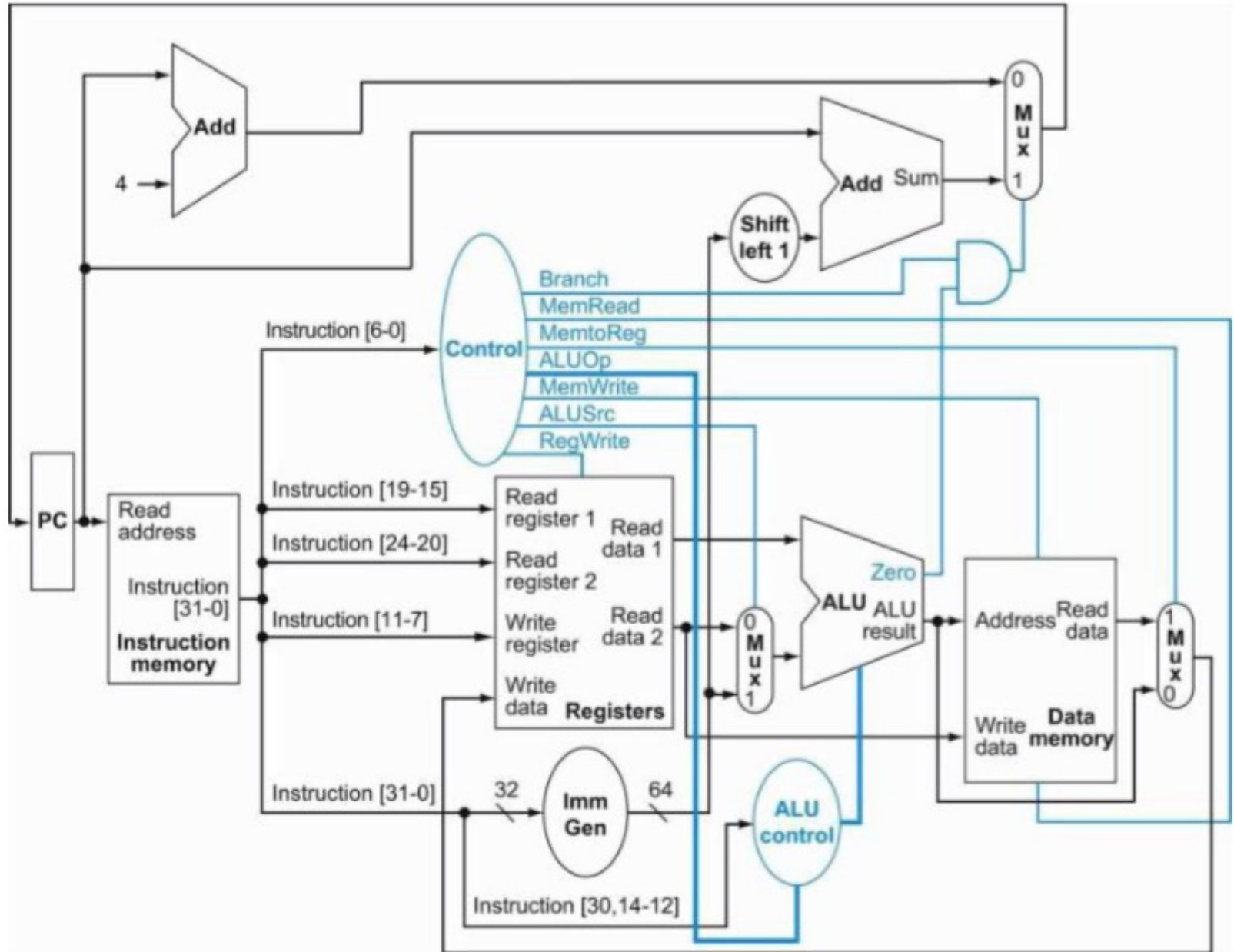


Lecture 7

Pipelining (I)

1. Pipelining
2. Data Hazards and Forwarding
3. Control Hazard Solution

Single Cycle Implementations



What's wrong with our CPI=1 processor?

Arithmetic & Logical



Load



Store

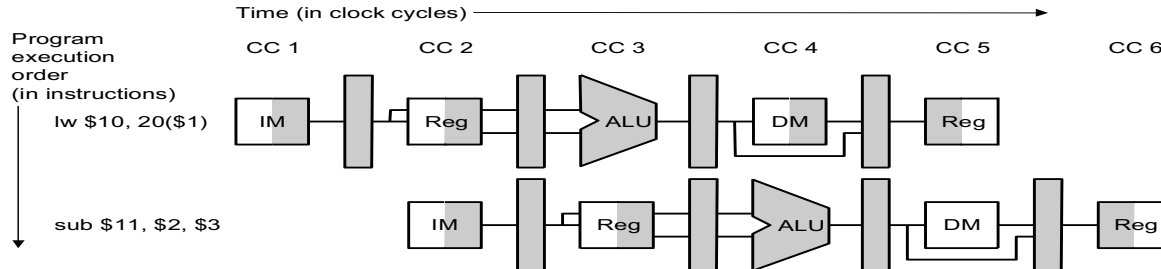


Branch



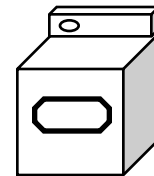
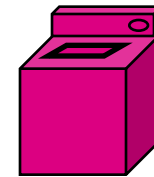
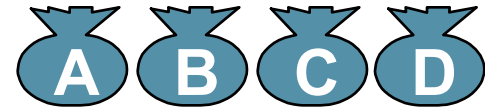
- Long Cycle Time
- All instructions take as much time as the slowest
- Real memory is not so nice as our idealized memory
 - cannot always get the job done in one (short) cycle

Pipelining

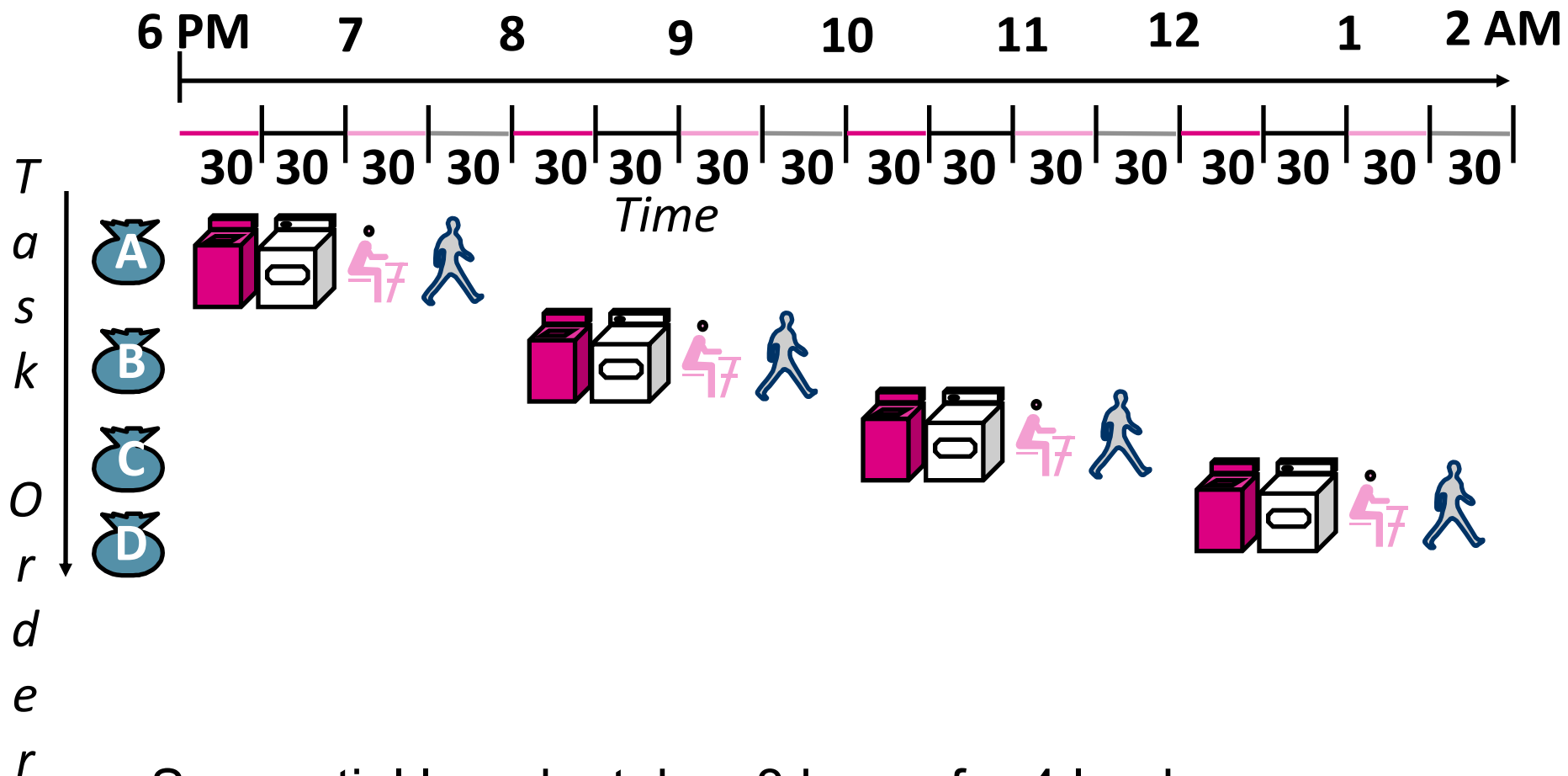


Pipelining is Natural!

- Laundry Example
- Ann, Brian, Cathy, David each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 30 minutes
- “Folder” takes 30 minutes
- Putting clothes into drawers takes 30 minutes

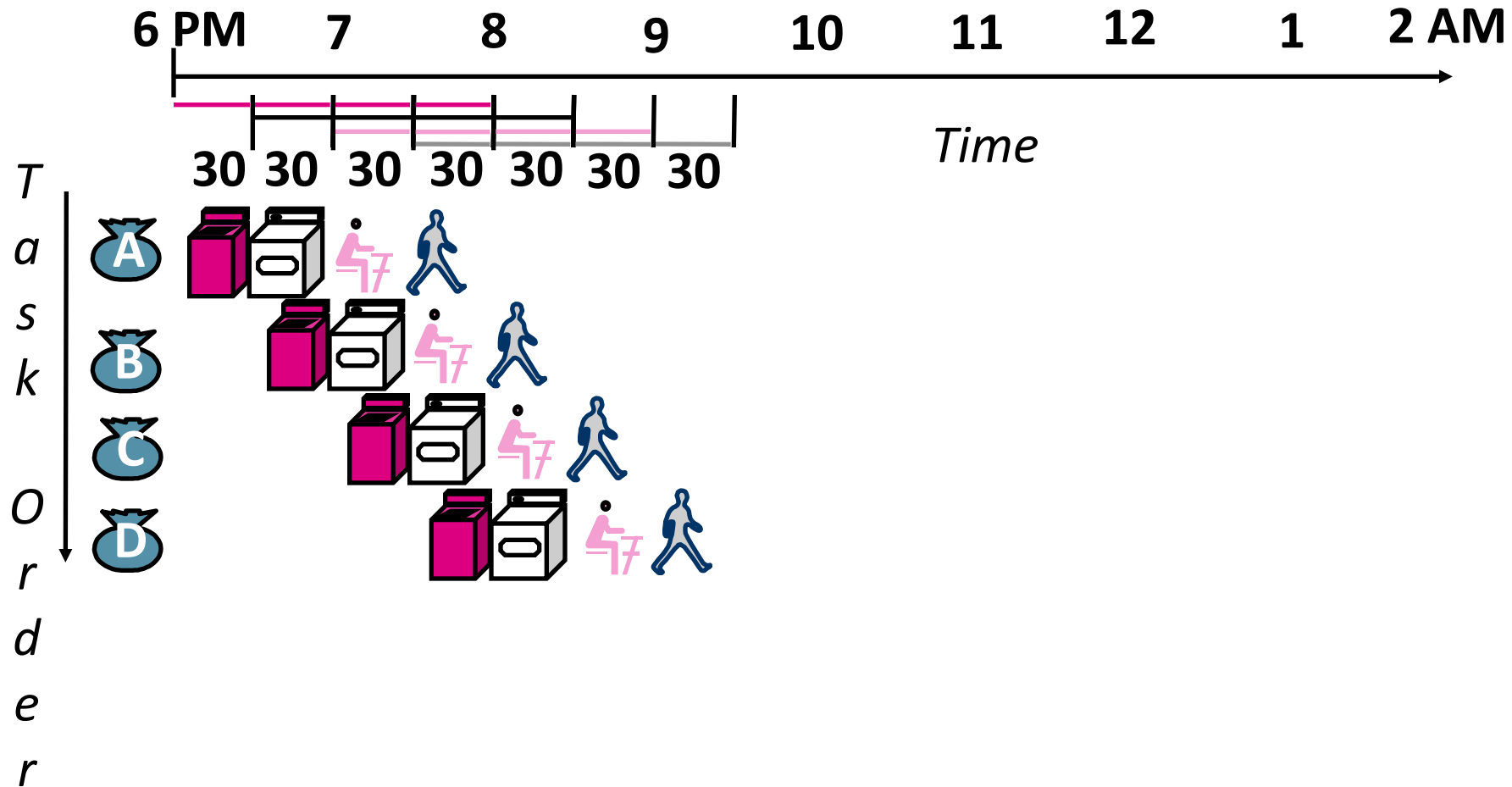


Sequential Laundry



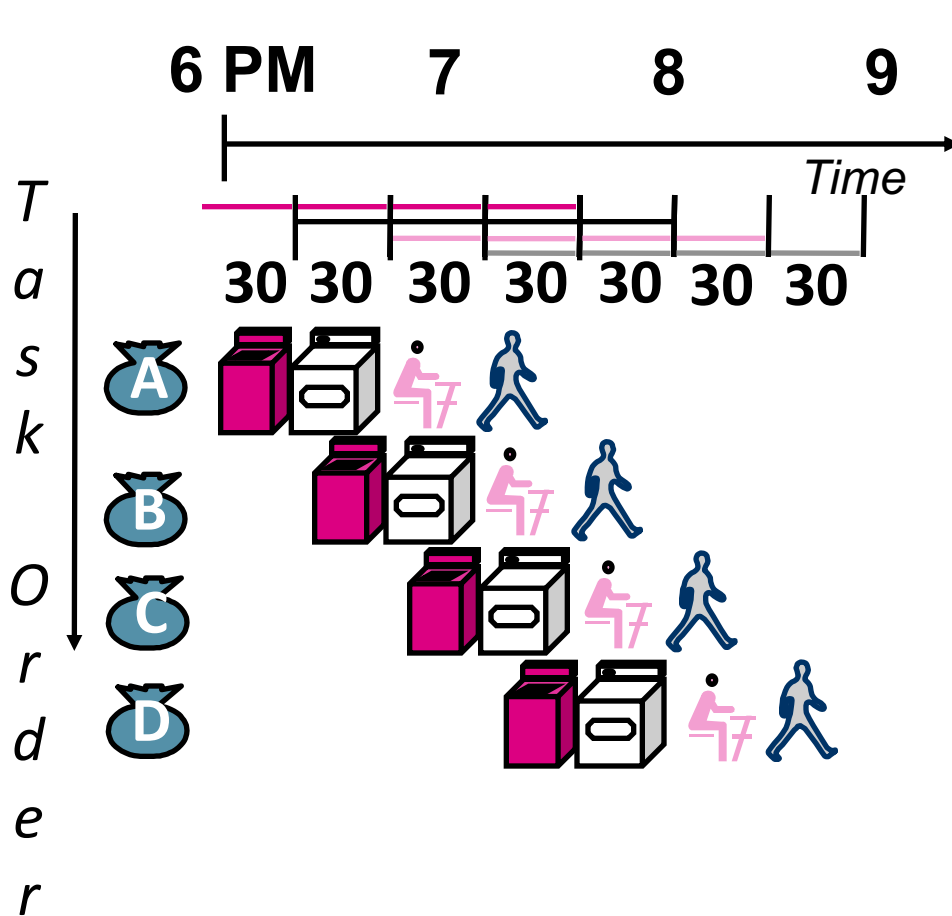
- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP



■ Pipelined laundry takes 3.5 hours for 4 loads!

Pipelining Lessons



- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- **Multiple** tasks operating simultaneously using different resources
- Pipeline rate limited by **slowest** pipeline stage
- Unbalanced lengths of pipe stages reduces speedup

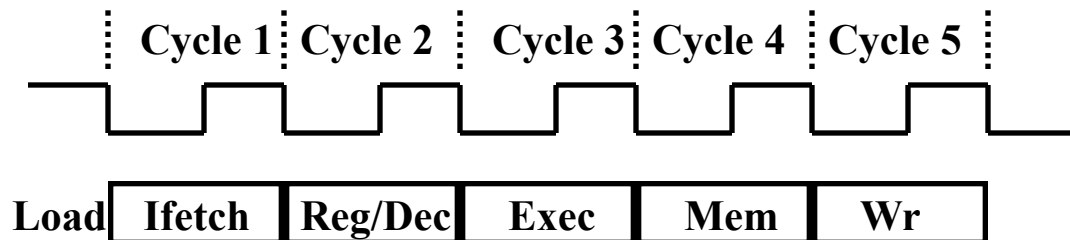
Pipelining in CPU

- Break the instruction into smaller steps
- Execute each step (instead of the entire instruction) in 1 clock cycle (multi-cycle implementation)
 - Cycle time: time it takes to execute the longest step
 - Try to make all the steps have similar length

5 steps in MIPS & RISC-V

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back

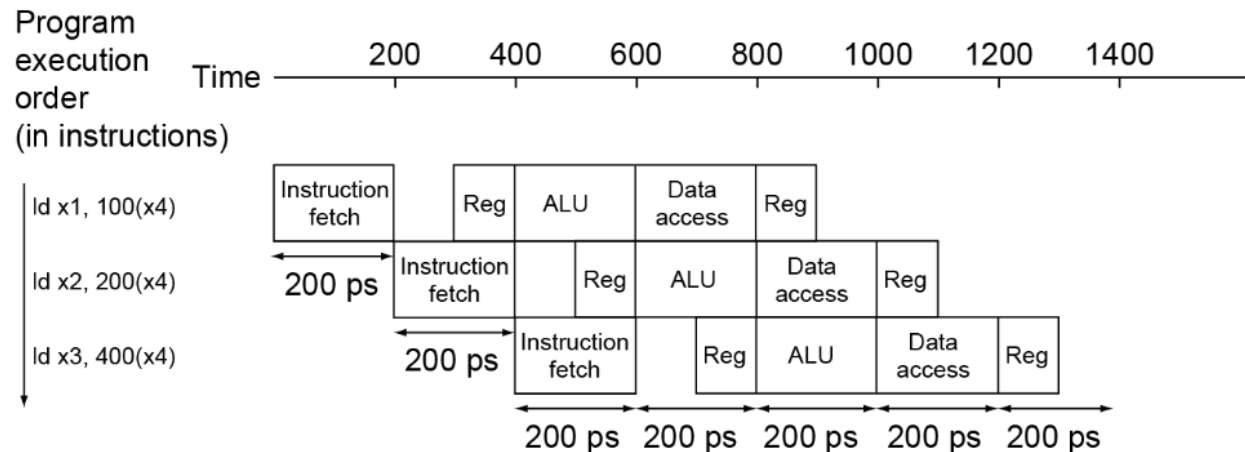
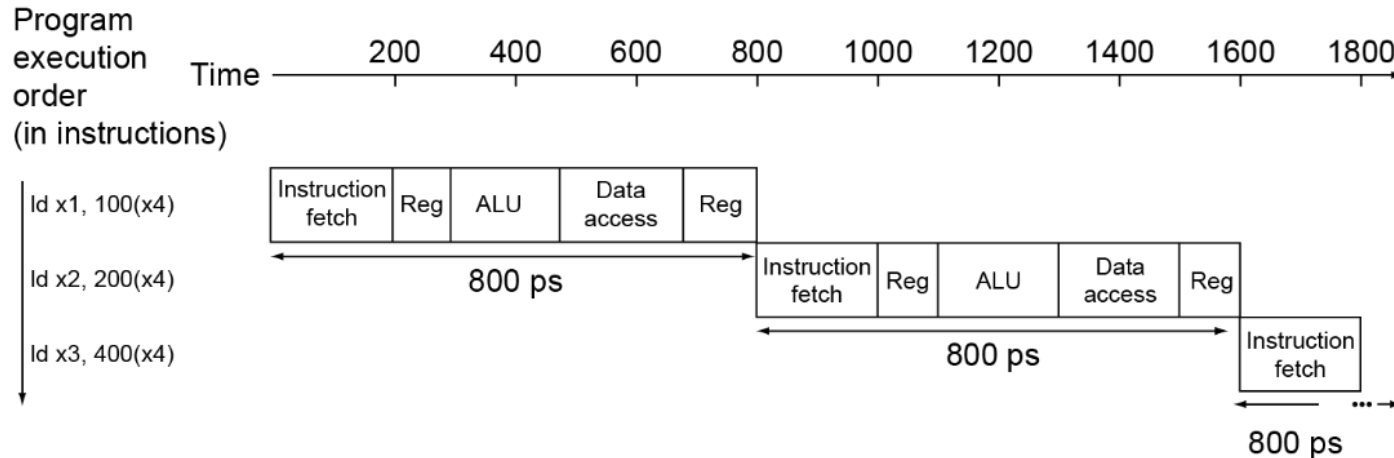
The Five Stages of Load



- **Ifetch: Instruction Fetch**
 - Fetch the instruction from the Instruction Memory
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec: Calculate the memory address**
- **Mem: Read the data from the Data Memory**
- **Wr: Write the data back to the register file**

Pipelining

■ Improve performance by increasing instruction throughput



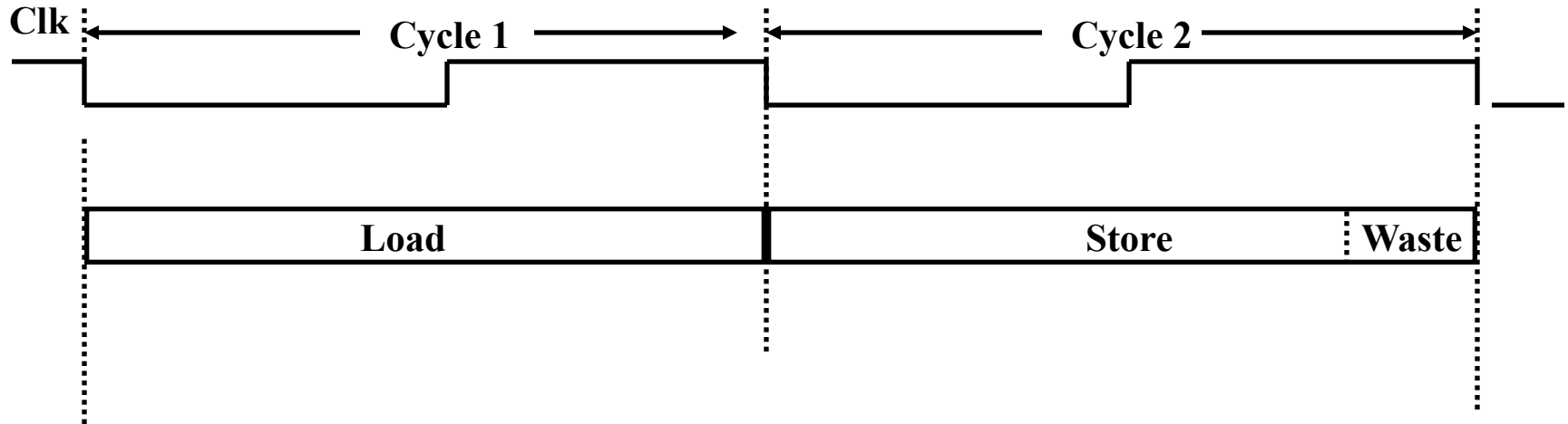
Time to execute 4 load instructions:

Single cycle: $0.8 \times 4 = 3.2$ ns

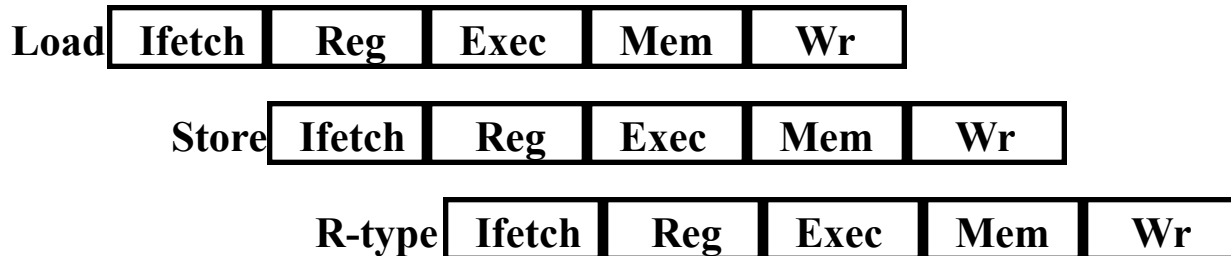
Pipeline: $4 \times 1 + 4$ (time to drain the pipeline) = 8 Cycle = 1.6 ns

Single Cycle vs. Pipeline

Single Cycle Implementation:



Pipeline Implementation:



Ideal Pipeline Performance

- Ideal CPI = 1
- Time to execute n instruction
 - $1 \times n + \text{time to drain the pipeline}$



- Ideal speedup from pipelining == # of pipeline stages
 - If the stages are perfectly balanced, and a large number of instructions

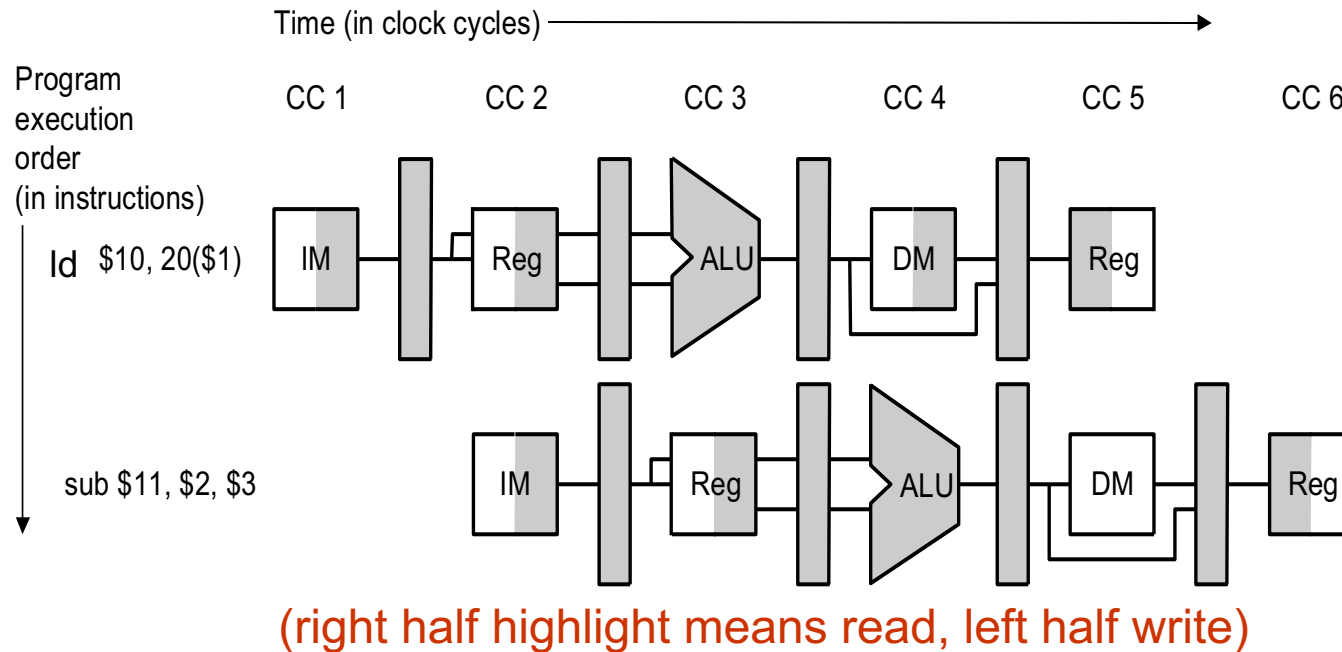
$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

$$\frac{(S-1) + N}{N} \approx 1$$

$$(S-1) + N$$

$$\frac{S \times N \times T}{[(S-1) + N] \times T} \approx S$$

Graphically Representing Pipelines



- Can help with answering questions like:
 - how many cycles does it take to execute this code?
 - what is the ALU doing during cycle 4?
 - use this representation to help understand datapaths

Pipelining and ISA Design

- RISC-V ISA designed for pipelining
 - All instructions are 32-bits
 - Easier to fetch and decode in one cycle
 - c.f. x86: 1- to 17-byte instructions
 - Few and regular instruction formats
 - Can decode and read registers in one step
 - Load/store addressing
 - Can calculate address in 3rd stage, access memory in 4th stage

Can pipelining get us into trouble?

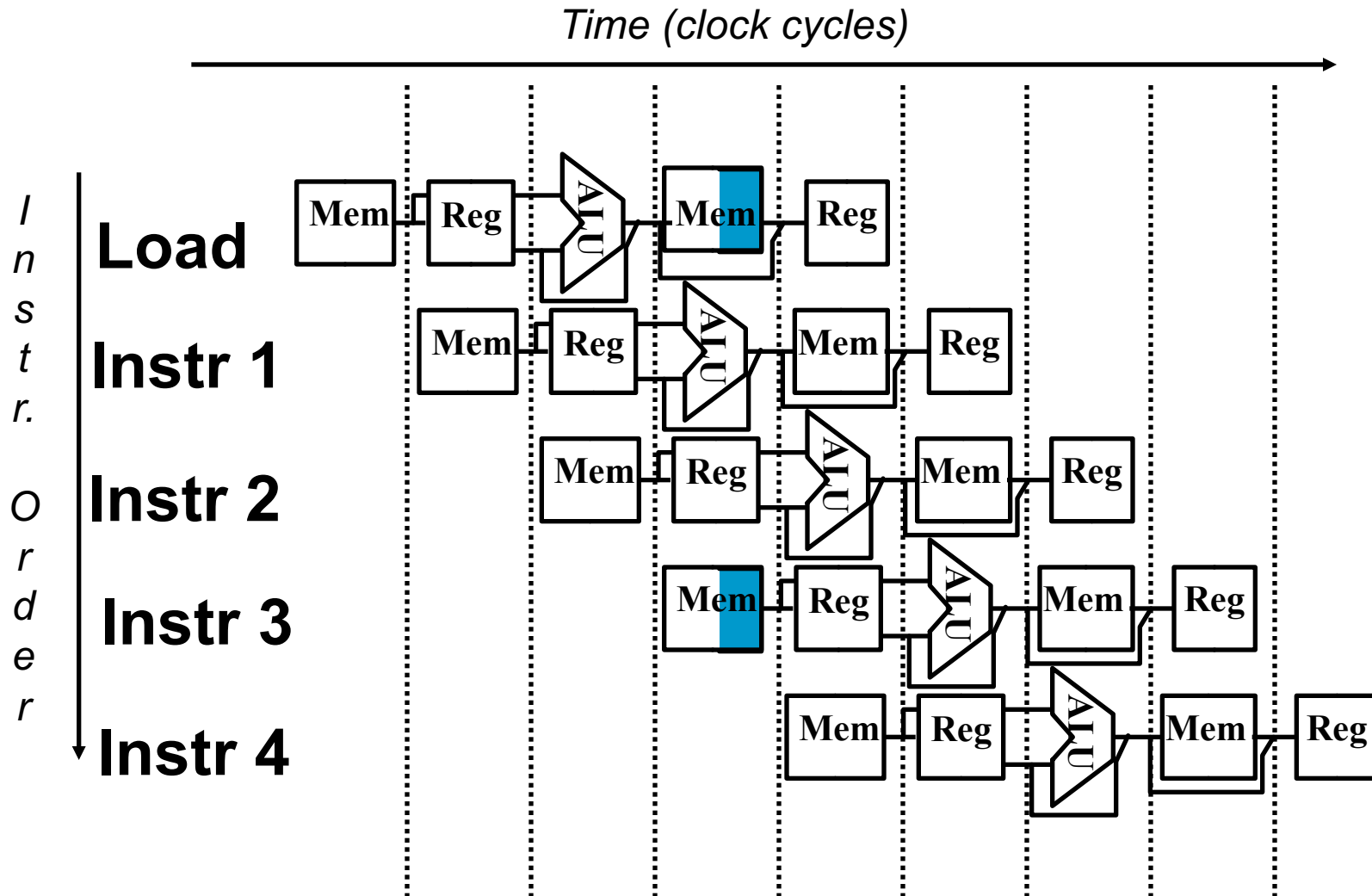
■ Yes: Pipeline Hazards

- **structural hazards**: attempt to use the same resource two different ways at the same time
 - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
- **data hazards**: attempt to use item before it is ready
 - E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
 - instruction depends on result of prior instruction still in the pipeline
- **control hazards**: attempt to make a decision before condition is evaluated
 - E.g., washing football uniforms and need to get proper detergent level ; need to see after dryer before next load in
 - branch instructions

■ Can always resolve hazards by **waiting**

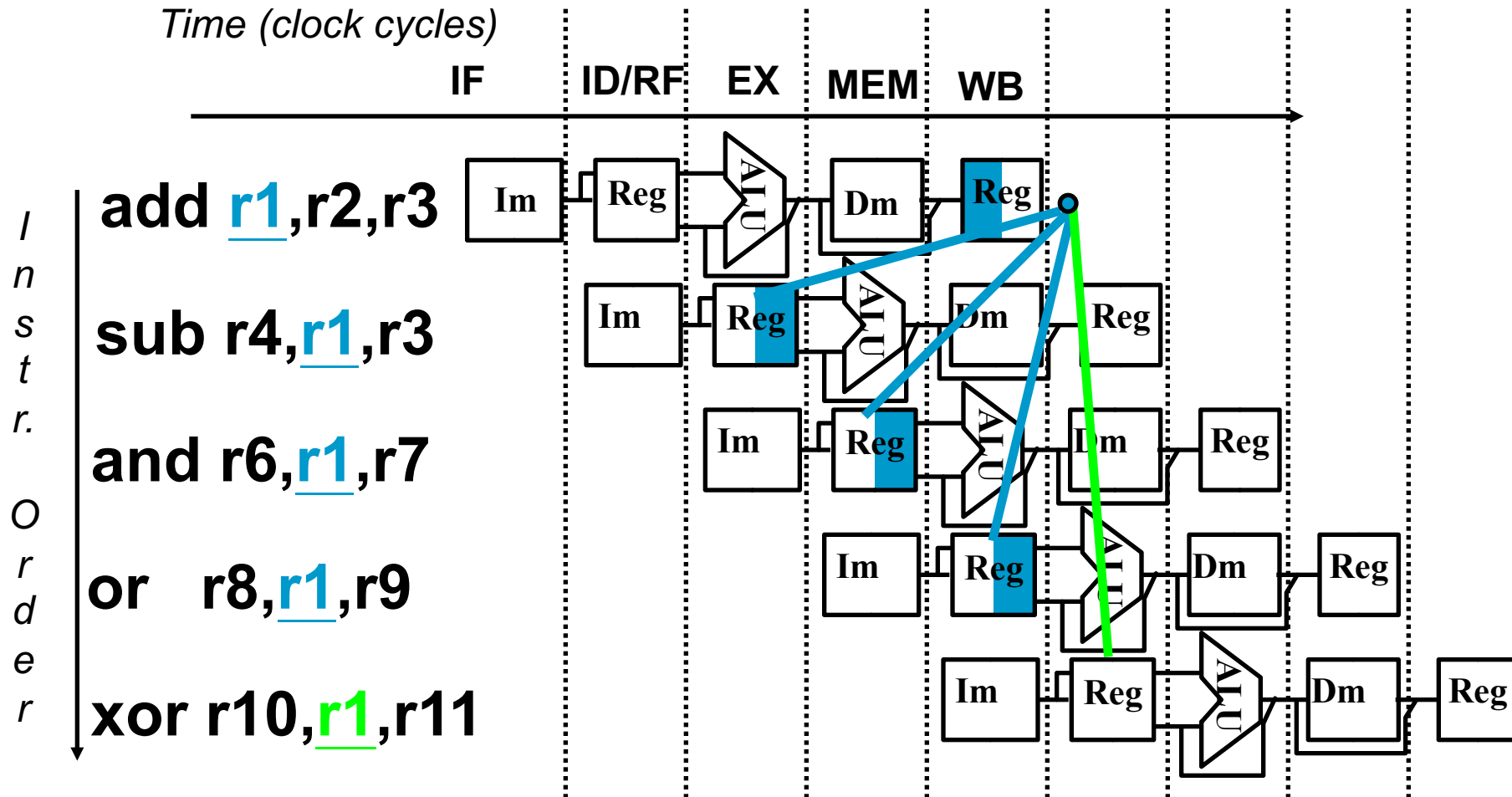
- pipeline control must detect the hazard
- take action (or delay action) to resolve hazards

Single Memory is a Structural Hazard

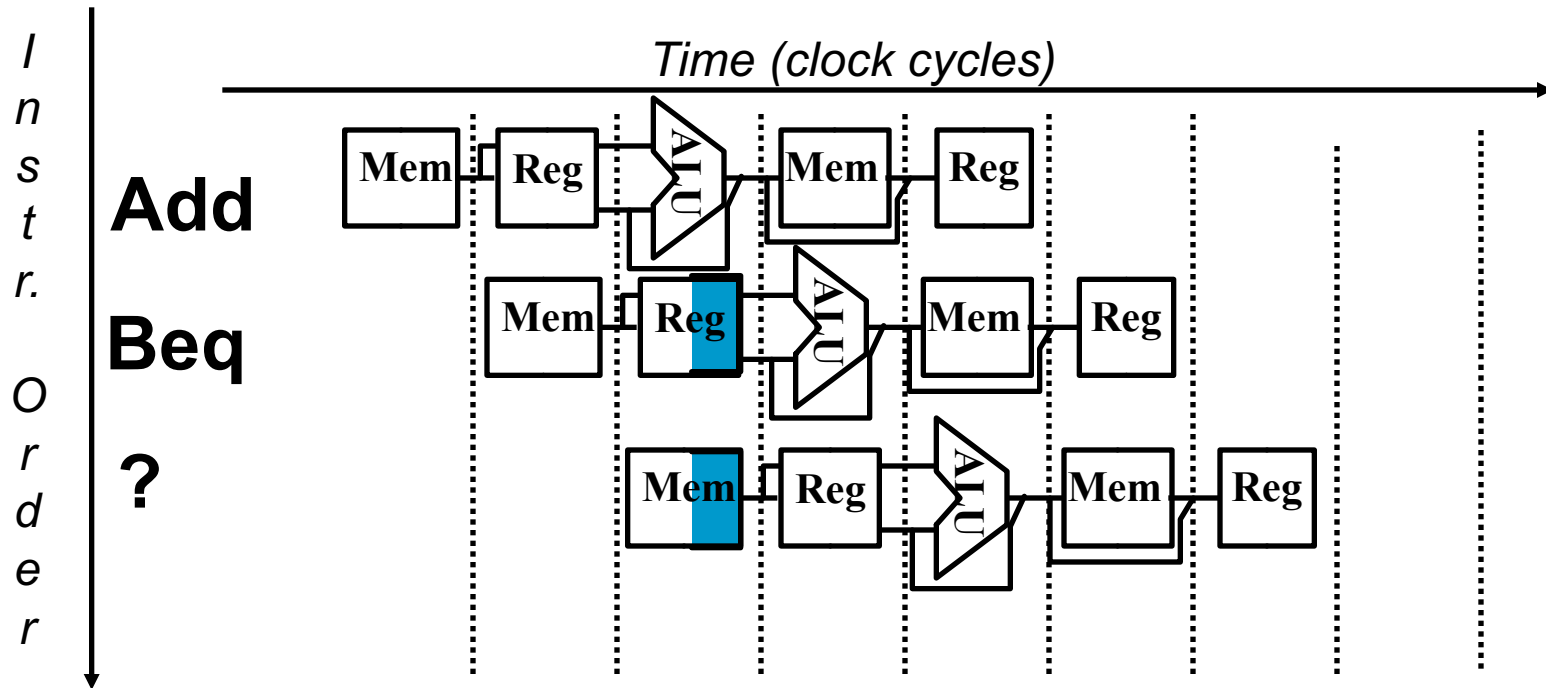


Data Hazard on r1:

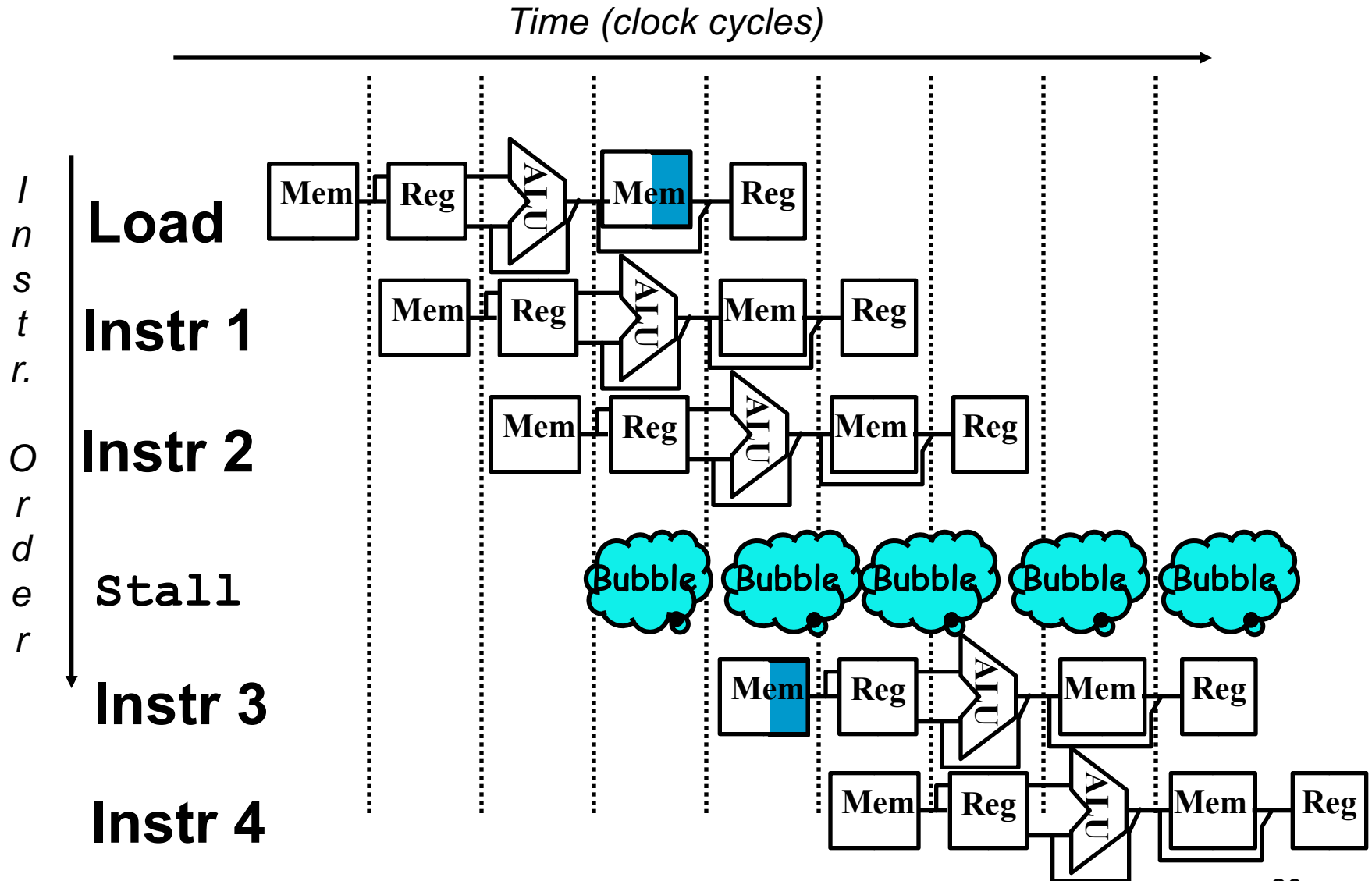
- Dependencies** backwards in time are **hazards**



