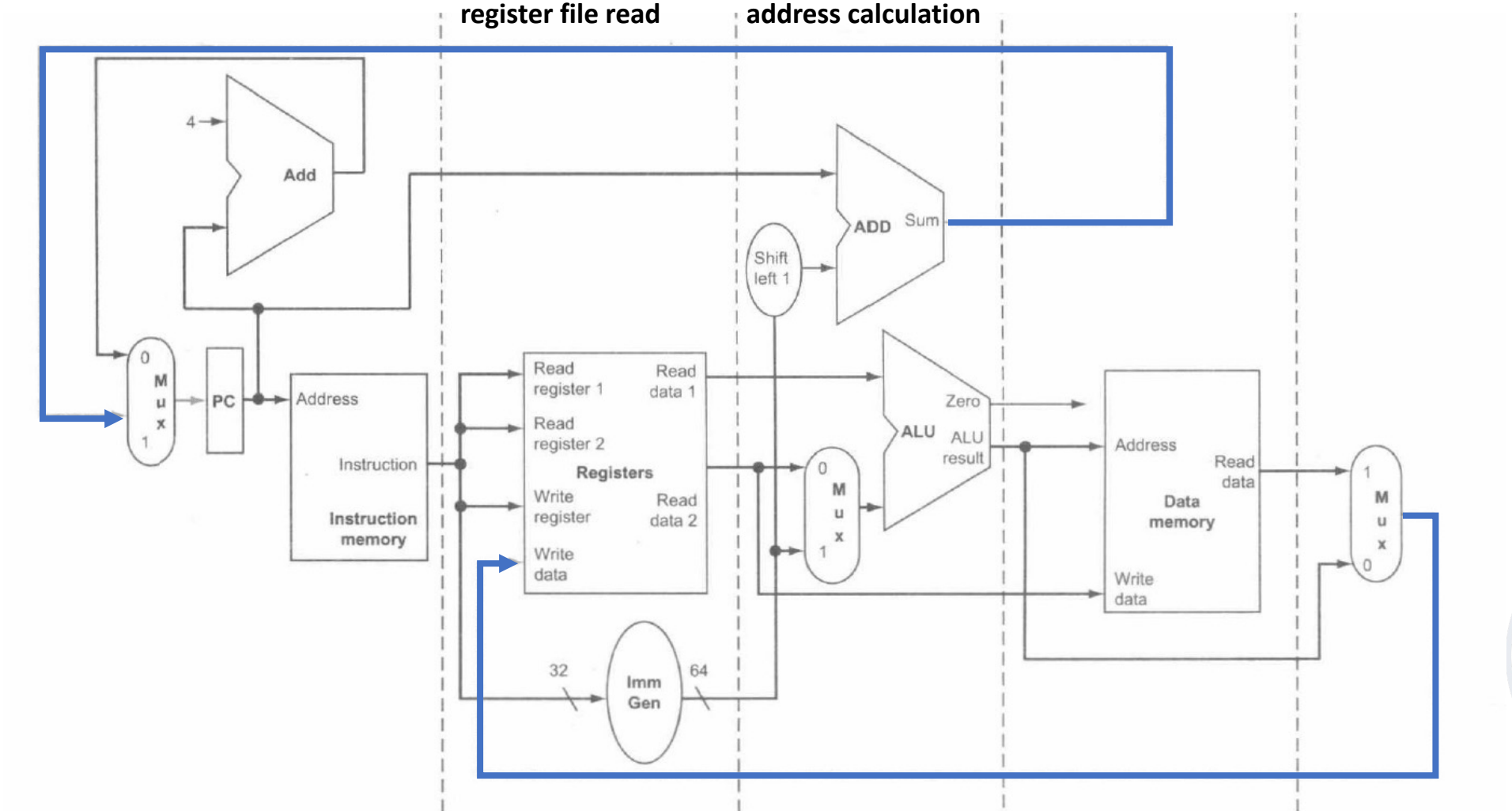
Let's find two exceptions to this left-to-right flow of instructions

IF: Instruction fetch **ID: Instruction decode/ register file read** **EX: Execute/ address calculation** **MEM: Memory access** **WB: Write back**



