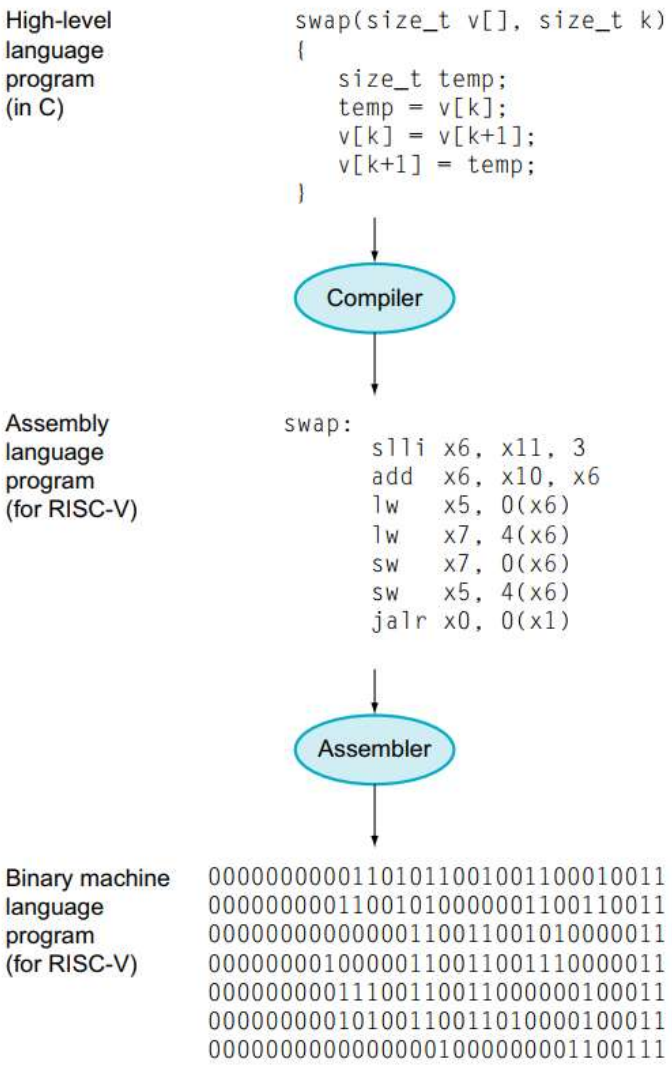
Computer Architecture

Ngo Lam Trung, Pham Ngoc Hung, Hoang Van Hiep
Department of Computer Engineering
School of Information and Communication Technology (SoICT)
Hanoi University of Science and Technology
E-mail: [trungnl, hungpn, hiephv]@soict.hust.edu.vn

Chapter 5: The Processor

[with materials from *Computer Organization and Design RISC-V, 2nd Edition*, Patterson & Hennessy, 2021, and M.J. Irwin's presentation, PSU 2008]

Review



Performance metric

CPU time = IC * CPI * CC

CPI: cycle per instruction
CC: clock cycle
IC: instruction count

How to improve?

- IC: ISA and compiler
- CC: hardware manufacturing
- CPI: CPU (logic) implementation

In this chapter

- Implementation of datapath
- How to improve CPI

Introduction

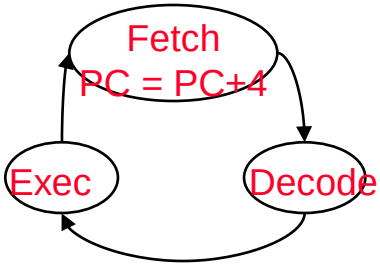
- We will examine two CPU implementations
  - A simplified version, to see the main components inside a CPU.
  - A more realistic pipelined version, to see how CPI can be improved (based on the pipeling technique).
- Simple subset which shows most aspects of RISC-V ISA.
  - Memory reference: `lw`, `sw`
  - Arithmetic and logical: `add`, `sub`, `and`, `or`
  - Branching: `beq`

31 27	26 25	24 20	19 15	14 12	11 7	6 0		
funct7		rs2	rs1	funct3	rd	opcode	R-type	add, sub, and, or
imm[11:0]			rs1	funct3	rd	opcode	I-type	lw
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type	sw
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type	beq
imm[31:12]					rd	opcode	U-type	
imm[20|10:1|11|19:12]					rd	opcode	J-type	

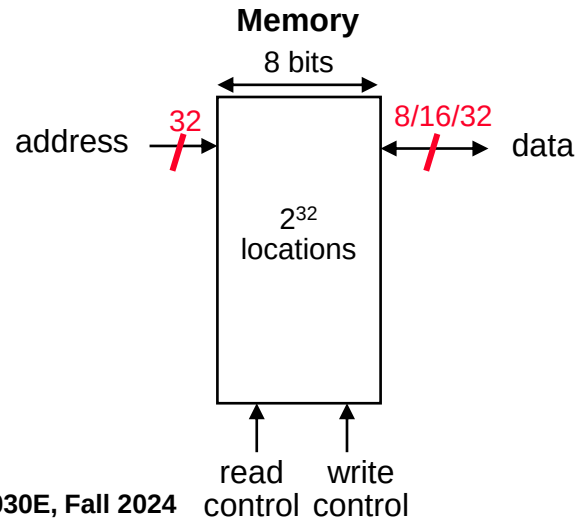
- Other instructions can be added later easily (hopefully).

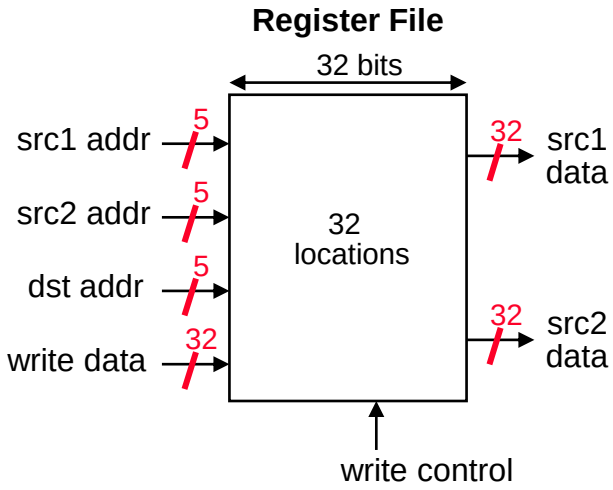
What we have so far

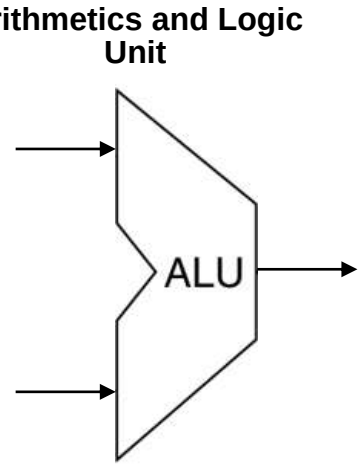
- Instruction cycle
  - Fetch the instruction from memory using PC and update PC.
  - Decode the instruction.
  - Execute the instruction.



- The operands and instruction set.
- Memory model, code and data segments.
- The module for add, sub, and other operations.