Control Hazard

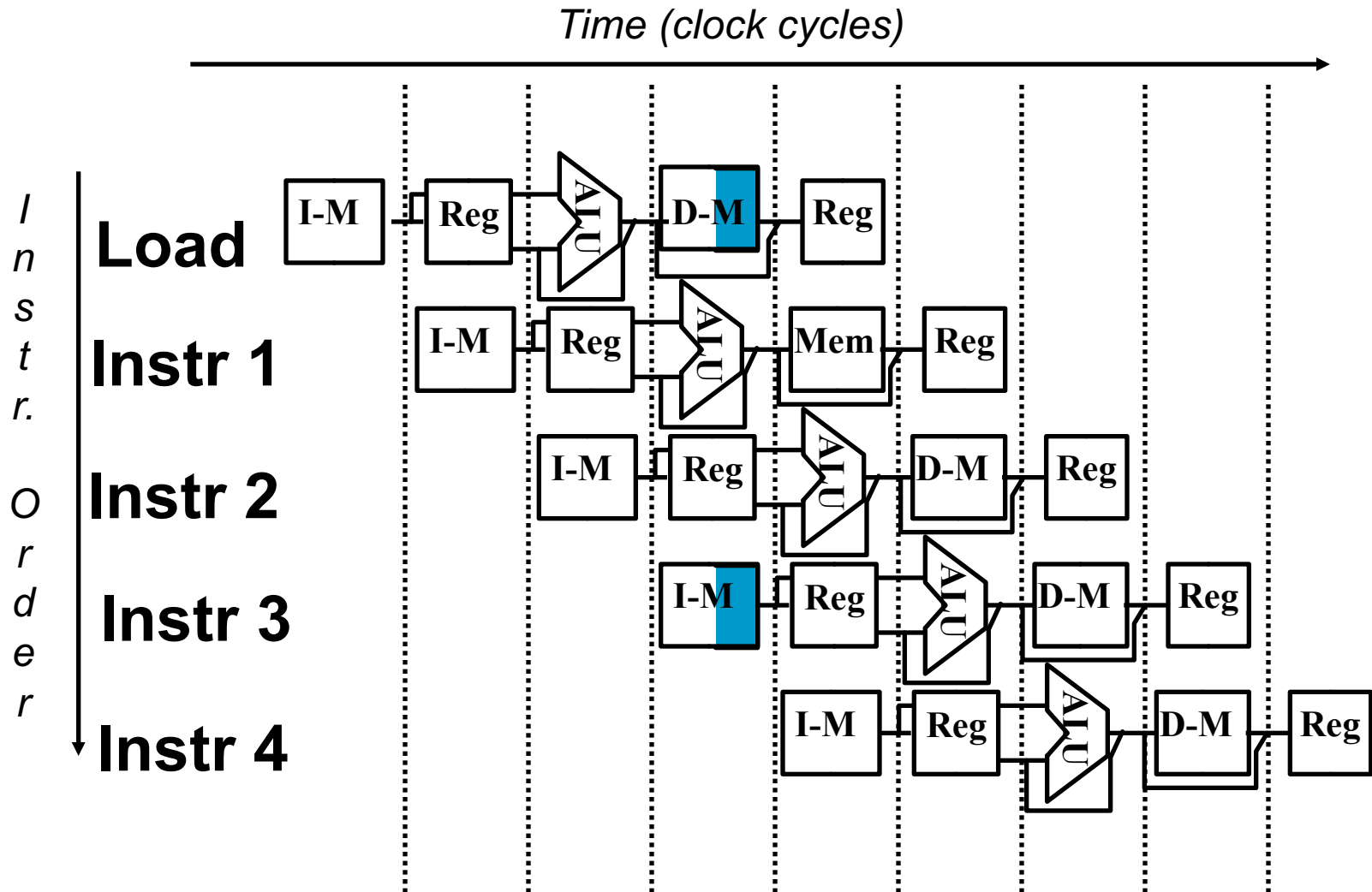


How to solve structure hazards: (1) Stall



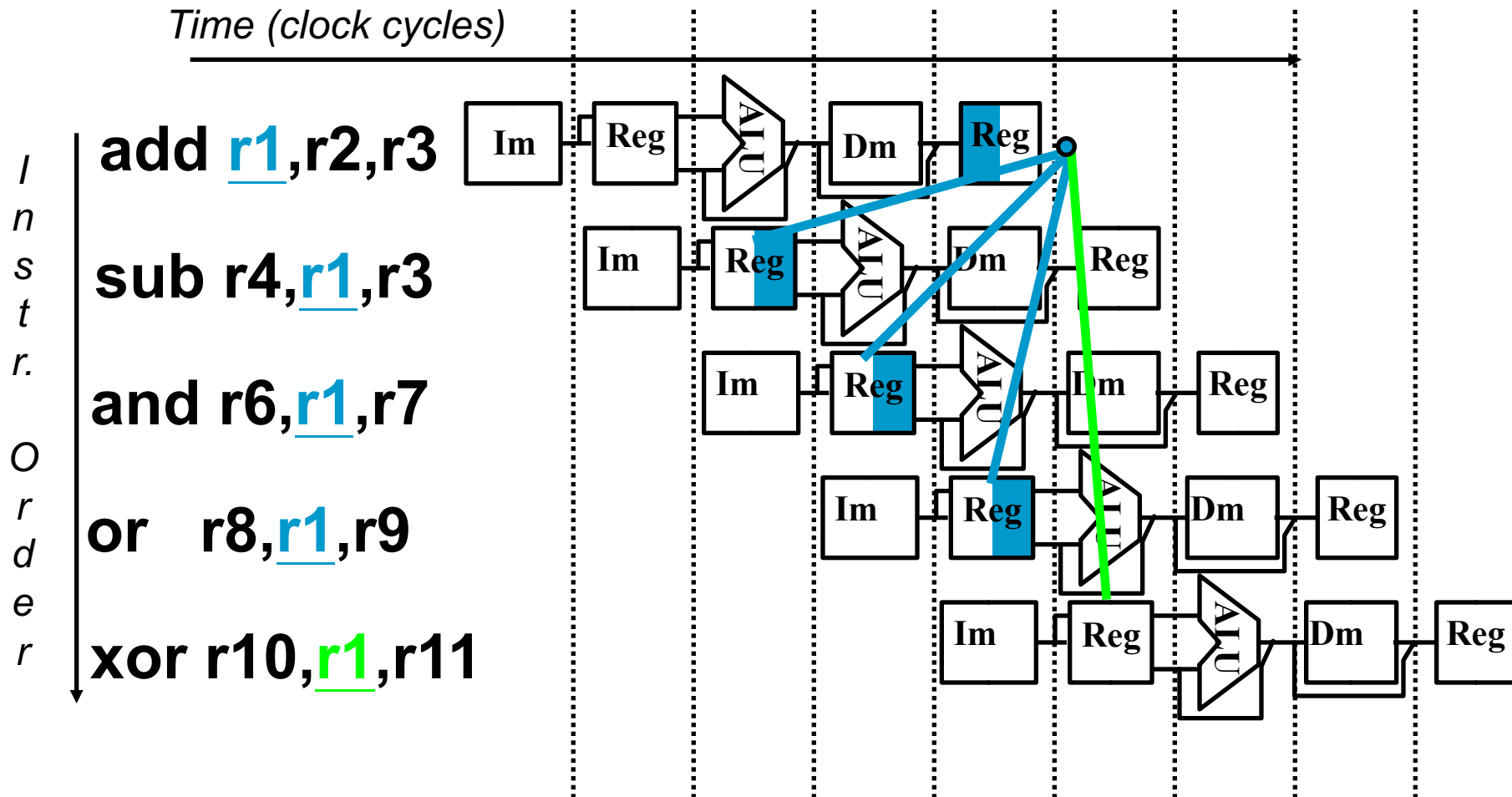
How to solve structure hazards: (2)

Split instruction and data memory



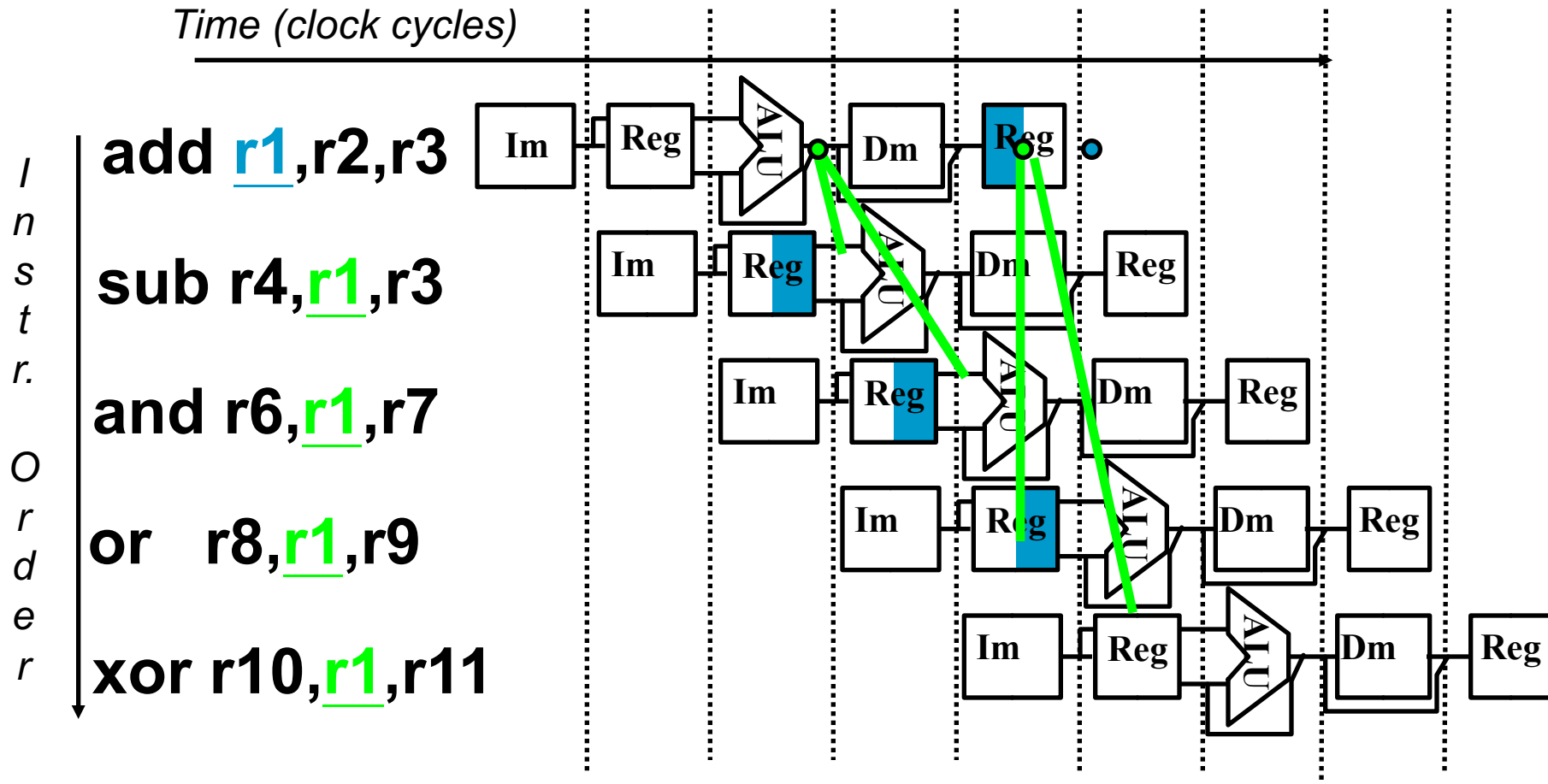
Data Hazard on r1:

- Dependencies backwards in time are hazards



Data Hazard Solution:

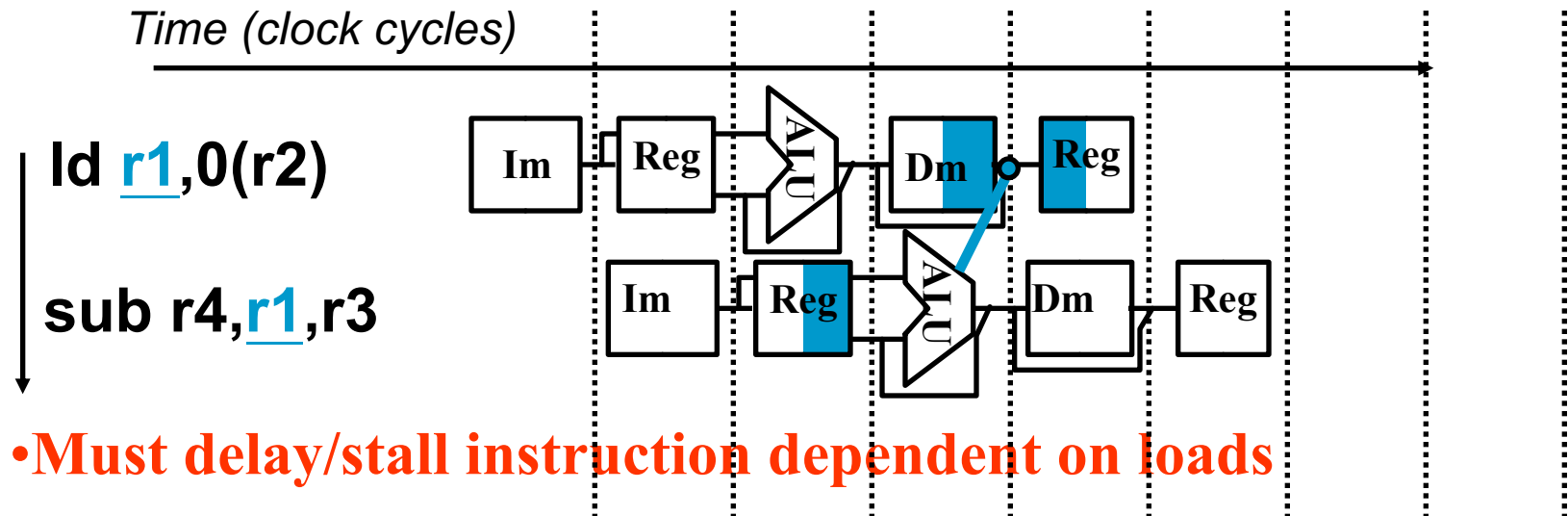
- “Forward” result from one stage to another



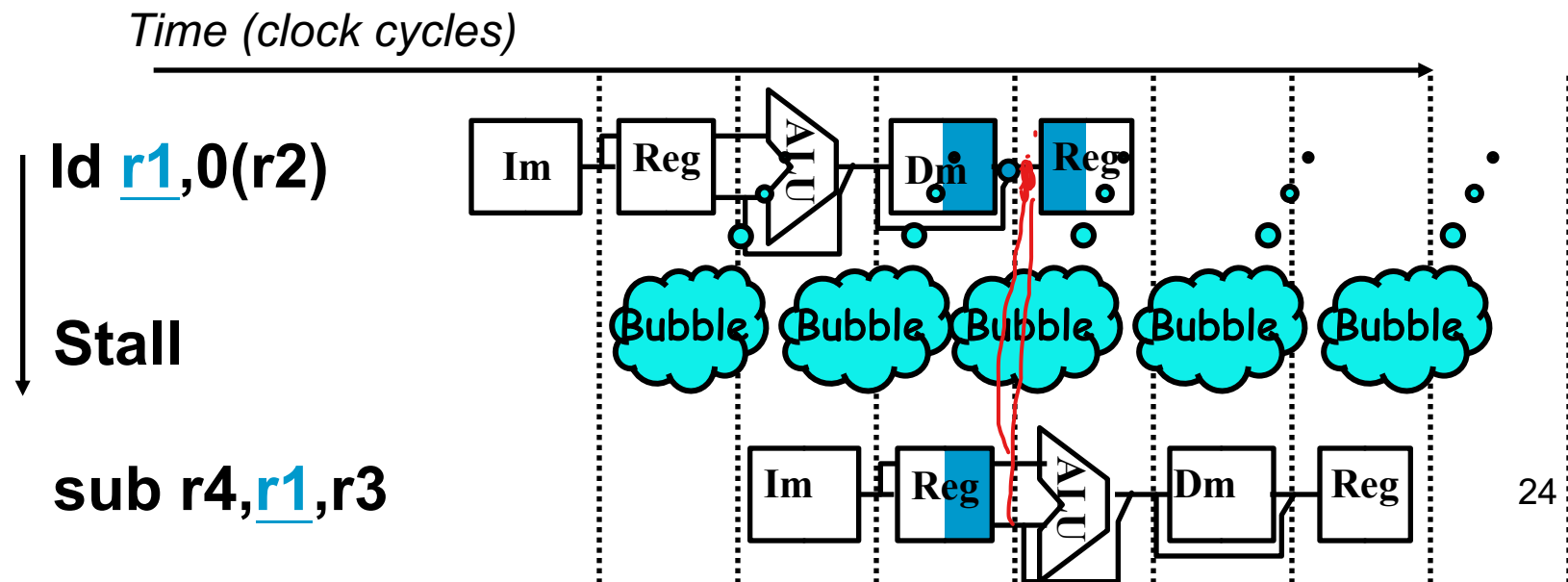
- “or” OK if register writes occur in the first half of the clock cycle, and register reads in the last half of the clock cycle

Forwarding (or Bypassing): What about Loads

- **Load-use data hazard: cannot solve with forwarding**



- **Must delay/stall instruction dependent on loads**



Reordering Code to Avoid Pipeline Stalls

A = B + E;

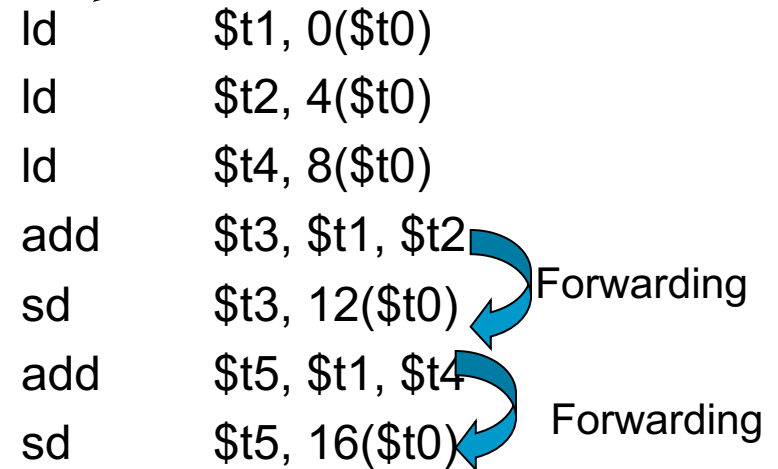
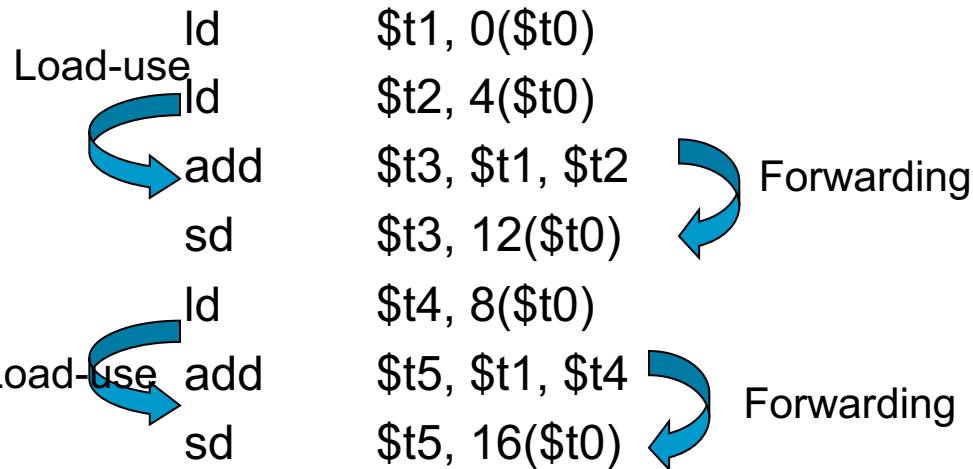
C = B + F;

```
ld      $t1, 0($t0)
ld      $t2, 4($t0)
add     $t3, $t1, $t2
sd      $t3, 12($t0)
ld      $t4, 8($t0)
add     $t5, $t1, $t4
sd      $t5, 16($t0)
```

Reordering Code to Avoid Pipeline Stalls

A = B + E;
C = B + F;

On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than left-hand side sequence.

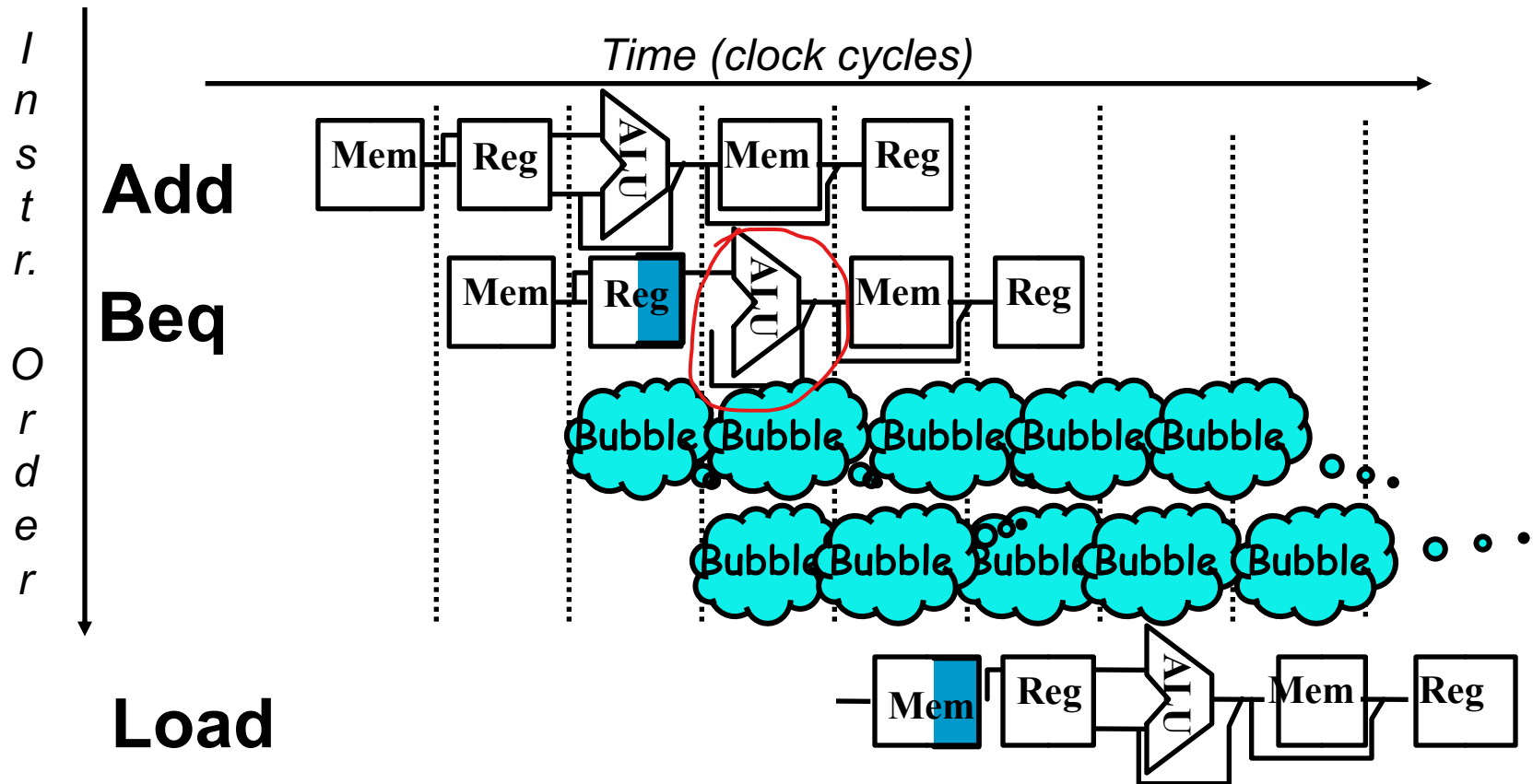


Control Hazard

- It arises from the need to make a decision based on the results of one instruction while others are executing.
- Two solutions to control hazards
 - Stall
 - After fetching a branch instruction, pipeline is to stall immediately, waiting until the outcome of the branch is known.
 - Slow down the pipeline.
 - Predict
 - Assumes a given outcome for the branch and proceeds from the assumption rather than waiting to ascertain the actual outcome.
 - Should rollback if the assumption is wrong,

Control Hazard Solutions (1) Stall

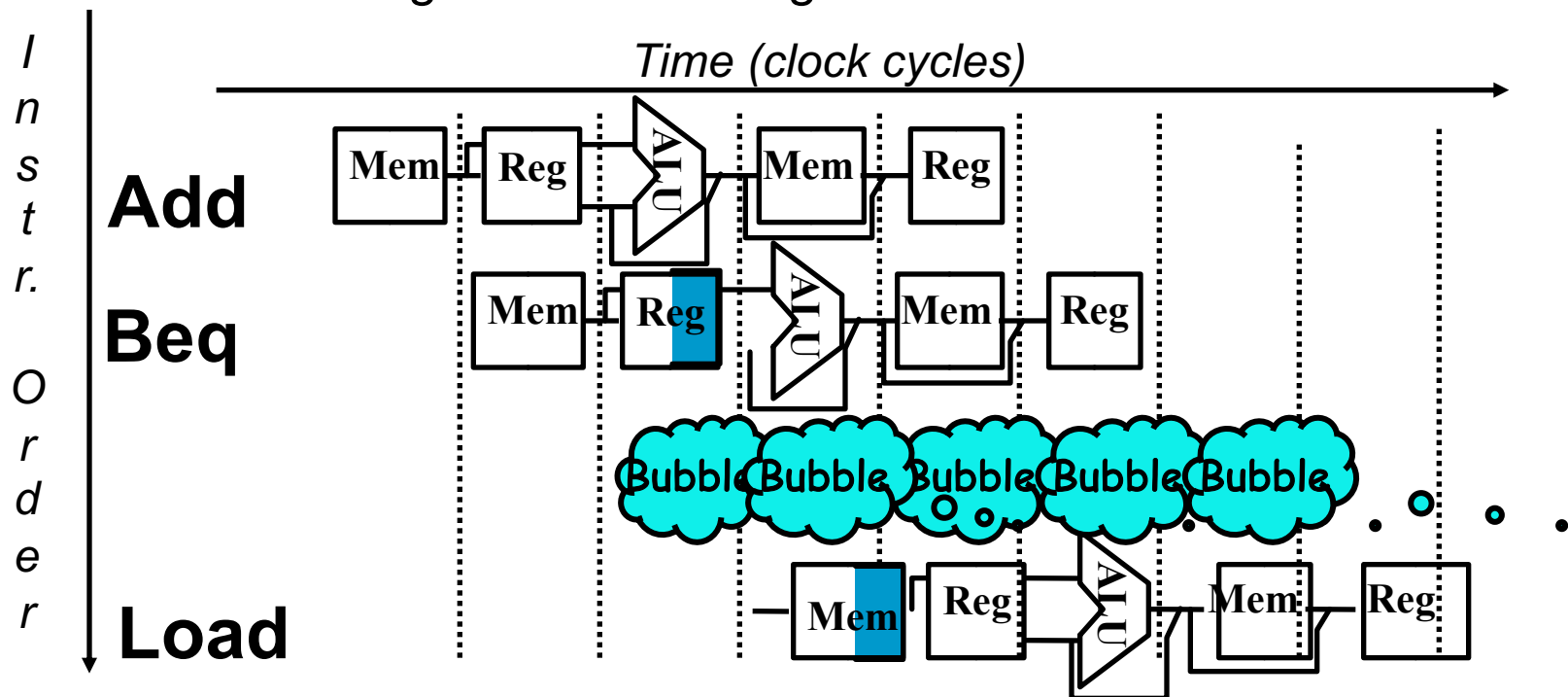
- When is a branch resolved?