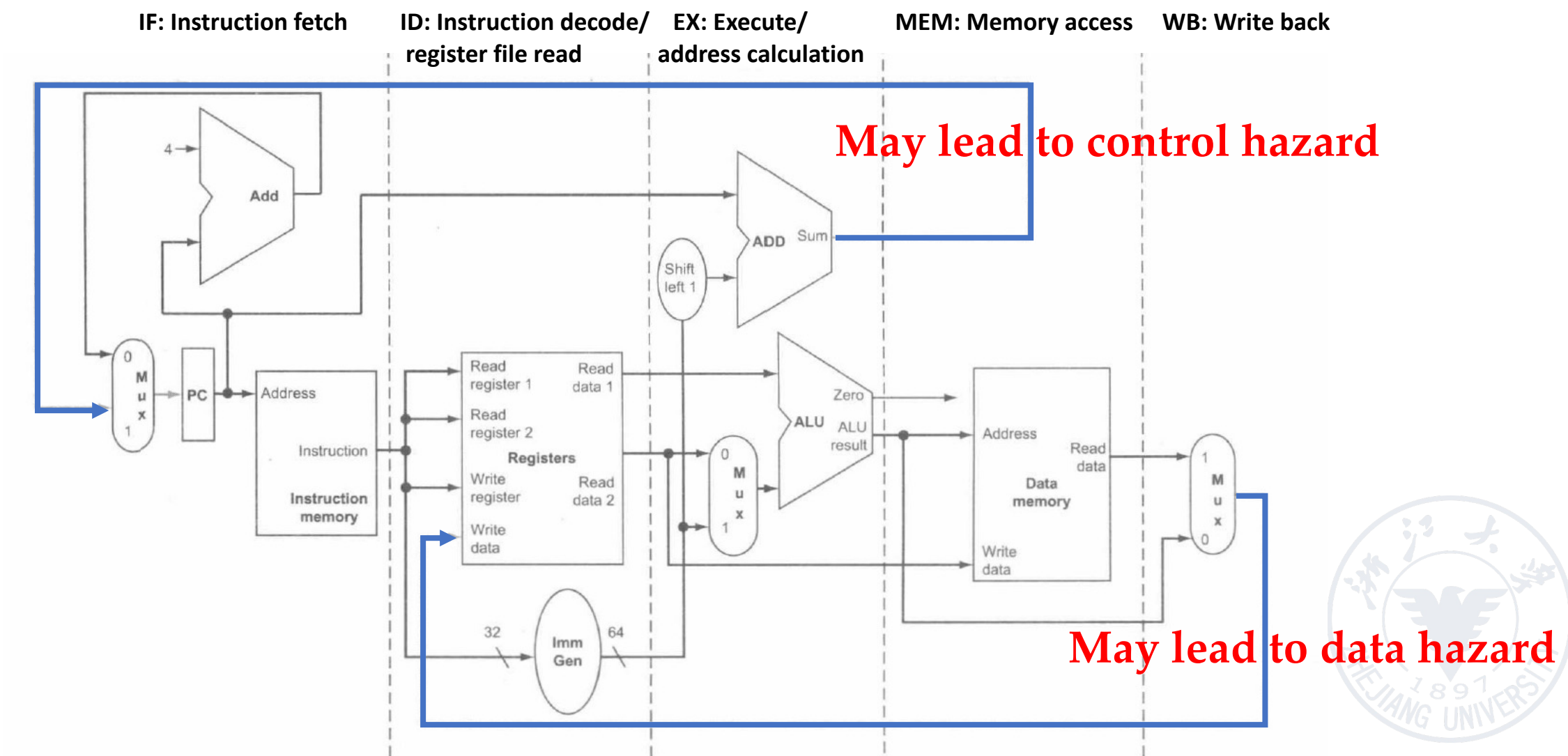
Single-Cycle Datapath

Question: The impacts of the two exceptions



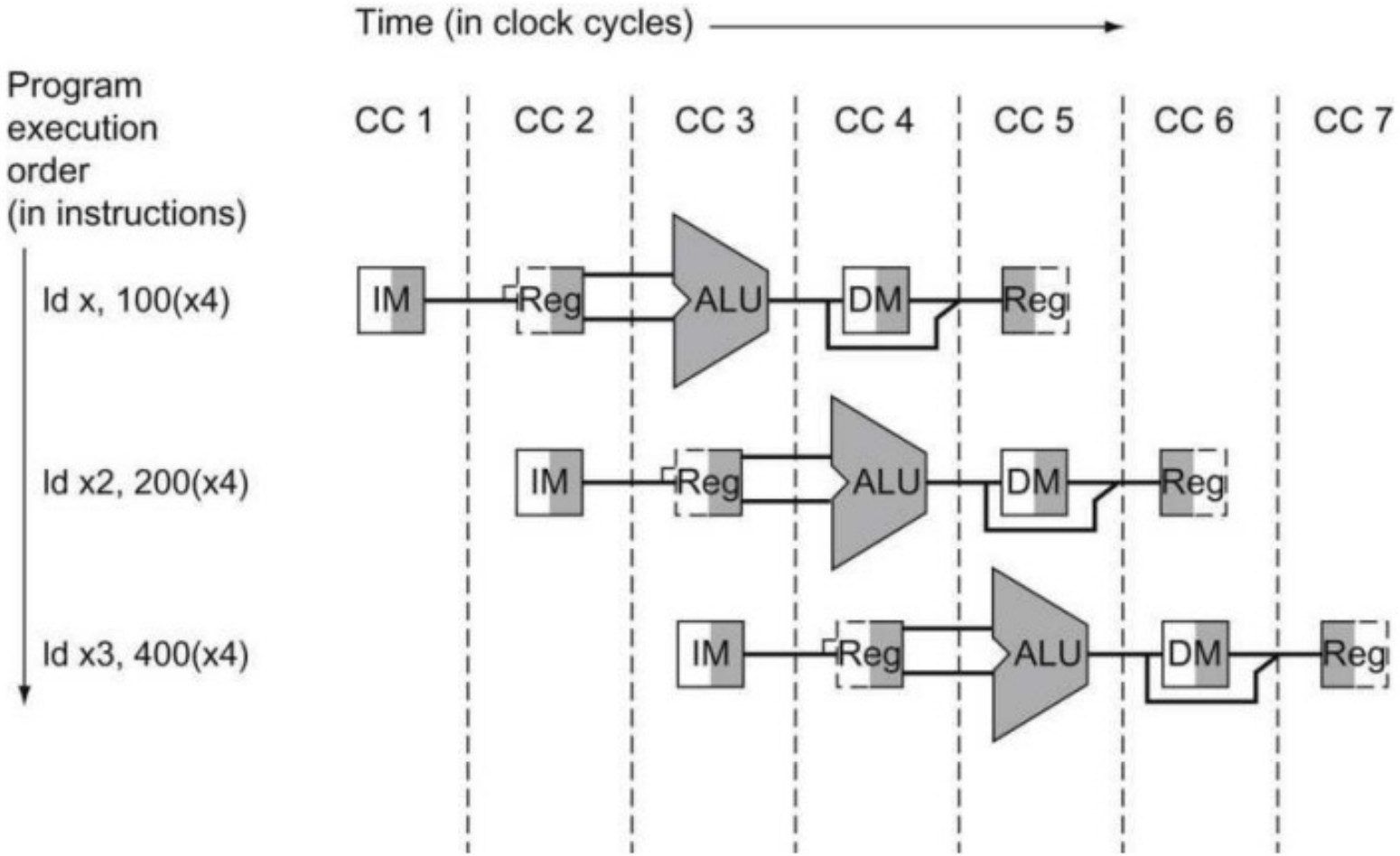
Single-Cycle Datapath

Question: The impacts of the two exceptions

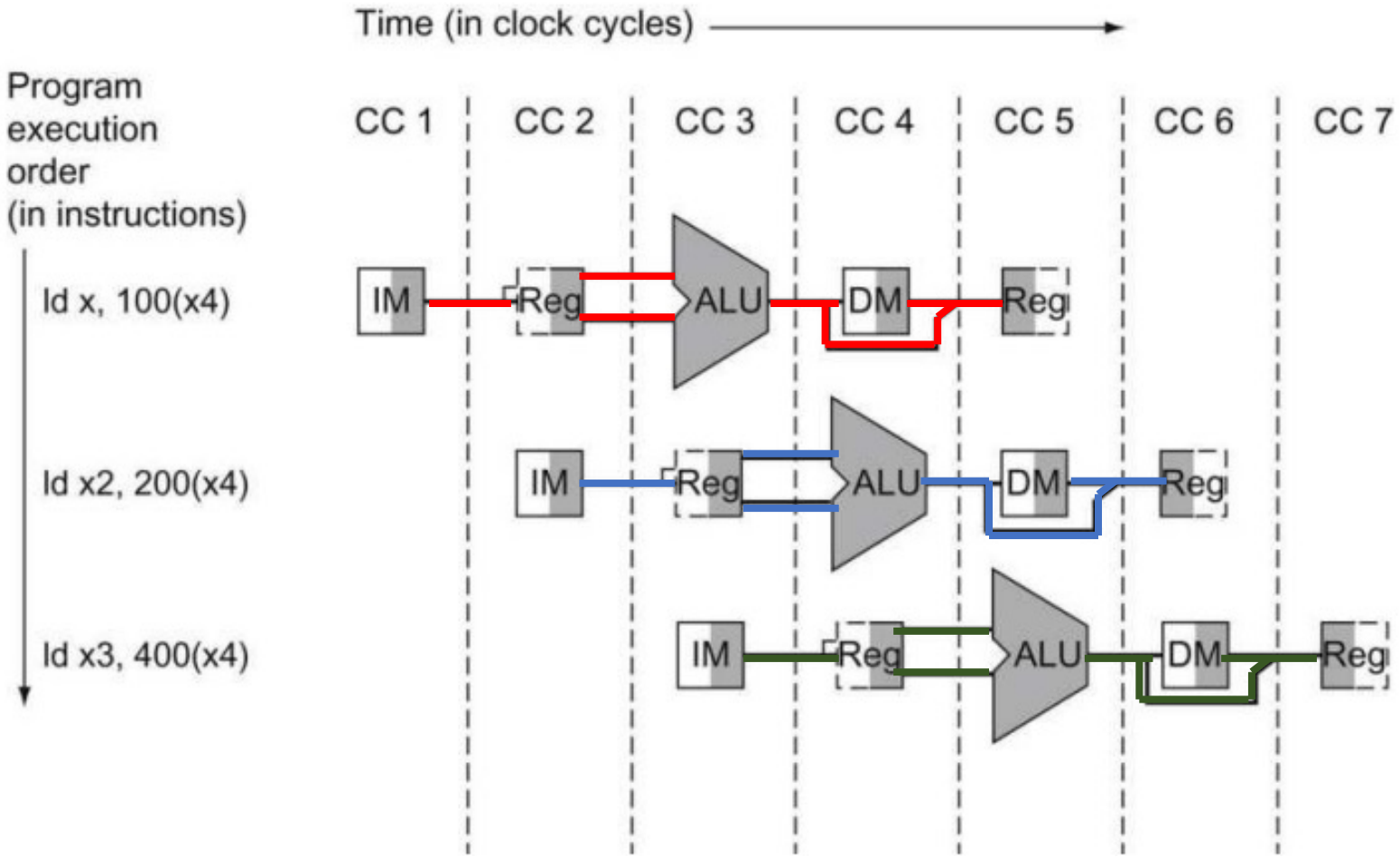


Single-Cycle Datapath to the Pipelined Version

Instructions Being Executed Using the Single-Cycle Datapath

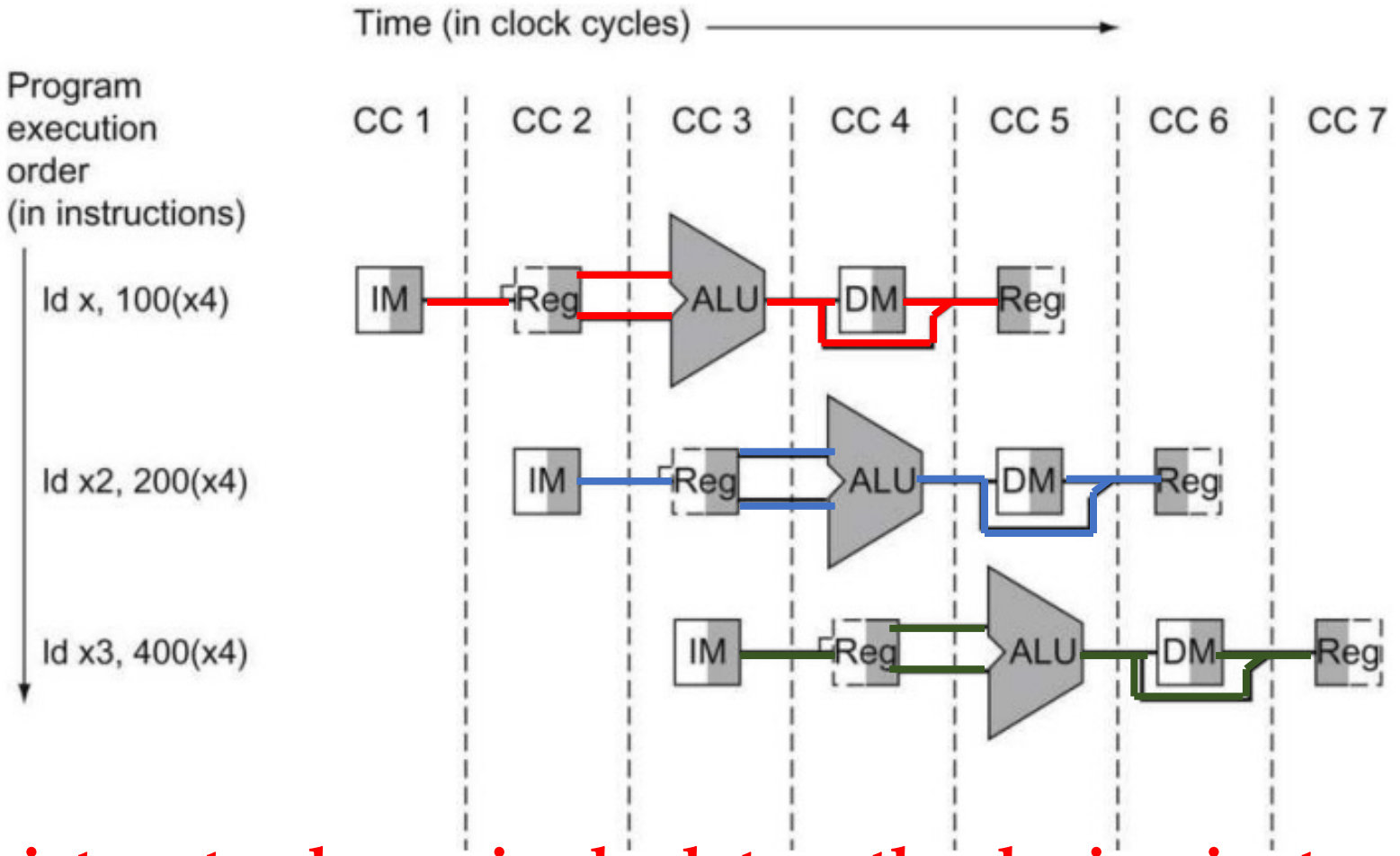


Instructions Being Executed Using the Single-Cycle Datapath



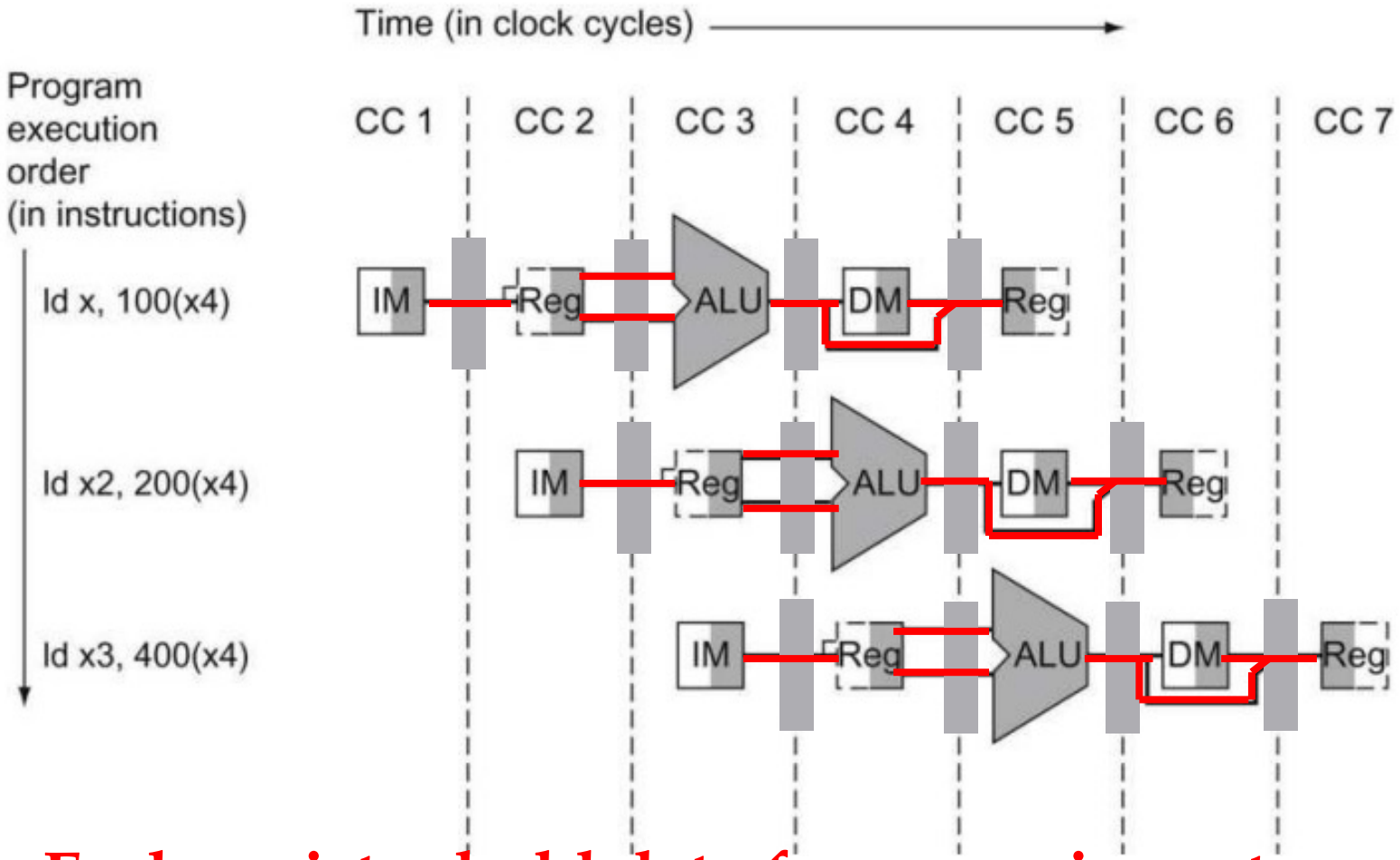
Three instructions need three datapaths

Instructions Being Executed Using the Single-Cycle Datapath



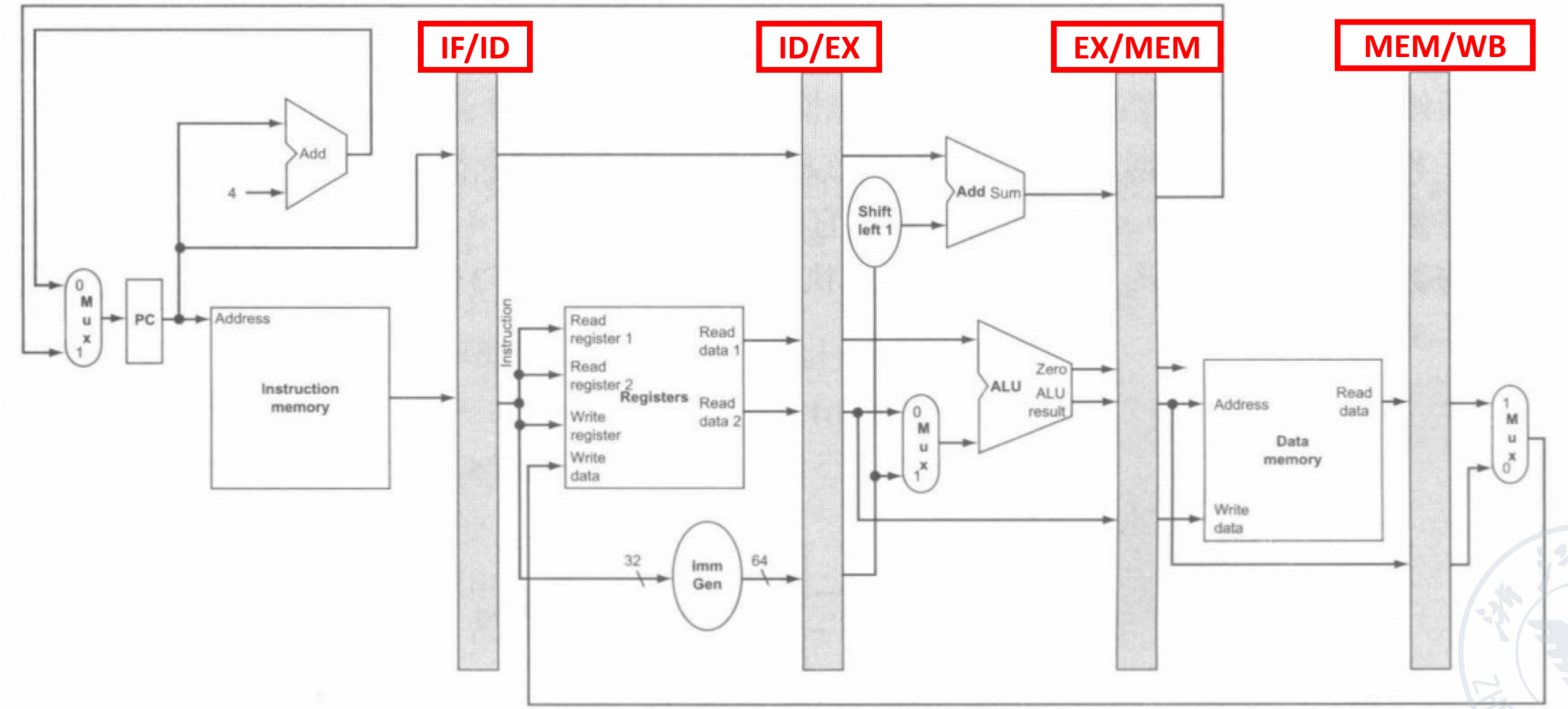
Let's add registers to share single datapaths during instruction execution

Instructions Being Executed Using the Single-Cycle Datapath



Each register hold data from previous stage

Pipelined Version of the Datapath

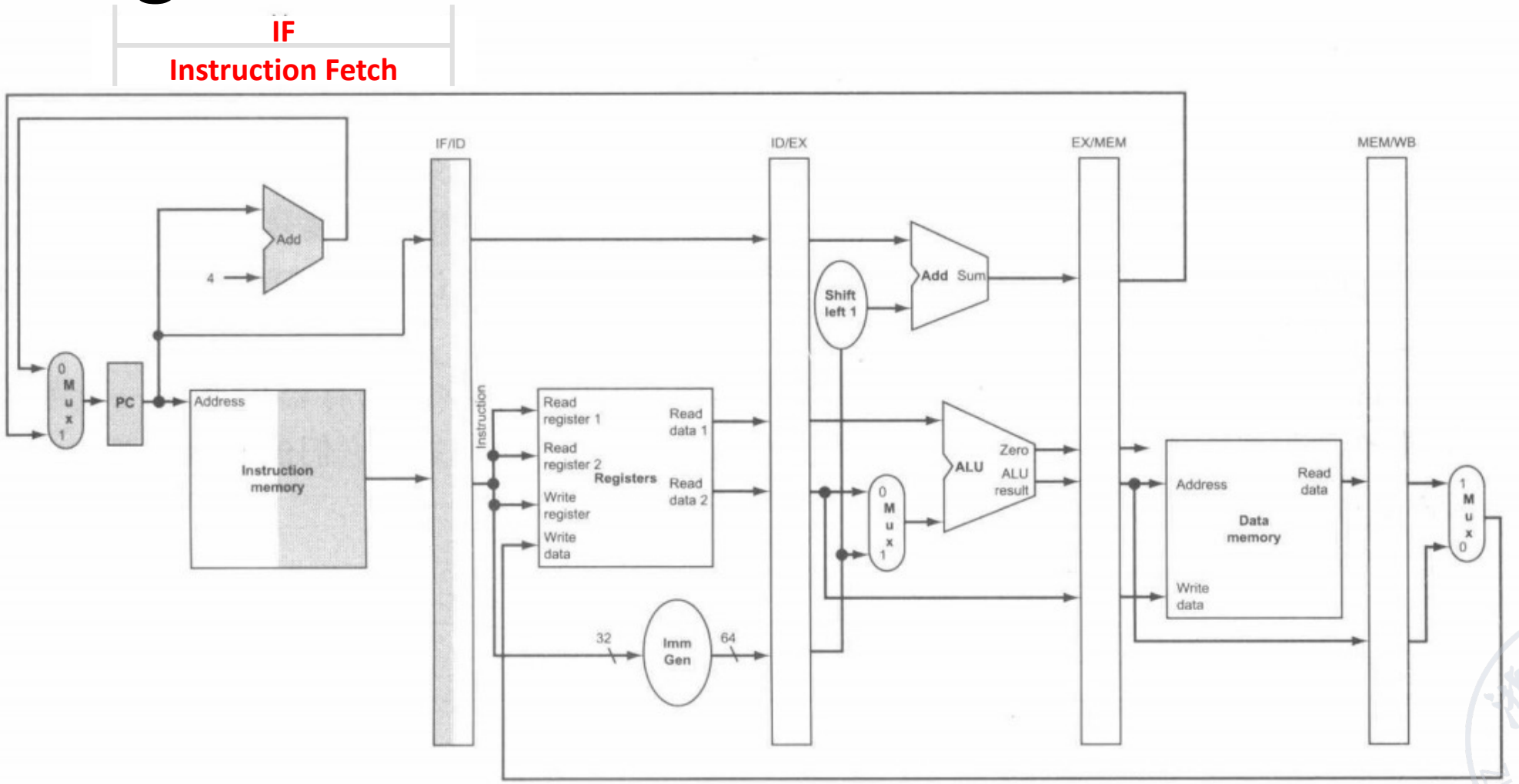


Understand How Pipelined Datapath Work

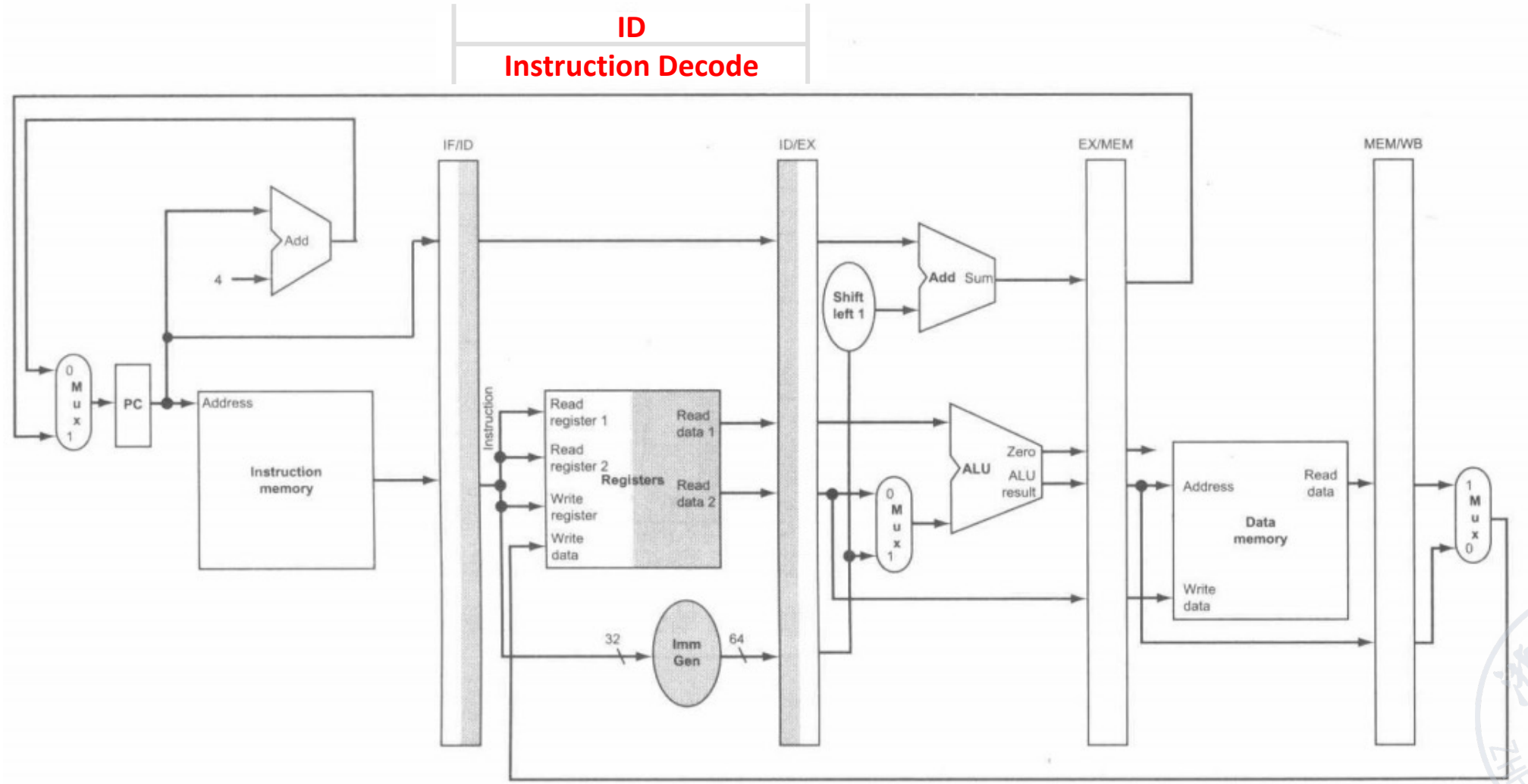
Through Load Instruction and Store Instruction