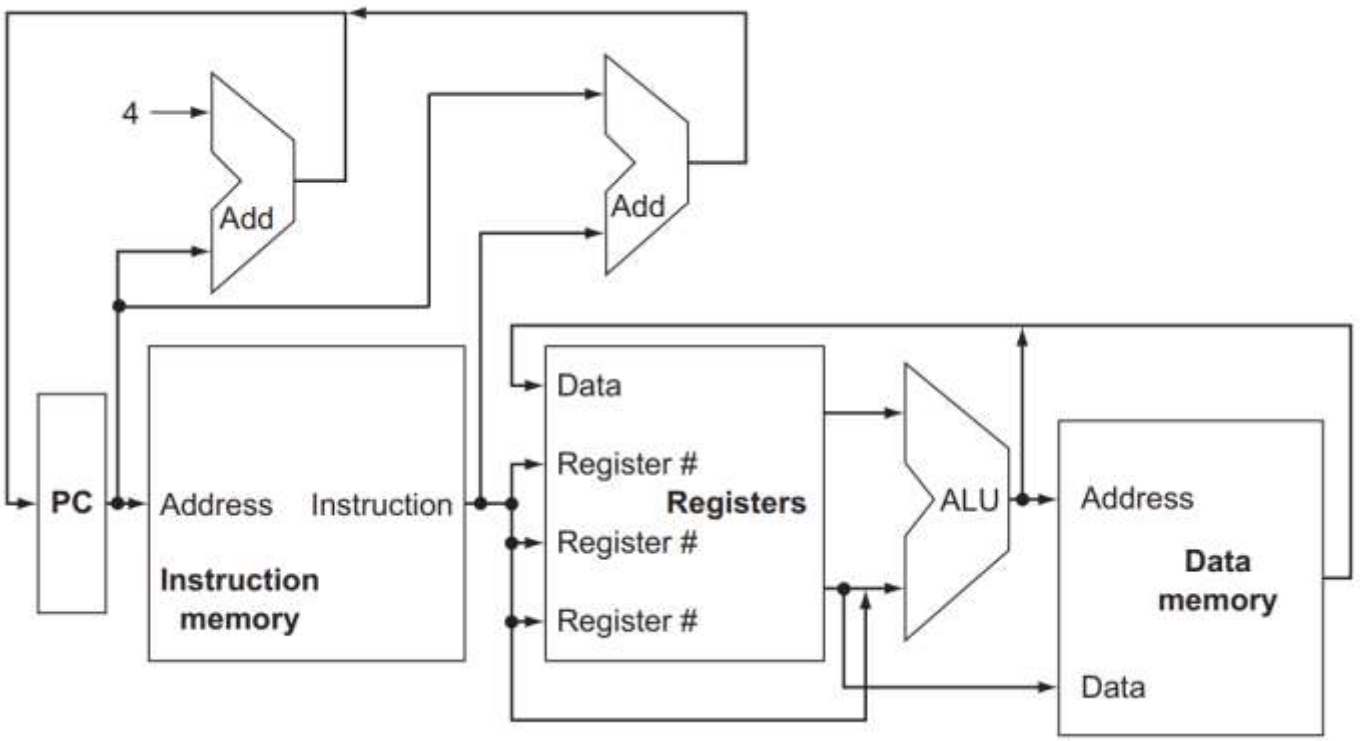






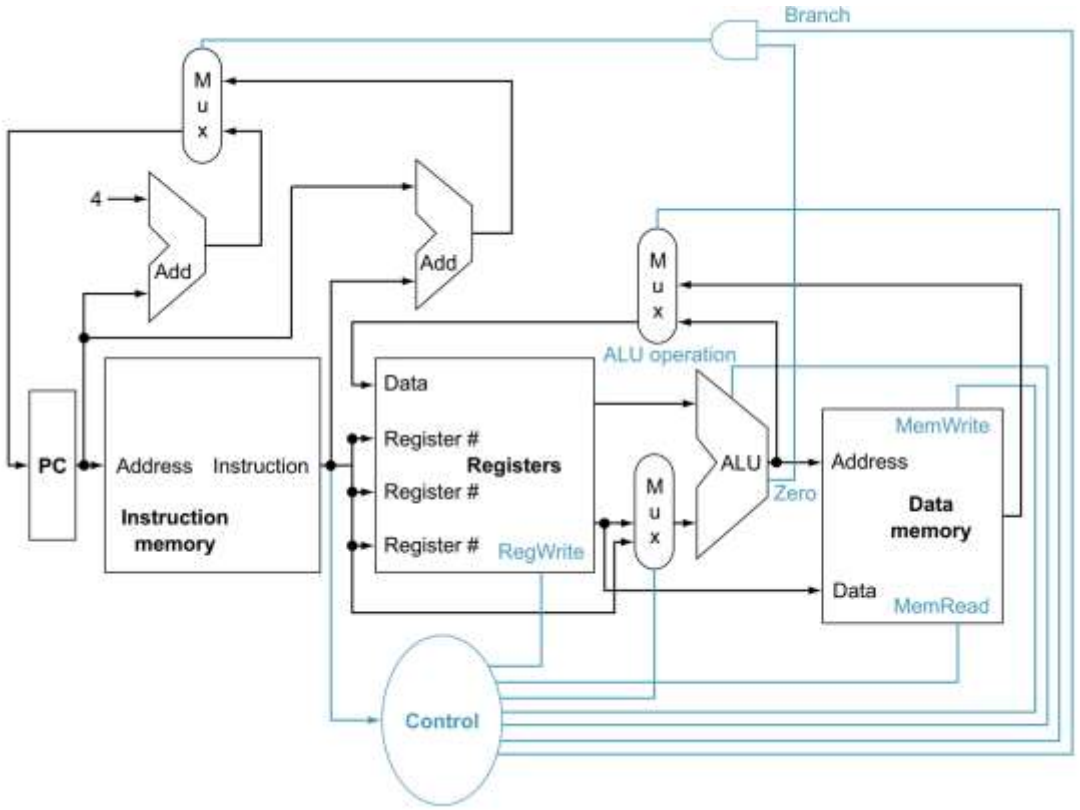
Simple datapath overview (wo. multiplexor)

- CPU that can execute lw, sw, add, sub, and, or, beq.
- We'll build this incrementally.



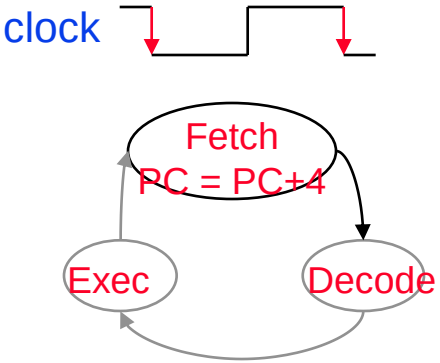
Simple datapath overview (w. multiplexor and control)

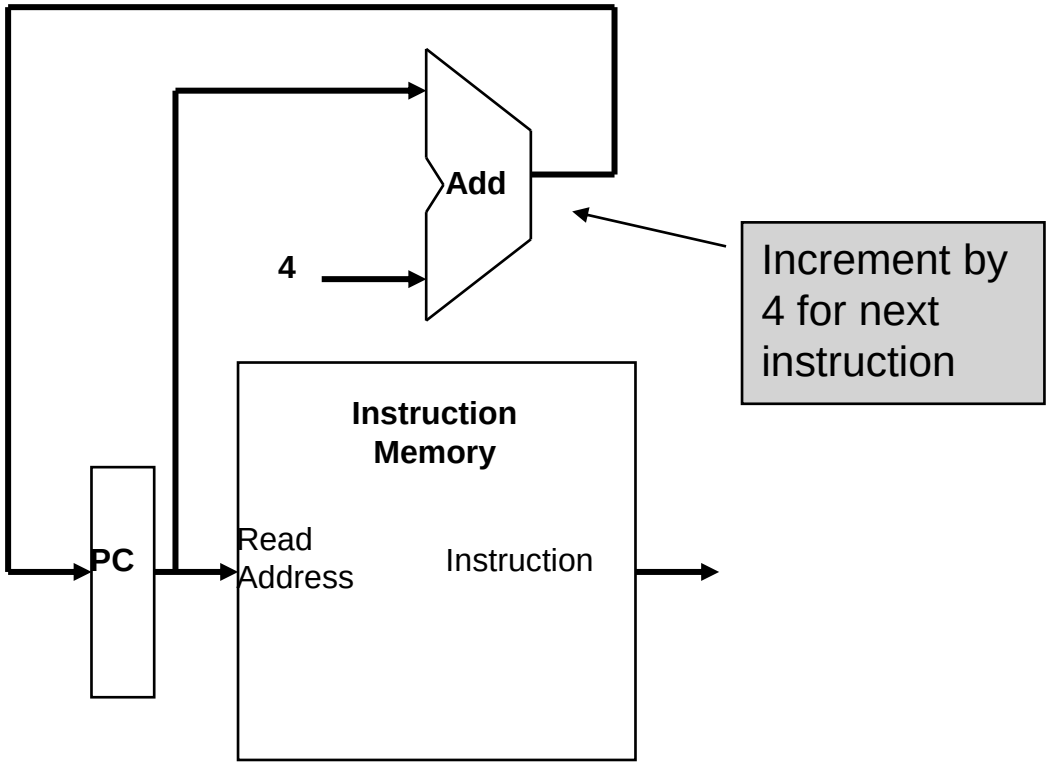
- CPU that can execute lw, sw, add, sub, and, or, beq.
- We'll build this incrementally.
- ..then refine it to improve performance.



Fetching Instructions

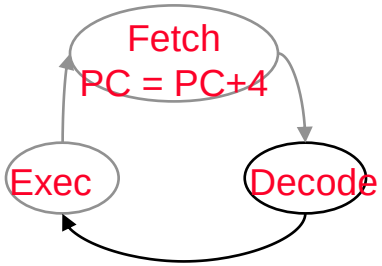
- Fetching instruction involves
  - reading the instruction from the Instruction Memory
  - updating the PC value to be the address of the next instruction in memory

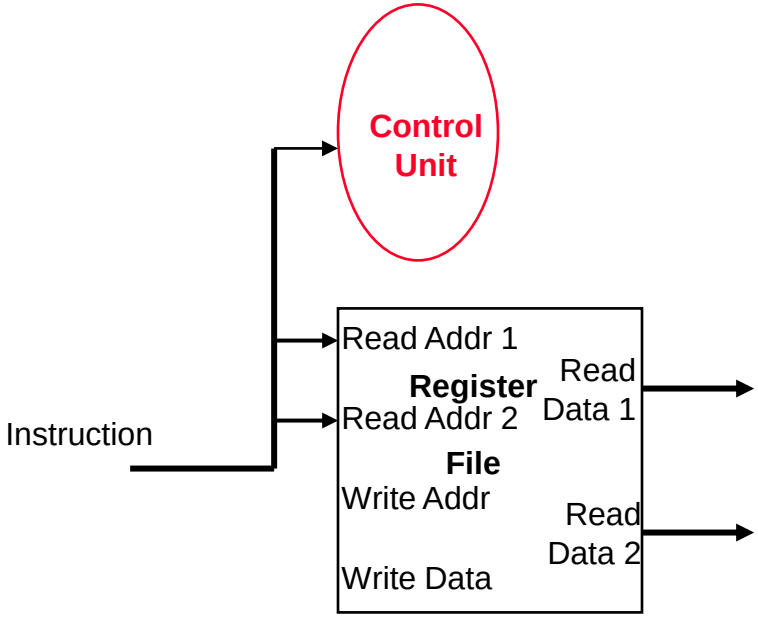




Decoding Instructions

- Decoding instruction involves
  - Sending the fetched instruction's opcode and function field bits to the control unit
  - The control unit send appropriate control signals to other parts inside CPU to execute the operations corresponds to the instruction





- Example: reading two values from the Register File

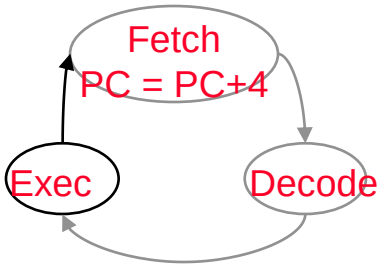
  →Register File addresses are contained in the instruction

Executing R-format instructions (ALU instructions)

- R format operations (**`add, sub, and, or`**)

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

  - read two register operands rs1 and rs2
  - perform operation (opcode and funct7, funct3) on values in rs1 and rs2
  - store the result back into the Register File (into location rd)



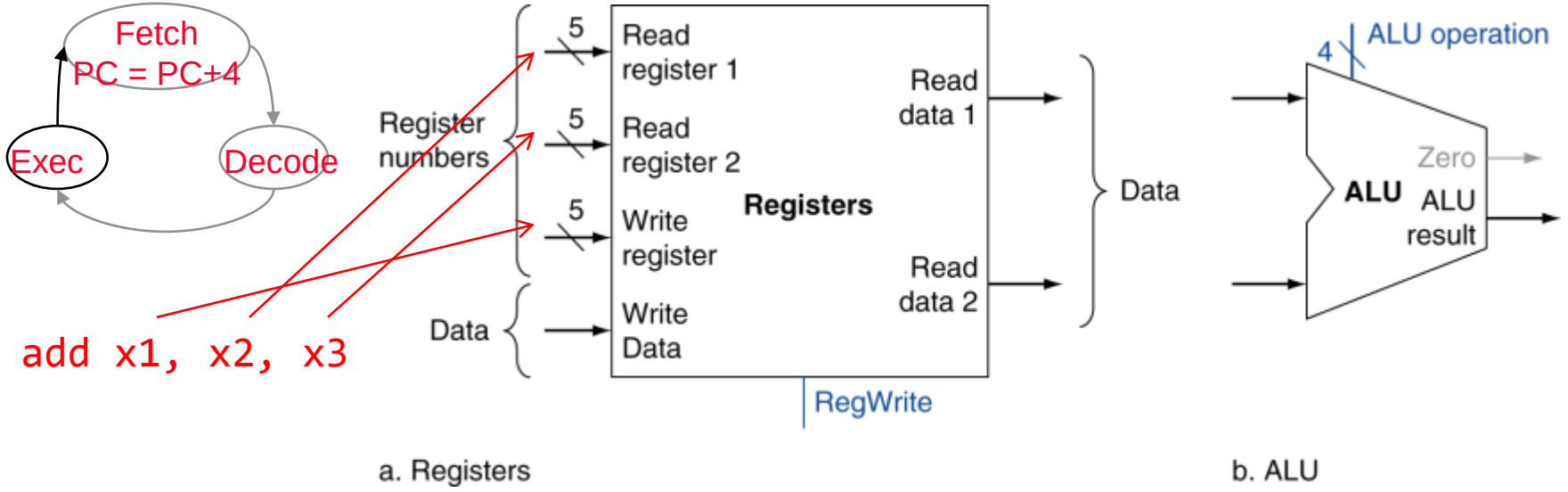

```
Example: add x1, x2, x3
- Value of x2 and x3 are sent to ALU
- ALU execute the x2 + x3 operation
- Result is store into x1
```