- Impact: 3 clock cycles per branch instruction
=> slow

Control Hazard Solutions (2) Resolve branch earlier

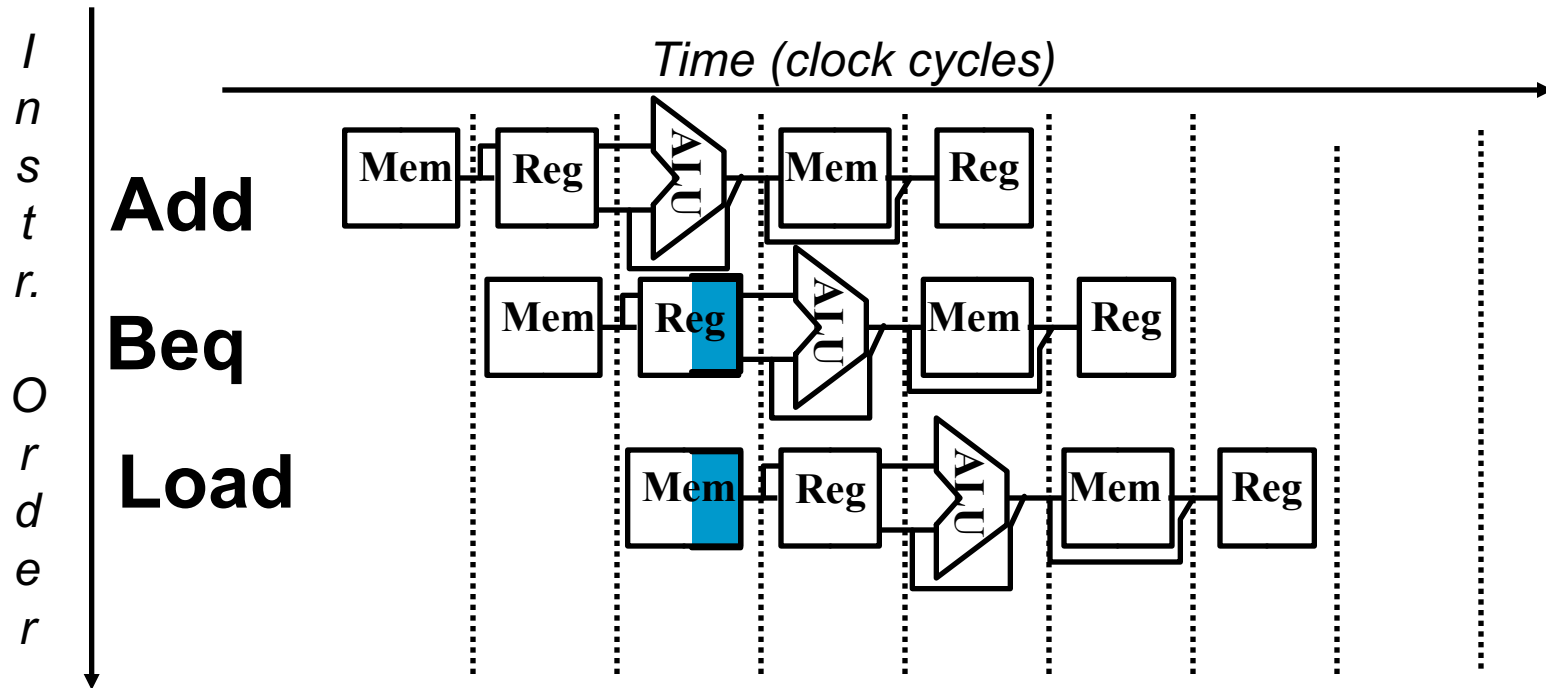
- What if a branch is resolved in the second stage (ID)?
 - Assume that we can put in enough extra hardware so that we can test registers, calculate the branch address, and update the PC during the second stage



- Impact: 2 clock cycles per branch instruction
- As the branch are 13% of the instructions executed in SPECint2000, the CPI is slowdown of 1.13 versus the ideal case.

Control Hazard Solutions (3) Predicted Untaken

- Predict: guess one direction then back up if wrong
 - Predict not taken



- Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right 50% of time)
- More **dynamic scheme**: history of 1 branch

Predict branch not taken

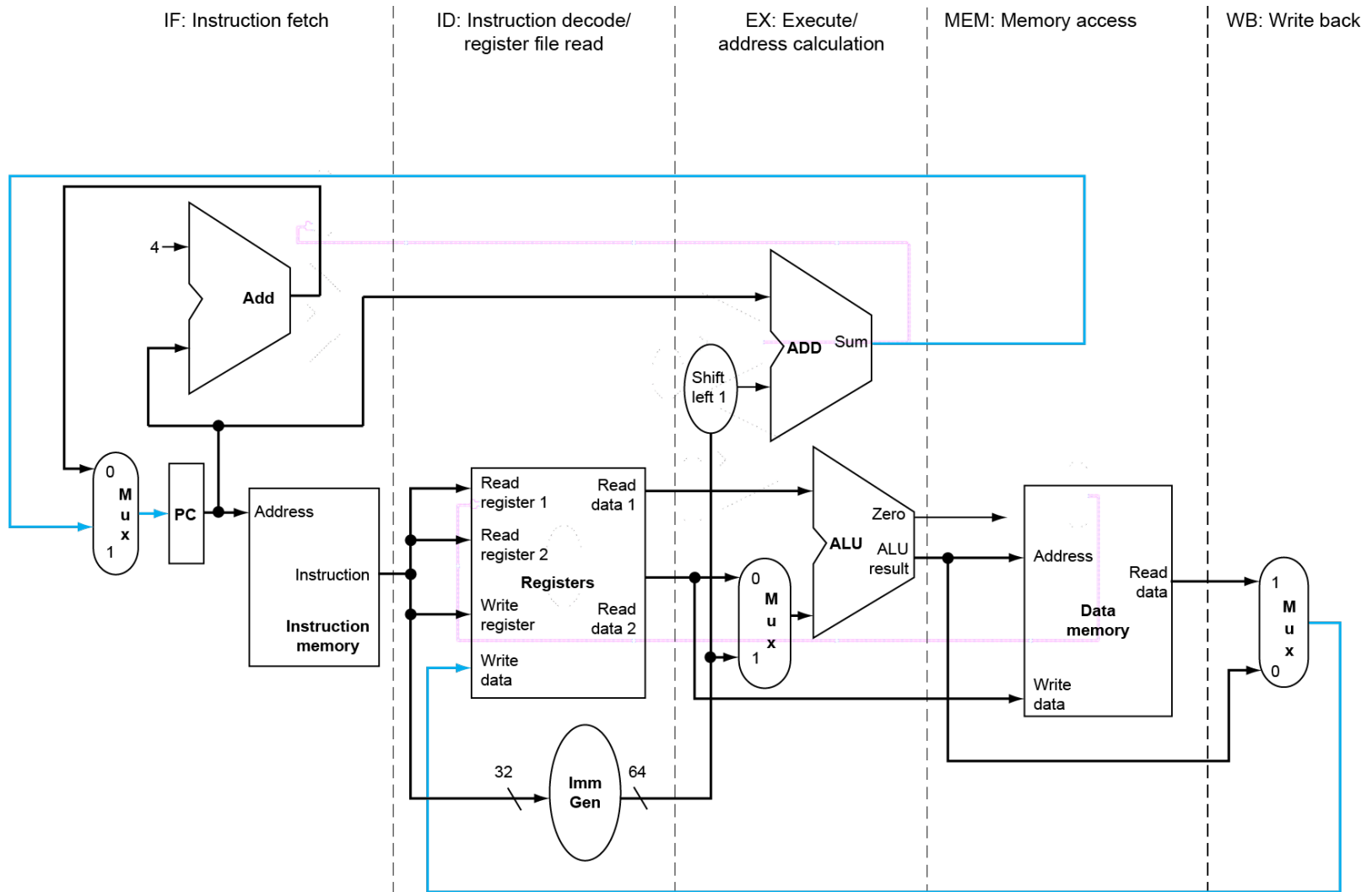
Branch Inst (i)	IF	ID	EX	MEM	WB			
Inst i+1		IF	ID	EX	MEM	WB		
Inst i+2			IF	ID	EX	MEM	WB	
Inst i+3				IF	ID	EX	MEM	WB
Inst i+4					IF	ID	EX	MEM

Correct Prediction : Zero Cycle Branch Penalty!

Branch Inst (i)	IF	ID	EX	MEM	WB			
Inst i+1		IF	nop	nop	nop	nop		
Branch target			IF	ID	EX	MEM	WB	

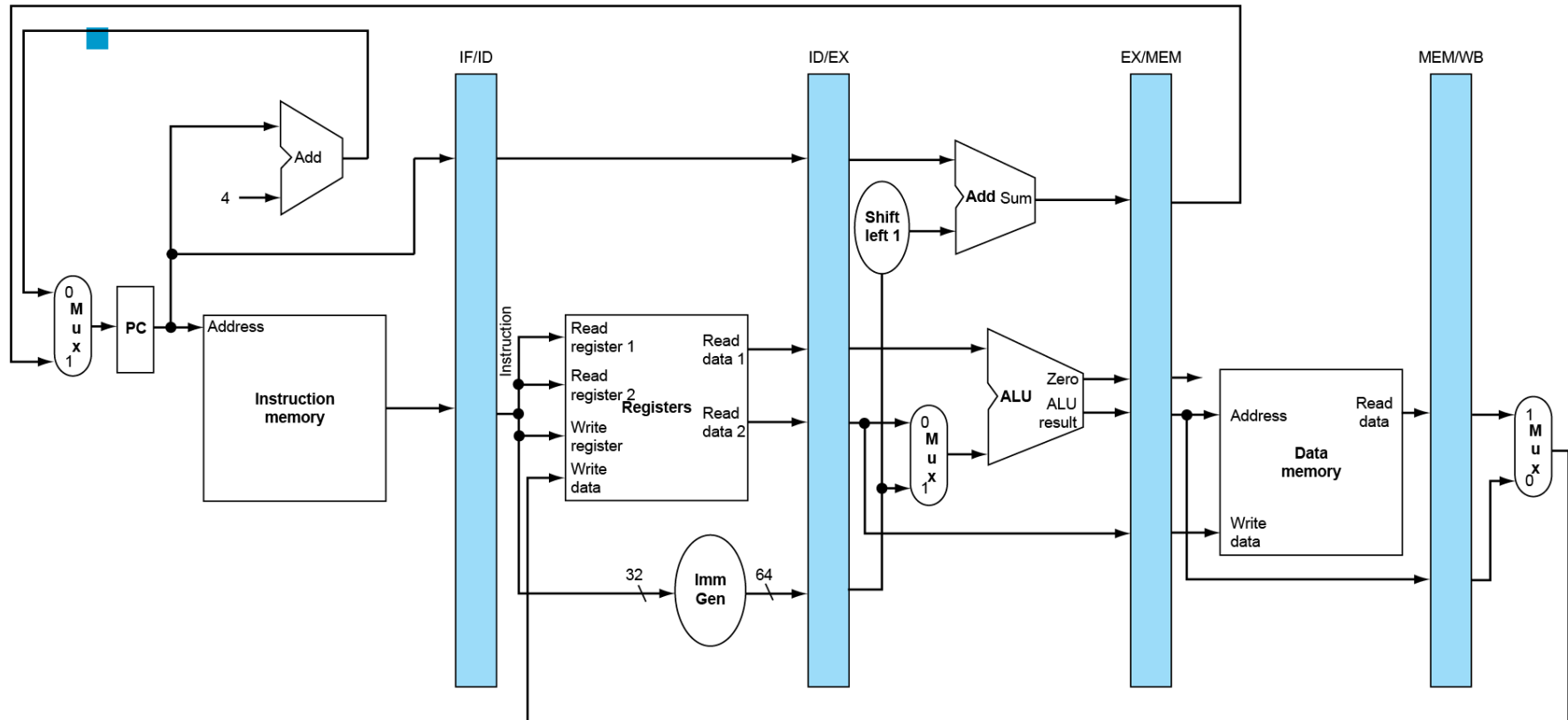
Incorrect Prediction - waste one cycle
How to flush pipeline?

A Pipelined Datapath: Basic Idea



■ What do we need to add to actually split the datapath into stages?

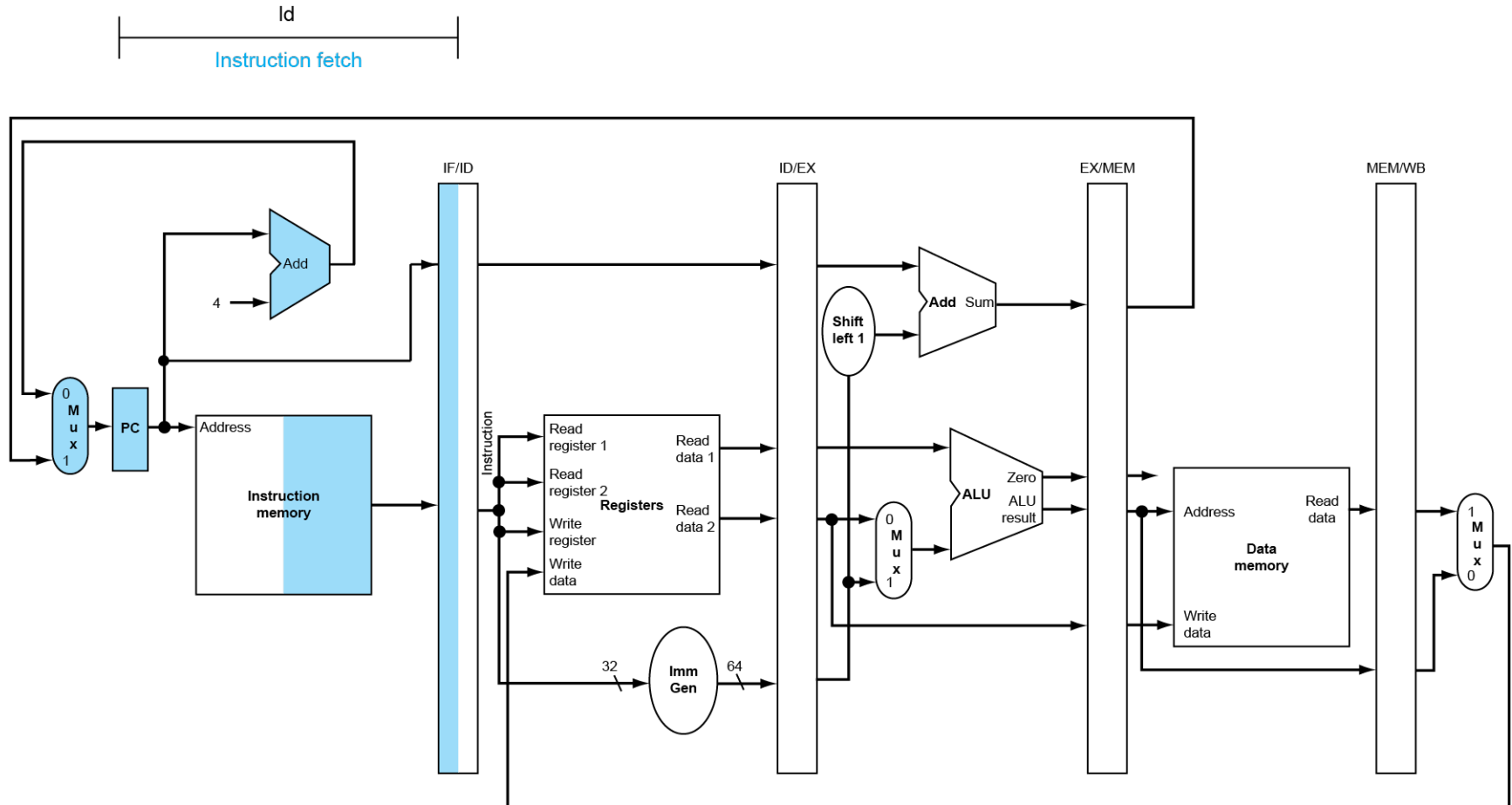
Pipelined Datapath



- Pipeline registers:
 - hold data needed for later pipeline stage
 - The registers are named for the two stages separated by that register.

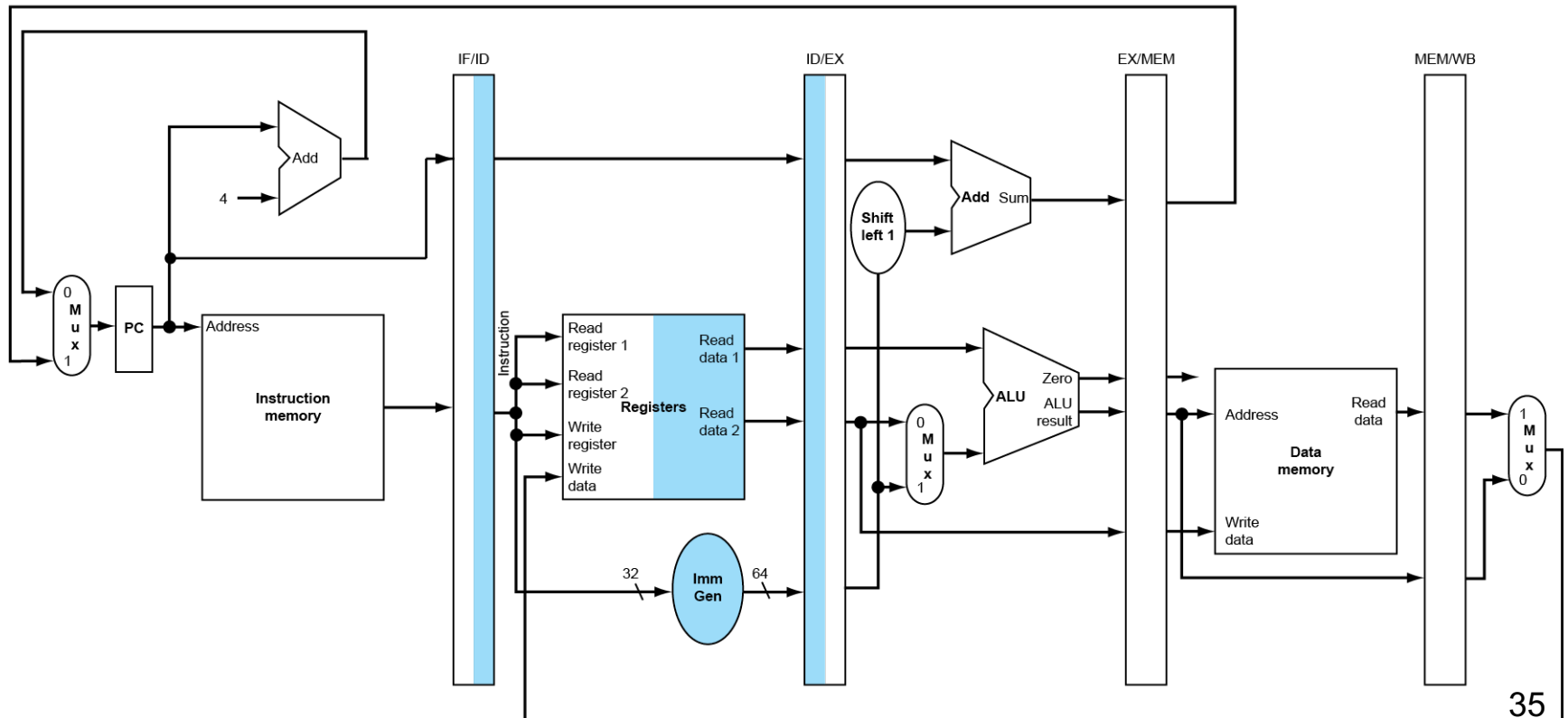
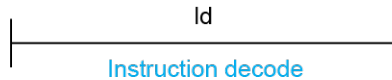
IF Stage of load

■ $IF/ID.IR = mem[PC]$; $IF/ID.PC = PC$; $PC = PC + 4$;



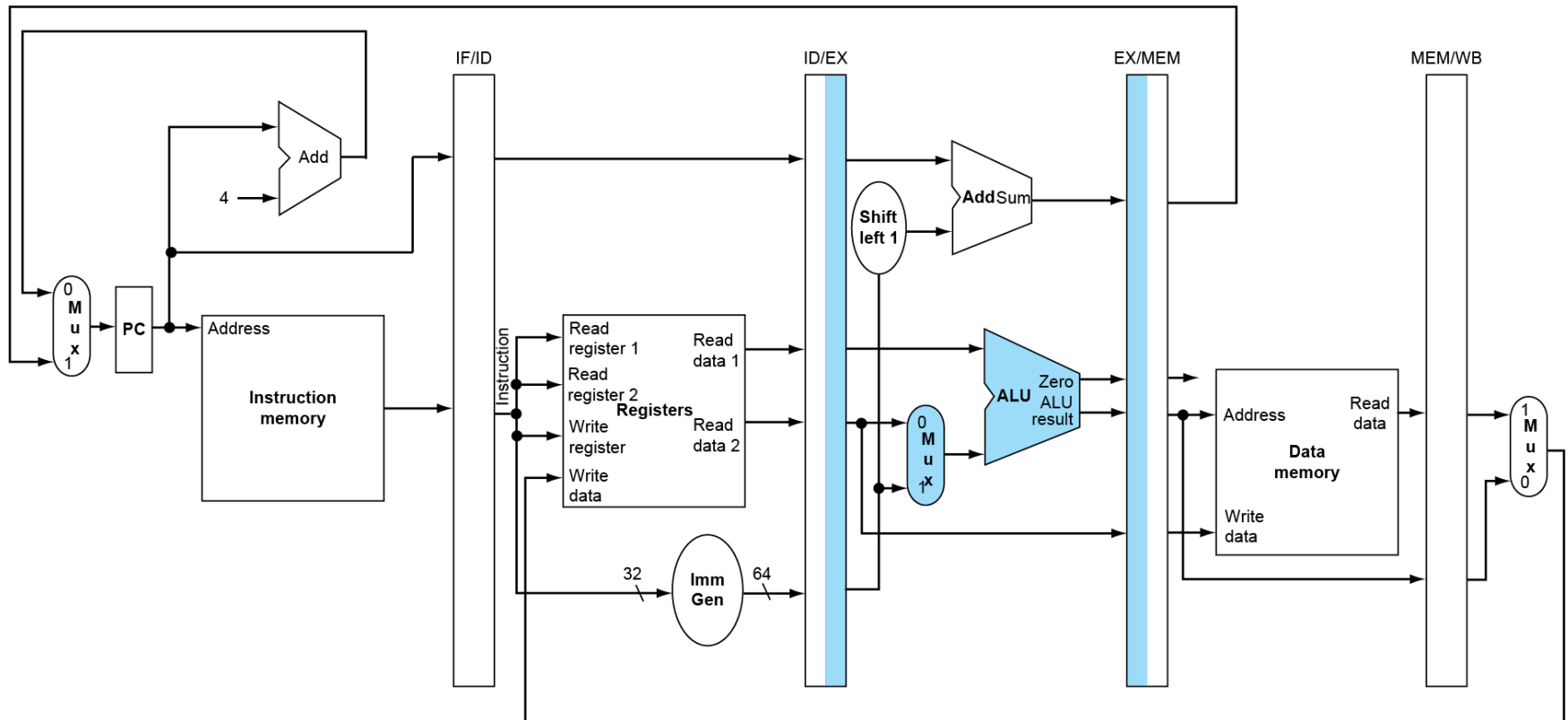
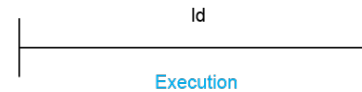
ID Stage of load

- $ID/EX.A = \text{Reg}[IF/ID.IR[19-15]]$
- $ID/EX.B = \text{Reg}[IF/ID.IR[11-7]]$
- $ID/EX.Immediate = (\text{sign-ext}(IF/ID.IR[31-20]))$
- $ID/EX.PC = IF/ID.PC$