Load Instruction

IF Stage of Load Instruction

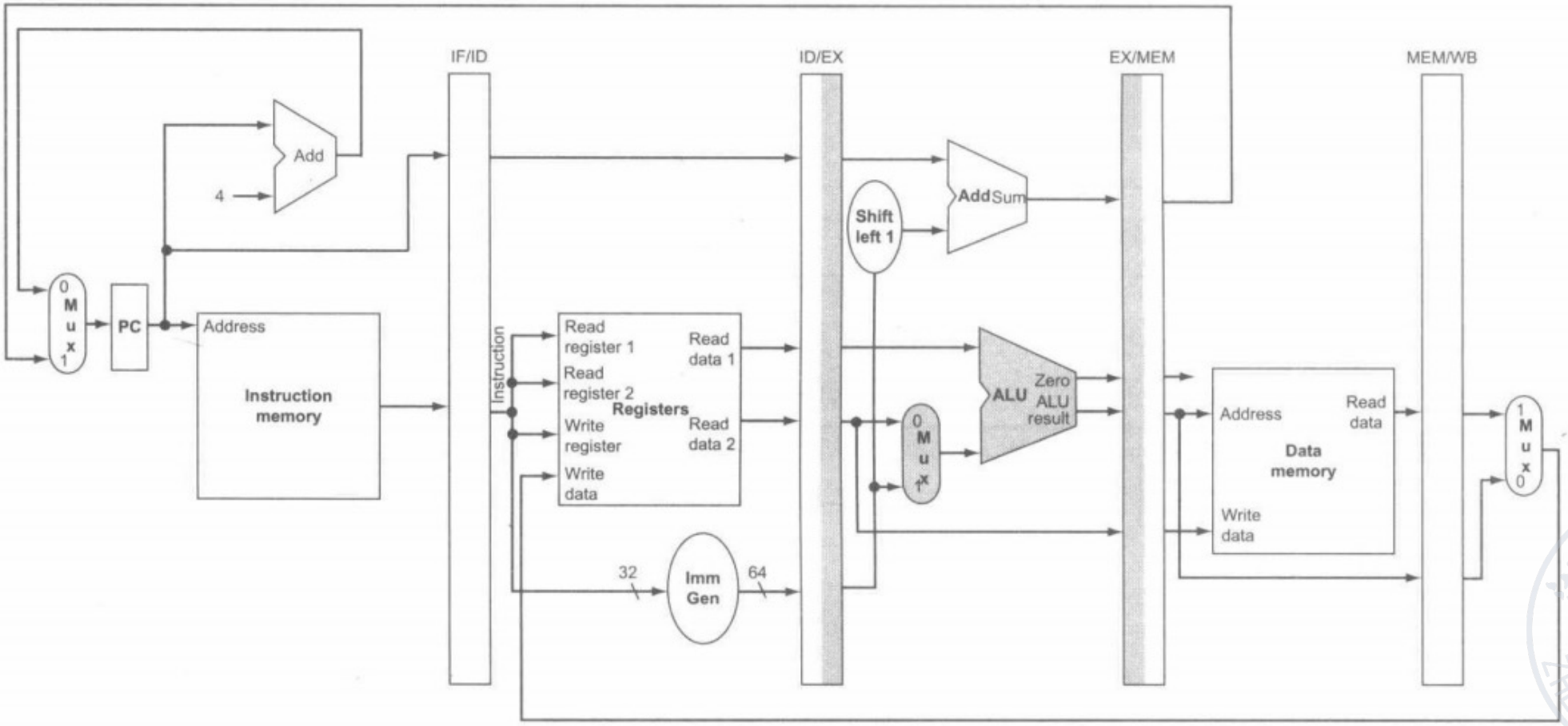


ID Stage of Load Instruction



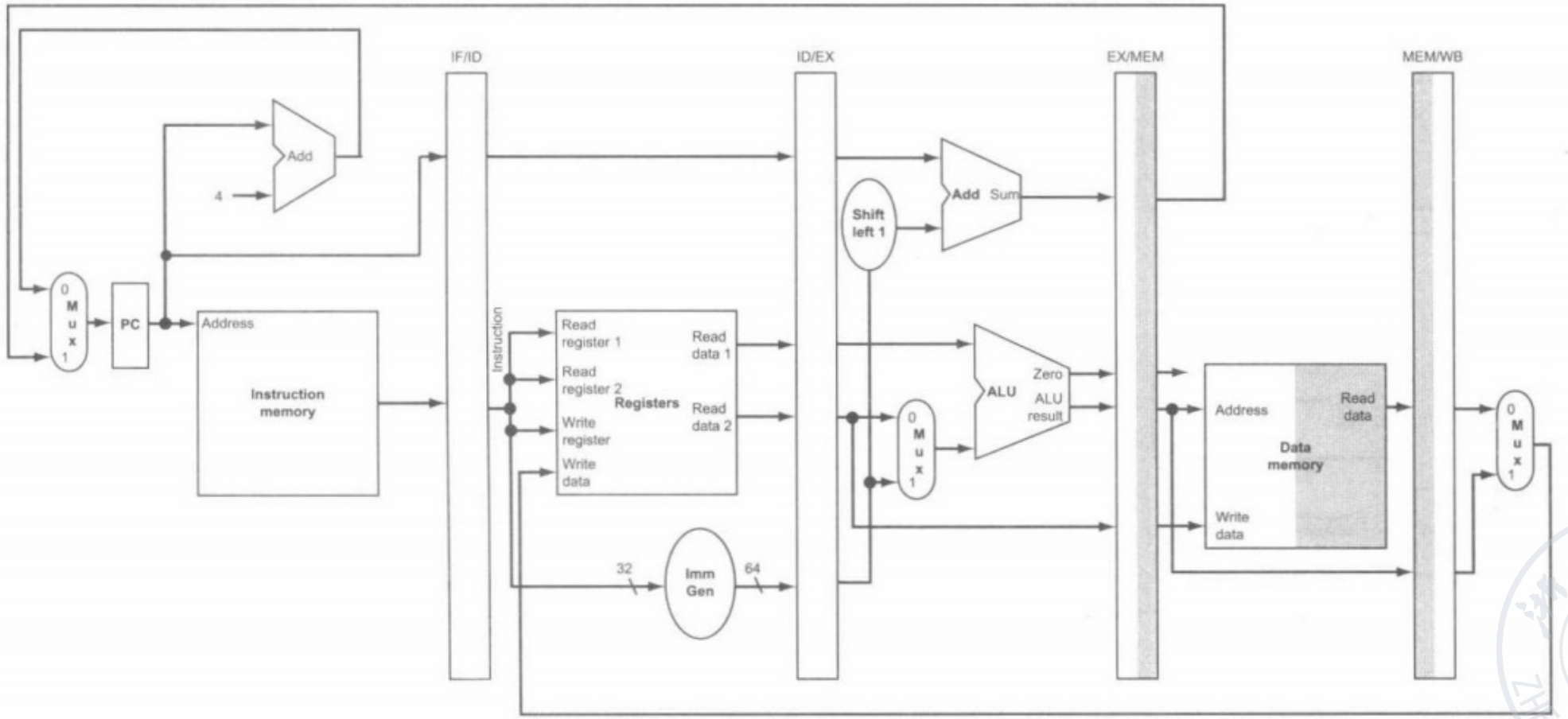
EX Stage of Load Instruction

EX
Execution

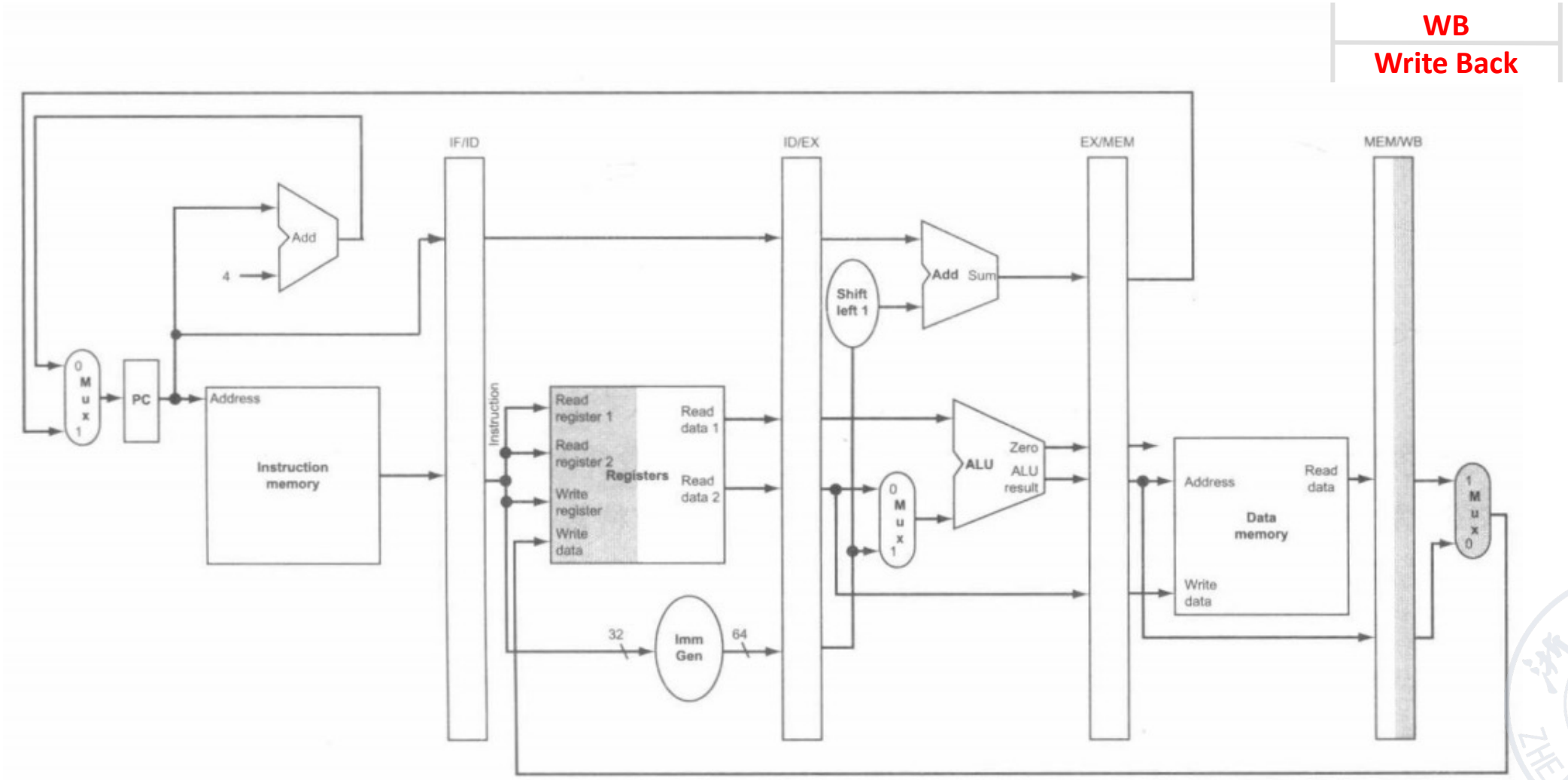


MEM Stage of Load Instruction

MEM
Memory Access

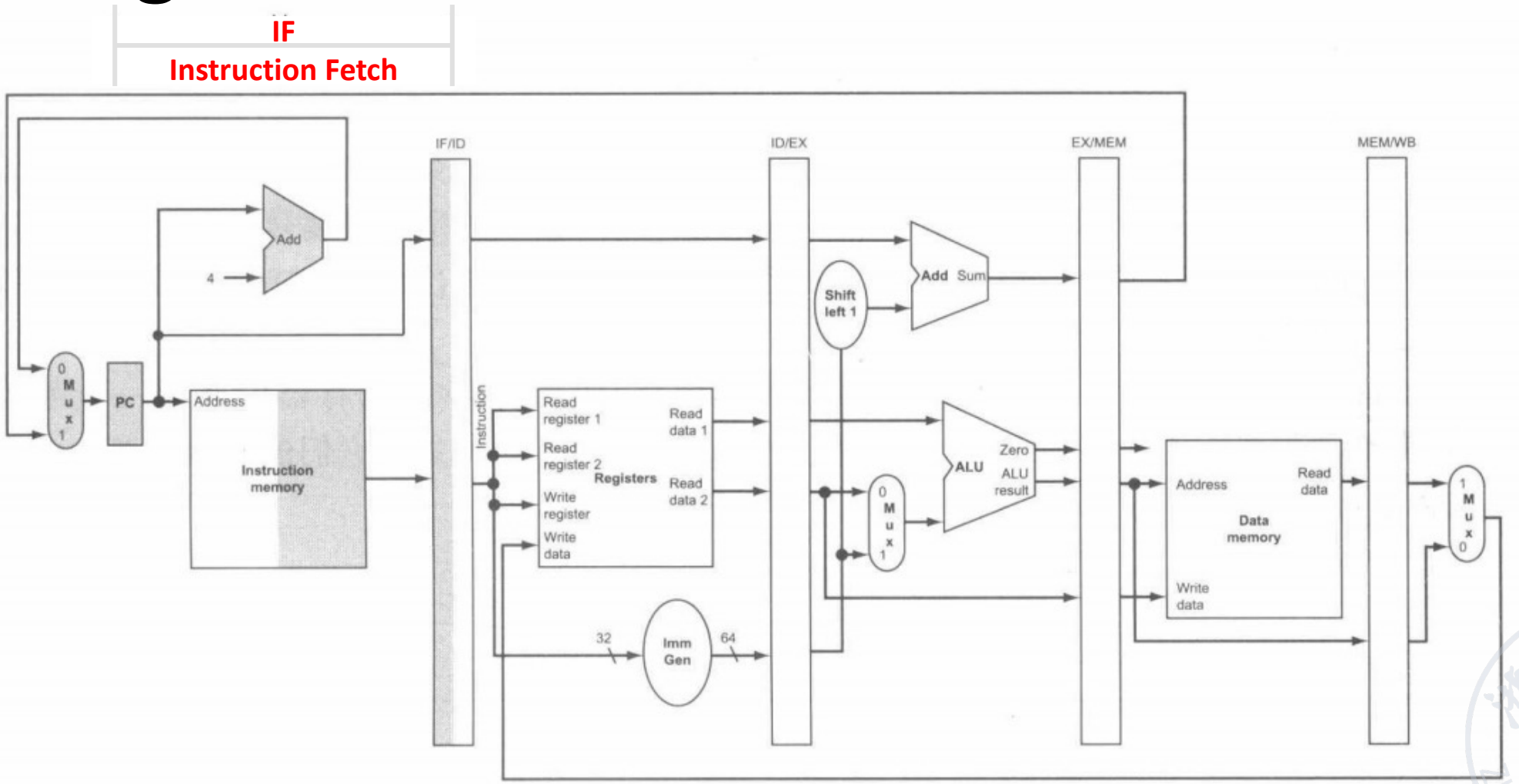


WB Stage of Load Instruction

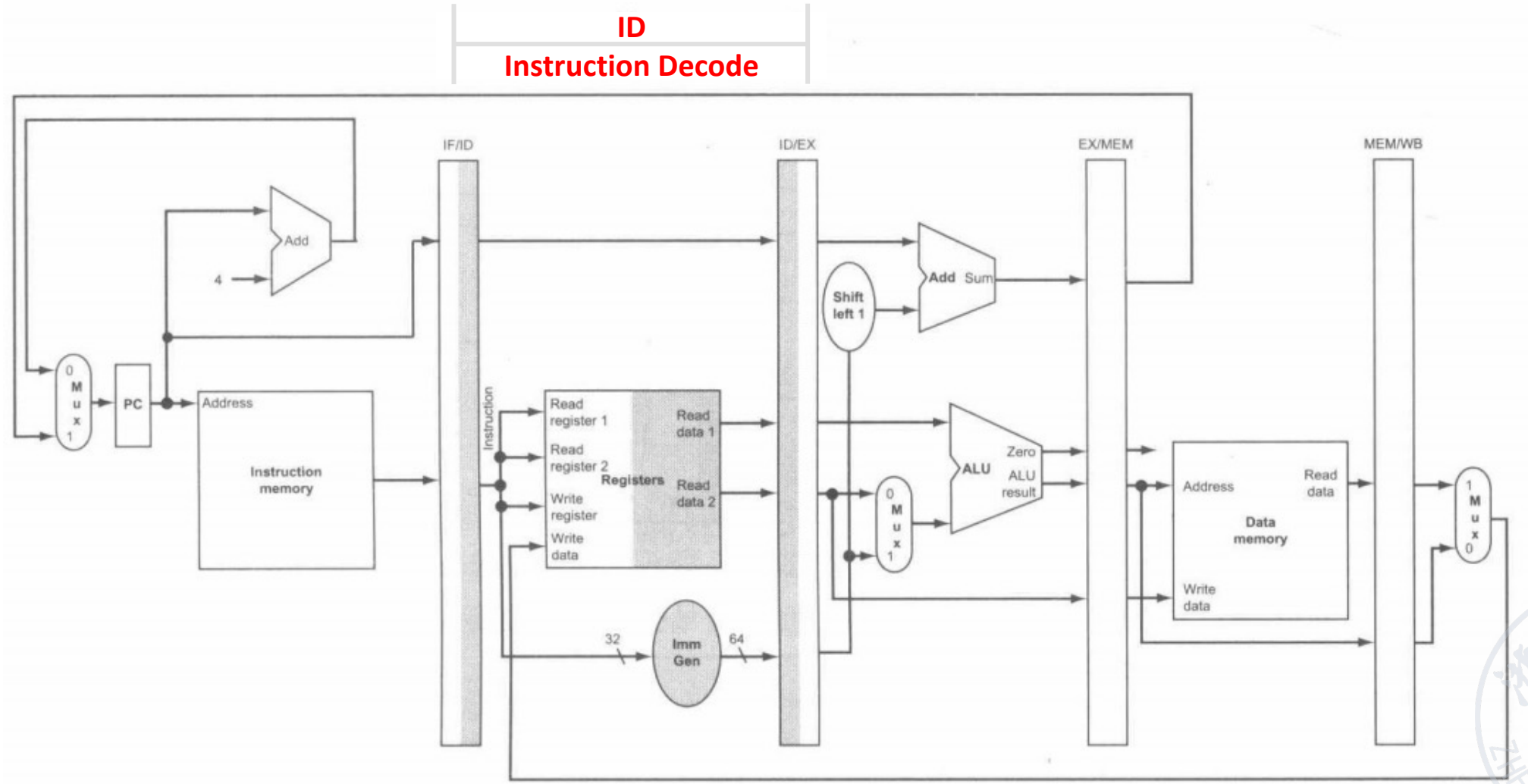


Store Instruction

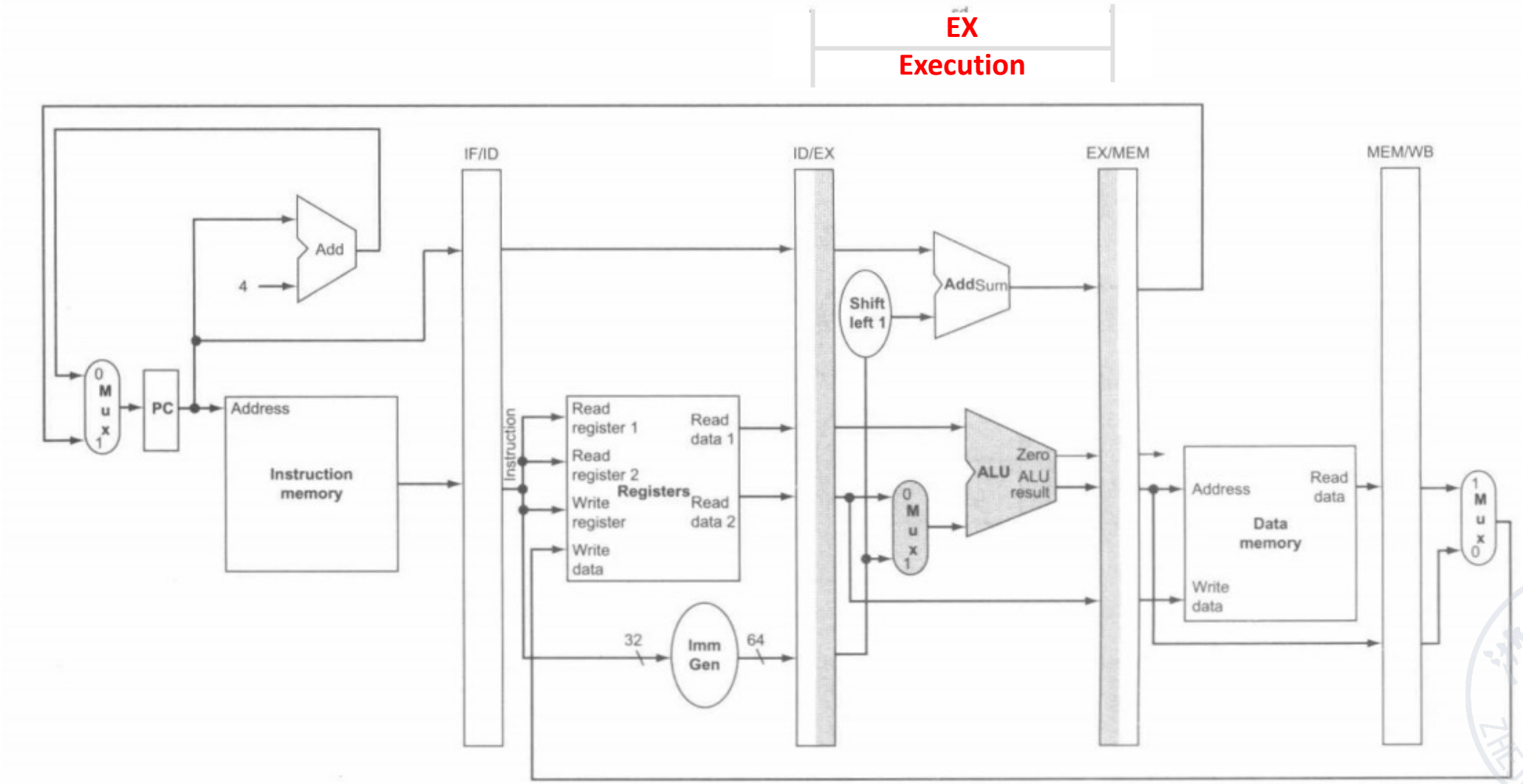
IF Stage of Store Instruction



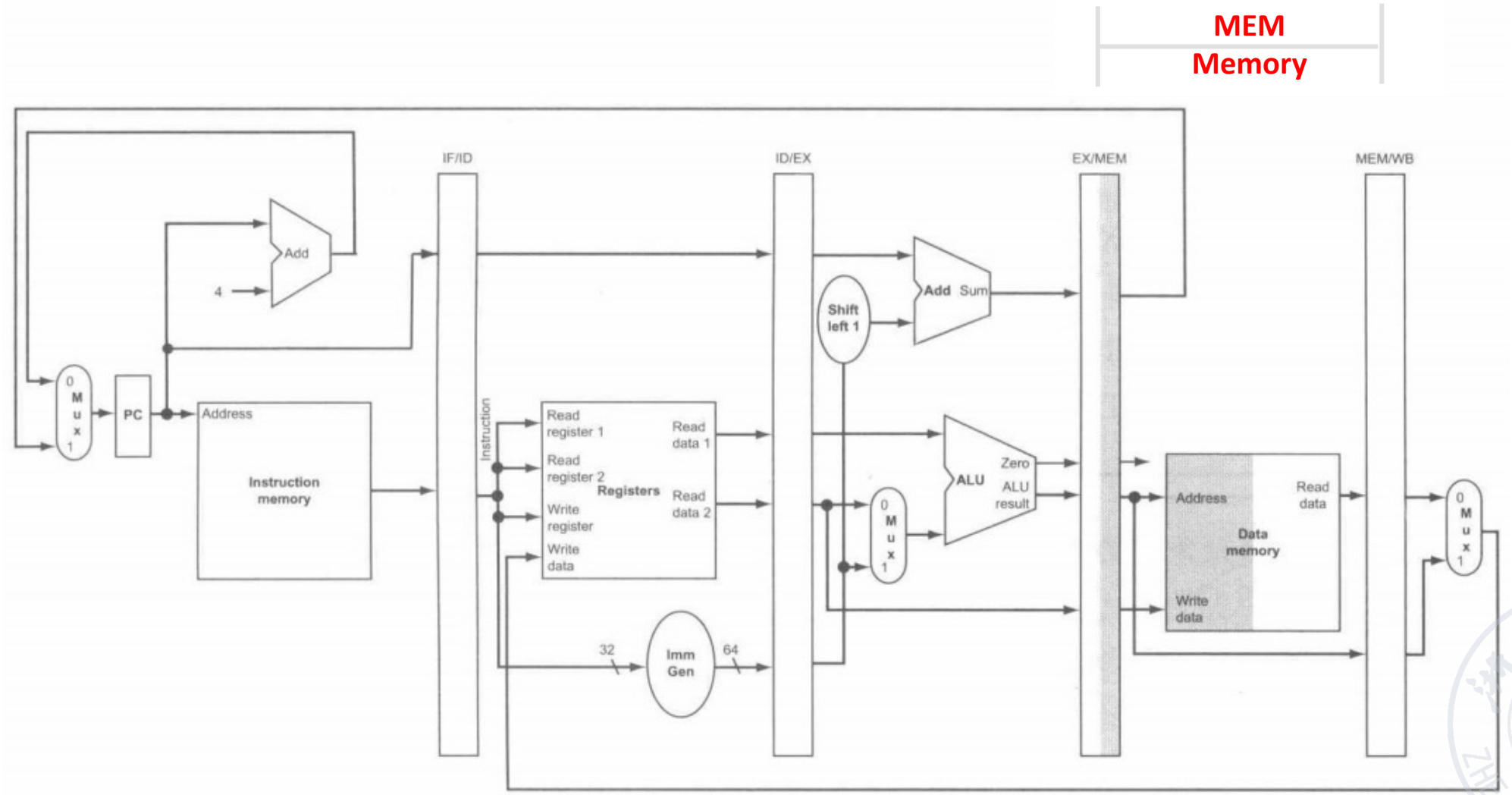
ID Stage of Store Instruction



EX Stage of Store Instruction

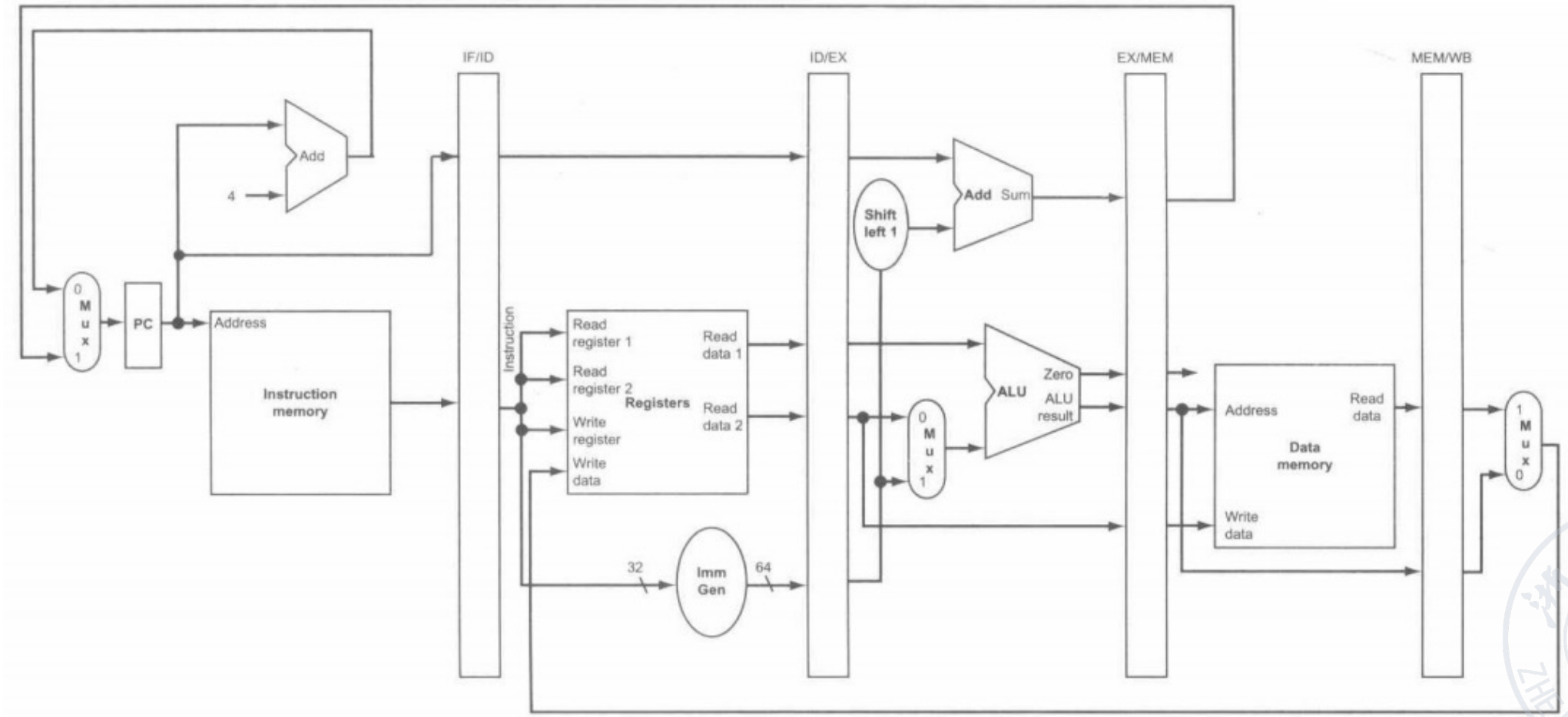


MEM Stage of Store Instruction



WB Stage of Store Instruction

WB
Write Back