Executing R-format instructions (ALU instructions)

- R format operations (**`add, sub, and, or`**)

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

  - read two register operands rs1 and rs2
  - perform operation (opcode and funct7, funct3) on values in rs1 and rs2
  - store the result back into the Register File (into location rd)



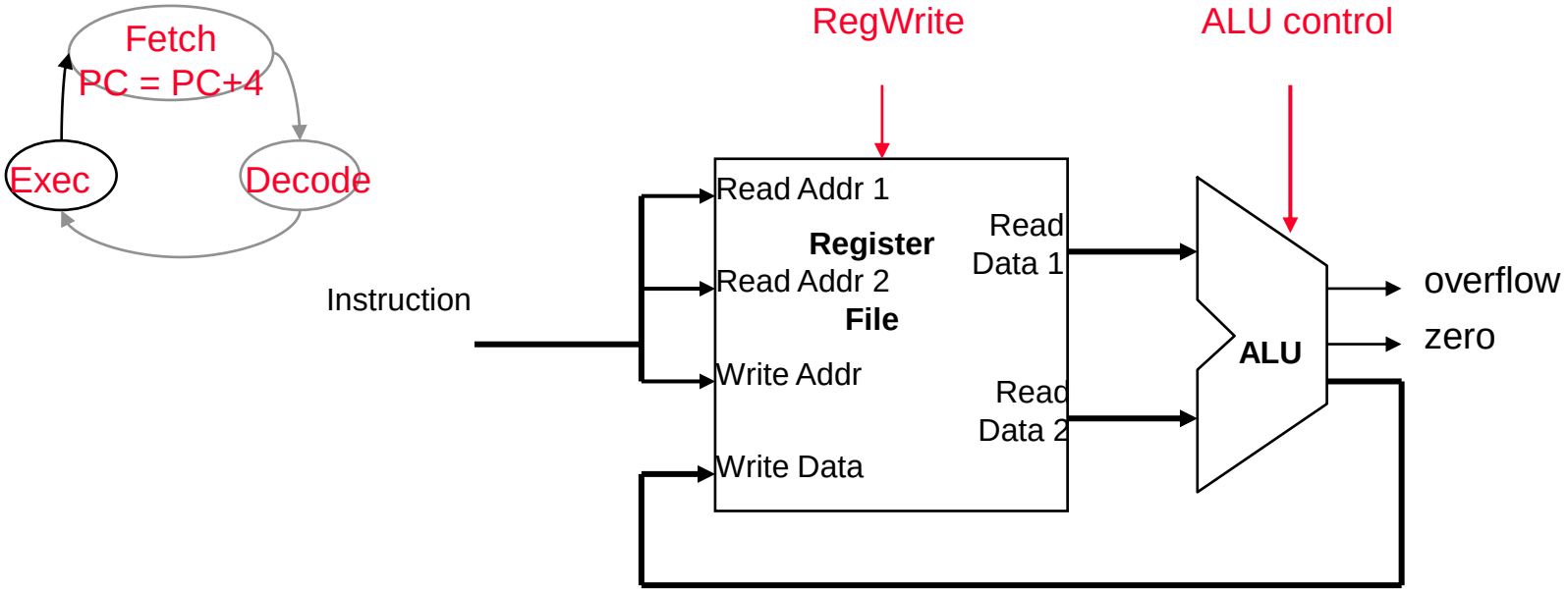
Draw connection between a and b to form the execution unit?

Executing R-format instructions (ALU instructions)

- R format operations (**`add, sub, and, or`**)

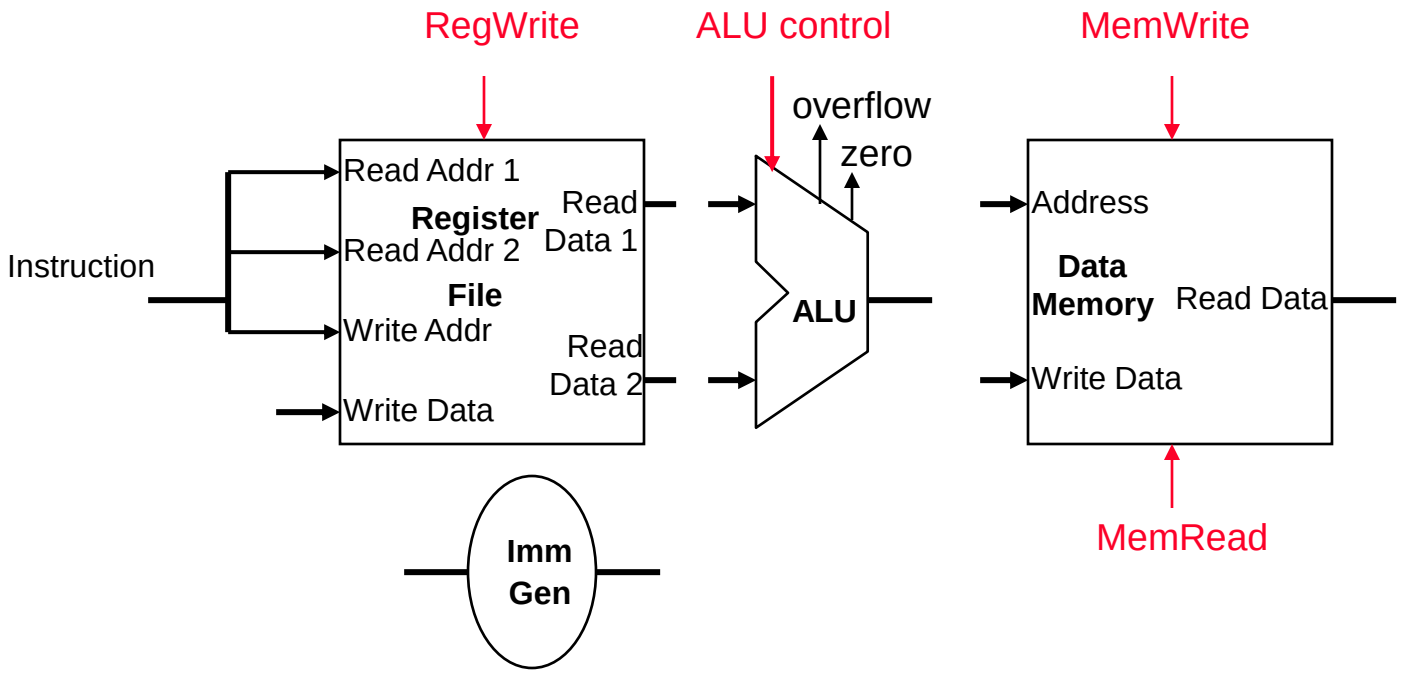
funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- read two register operands rs1 and rs2
- perform operation (opcode and funct7, funct3) on values in rs1 and rs2
- store the result back into the Register File (into location rd)



Executing Load and Store (Memory instructions)

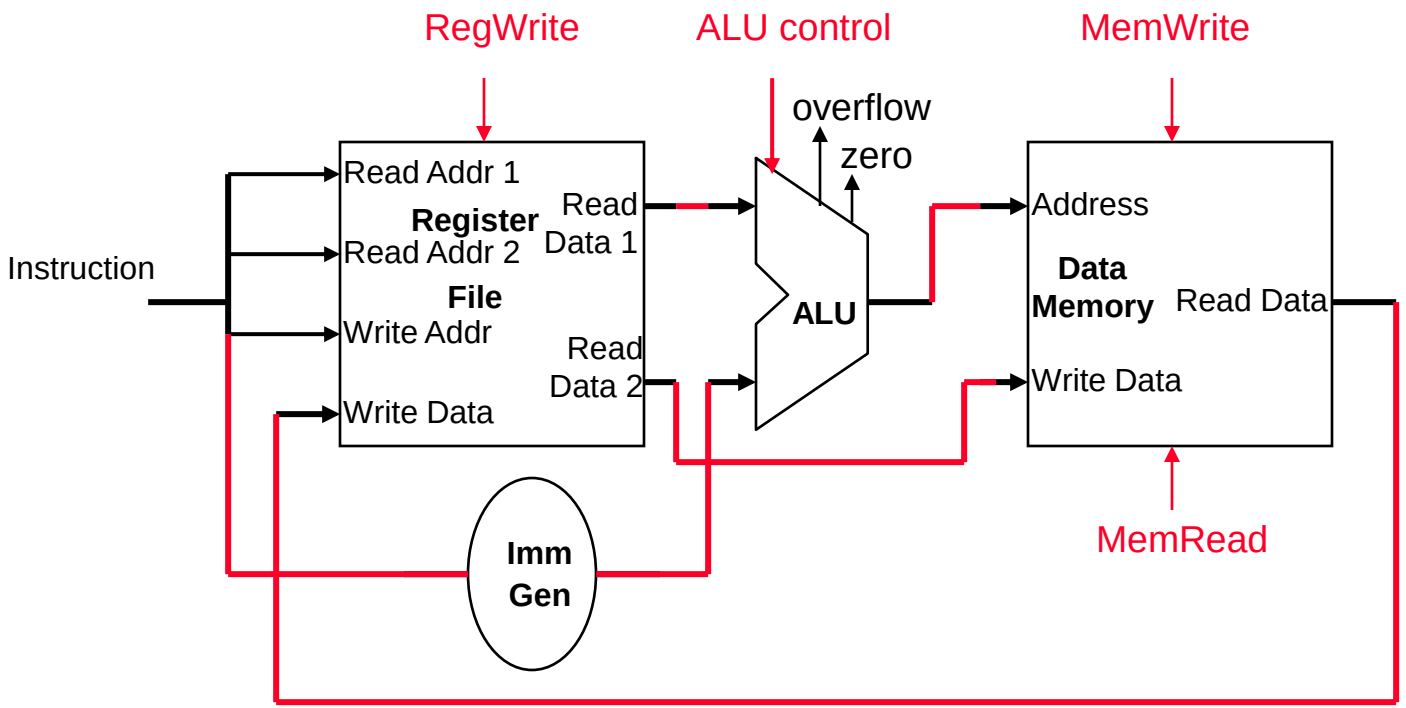
- Load and store operations involves
  - read register operands
  - Calculate address using 12-bit offset
    - Use ALU, but sign-extend offset
  - store: read from the Register File, write to the Data Memory
  - load: read from the Data Memory, write to the Register File