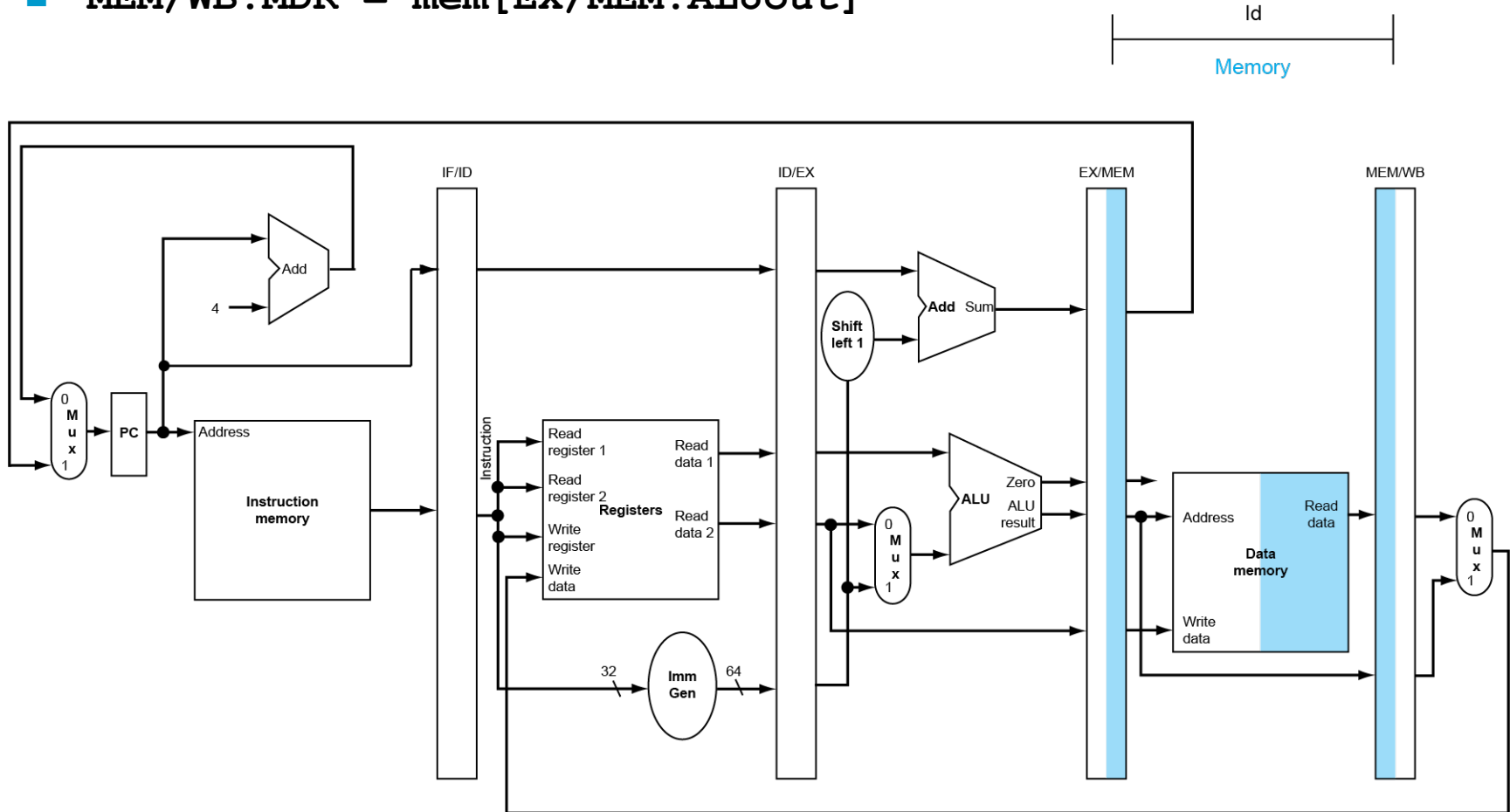
EX Stage of load

■ $EX/MEM.ALUout = ID/EX.A + ID/EX.immediate$



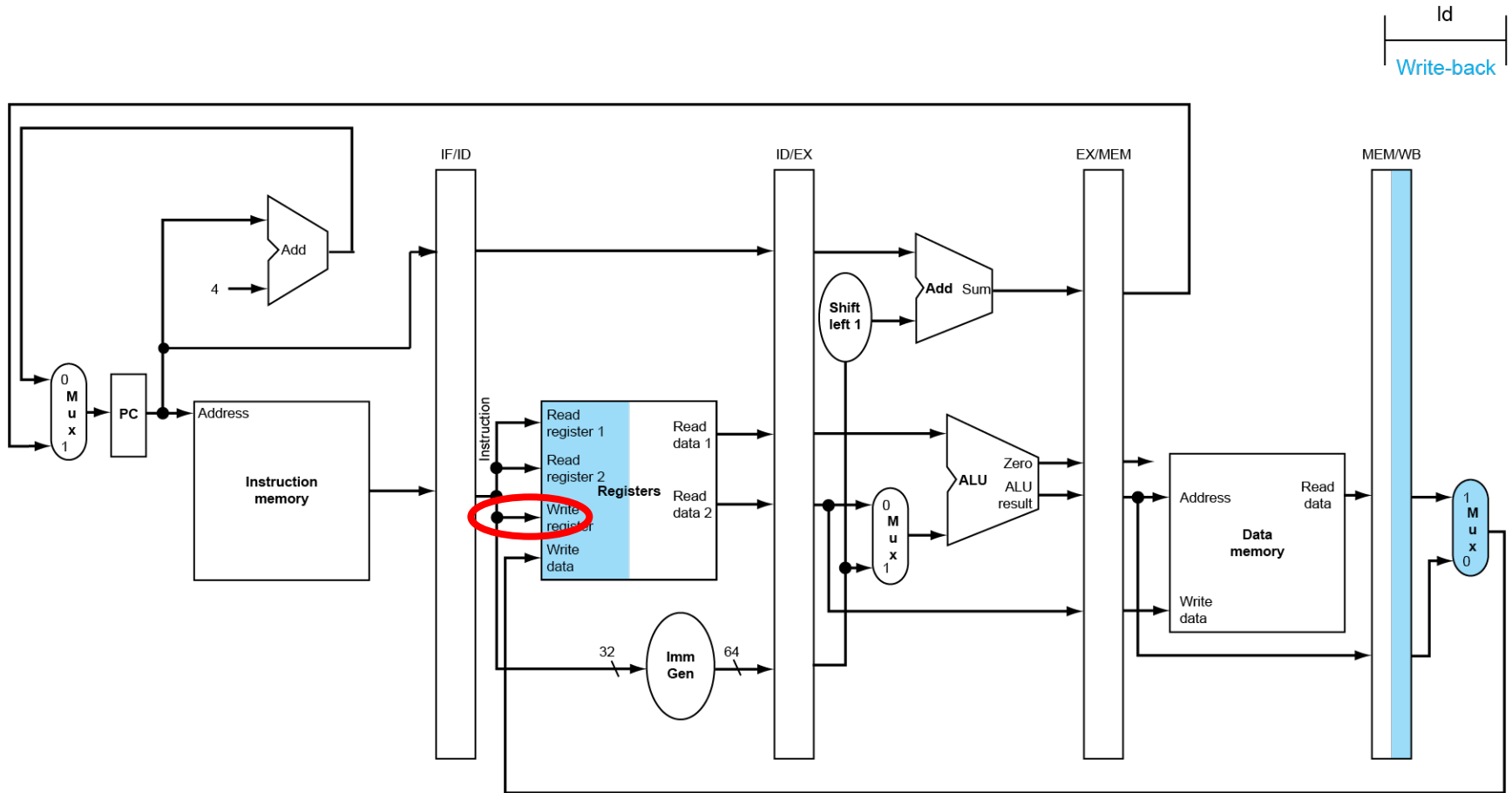
MEM Stage of load

■ $\text{MEM/WB.MDR} = \text{mem}[\text{EX/MEM.ALUout}]$



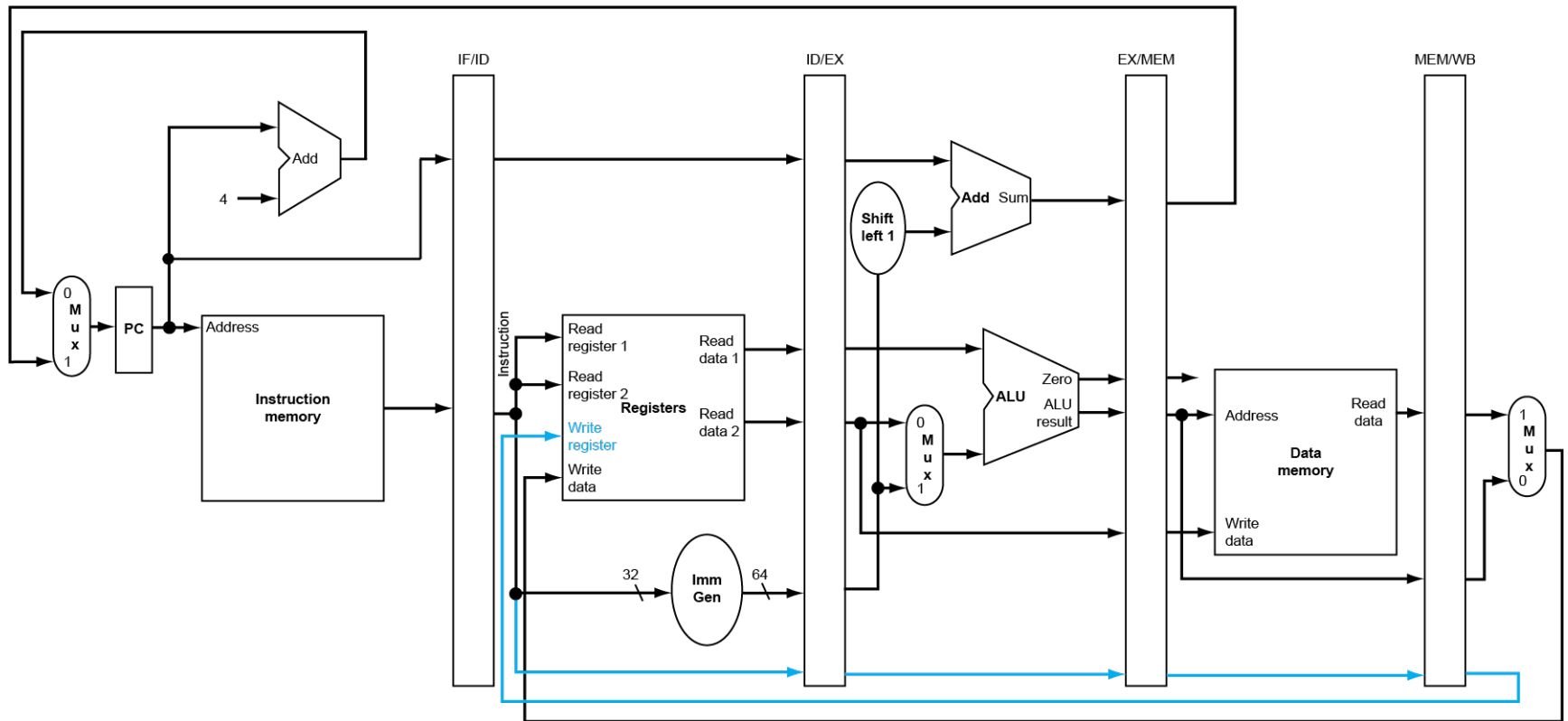
WB Stage of load

- $\text{Reg}[\text{Rd}] = \text{MEM/WB.MDR}$
 - Where does Rd come from?



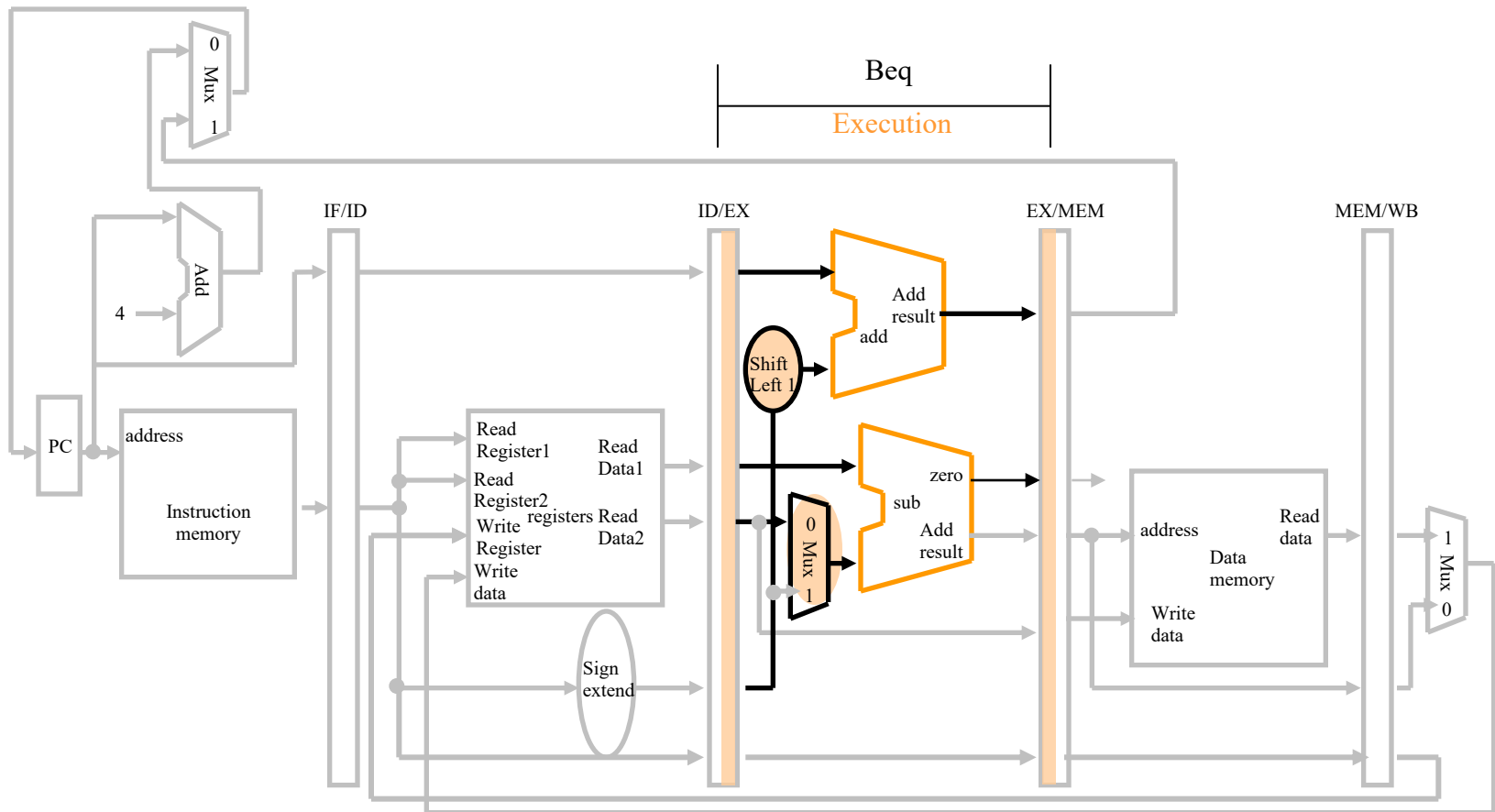
- Load must pass the register number from ID/EX through EX/MEM to the MEM/WB pipeline register for use in the WB stage

Corrected Datapath



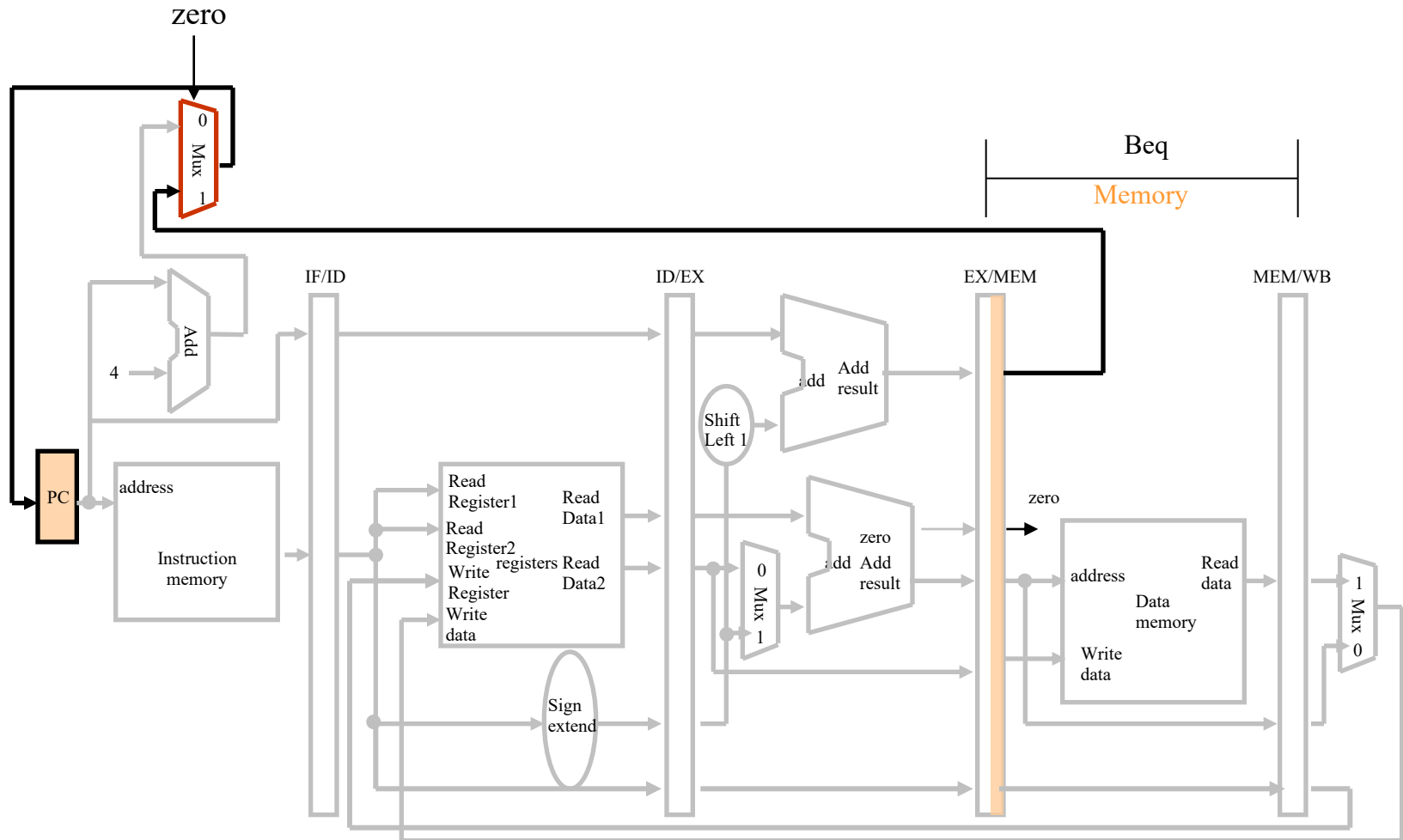
EX Stage of beq

- $EX/MEM.PC = ID/EX.PC + (ID/EX.immediate \ll 1);$
- $EX/MEM.Zero = (ID/EX.A - ID/EX.B) ? 0 : 1$

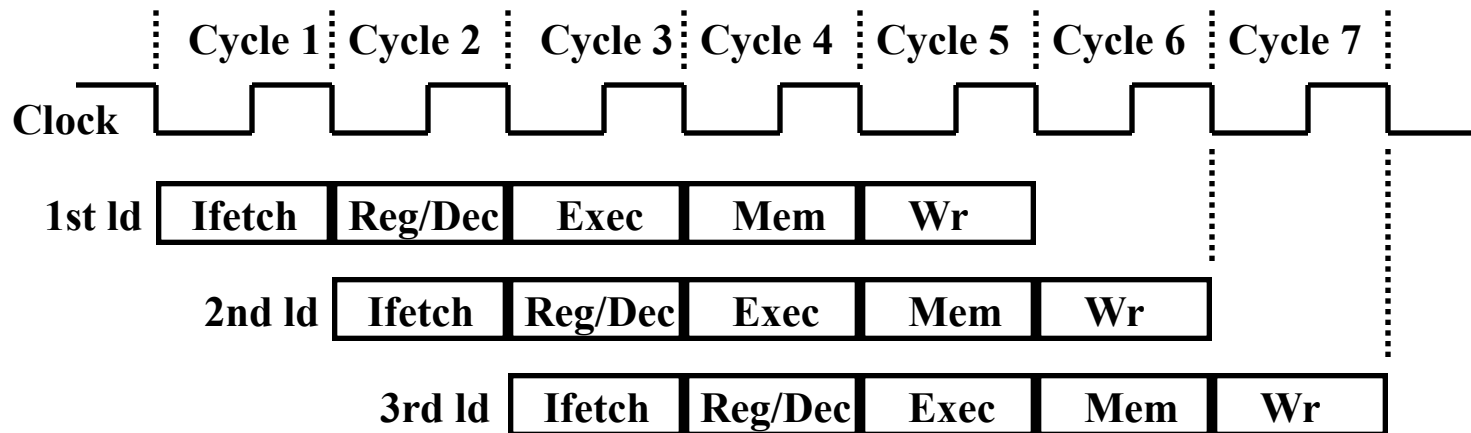


MEM Stage of beq

- If (EX/MEM.zero) PC = EX/MEM.PC

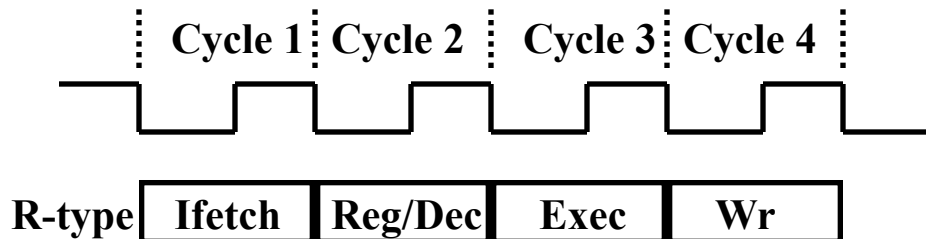


Pipelining the Load Instruction



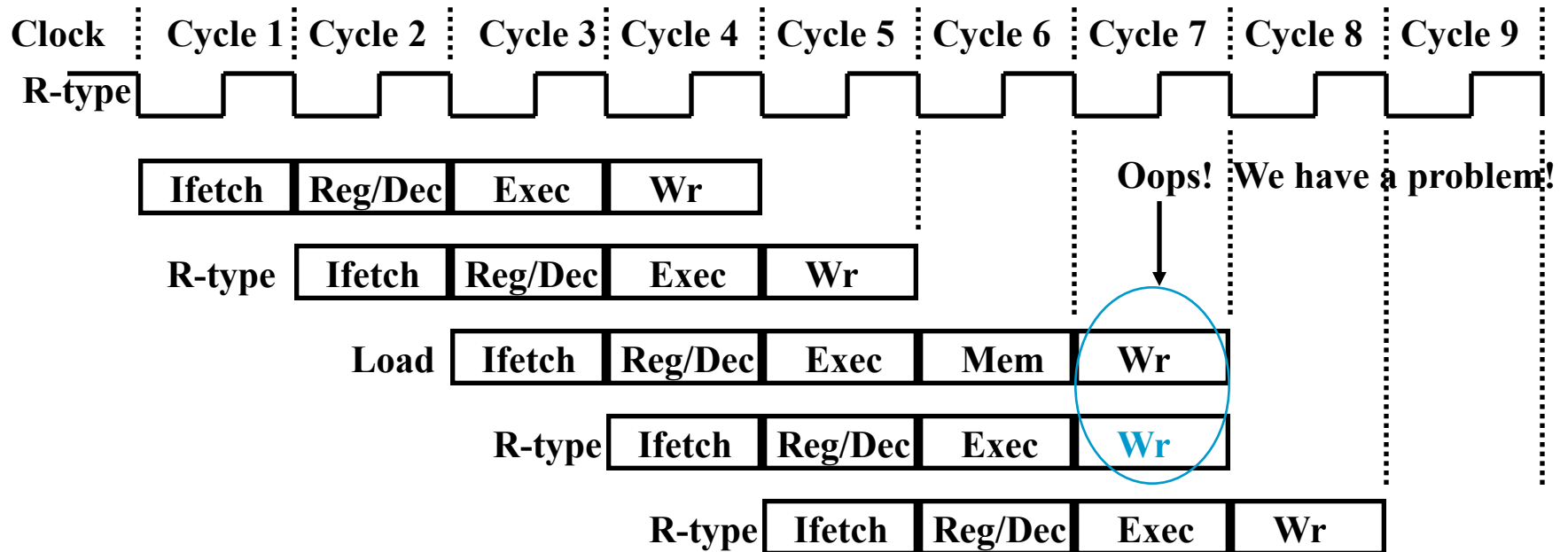
- The five independent functional units in the pipeline datapath are:
 - Instruction Memory for the **Ifetch** stage
 - Register File's Read ports (bus A and busB) for the **Reg/Dec** stage
 - ALU for the **Exec** stage
 - Data Memory for the **Mem** stage
 - Register File's **Write** port (bus W) for the **Wr** stage