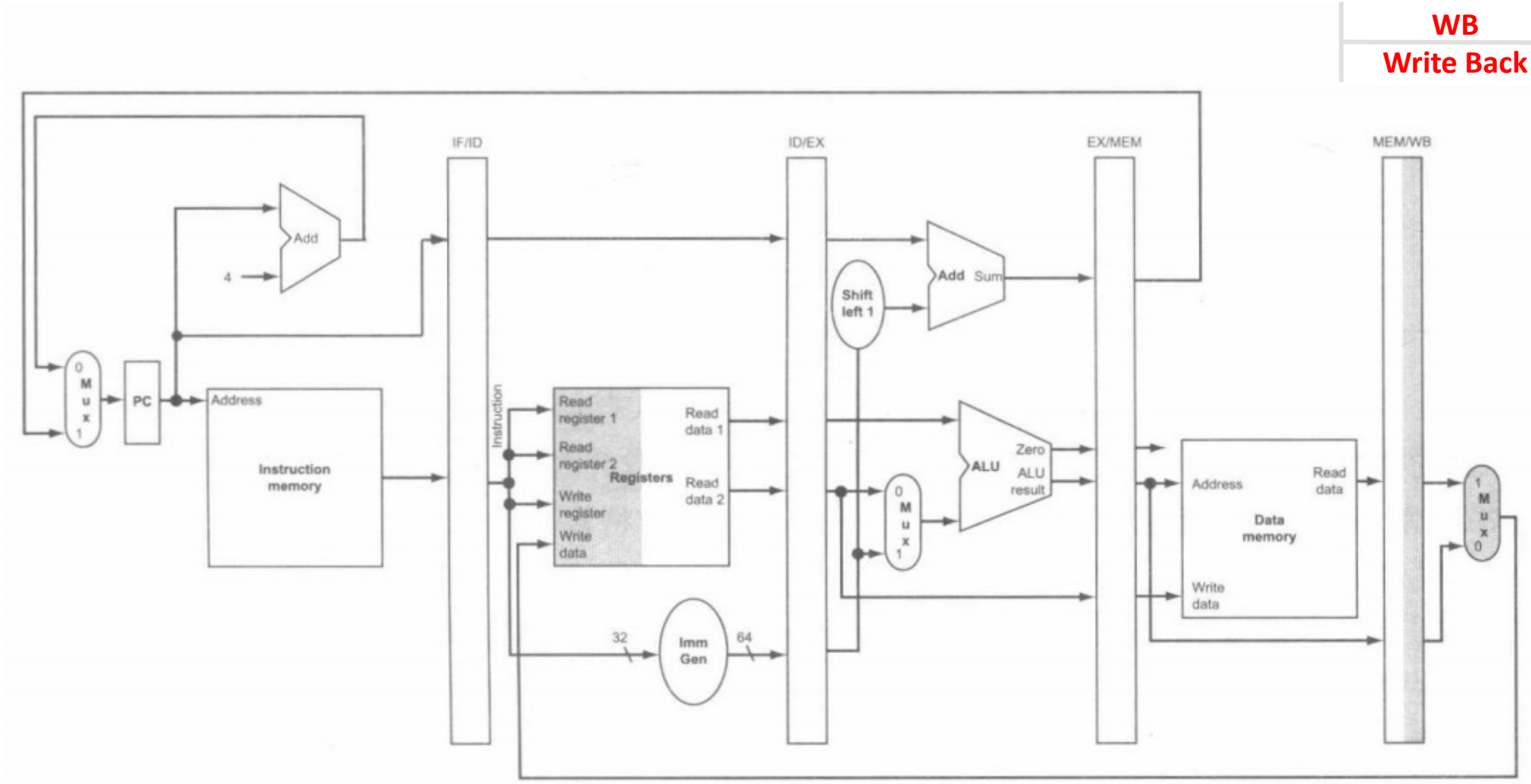


Review of Store Instruction

To pass something from an early pipe stage to a later pipe stage, the information must be placed in a pipeline register

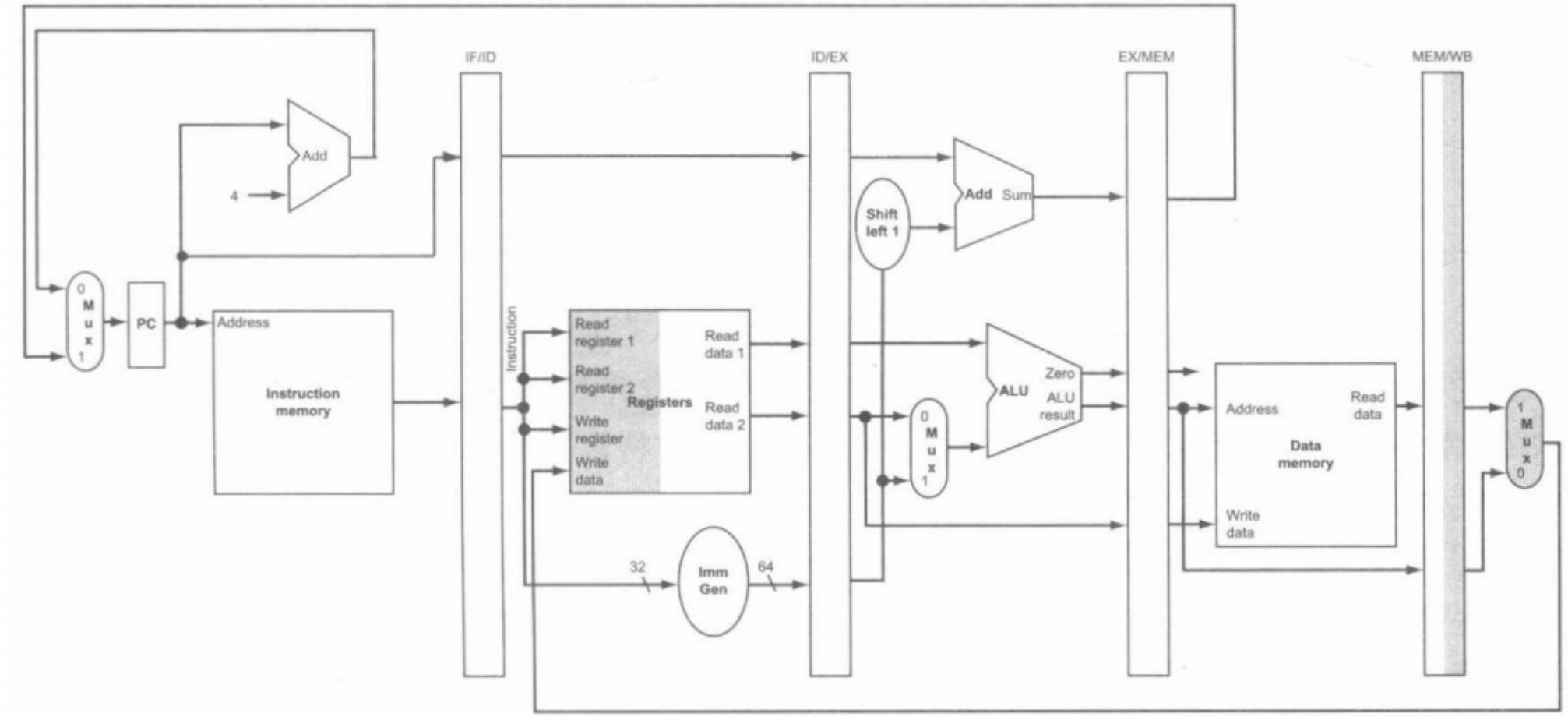
Review of Load Instruction in WB Stage

WB
Write Back



Review of Load Instruction in WB Stage

WB
Write Back



Let's uncover a bug in the design of the load instruction

Review the Load Instruction in WB Stage

Write Back to the Register

- **MEM/WB** — Supplies data
- **ID/EX Register** — Supplies the write register number

Review the Load Instruction in WB Stage

Write Back to the Register

- **MEM/WB** — Supplies data
- **ID/EX Register** — Supplies the write register number

Question: Is it the right write register number?