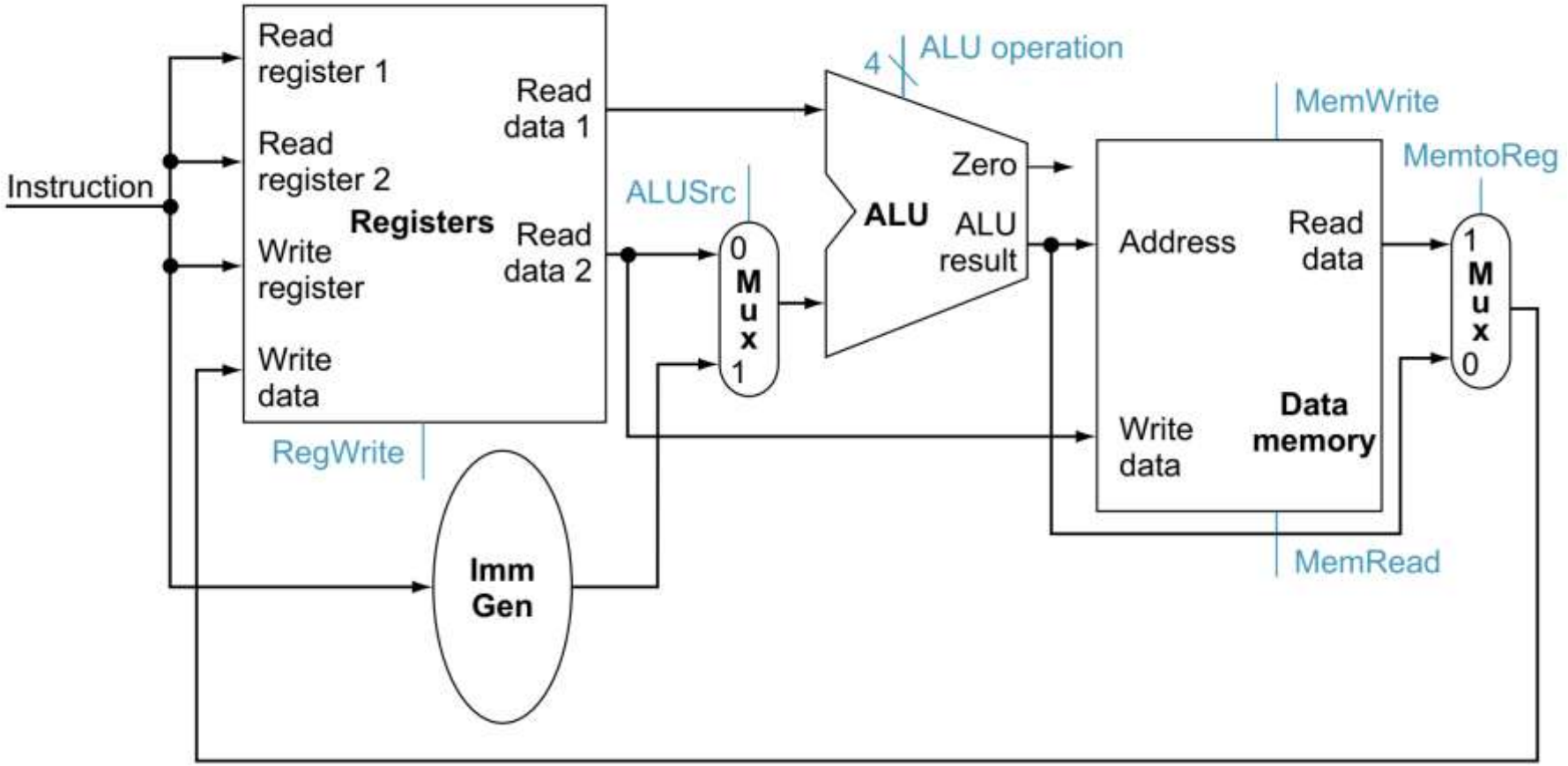
Draw necessary connections to form execution unit?

Executing Load and Store (Memory instructions)

- Load and store operations involves
  - read register operands
  - Calculate address using 12-bit offset
    - Use ALU, but sign-extend offset
  - store: read from the Register File, write to the Data Memory
  - load: read from the Data Memory, write to the Register File