The Four Stages of R-type



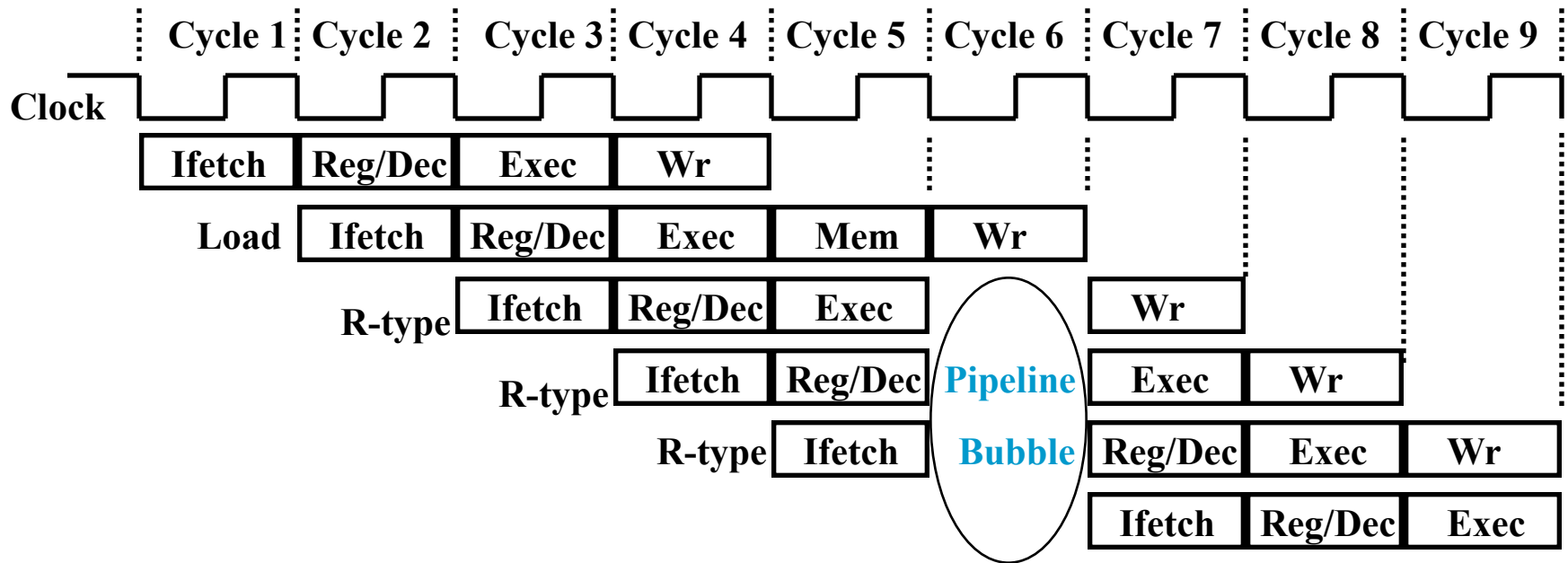
- **Ifetch: Instruction Fetch**
 - Fetch the instruction from the Instruction Memory
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec:**
 - ALU operates on the two register operands
- **Wr: Write the ALU output back to the register file**

Pipelining the R-type and Load Instruction



- We have pipelined conflict or structural hazard:
 - Two instructions try to write to the register file at the same time!
 - Only one write port

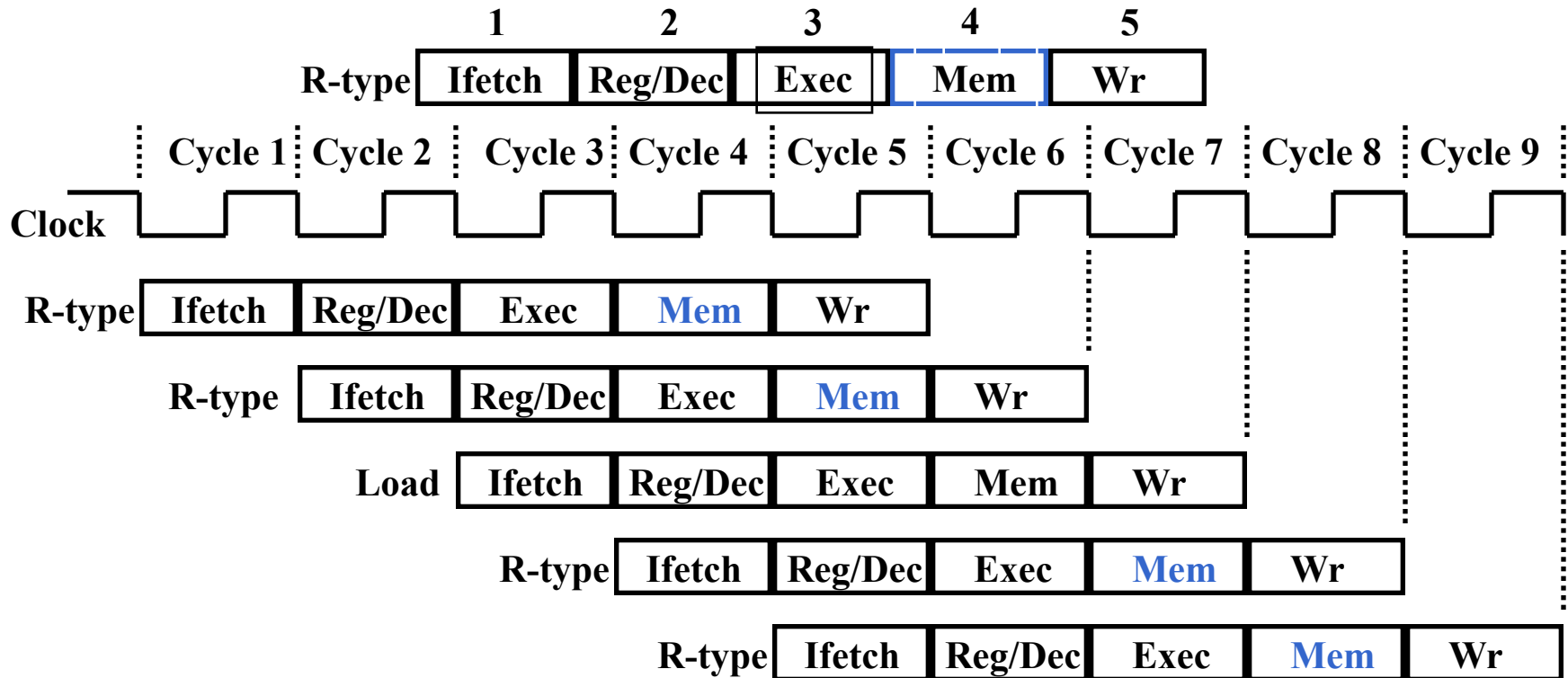
Solution 1: Insert “Bubble” into the Pipeline



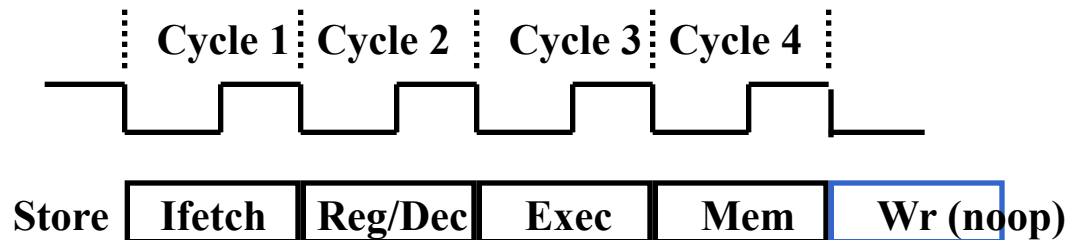
- Insert a “bubble” into the pipeline to prevent 2 writes at the same cycle
 - The control logic can be complex.
 - Lose instruction fetch and issue opportunity.
 - No instruction is started in Cycle 6!

Solution 2: Delay R-type's Write by One Cycle

- Delay R-type's register write by one cycle:
 - Now R-type instructions also use Reg File's write port at Stage 5
 - Mem stage is a **NOOP** stage: nothing is being done.

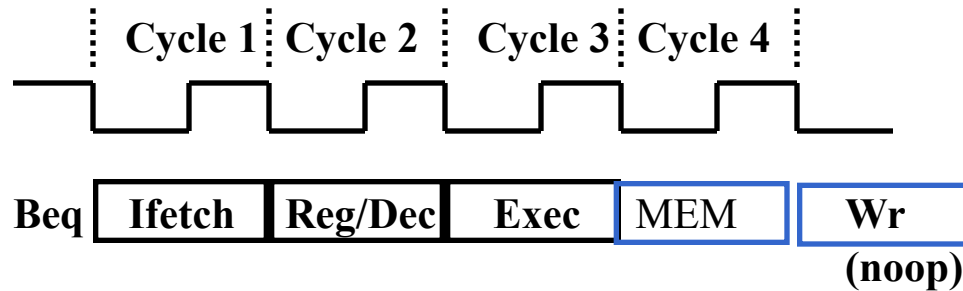


The Four Stages of Store



- Ifetch: Instruction Fetch
 - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

The Four Stages of Beq



■ Ifetch: Instruction Fetch

- Fetch the instruction from the Instruction Memory

■ Reg/Dec:

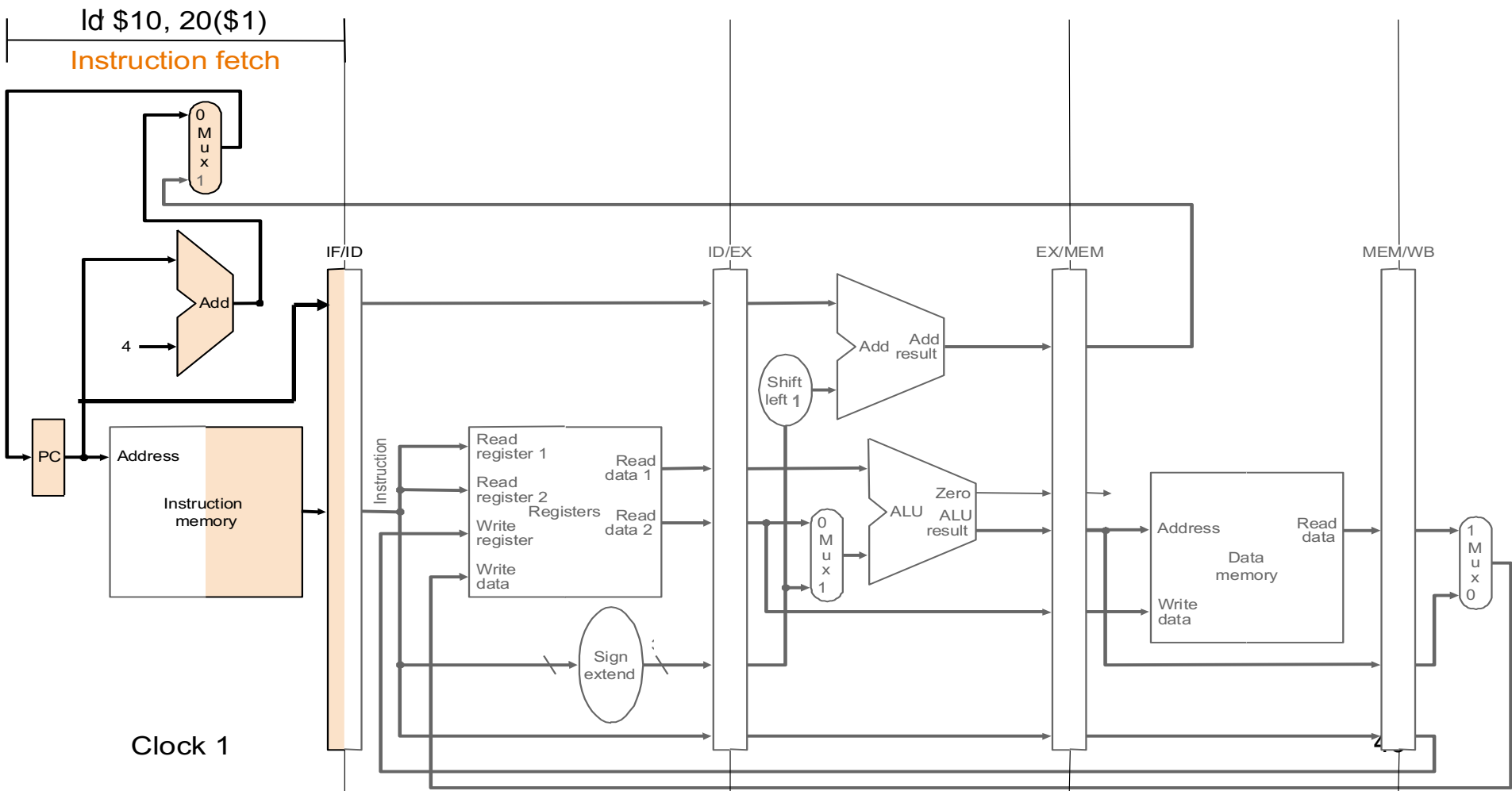
- Registers Fetch and Instruction Decode

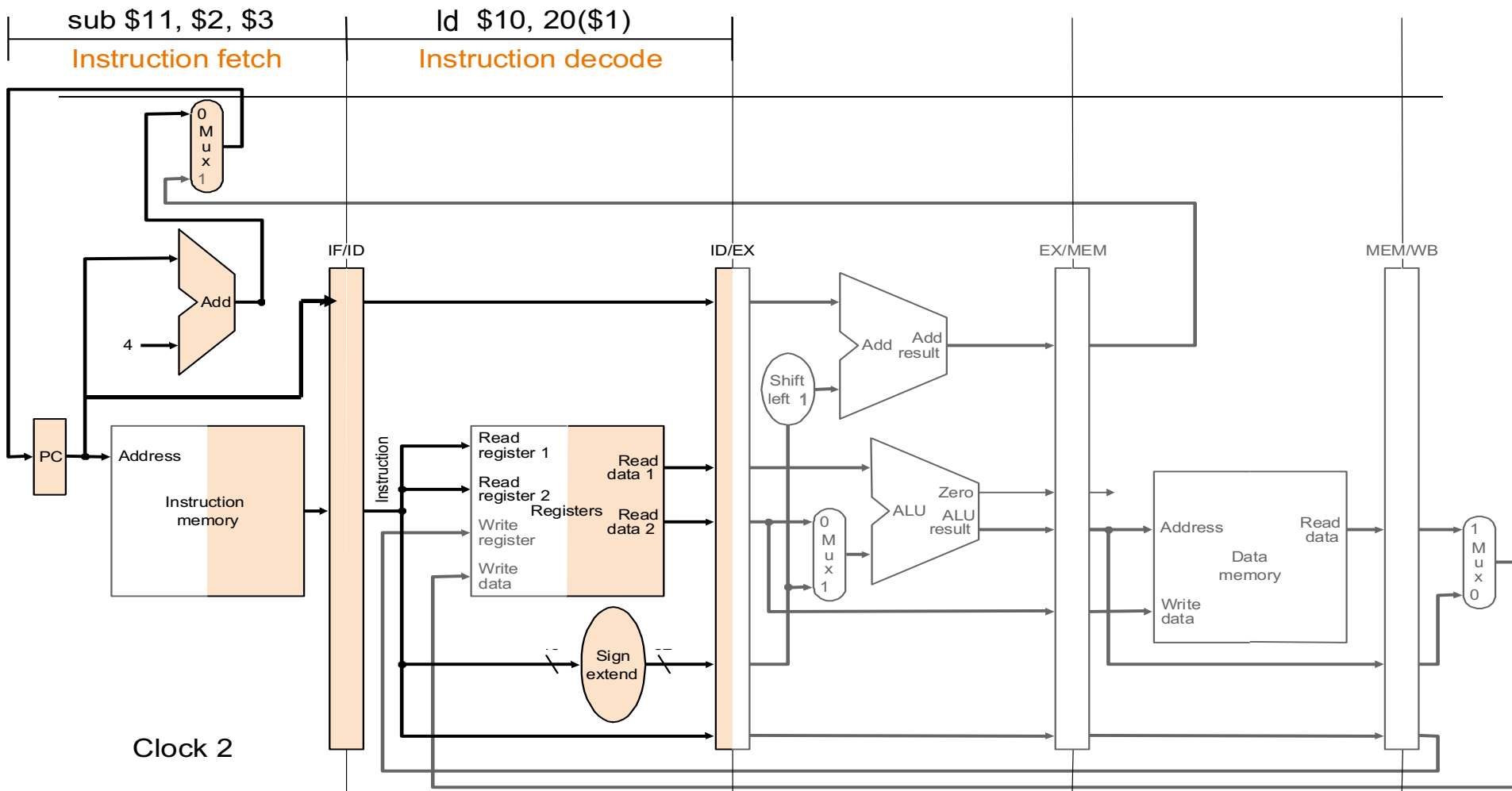
■ Exec:

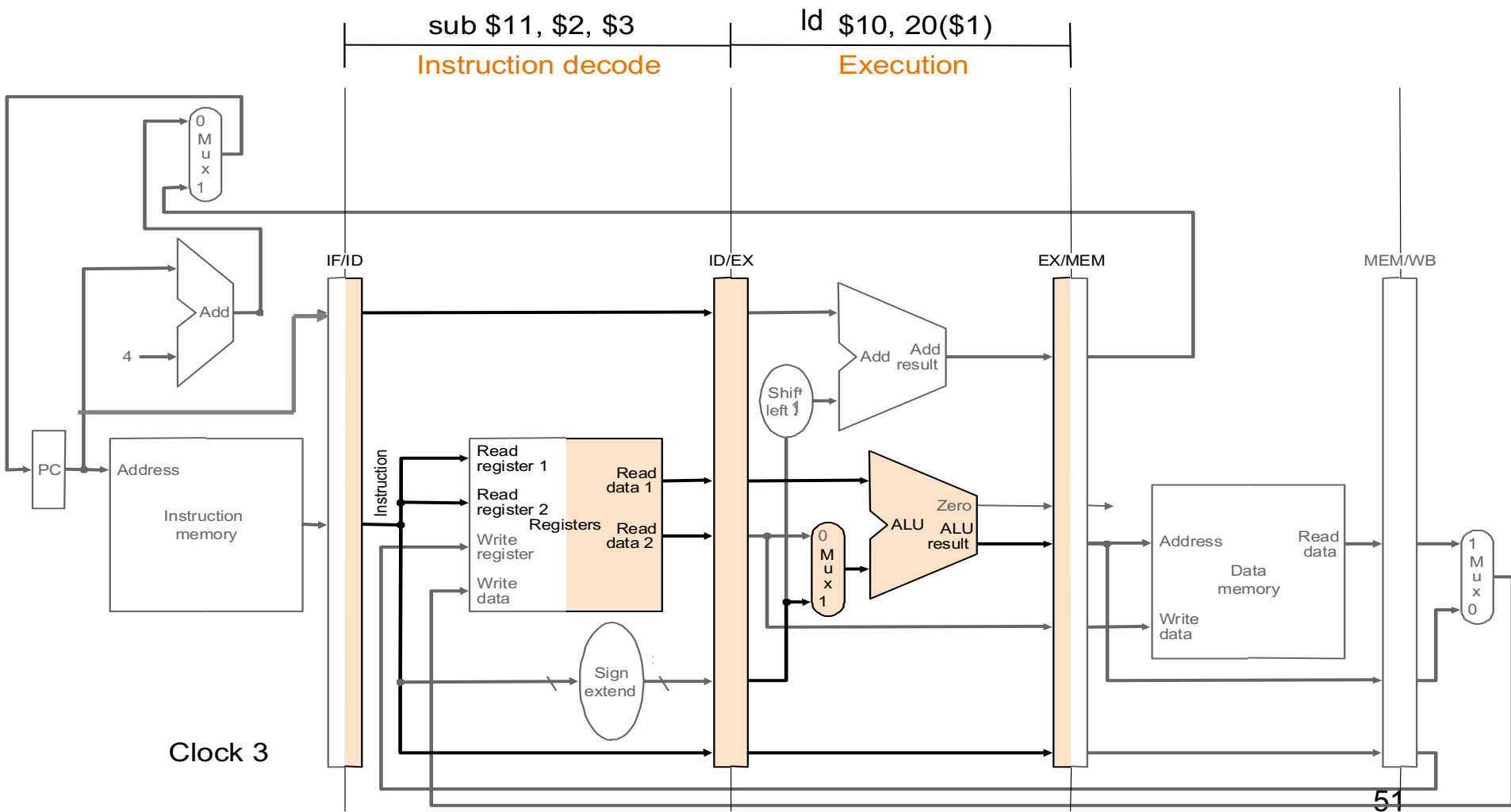
- compares the two register operand,

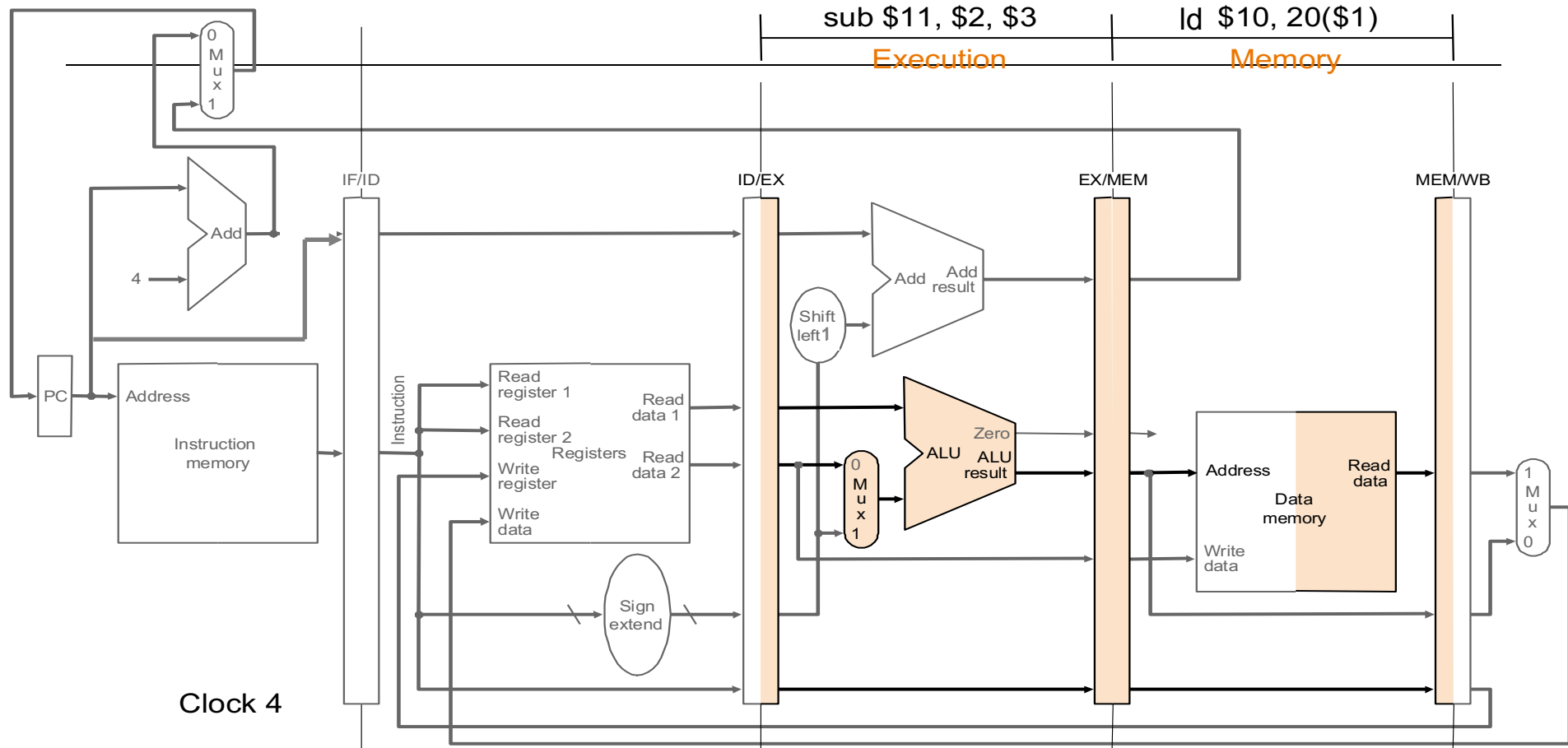
■ MEM

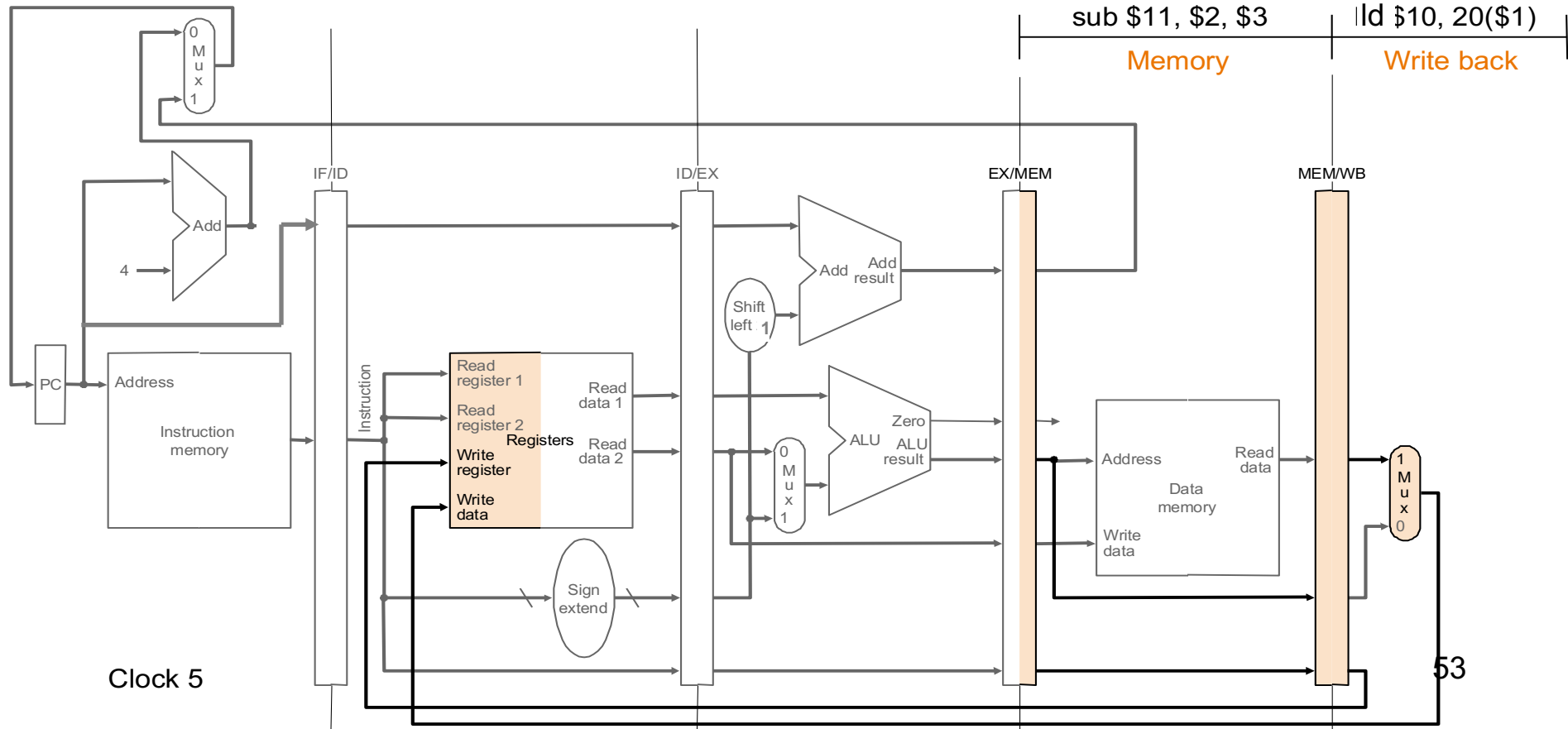
- select correct branch target address
- latch into PC











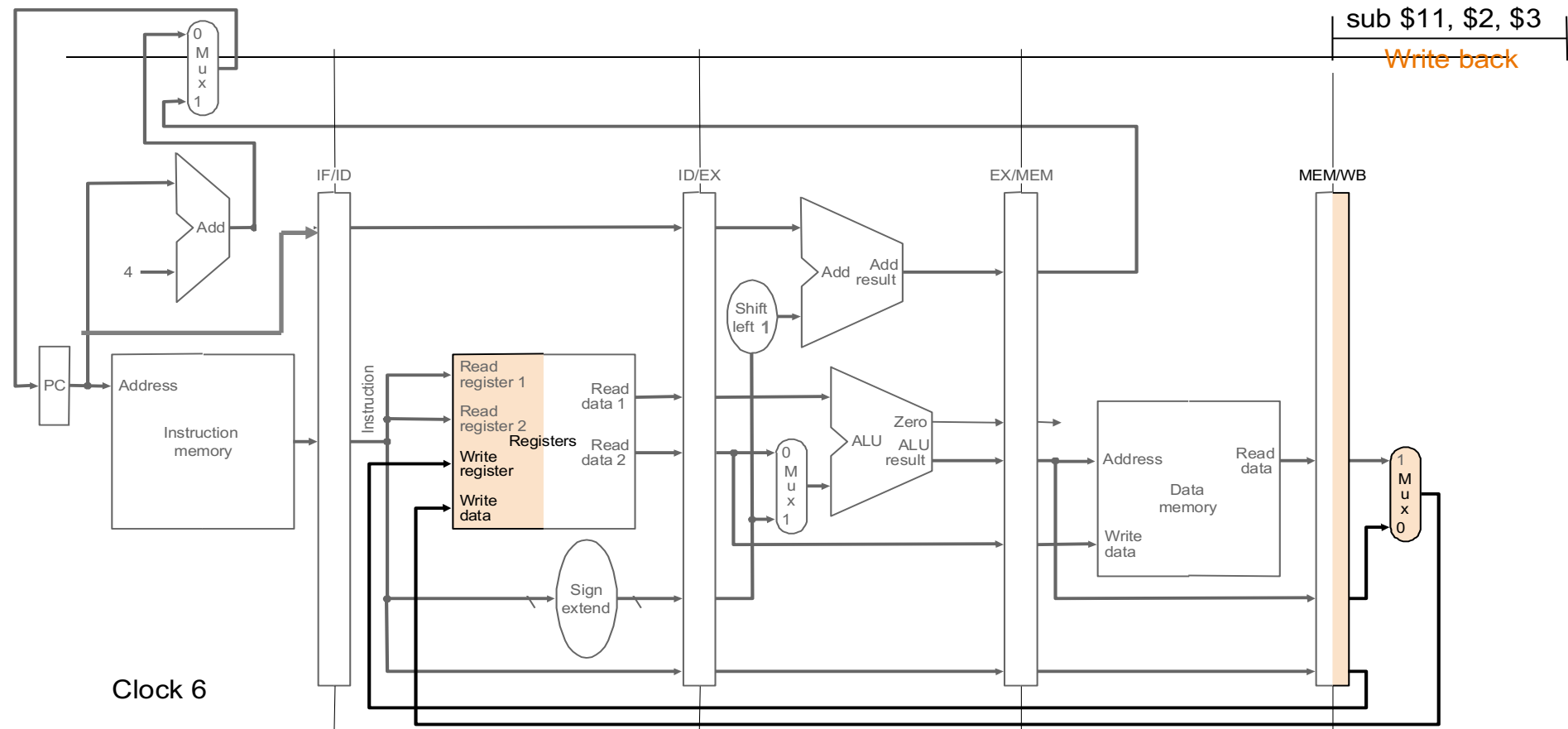
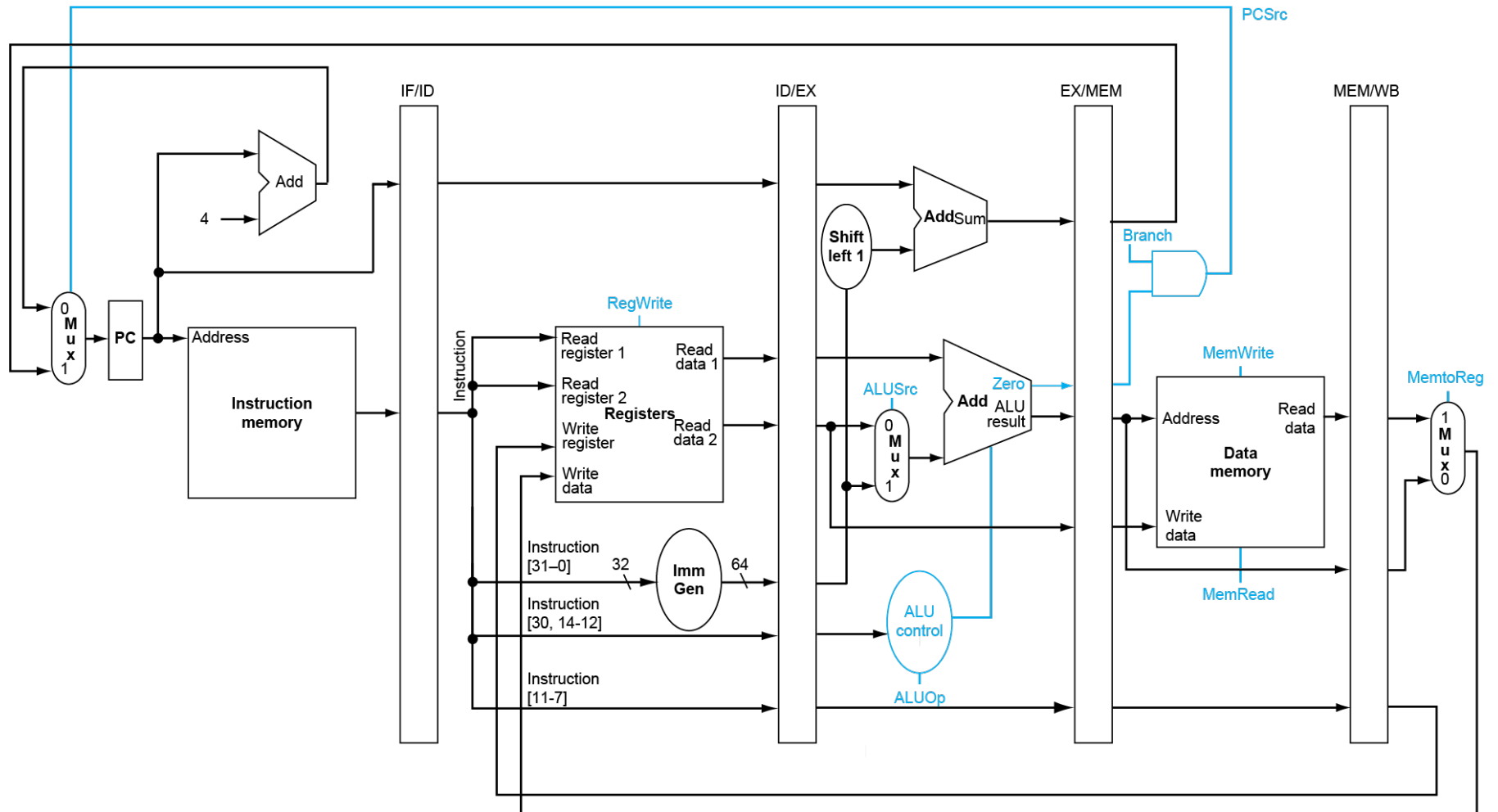


Fig. 6.24

Pipeline control

- We have 5 stages. What needs to be controlled in each stage?
 - Instruction Fetch and PC Increment
 - Instruction Decode / Register Fetch
 - Execution
 - Memory Stage
 - Write Back

Pipelined Datapath with Control Signals Identified



1. Program counter is updated in the mem stage for beq
2. Do we need PCwrite control signal?

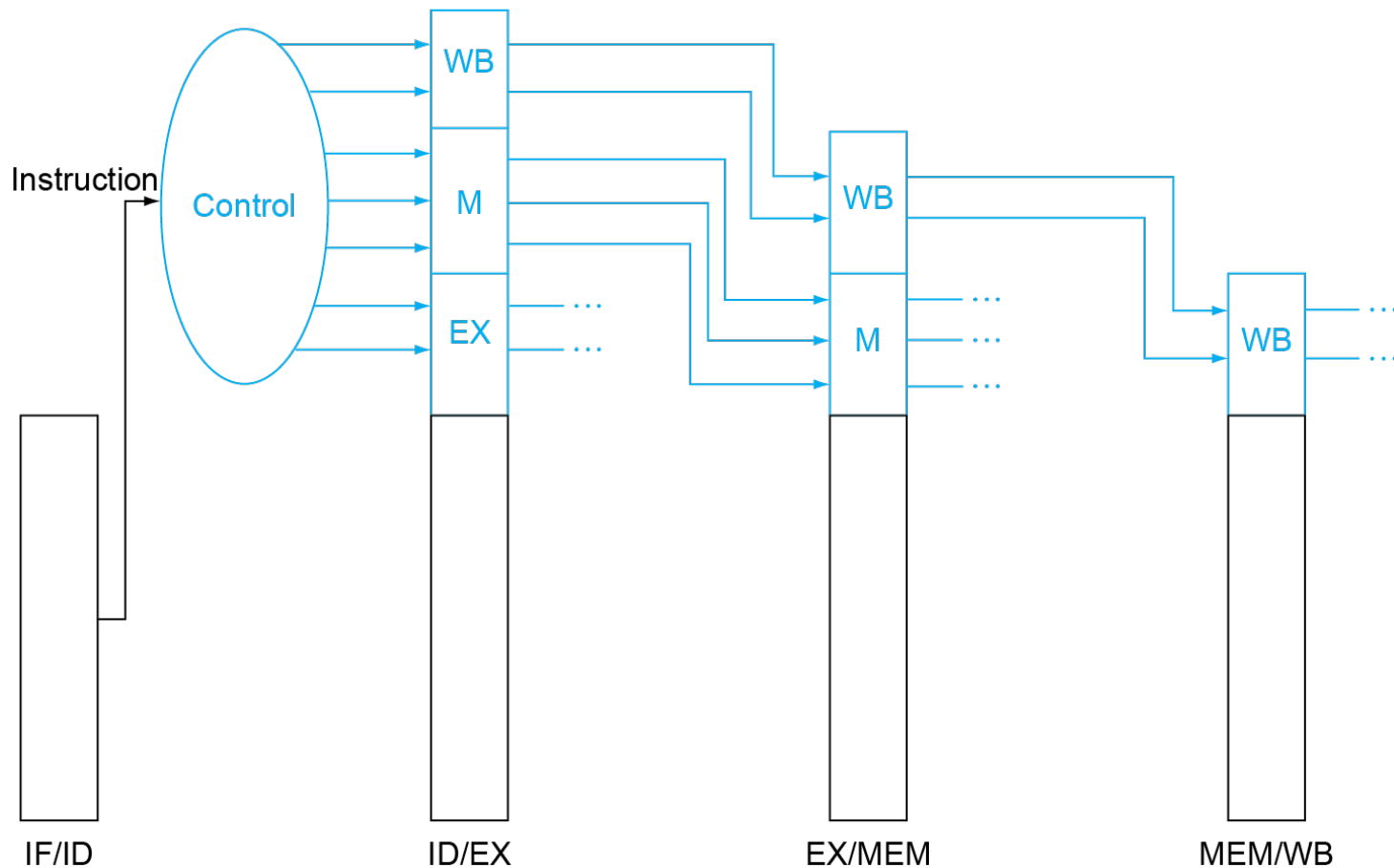
Group Signals According to Stages

- Can use control signals of single-cycle CPU

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

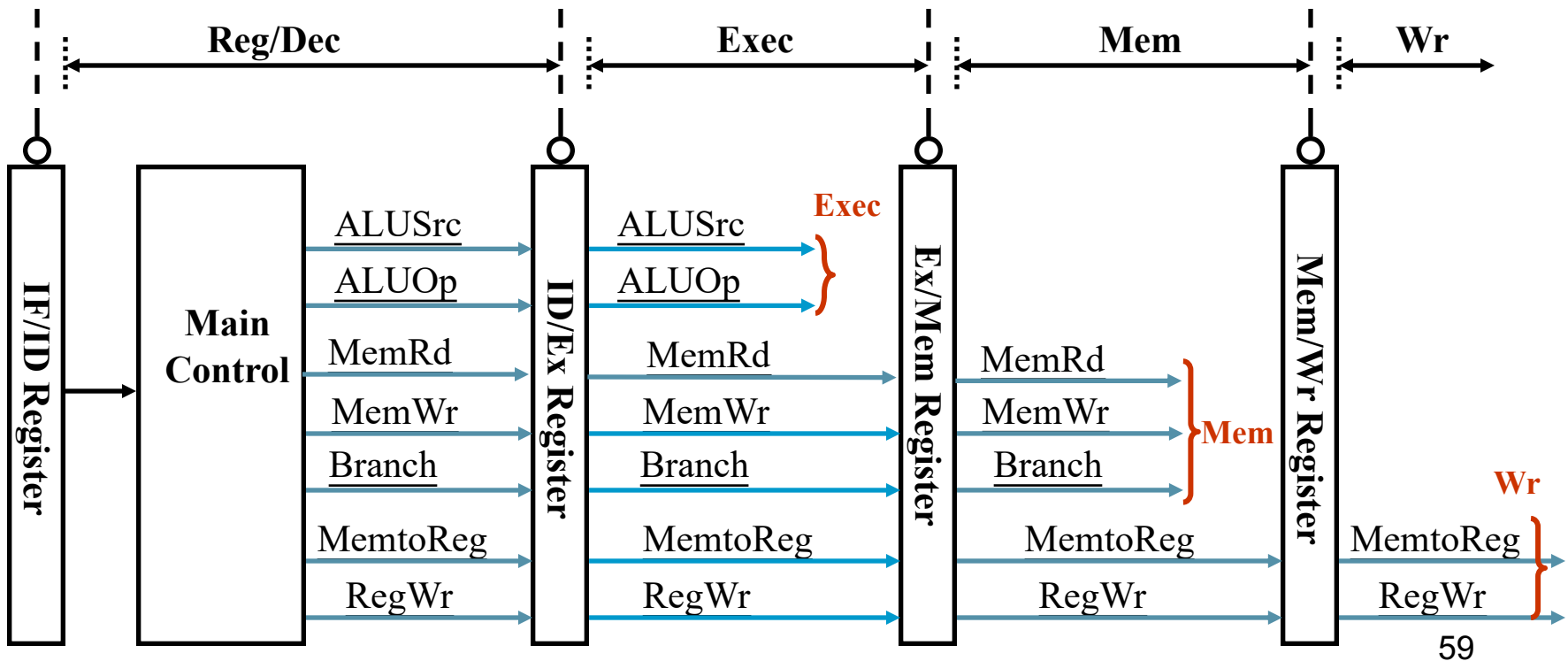
Control Lines for the Final Three Stages

- Pass control signals along just like the data
 - Main control generates control signals during ID

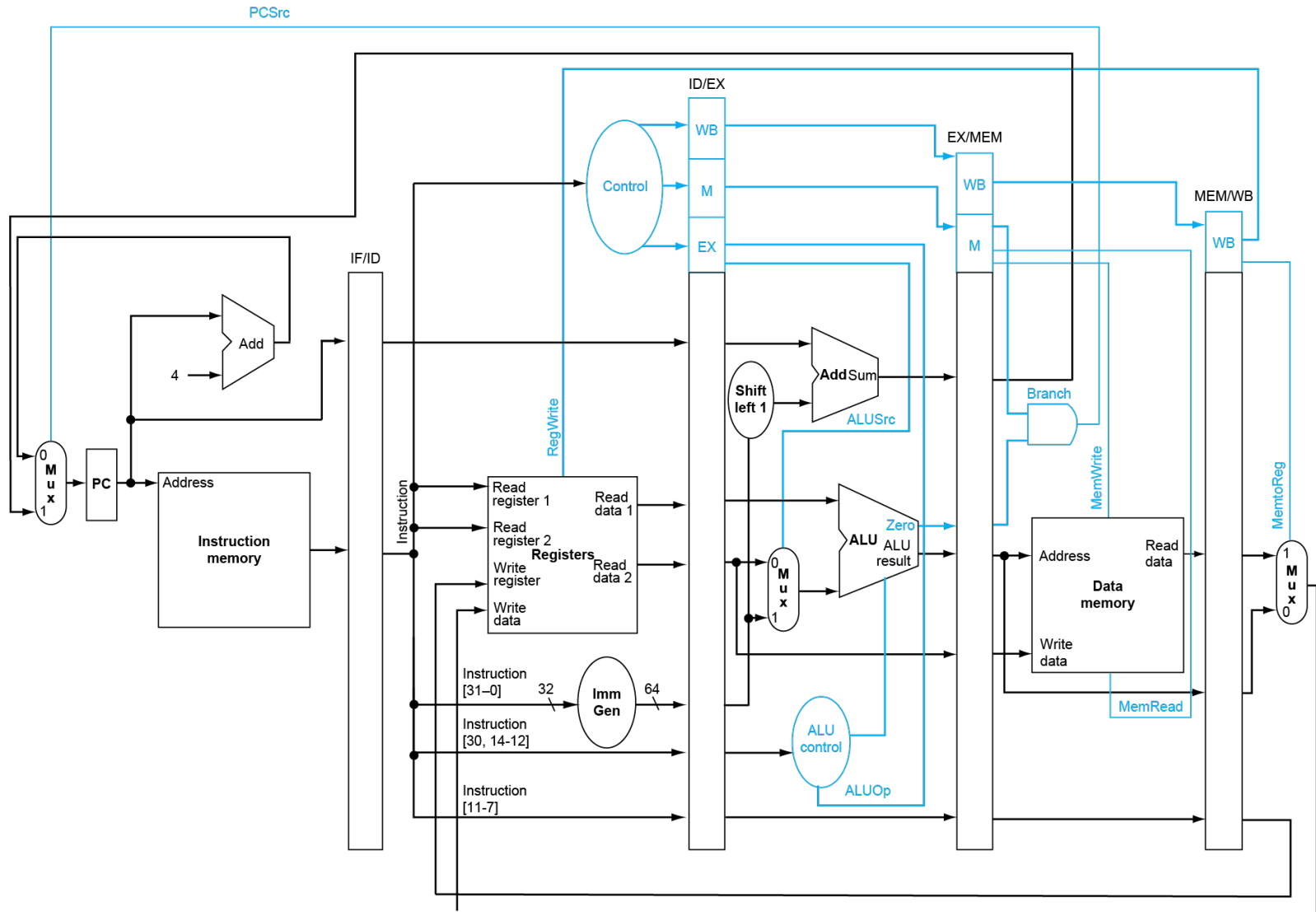


Pipeline Control

- The Main Control generates the control signals during Reg/Dec
 - Control signals for Exec (ALUSrc, ...) are used 1 cycle later
 - Control signals for Mem (MemWr, ...) are used 2 cycles later
 - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