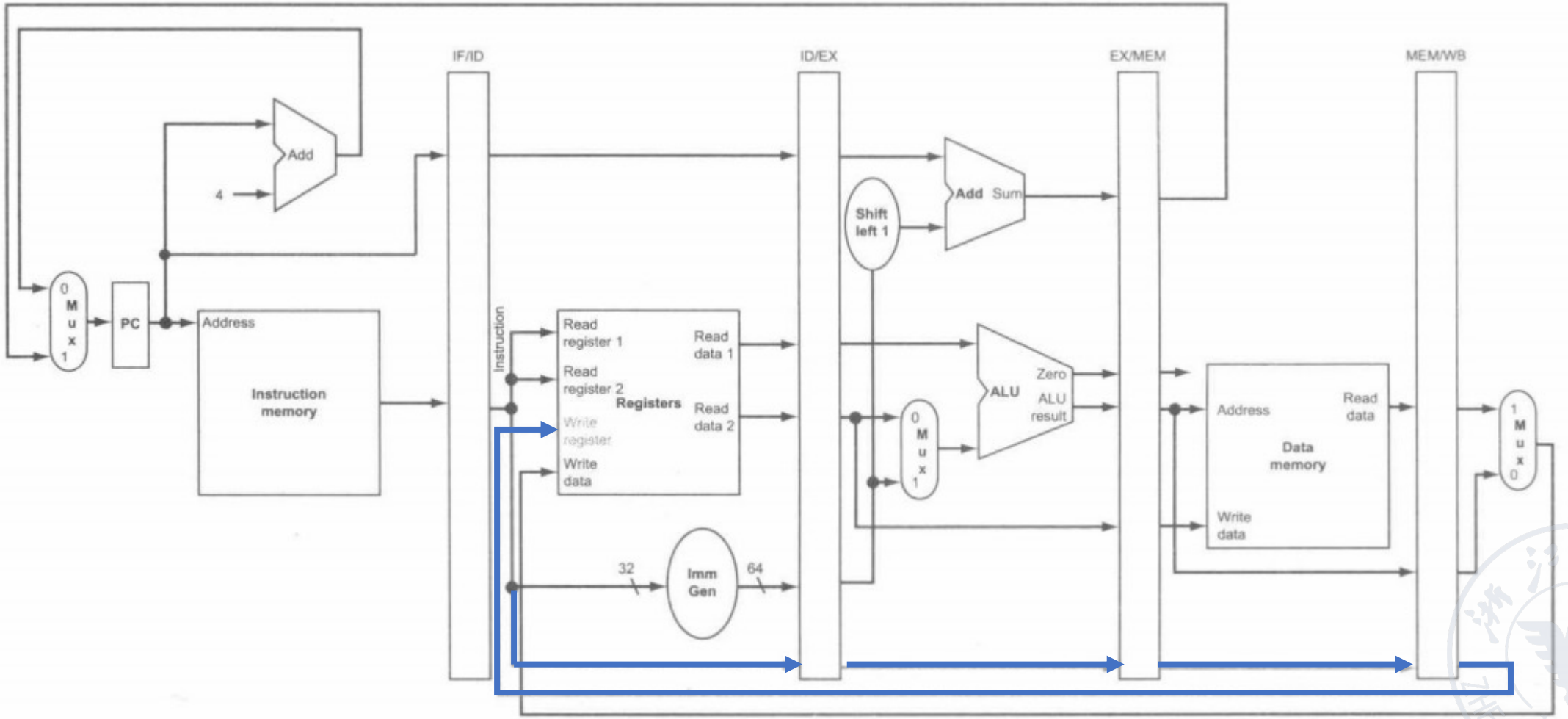
Review the Load Instruction in WB Stage

Write Back to the Register

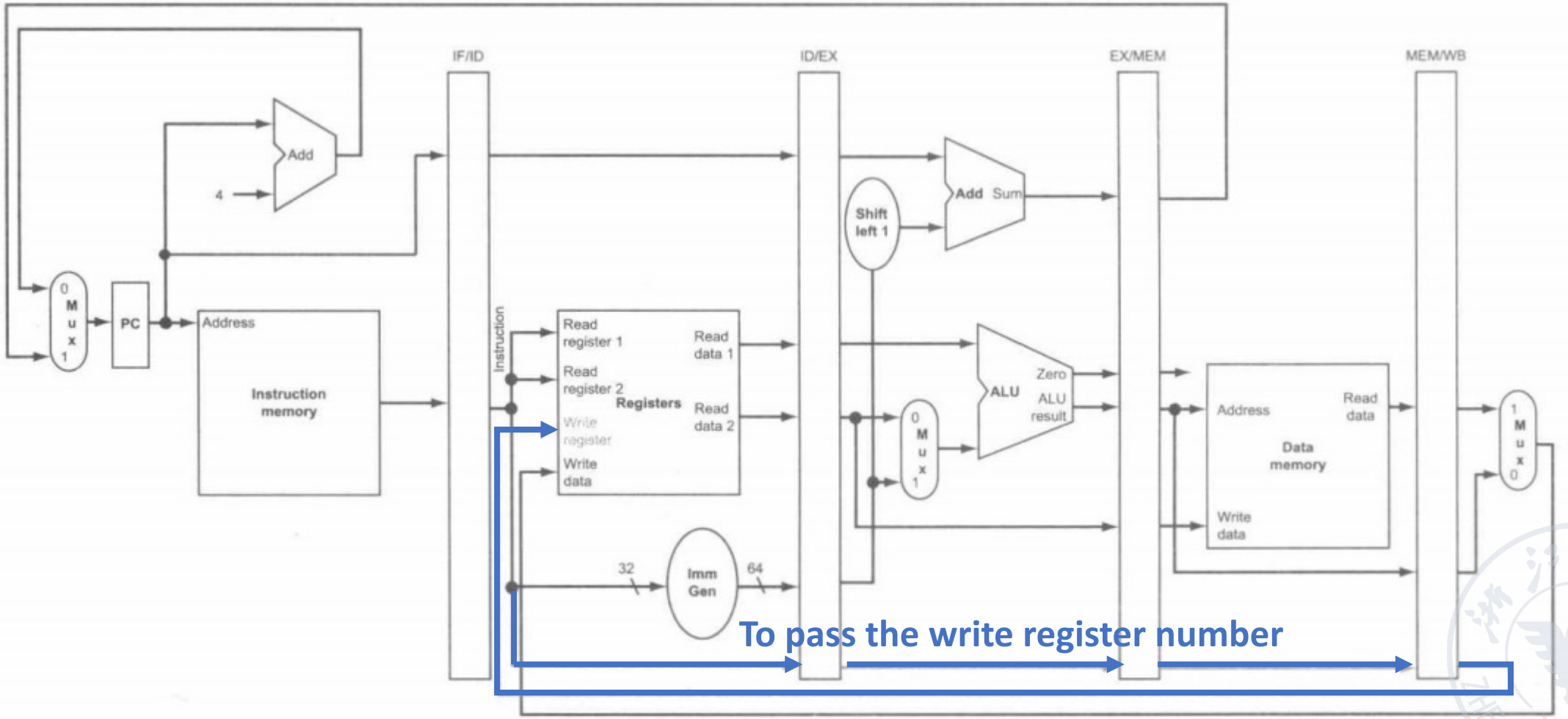
- **MEM/WB** — Supplies data
- **ID/EX Register** — Supplies the [illegible]

ID/EX Register has been modified by another instruction

The Corrected Pipelined Datapath to Handle the Load Instruction Properly

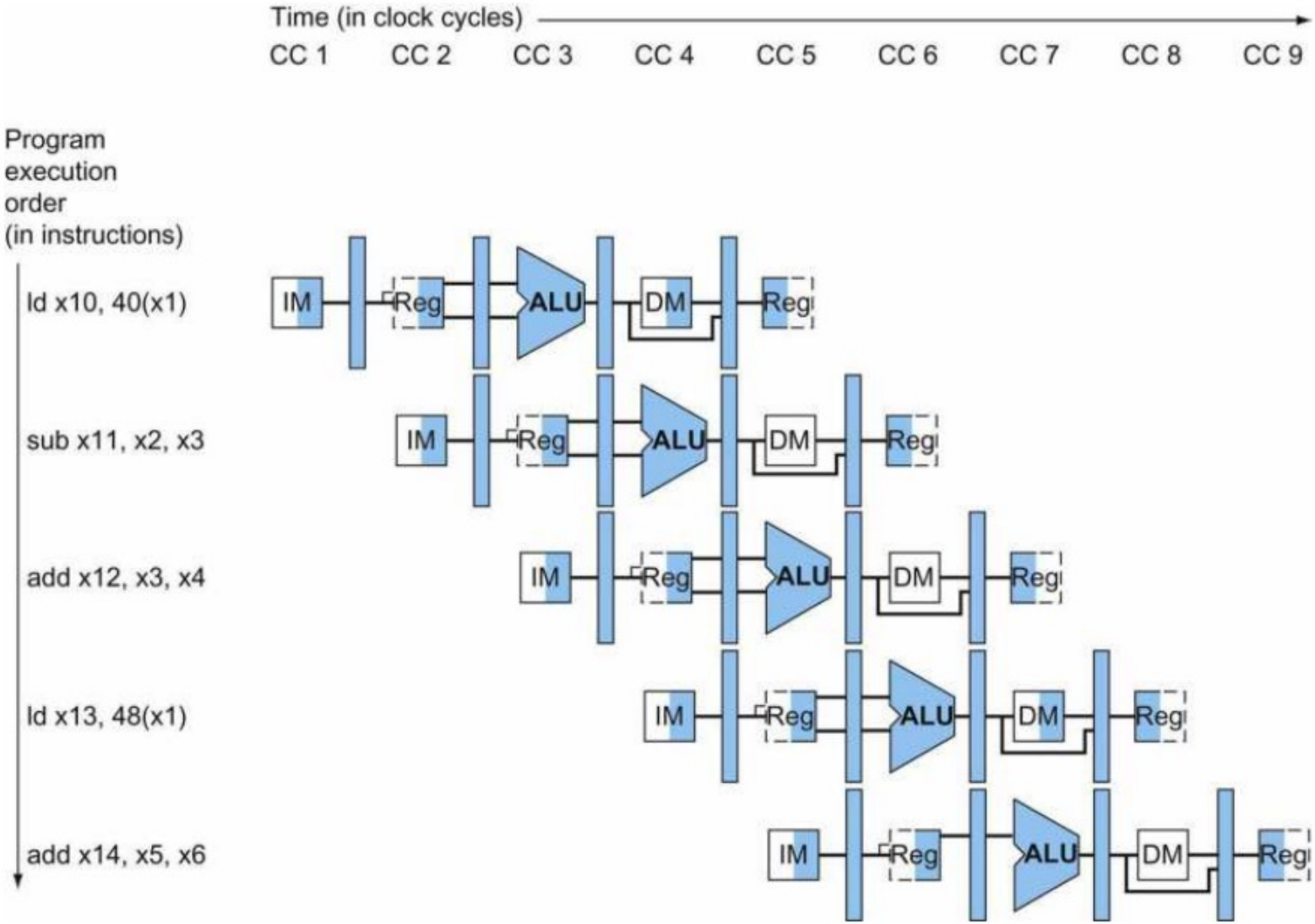


The Corrected Pipelined Datapath to Handle the Load Instruction Properly

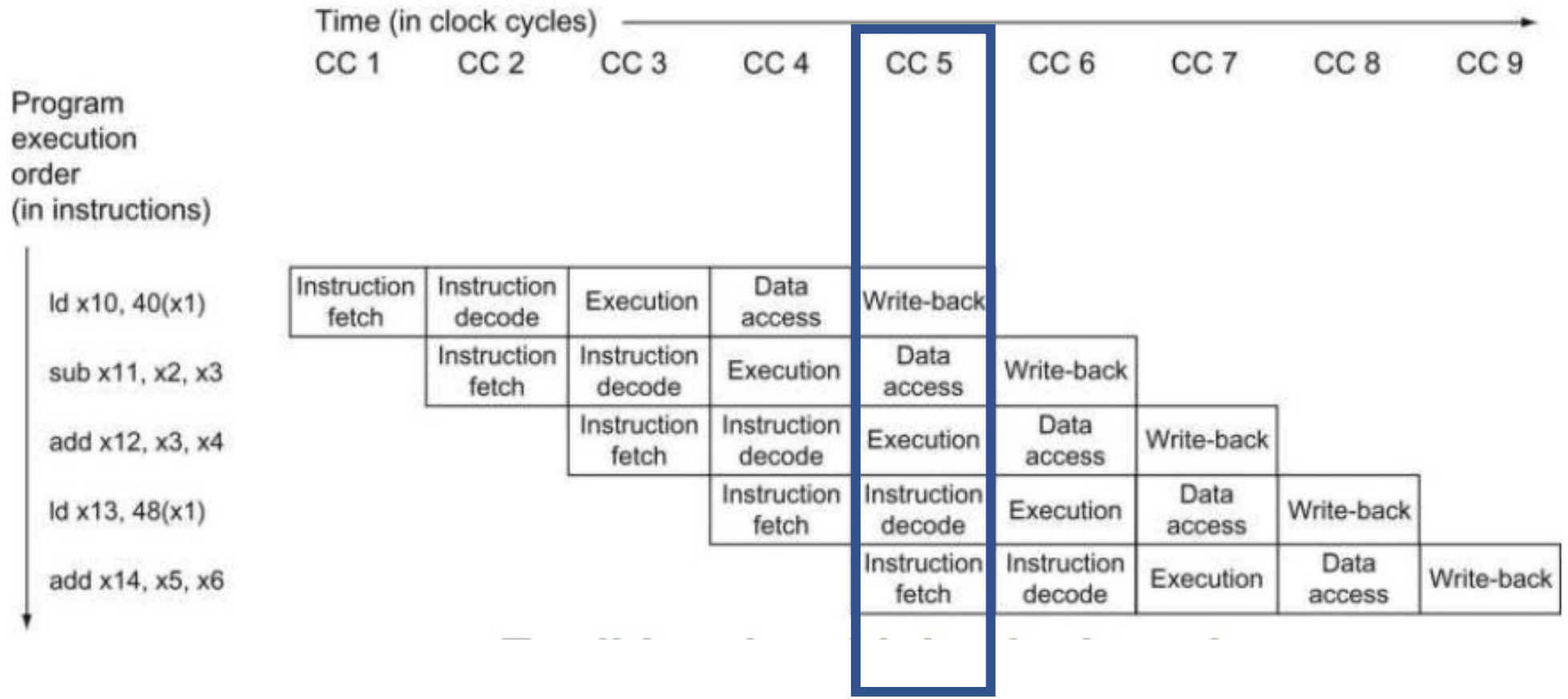


Graphically Representing Pipelines

Multiple-Clock-Cycle Pipeline Diagram of Five Instructions

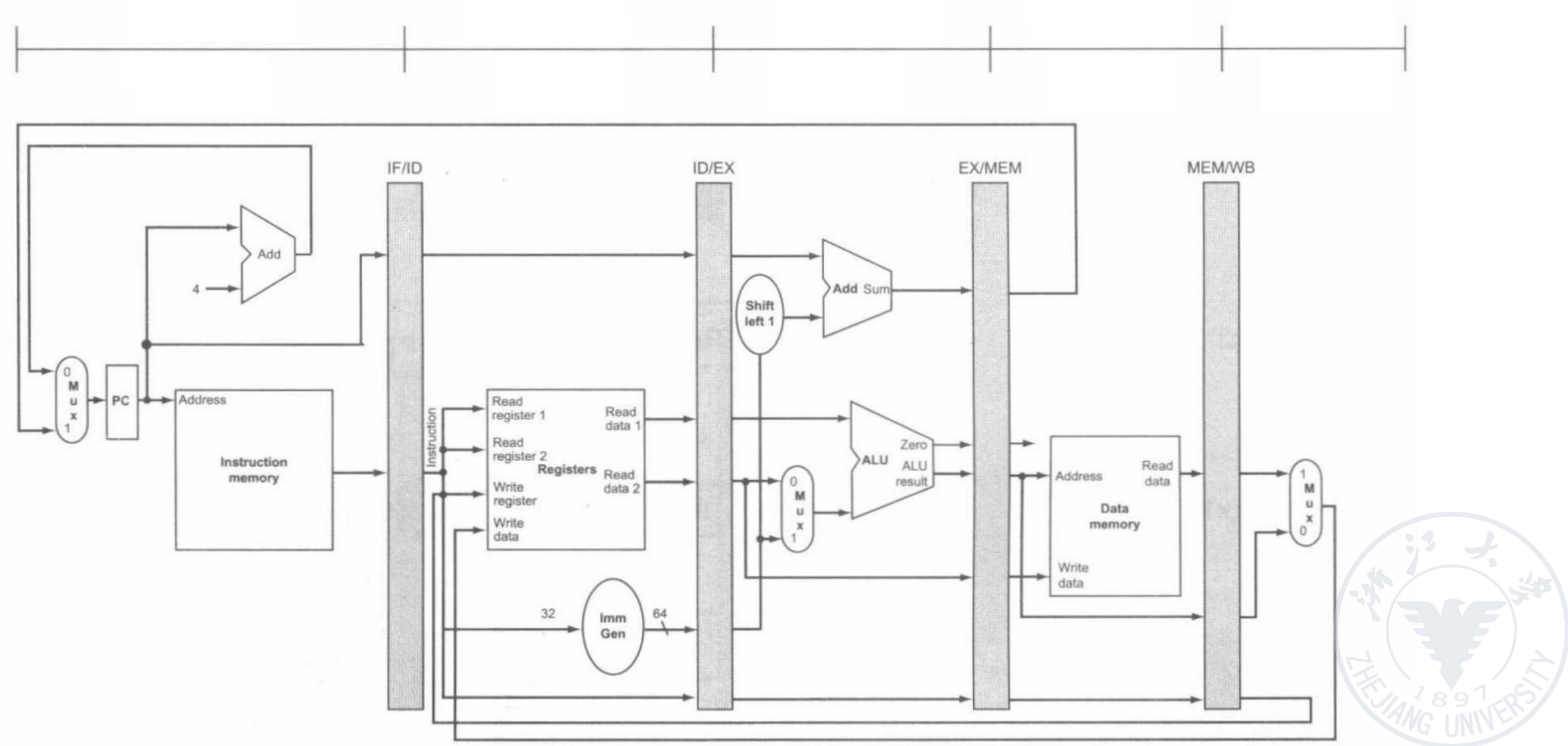


Traditional Multiple-Clock-Cycle Pipeline Diagram



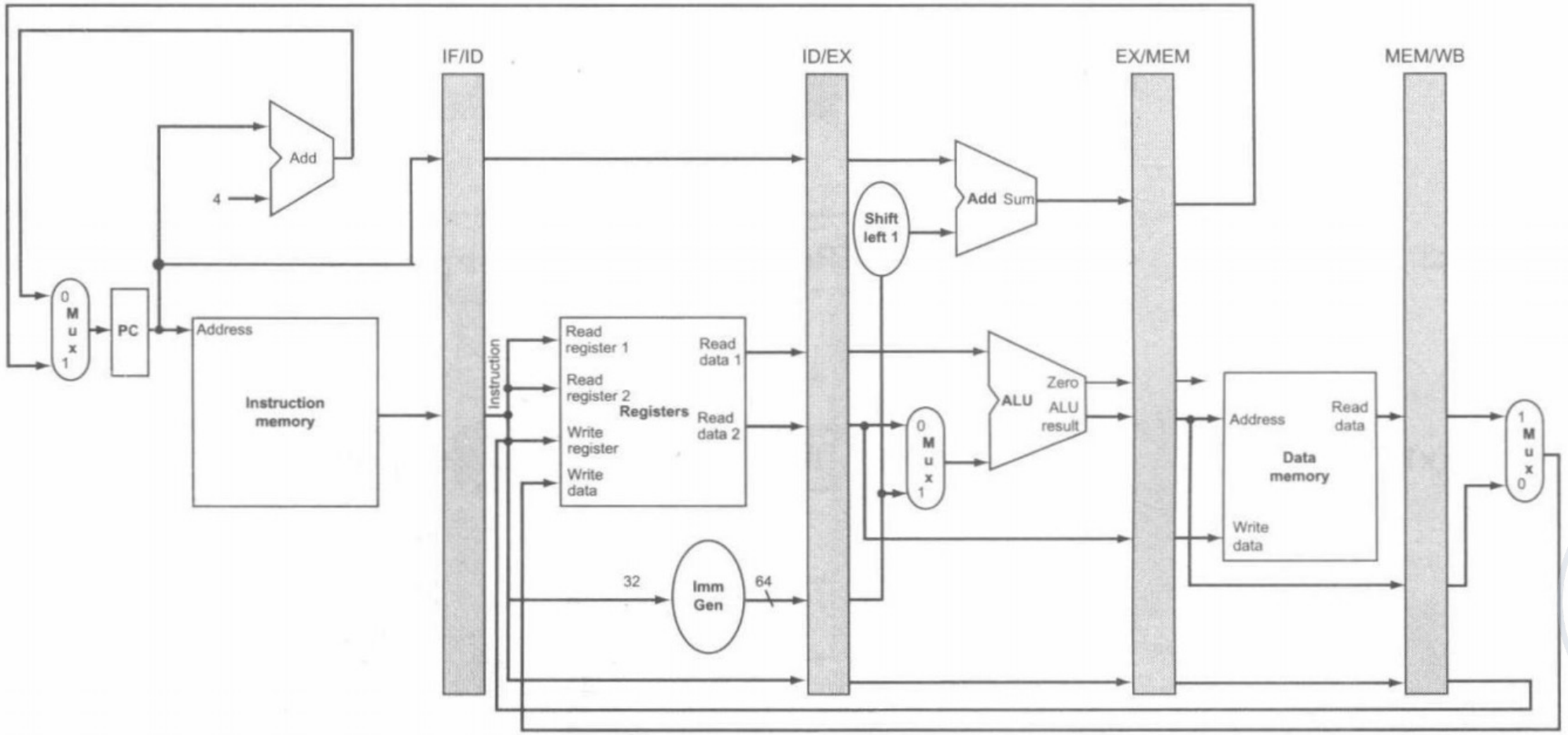
Now draw the Single-clock-cycle pipeline diagram at CC5