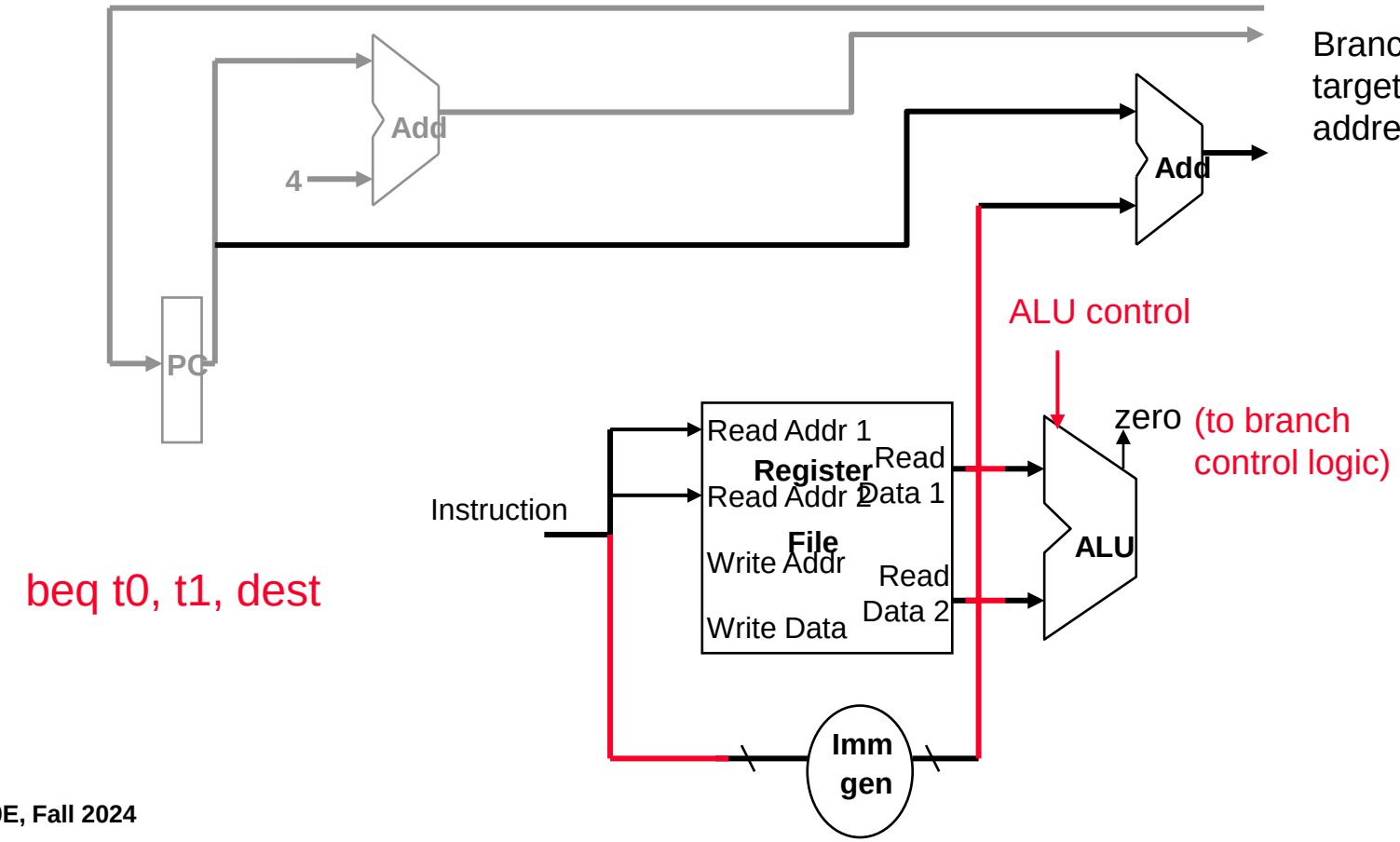
Combining ALU and Memory instructions



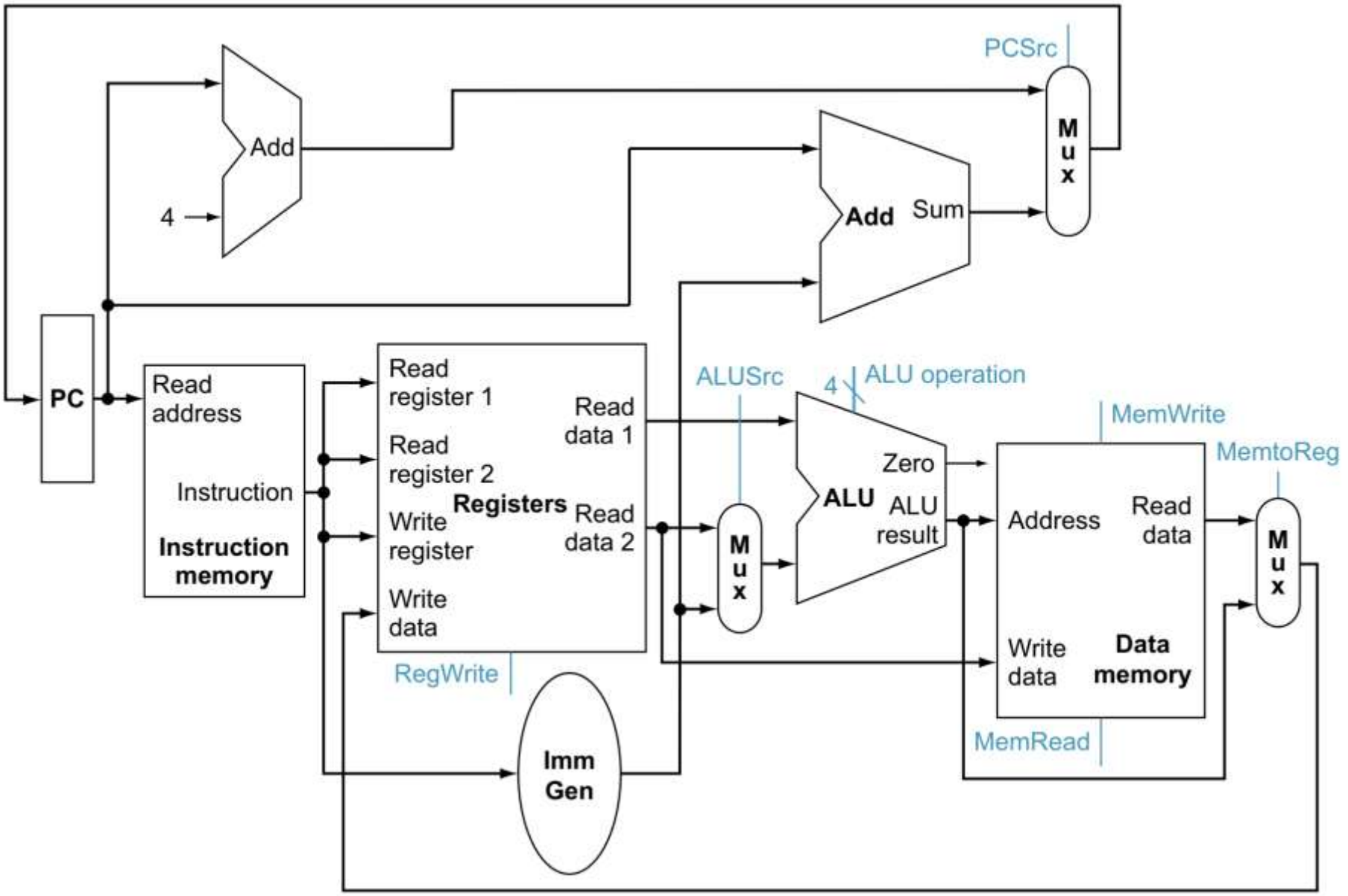
Note: multiplexors are added when connecting multiple inputs to one output

Executing Branch instruction (beq)

- Branch operations involves
  - read register operands
  - compare the operands (subtract, check `zero` ALU output)
  - compute the branch target address: adding the PC to the signed-extended offset shifted left 1 bit.

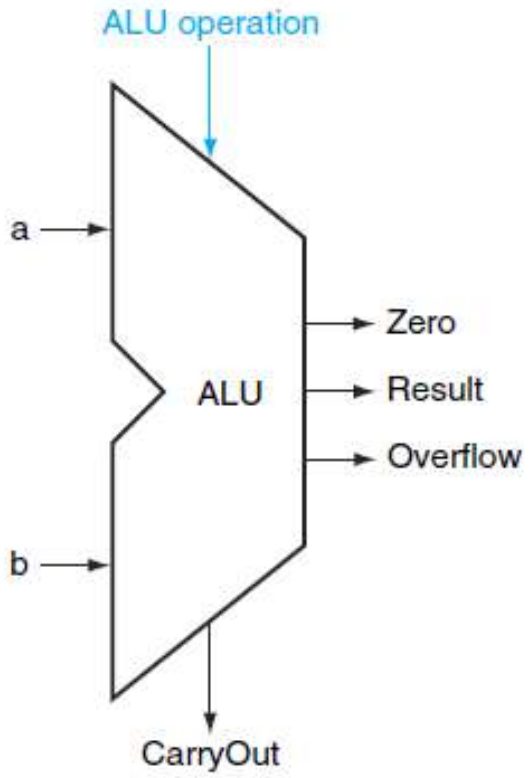


Full datapath for ALU, Memory, Branching instructions

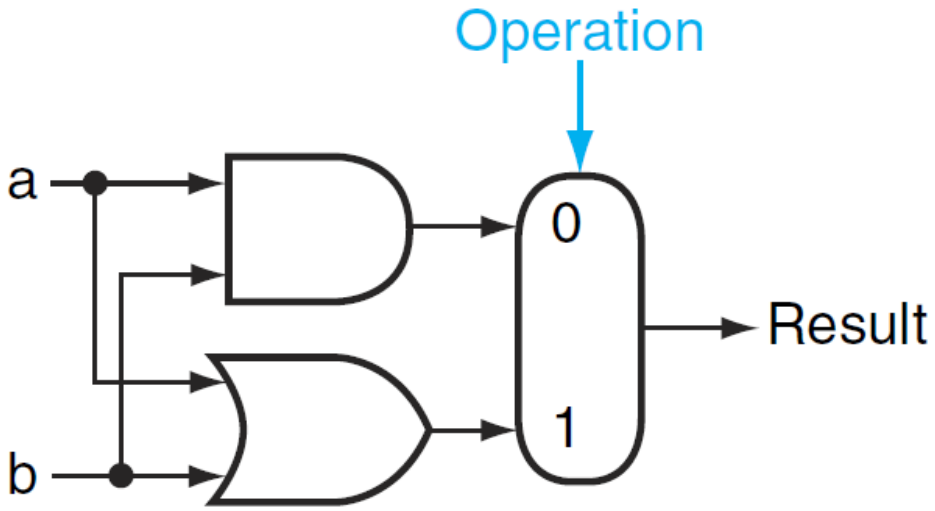


Designing a (very simple) ALU

- Input/output
  - Two data input: a, b
  - ALU control signals
  - Data out
  - Flags out
- Operations
  - and, or
  - add, subtract

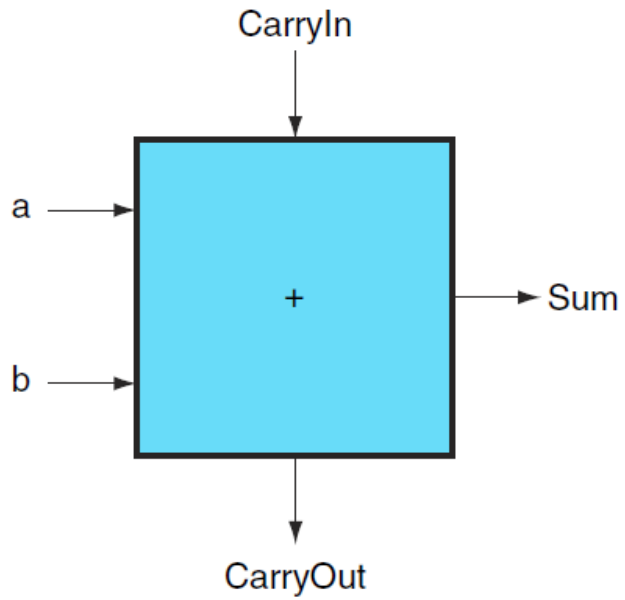


1-bit ALU with logic operation



- What do we have if
  - Operation = 0:
  - Operation = 1:

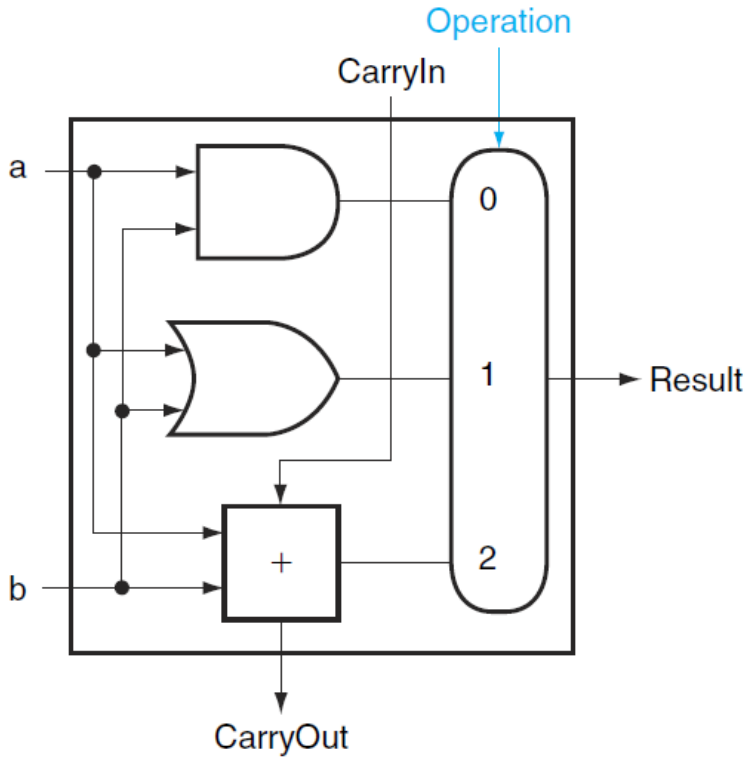
1-bit full-adder



$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

$$\text{Sum} = (a \cdot \overline{b} \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot \overline{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

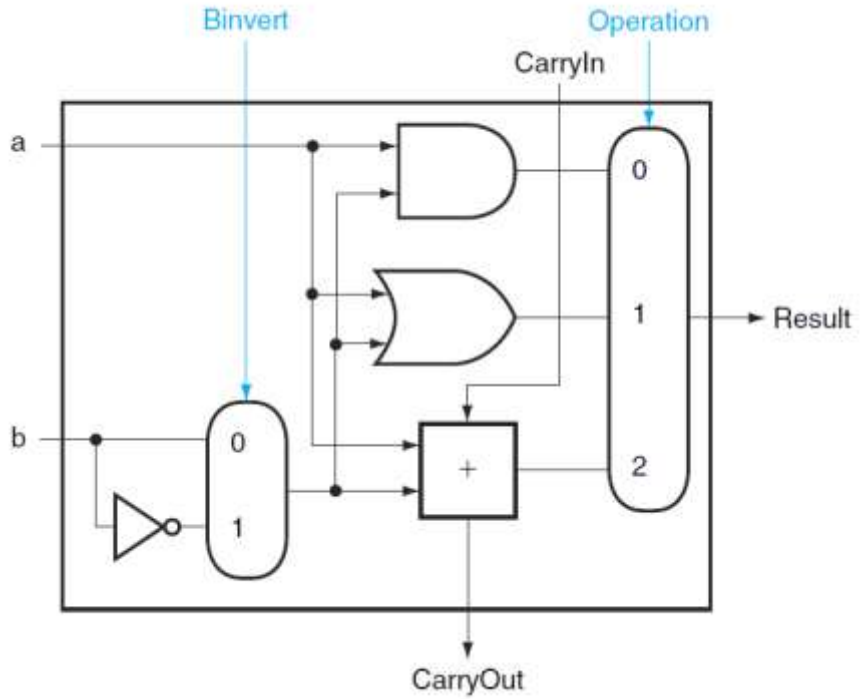
1-bit ALU with AND, OR, ADD



- Operation = 00:
- Operation = 01:
- Operation = 10:

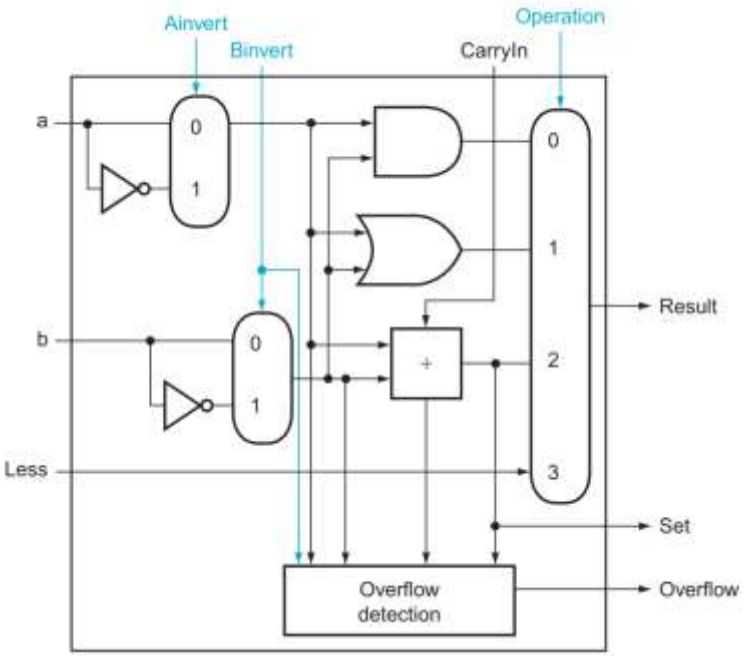
How about 1-bit ALU with AND, OR, ADD, SUB?

- a-b = a + (-b) = a + (2's complement of b)



- For SUB operation
  - Operation =
  - Binvert =
  - CarryIn =

Adding other operations, such as NOR and SLT



- Ainvert is added
- For NOR operation: $\overline{a + b} = \bar{a}.\bar{b}$
  - Ainvert =
  - Binvert =
  - Operation =

ALU control signals

- ALU operation:
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on opcode
- Operation (Function) is selected based on 4 control bits

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

The missing things: control signals

- Memory modules, register files, ALU, multiplexors require control signals to work.
  - ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch.
  - ALUOp (2 bits).
- Control signals are generated by:
  - “ALU control” unit: responsilble for the ALU control signals.
  - “Control” unit: read/write signals, multiplexors input selector, ALUOp to control “ALU control”.

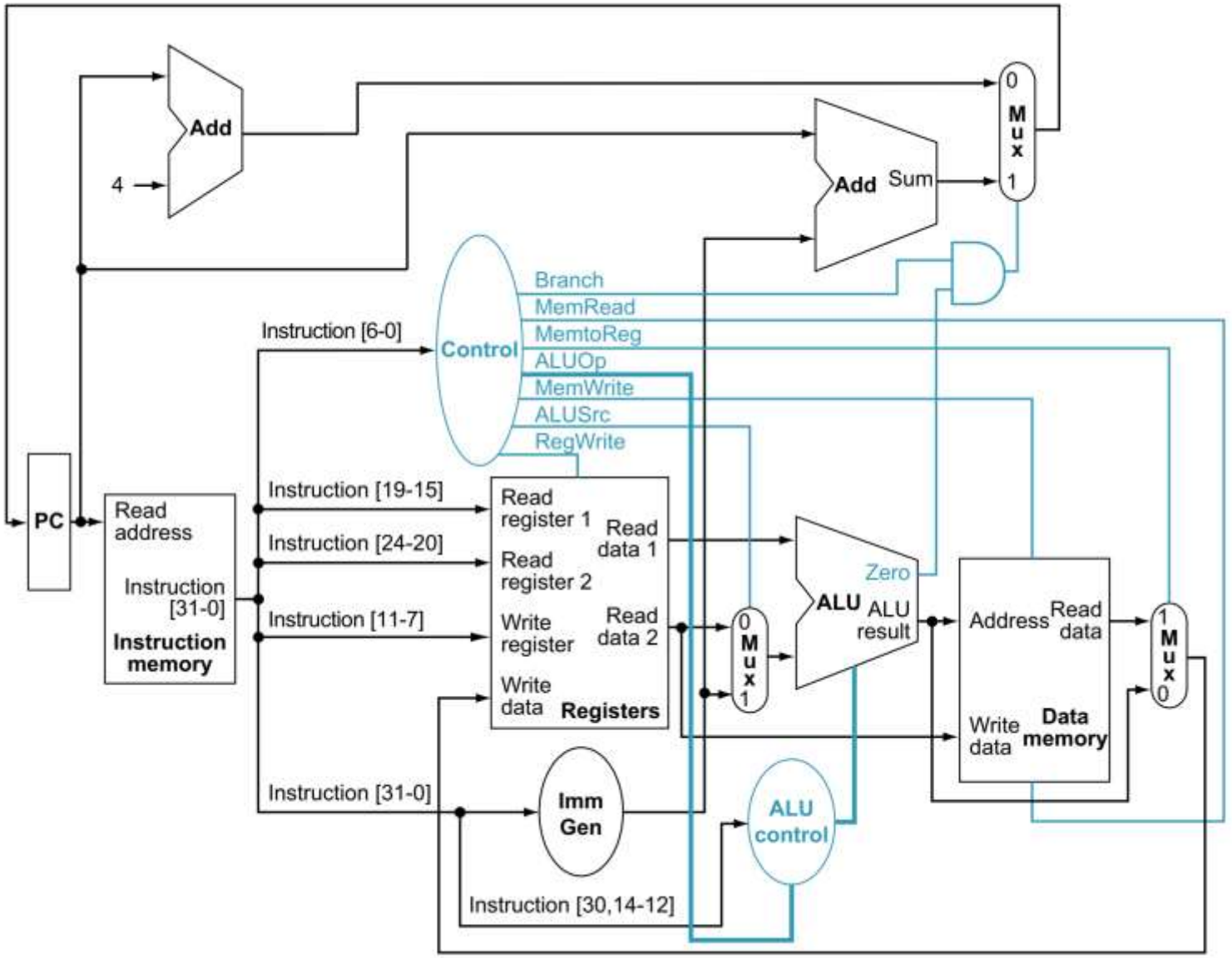
Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
lw	1	1	1	1	0	0	0	0
sw	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

Control signals generated by “Control” unit

The missing things: control signals

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.

Datapath with Control unit and signals



ALU control signals

- Set by “ALU control” unit.
- Based on 2-bit ALUOp and func3, func7 fields

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	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

ALU control signals

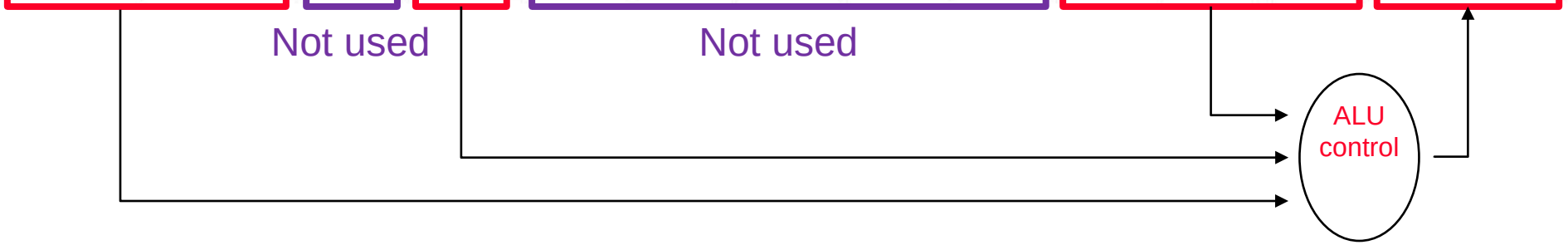
ALU control signals

- How ALU control signals are set?