Datapath with Control



Instructions Pipelining Example

ld \$10, 20(\$1)

sub \$11, \$2, \$3

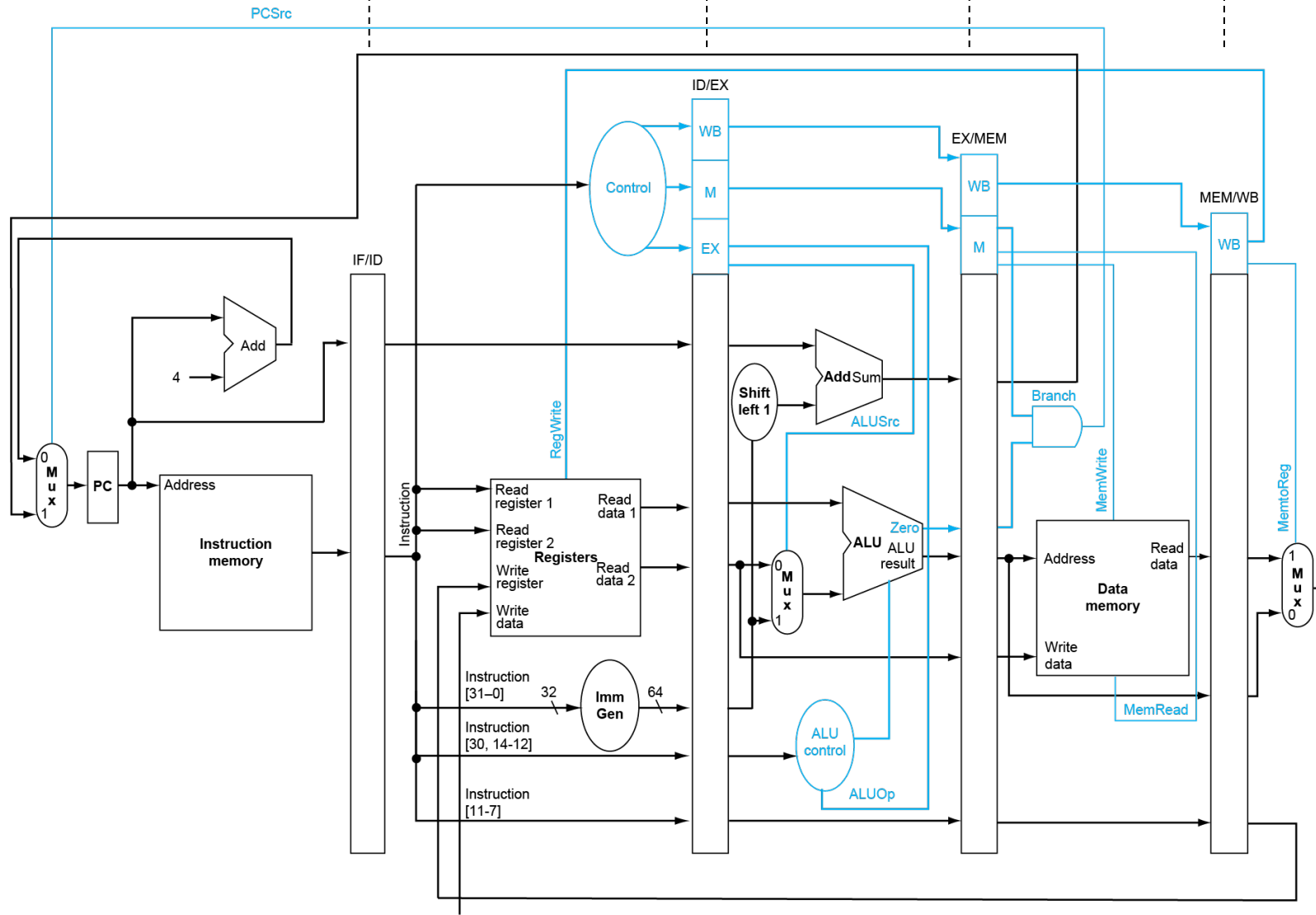
and \$12, \$4, \$5

or \$13, \$6, \$7

add \$14, \$8, \$9

Cycle 1

ld \$t0, 20(\$t1)



1d \$10, 20 (\$1)

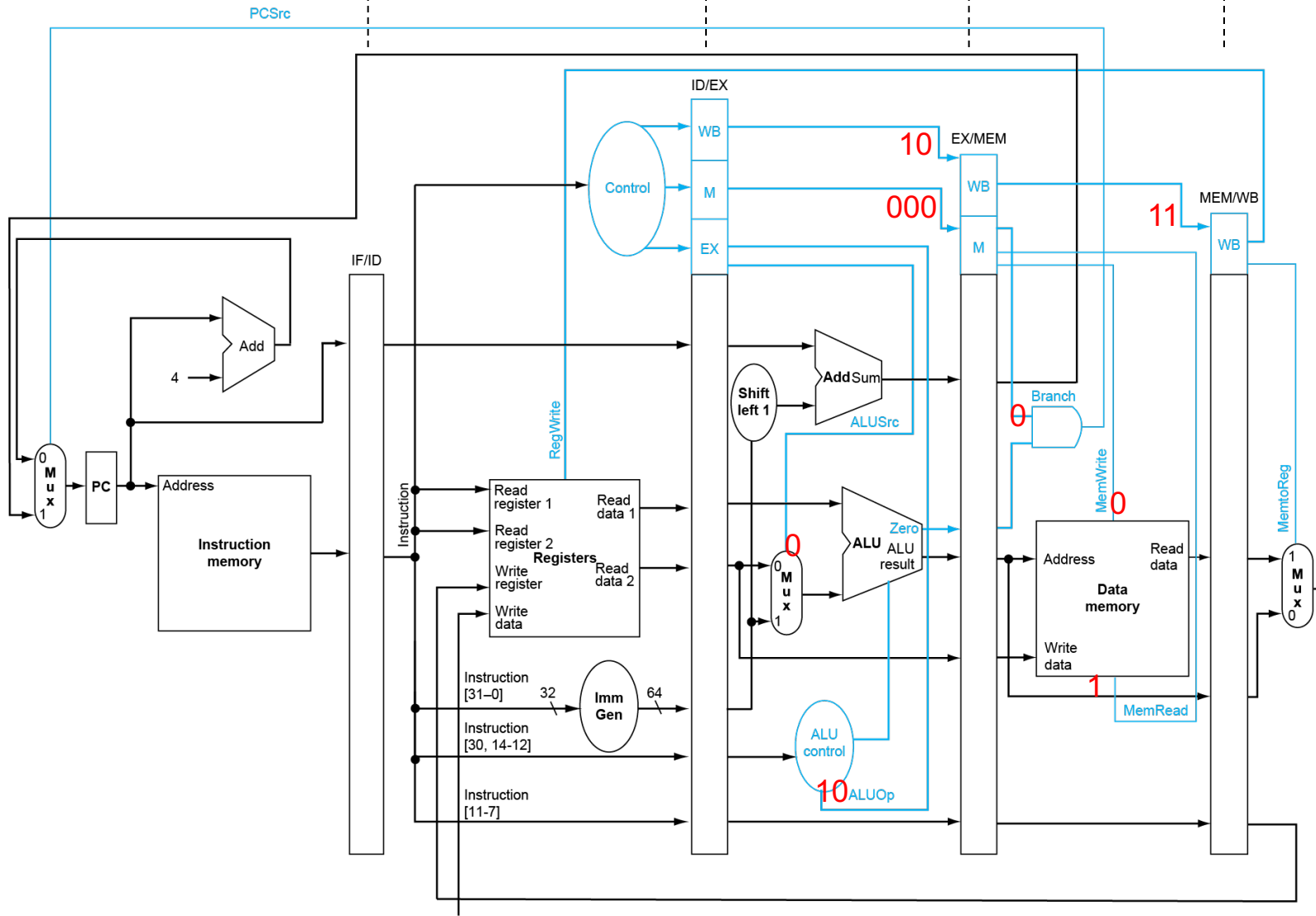


1d \$10,20 (\$1)



Cycle 4

or \$13,\$6,\$7 and \$12,\$4,\$5 sub \$11,\$2,\$3 ld \$10,20(\$1)



Cycle 5

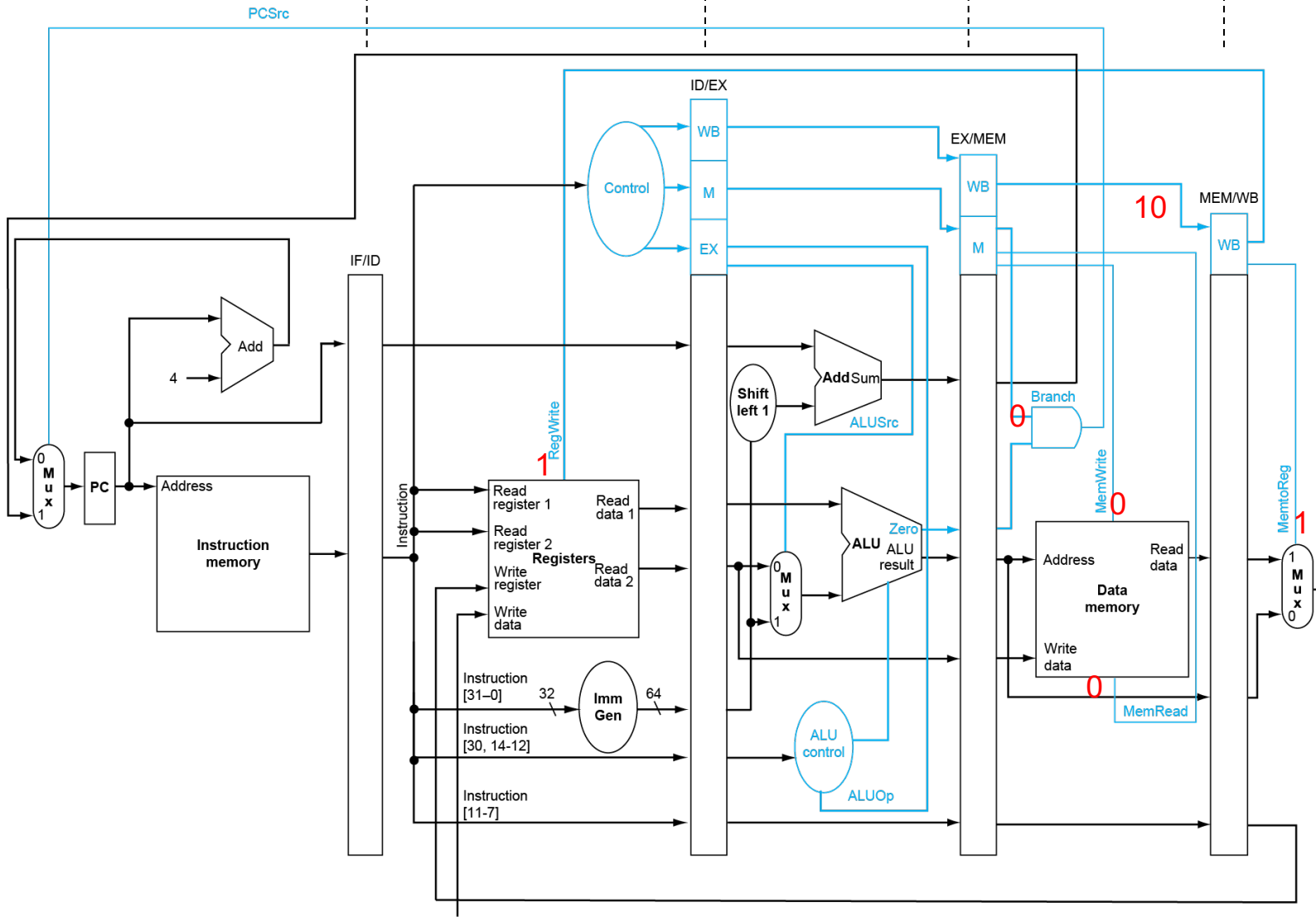
add \$14,\$8,\$9

or \$13,\$6,\$7

and \$12,\$4,\$5

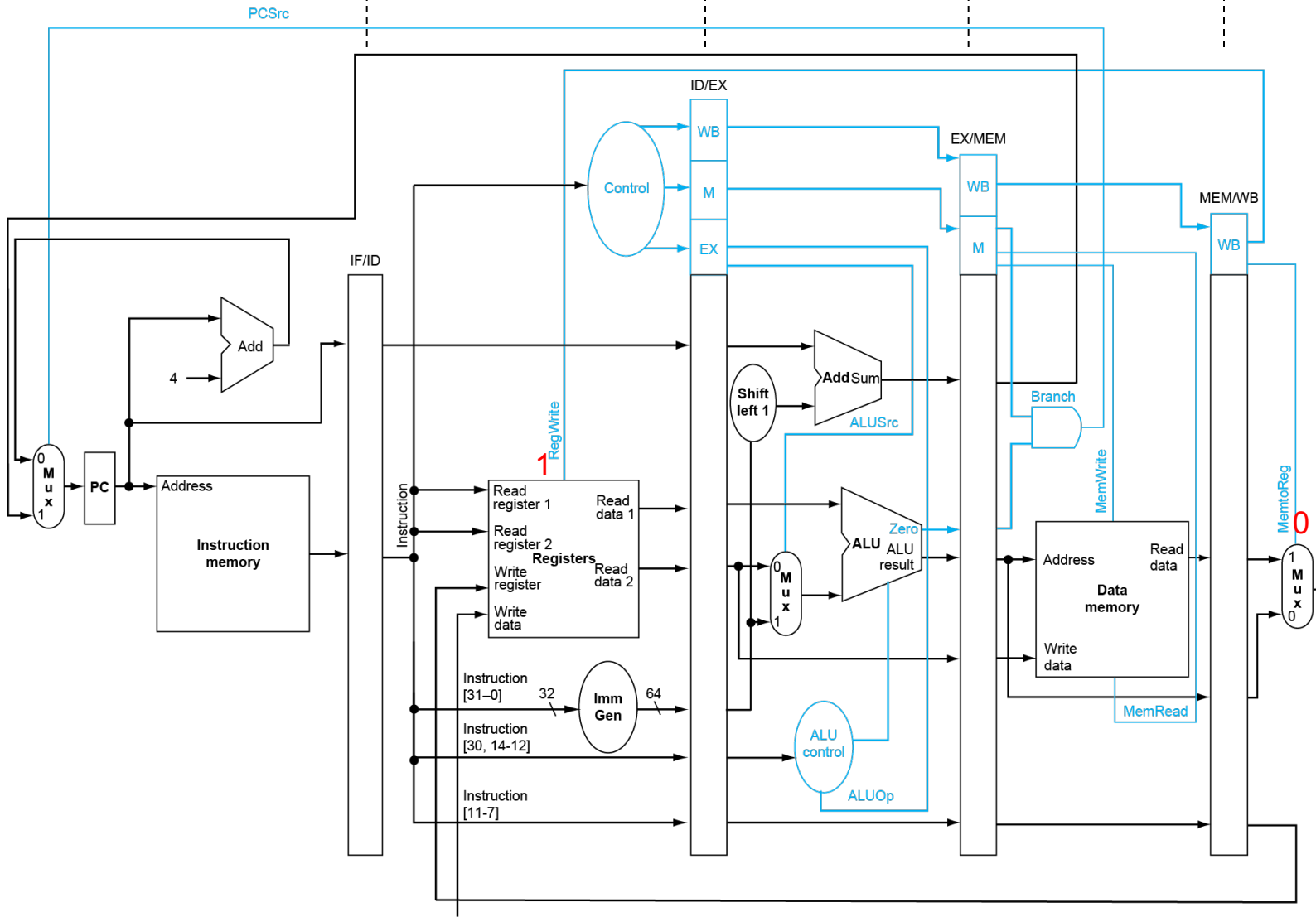
sub \$11,\$2,\$3

ld \$10,20(\$1)



Cycle 6

add \$14,\$8,\$9 or \$13,\$6,\$7 and \$12,\$4,\$5 sub \$11,\$2,\$3

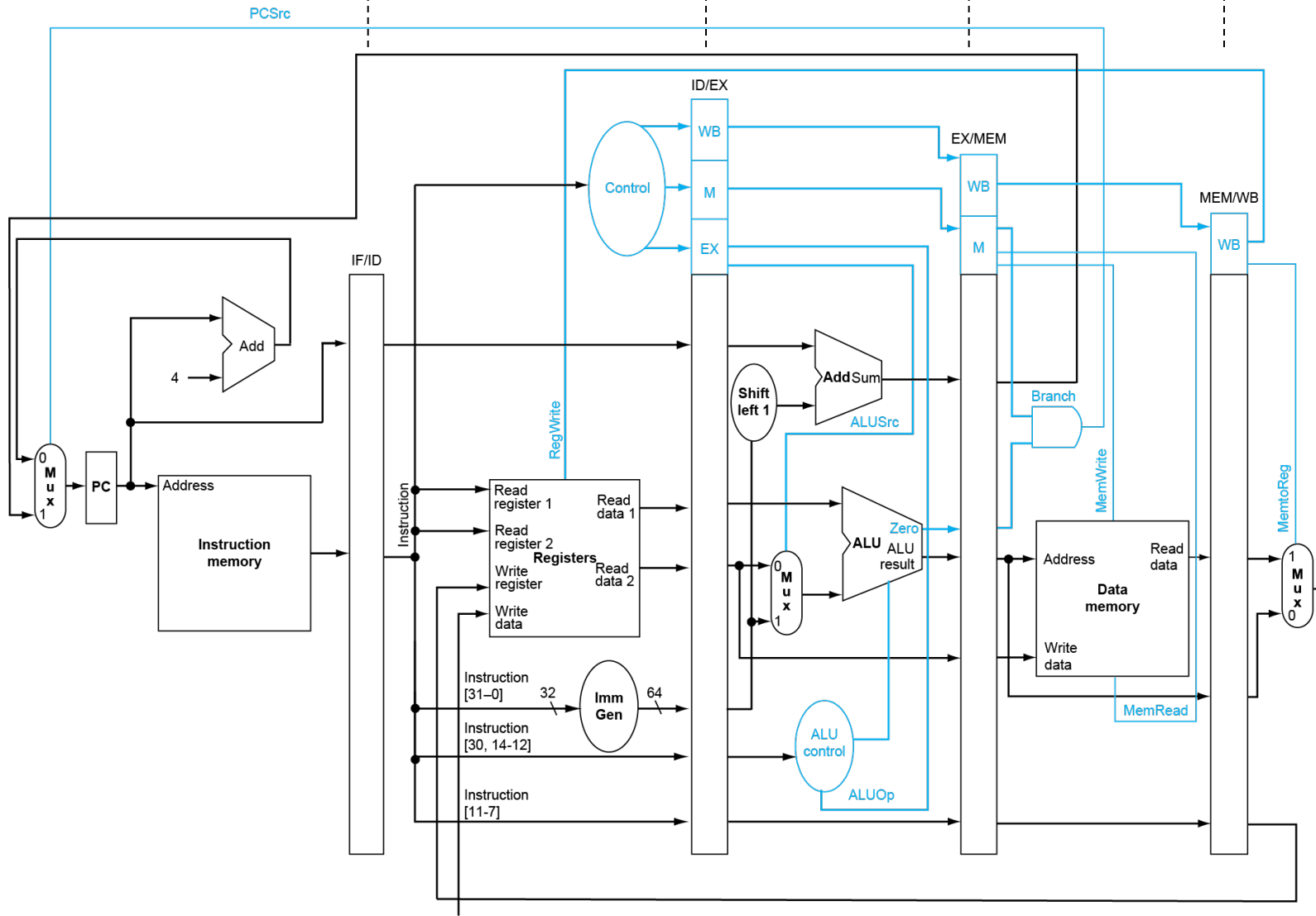


Cycle 7

add \$14,\$8,\$9

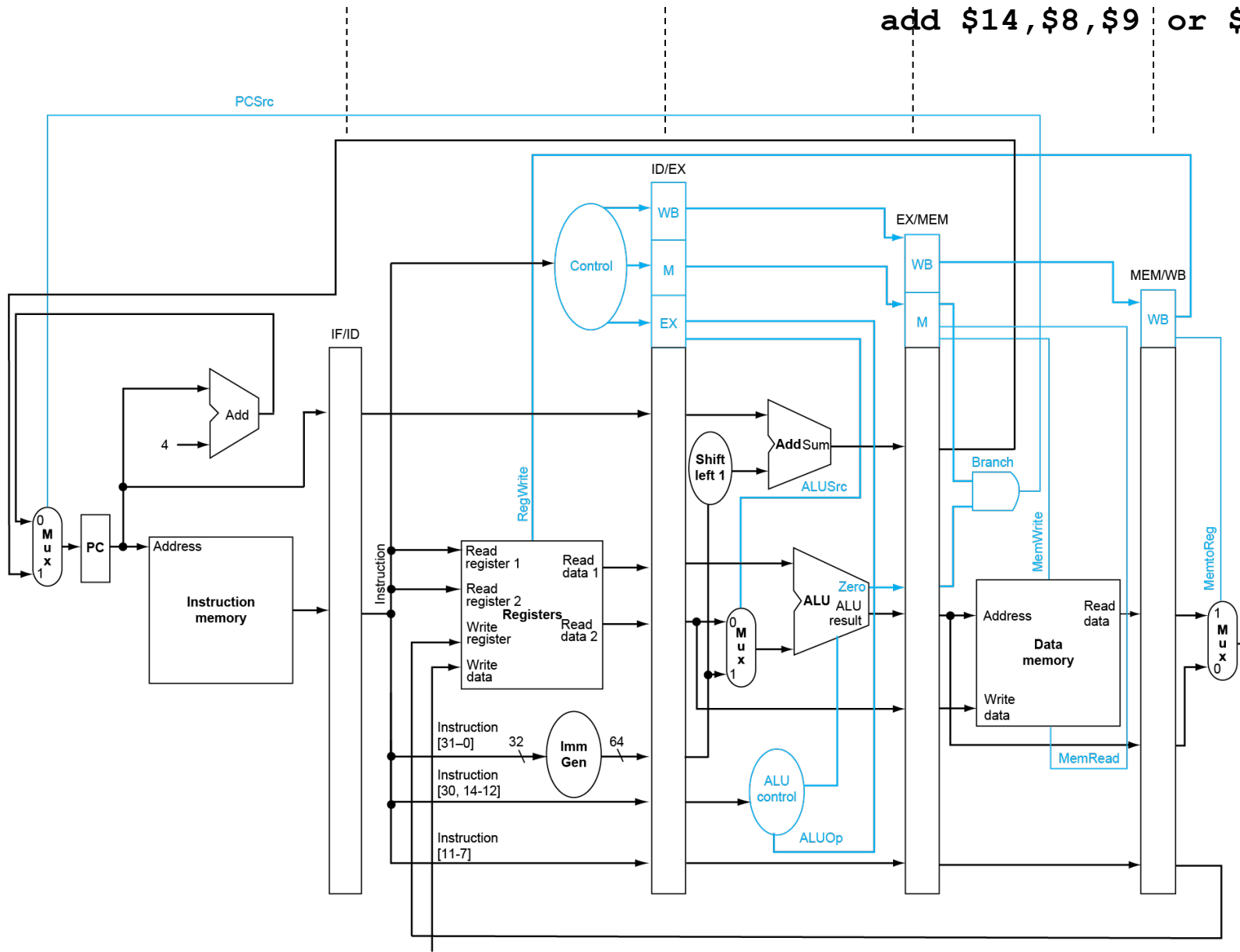
or \$13,\$6,\$7

and \$12,\$4,\$5



Cycle 8

add \$14,\$8,\$9 or \$13,\$6,\$7



add \$14, \$8, \$9



Summary of Pipeline

- Pipelining is a fundamental concept
 - Multiple steps using distinct resources
 - Utilize capabilities of data path by pipelined instruction processing
 - Start next instruction while working on the current one
 - Limited by length of longest stage (plus fill/flush)
 - Need to detect and resolve hazards
- What makes it easy in MIPS & RISC-V?
 - All instructions are of the same length
 - Just a few instruction formats
 - Memory operands only in loads and stores
- What makes pipelining hard? hazards