Single-Clock-Cycle Pipeline Diagram at CC5



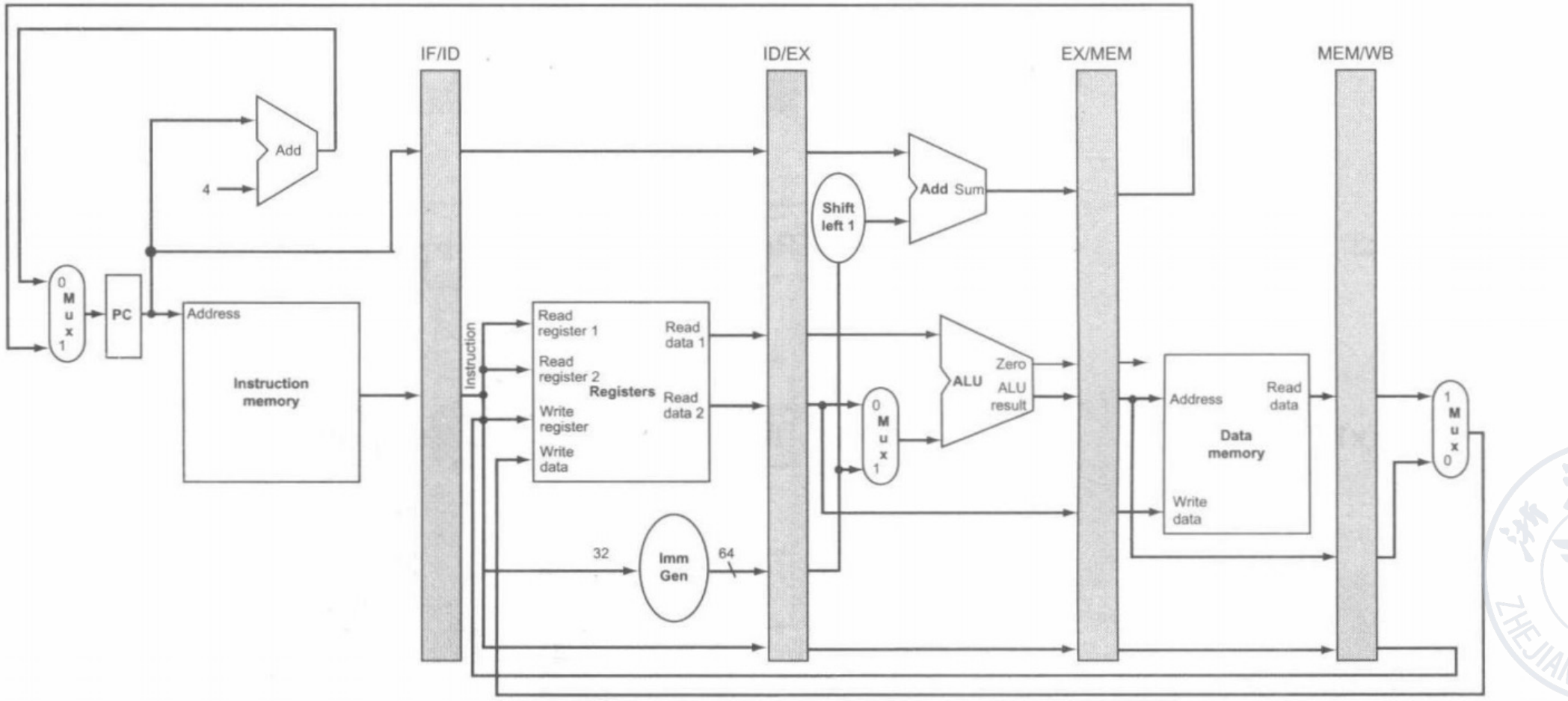
Single-Clock-Cycle Pipeline Diagram at CC5

Question: Write name and stage of instructions



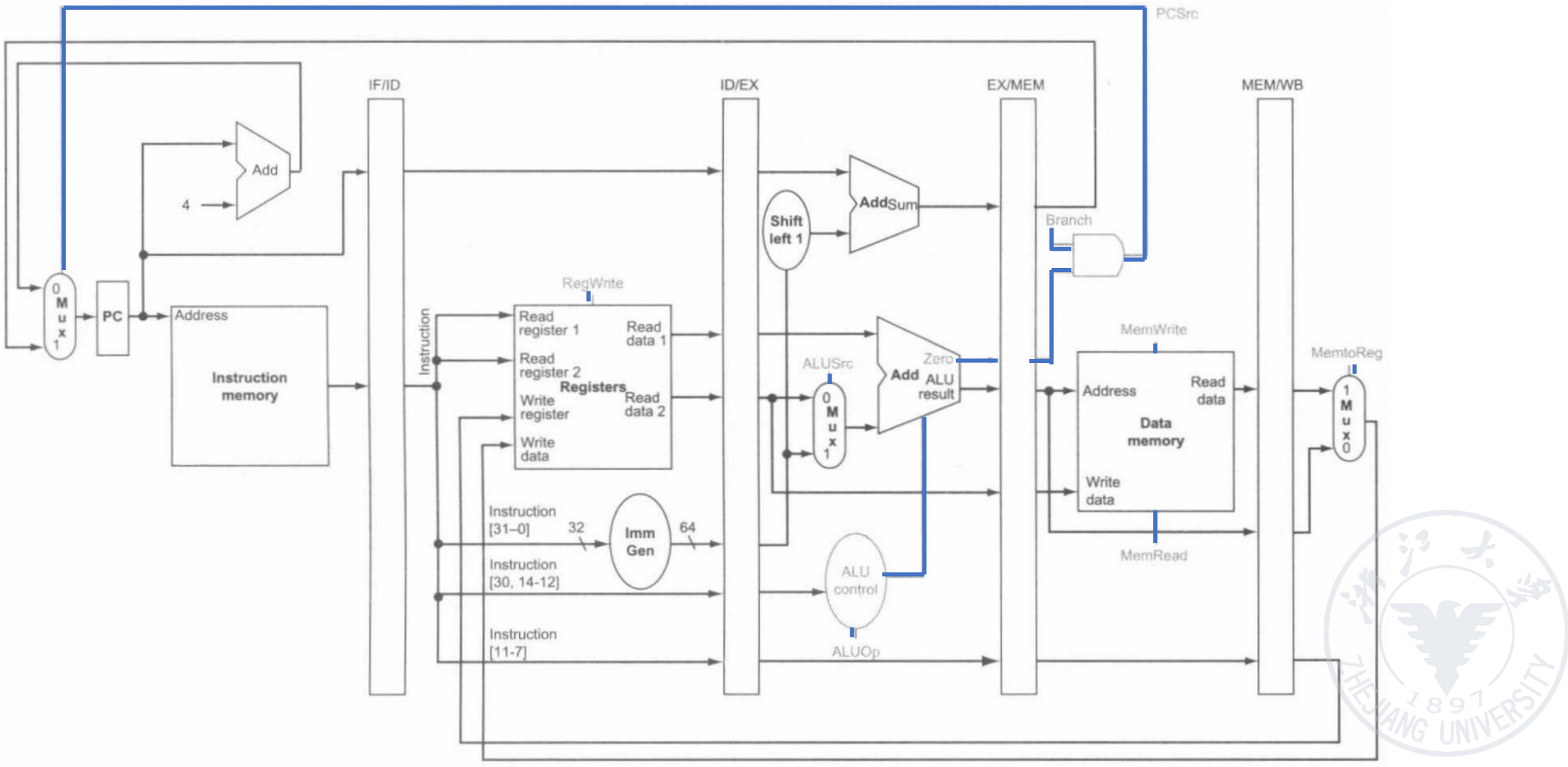
Single-Clock-Cycle Pipeline Diagram at CC5

add x14,x5,x6	ld x13,48(x1)	add x12,x3,x4	sub x11,x2,x3	ld x10,40(x1)
Instruction Fetch	Instruction Decode	Execution	Memory	Write Back