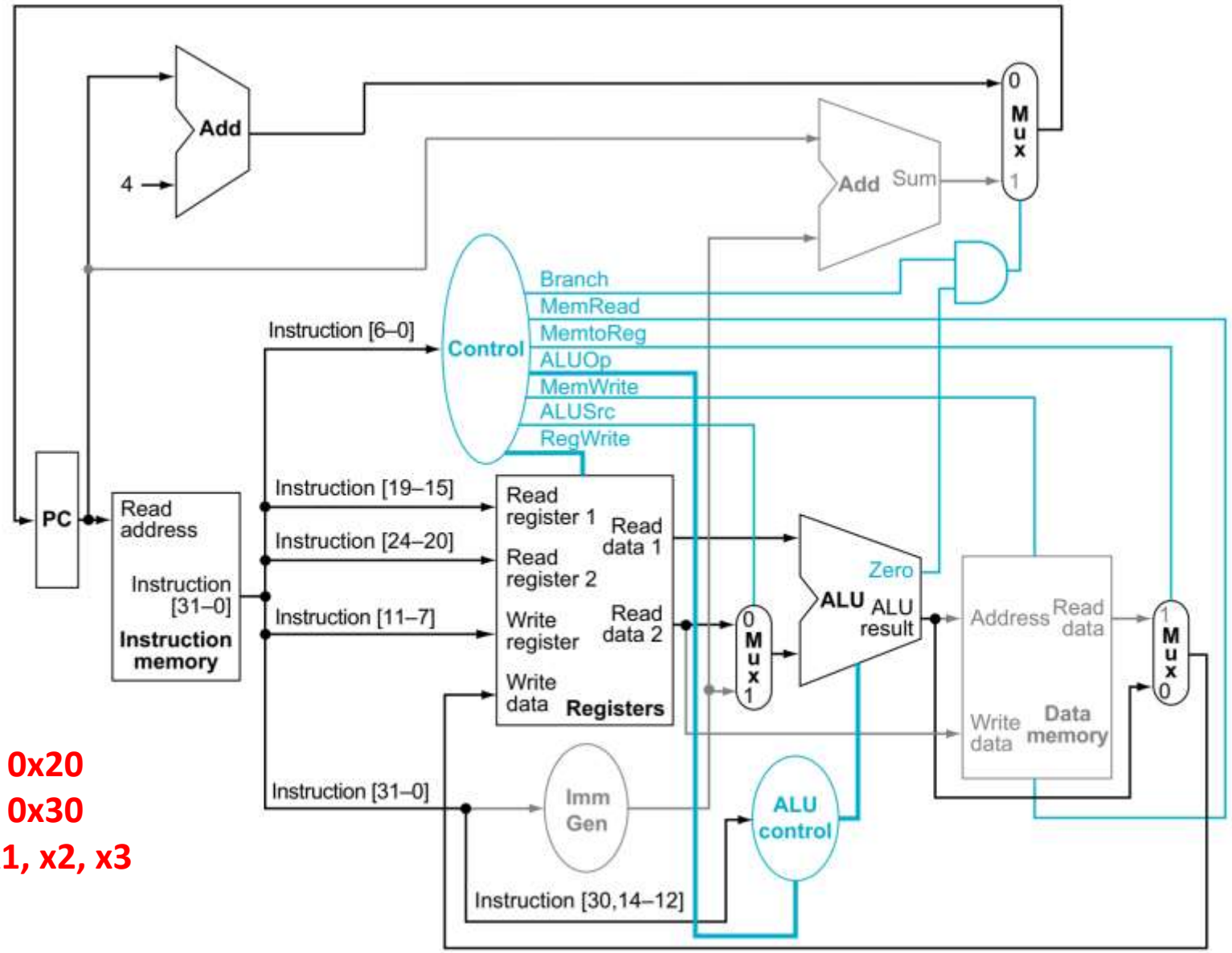
ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

Not used

Not used

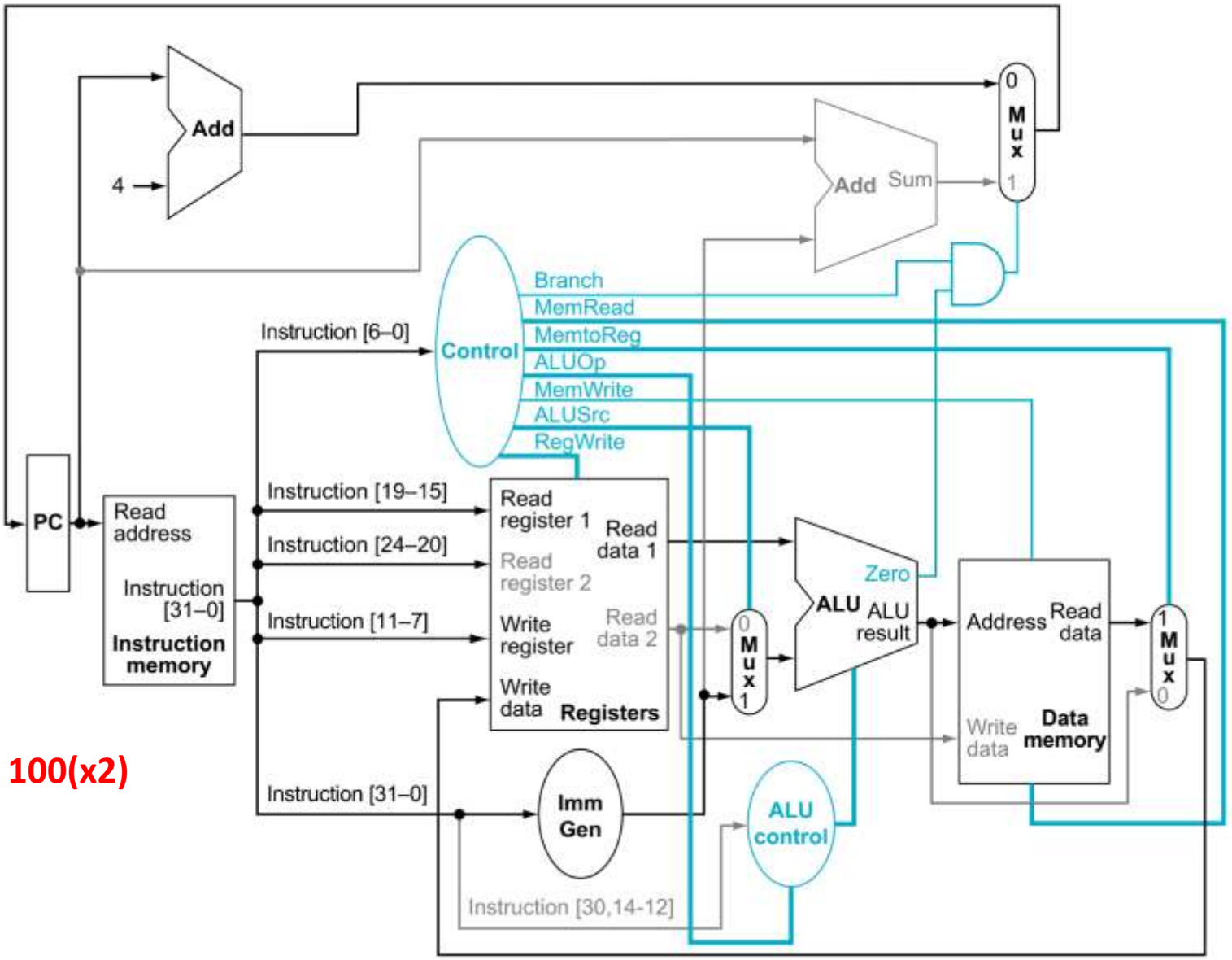


Datapath in operation for ALU instructions



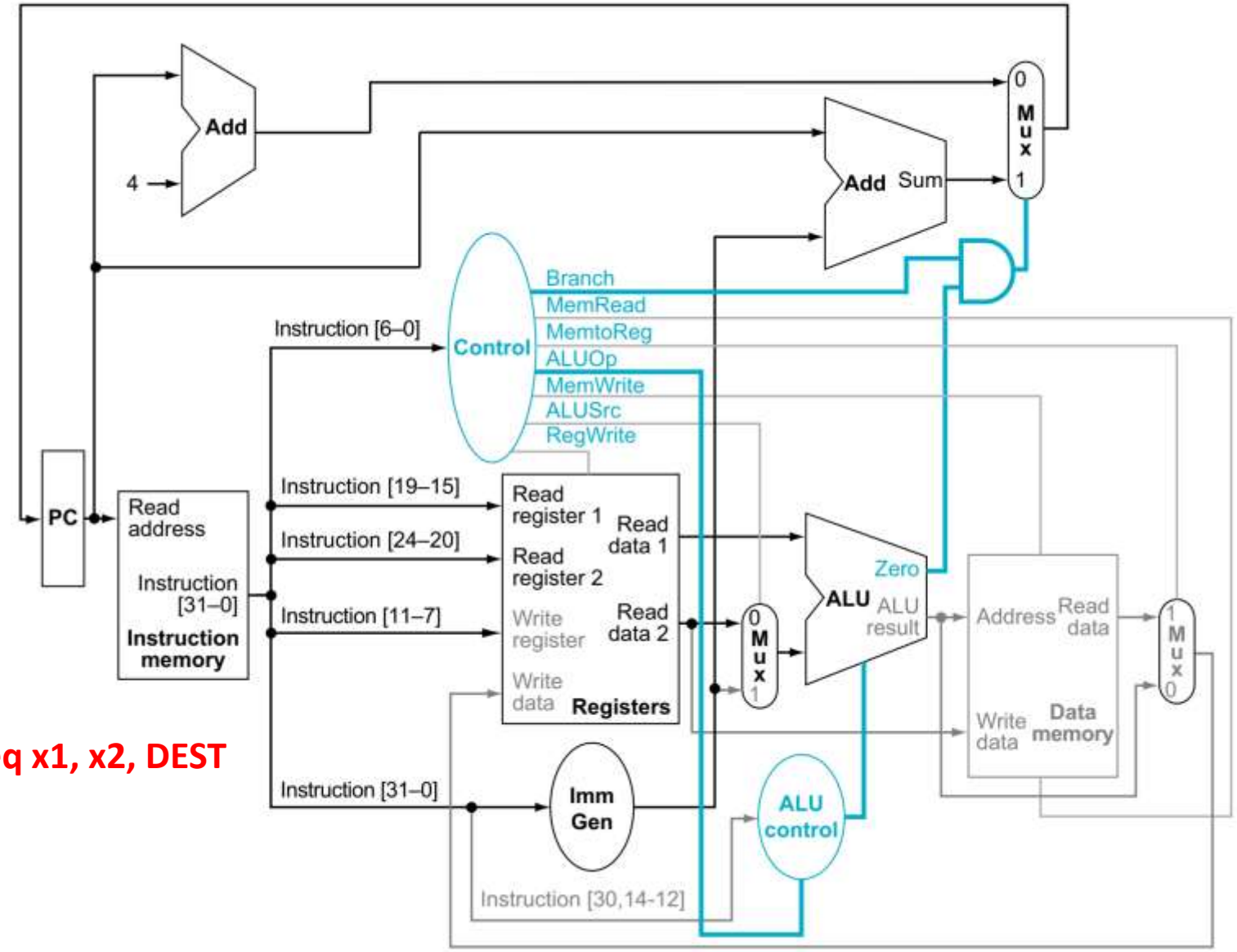
#x2 = 0x20
#x3 = 0x30
add x1, x2, x3

Datapath in operation for lw instruction



lw x1, 100(x2)

Datapath in operation for beq instruction



beq x1, x2, DEST

Instruction Times (Critical Paths)

- What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 1, wires, setup and hold times except:
  - Instruction Fetch and Data Access (200 ps)
  - ALU operation and adders (200 ps)
  - Register File access (reads or writes) (100 ps)

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total
Load (lw)						
Store (sw)						
R-format (add, sub, and, or)						
Branch (beq)						