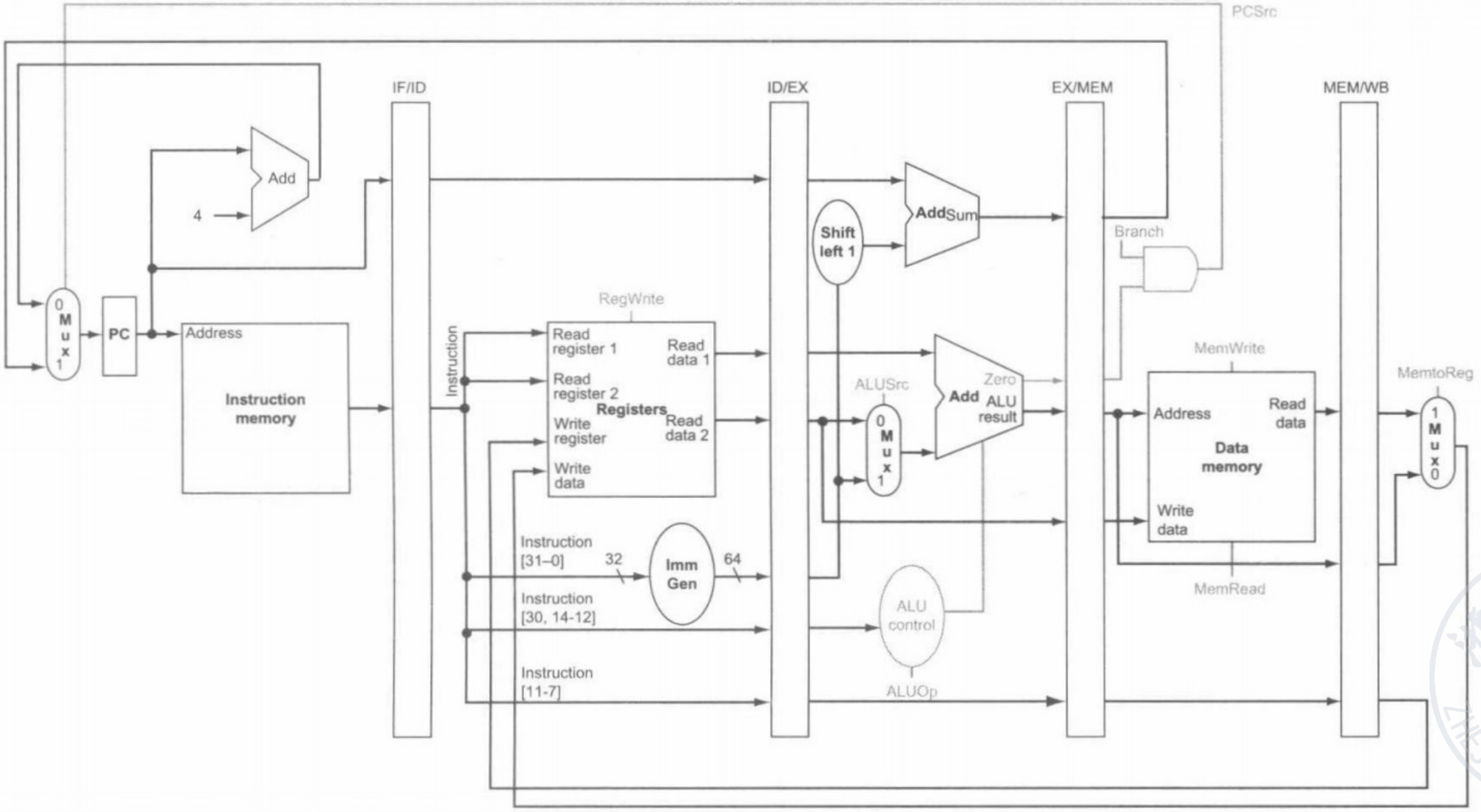


Pipelined Control

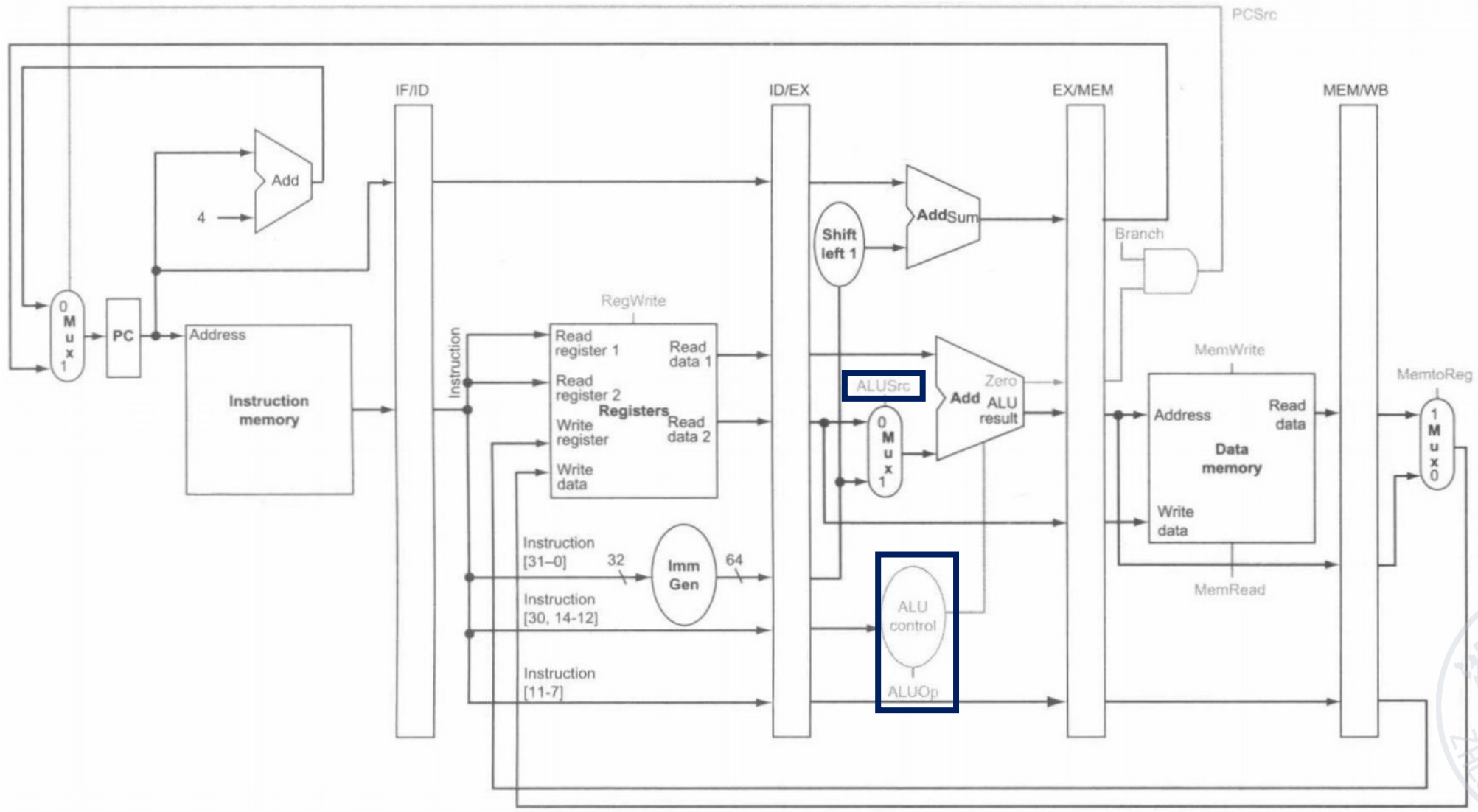
Pipelined Datapath with Control Signals



No Extra Design at Instruction Fetch and Instruction Decode



Control Signals at Execution Stage



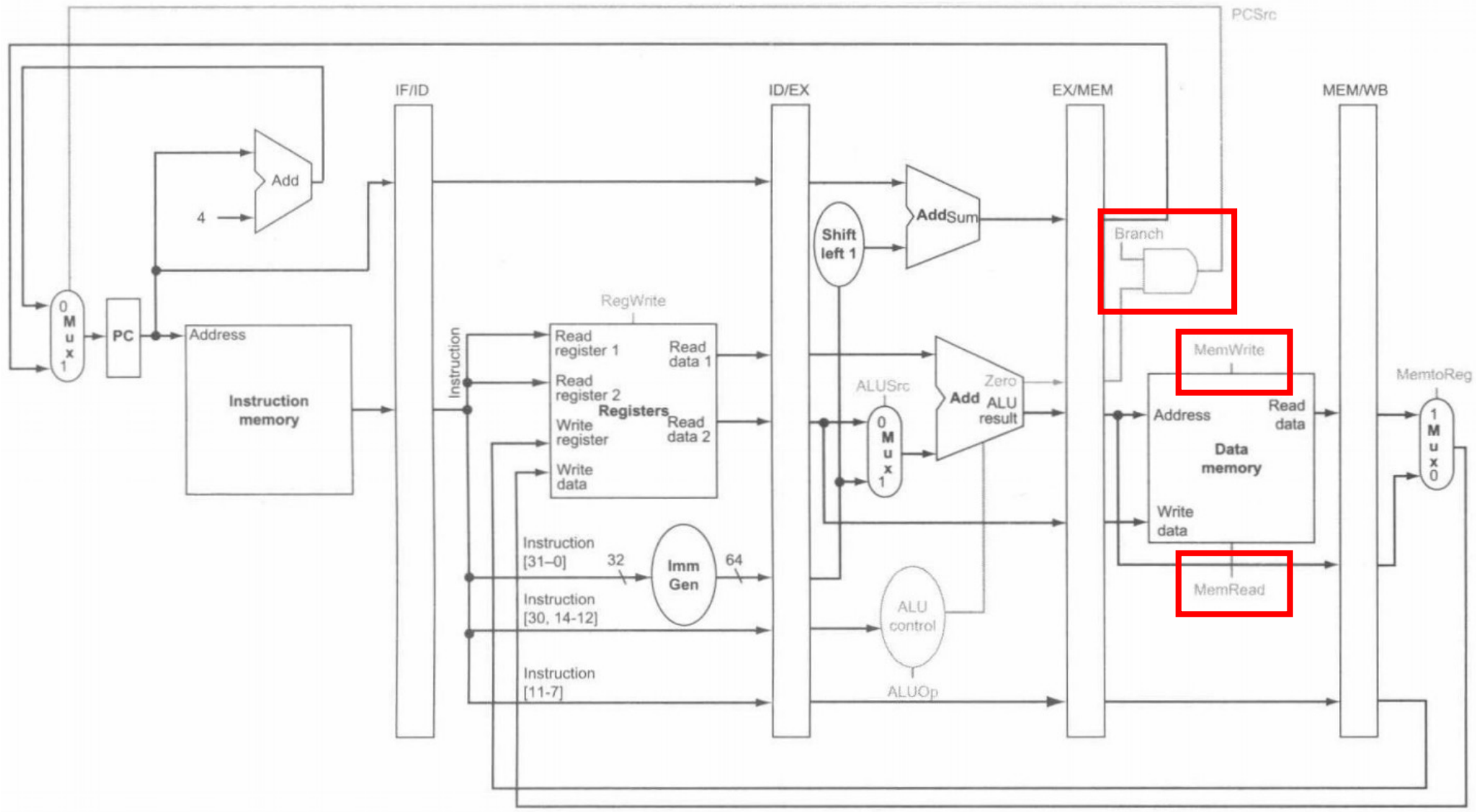
Signals Related to ALU Operations

Instruction	ALUOp	operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

These signals select the ALU operations

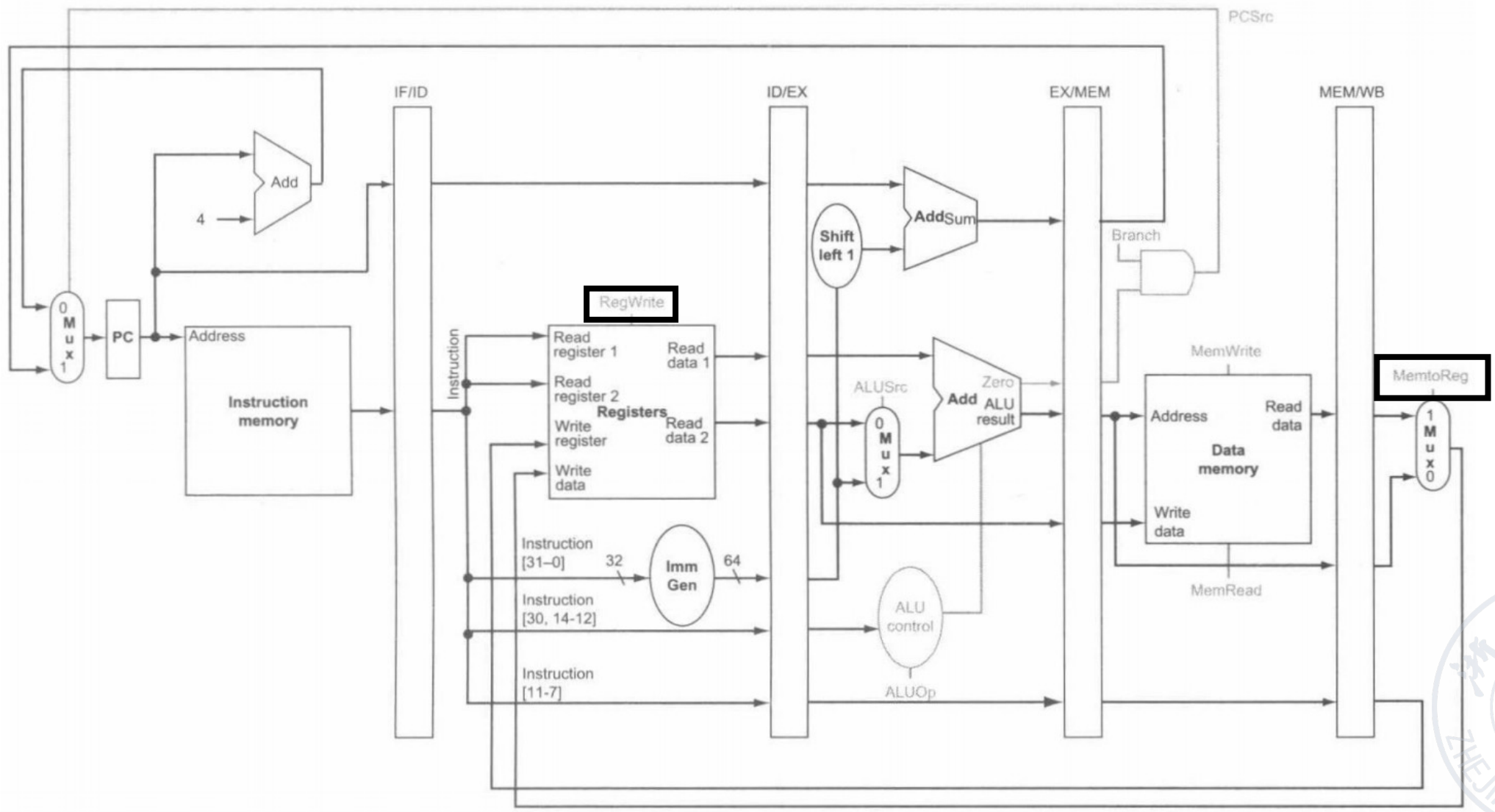
Control Signals at Memory Access Stage



Signals Related to Branch If Equal, Load, and Store Instructions

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

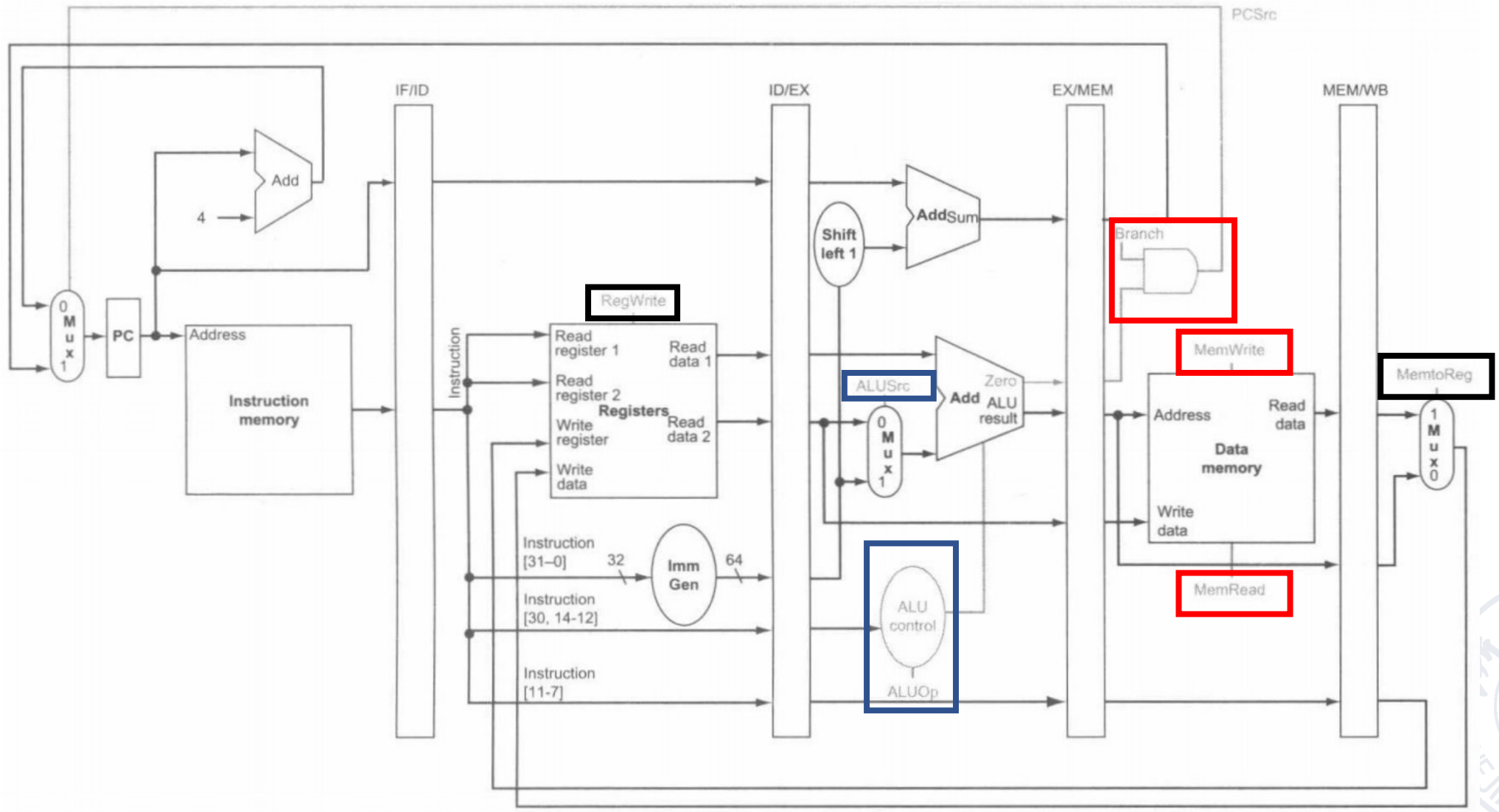
Control Signals at Write Back Stage



Signals Related to the Write Operation

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

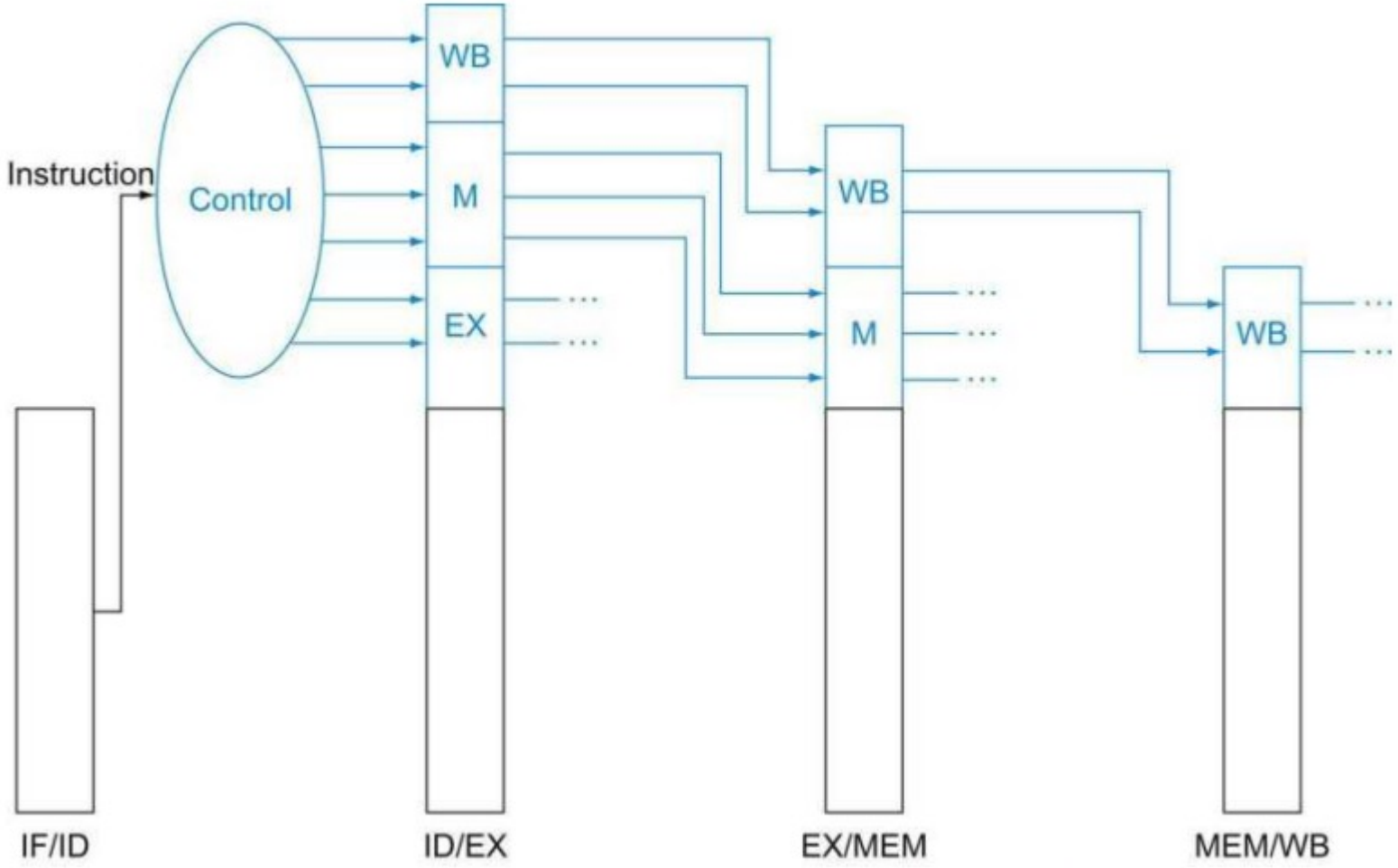
Review of 7 Control Lines



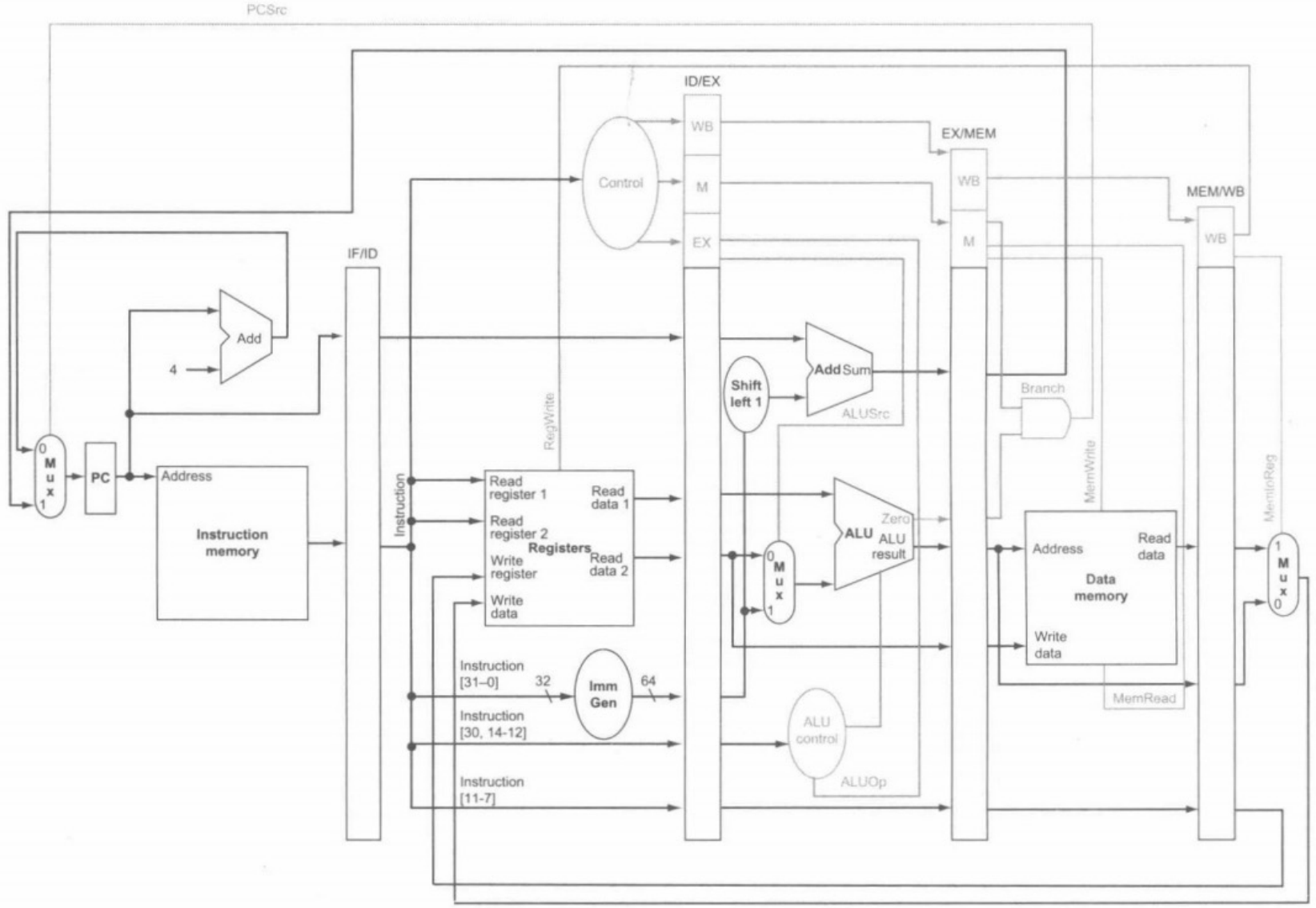
Seven Control Lines in Last Three Stages

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

Extend Pipeline Registers to Include Control Information



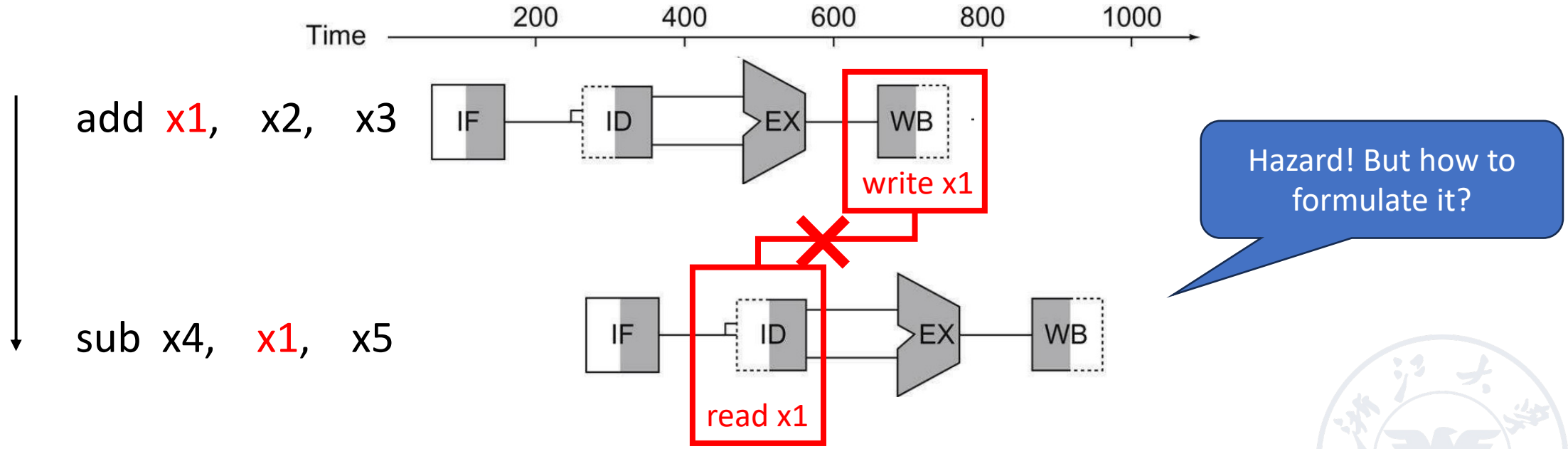
Pipelined Datapath with the Control Signals



Pipelining with Stall

Stall Condition Detection

- Data dependency between instructions



Stall Condition Detection

- Pass register numbers along pipeline
  - e.g., ID/EX.Rs1 = register number for Rs1 sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.Rs1, ID/EX.Rs2
- Data hazards when
  - 1a. EX/MEM. Rd = ID/EX. Rs1
  - 1b. EX/MEM. Rd = ID/EX. Rs2

} Fwd from EX/MEM pipeline reg

Stall Condition Detection

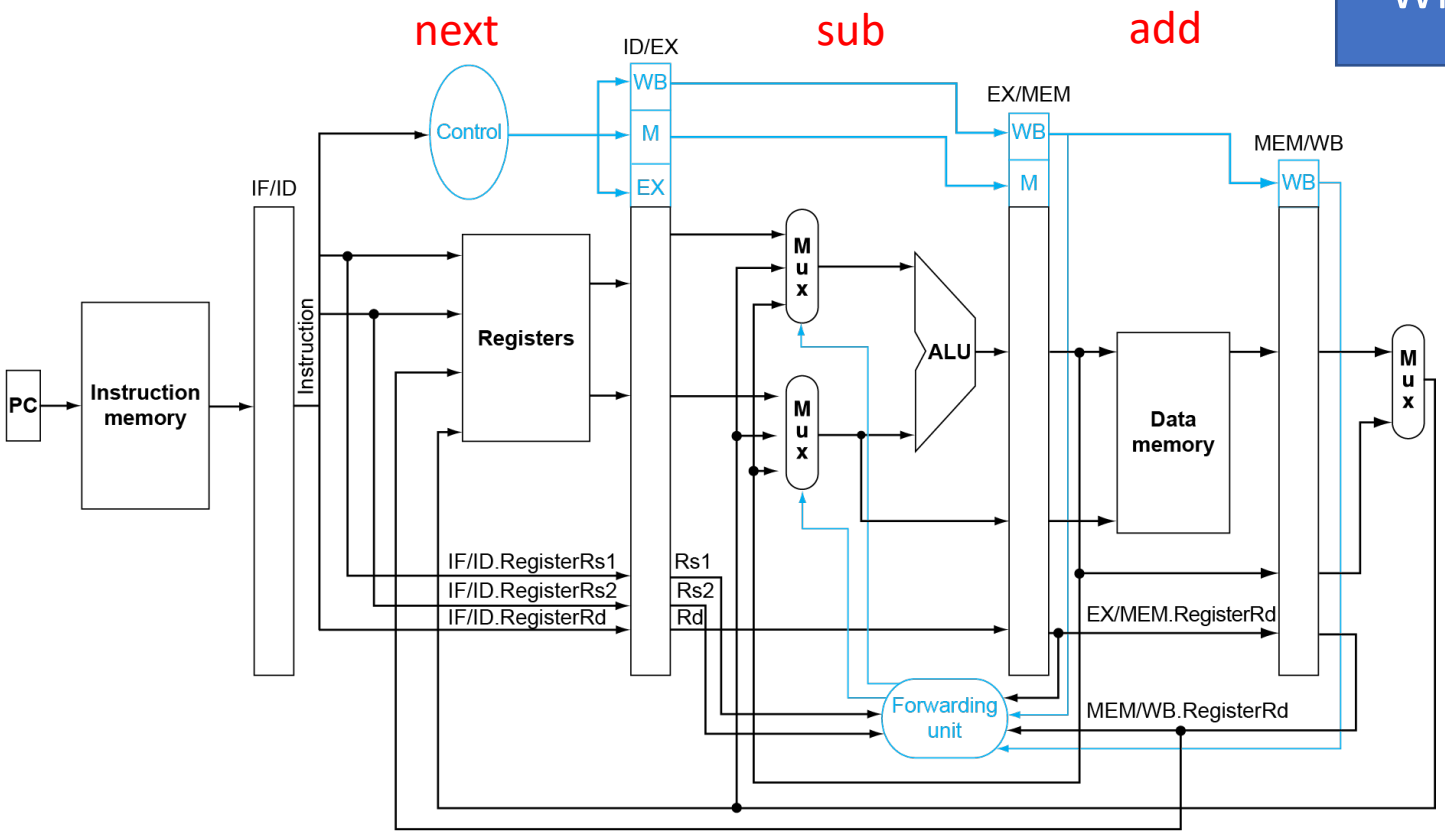
- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite

- And only if Rd for that instruction is not x0
  - EX/MEM. Rd ≠ 0

How to Stall the Pipeline

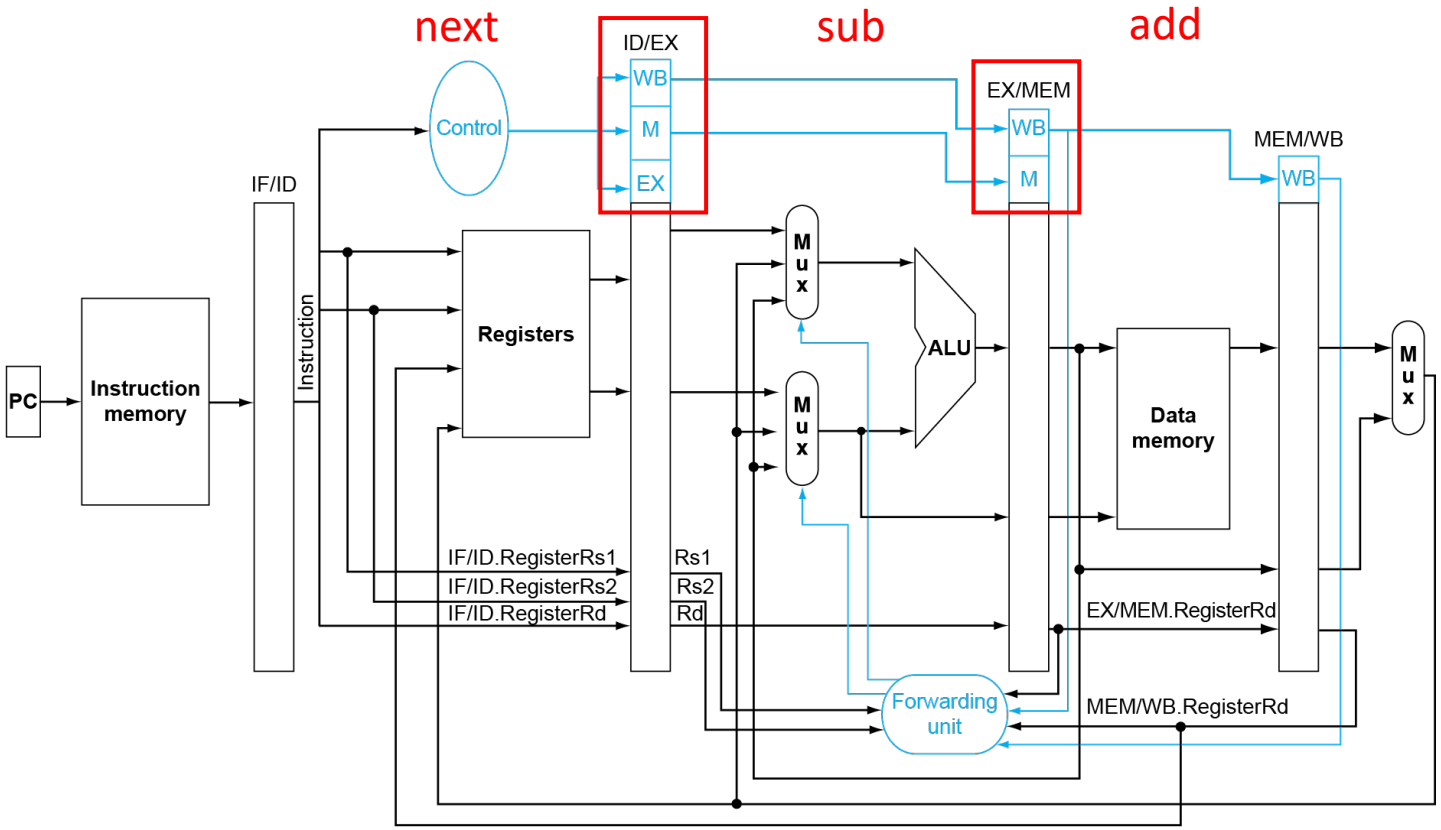
- Force control values in EX/MEM register to 0
  - EX, MEM and WB do nop (no-operation)

Which one?



How to Stall the Pipeline

- Force control values in EX/MEM register to 0
  - EX, MEM and WB do nop (no-operation)

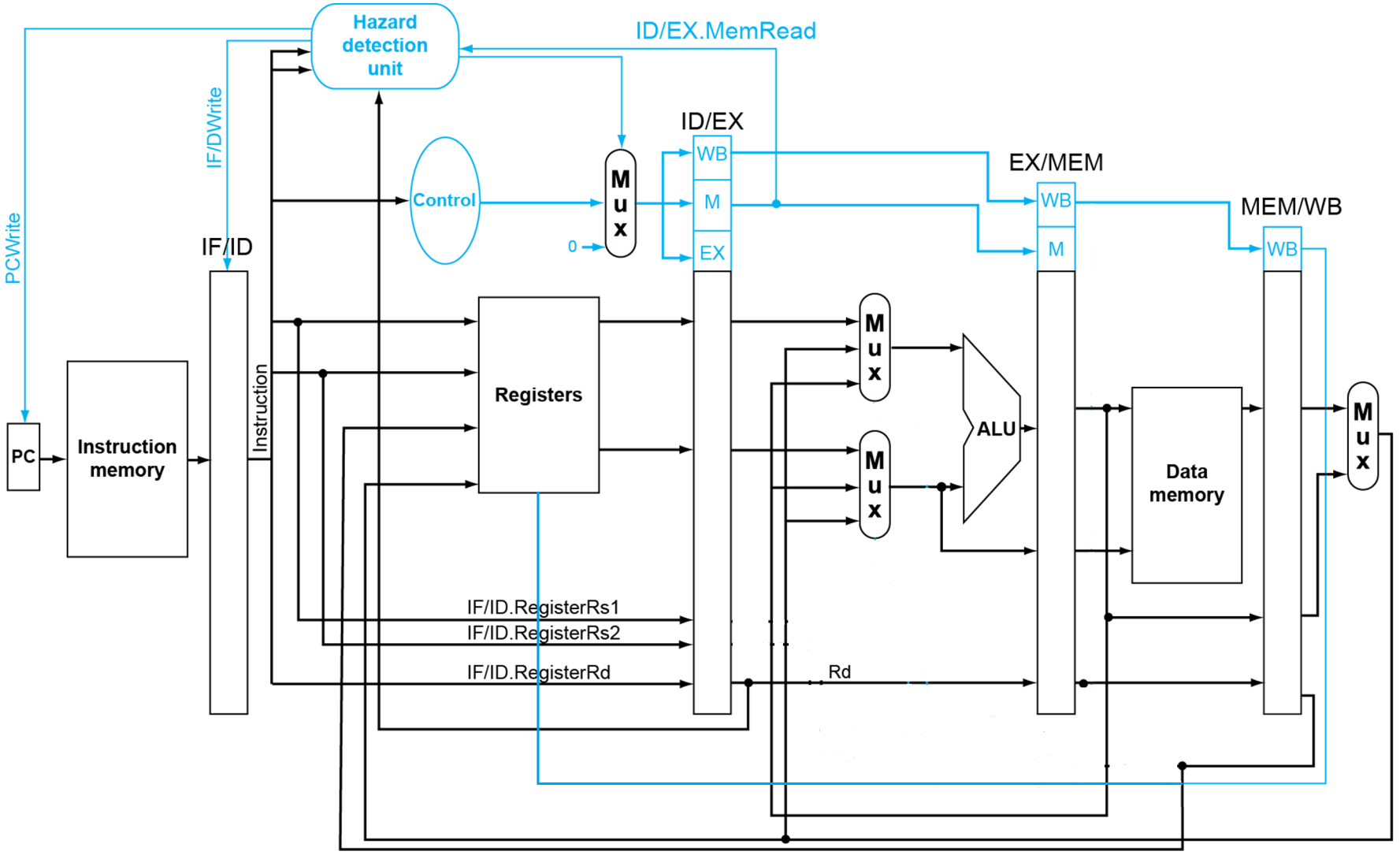


What's more?

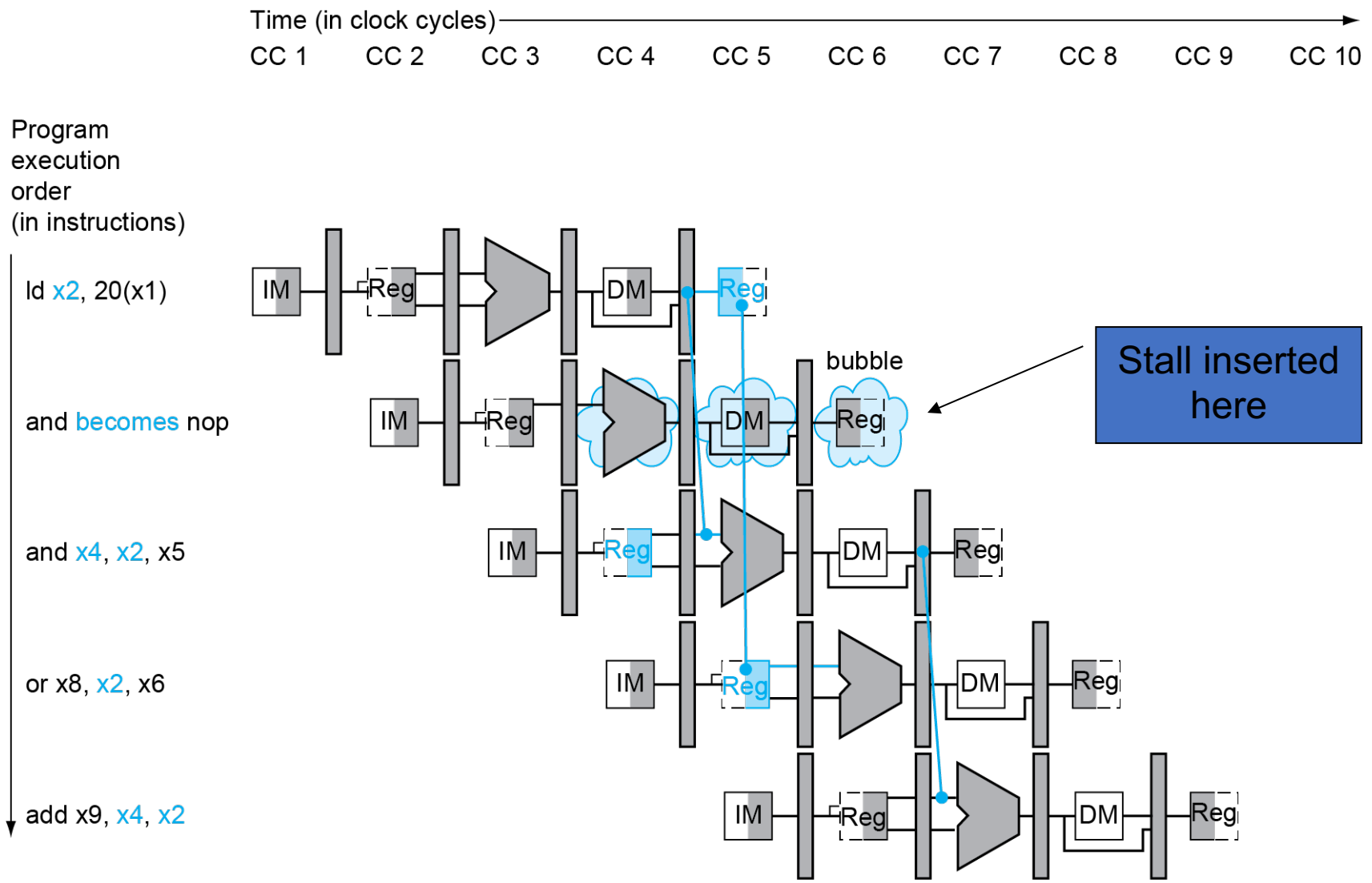
How to Stall the Pipeline

- Force control values in EX/MEM register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for ld
    - Can subsequently forward to EX stage

Datapath with Stall



Stall/Bubble in the Pipeline



Stall/Bubble in the Pipeline