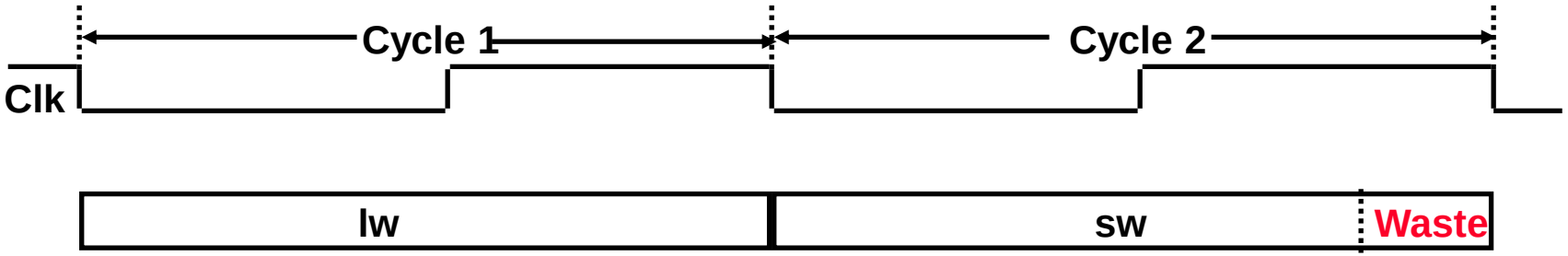
Instruction Times (Critical Paths)

- What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 1, wires, setup and hold times except:
  - Instruction Fetch and Data Access (200 ps)
  - ALU operation and adders (200 ps)
  - Register File access (reads or writes) (100 ps)

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total
Load (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  - especially problematic for more complex instructions like floating point multiply



- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

but

- Is simple and easy to understand

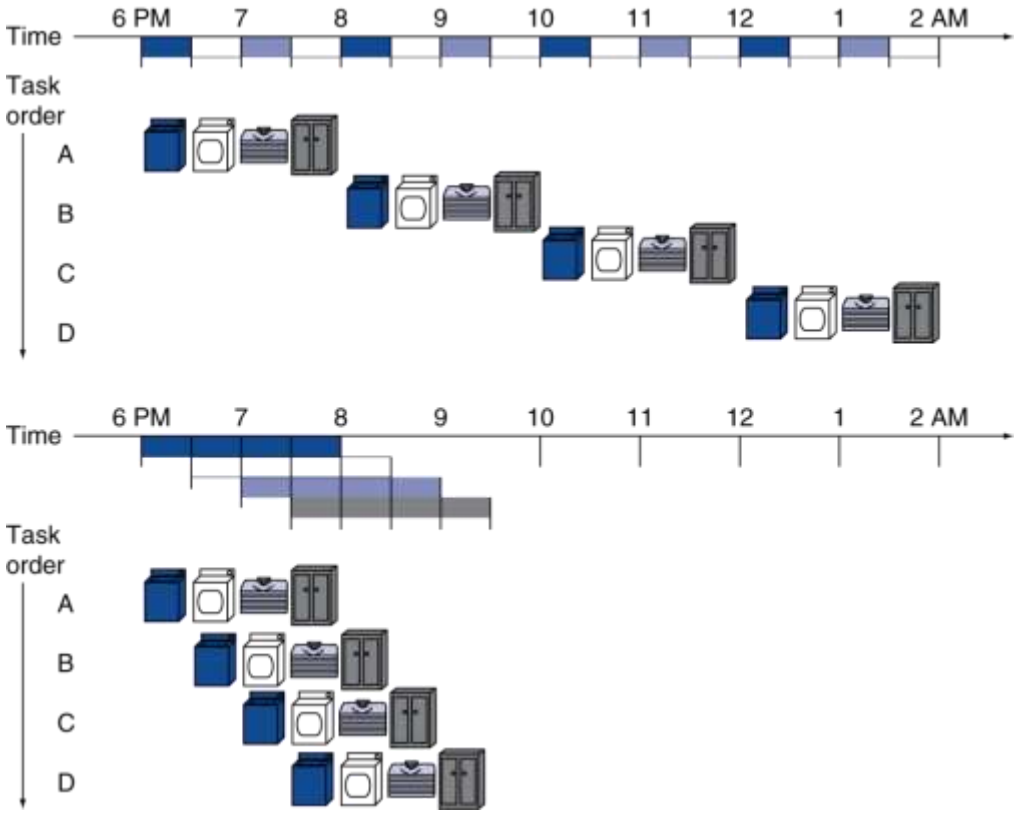
How Can We Make The Computer Faster?

- Divide instruction cycles into smaller cycles
- Executing instructions in parallel
  - With only one CPU?
- Pipelining:
  - Start fetching and executing the next instruction before the current one has completed
  - Overlapping execution

Pipeline in real life

A more serious example: laundry work

- Pipelined laundry boots performance up to 4 times



- With 4 loads

  T_{normal} = 4*2 = 8 hours

  $T_{pipeline}$ = 3.5 hours

- With n loads

  T_{normal} = n*2 hours

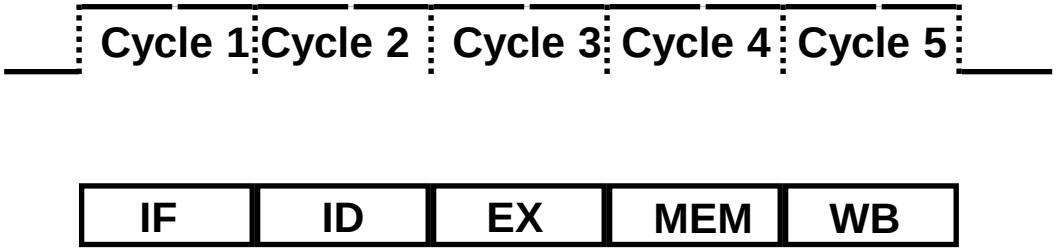
  $T_{pipeline}$ = (3+n)/2 hours

4 stages: washing, drying, ironing, folding

When n → ∞ : T_{normal} → 4*$T_{pipeline}$

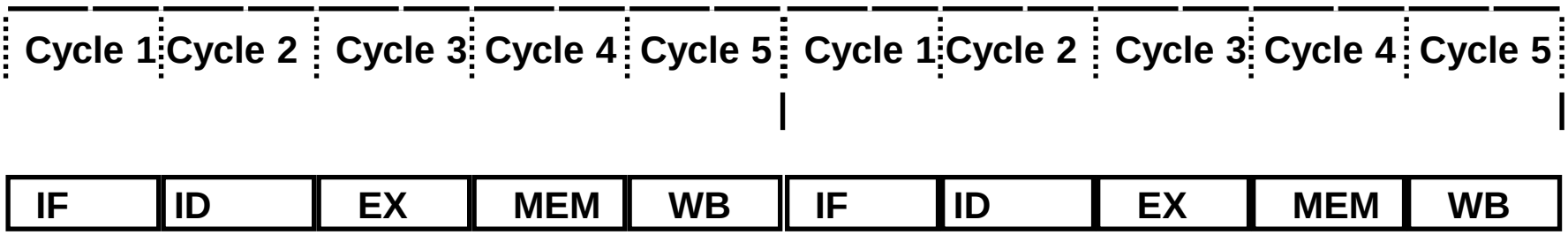
RISC-V Pipeline

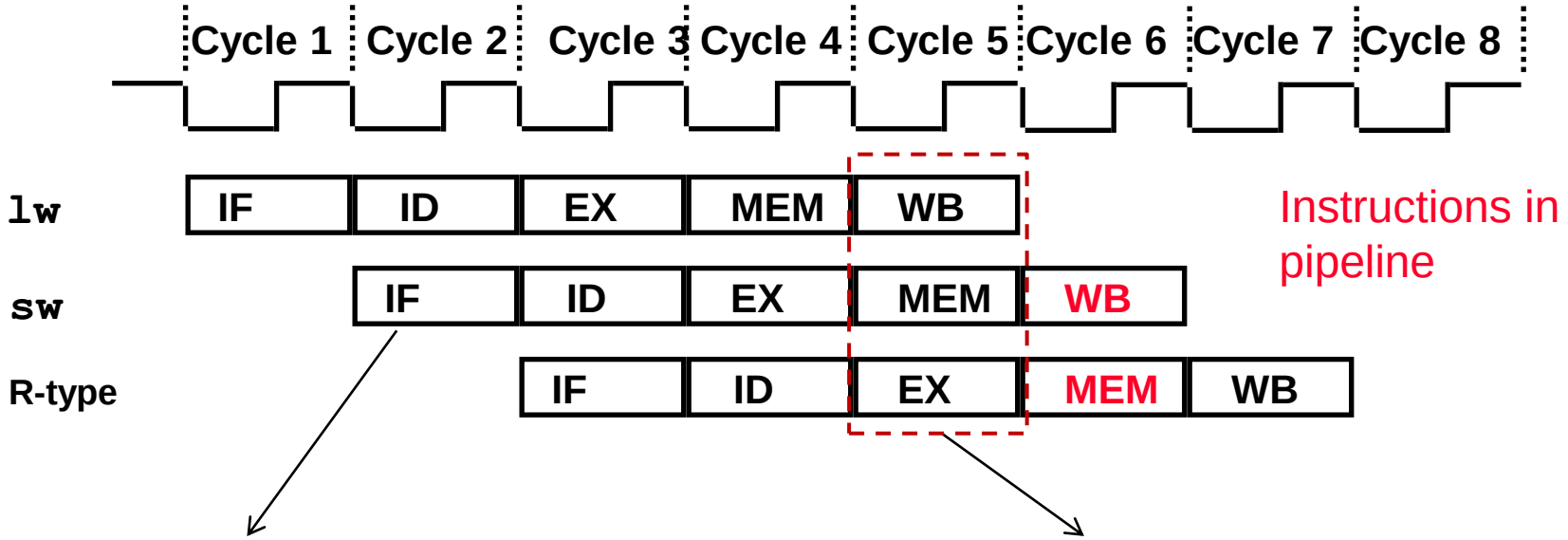
- Five stages, one step per stage
  - IF: Instruction Fetch from Memory and Update PC
  - ID: Instruction Decode and Register Read
  - EX: Execute R-type or calculate memory address
  - MEM: Read/write the data from/to the Data Memory
  - WB: Write the result data into the register file



Execution time for a single instruction is always 5 cycles, regardless of instruction operation

Instruction pipeline





Start fetching and executing the next instruction before the current one has completed

More than one instruction are executed at a time

Pipeline performance

- All modern processors are pipelined for performance
  - Remember *the* performance equation:

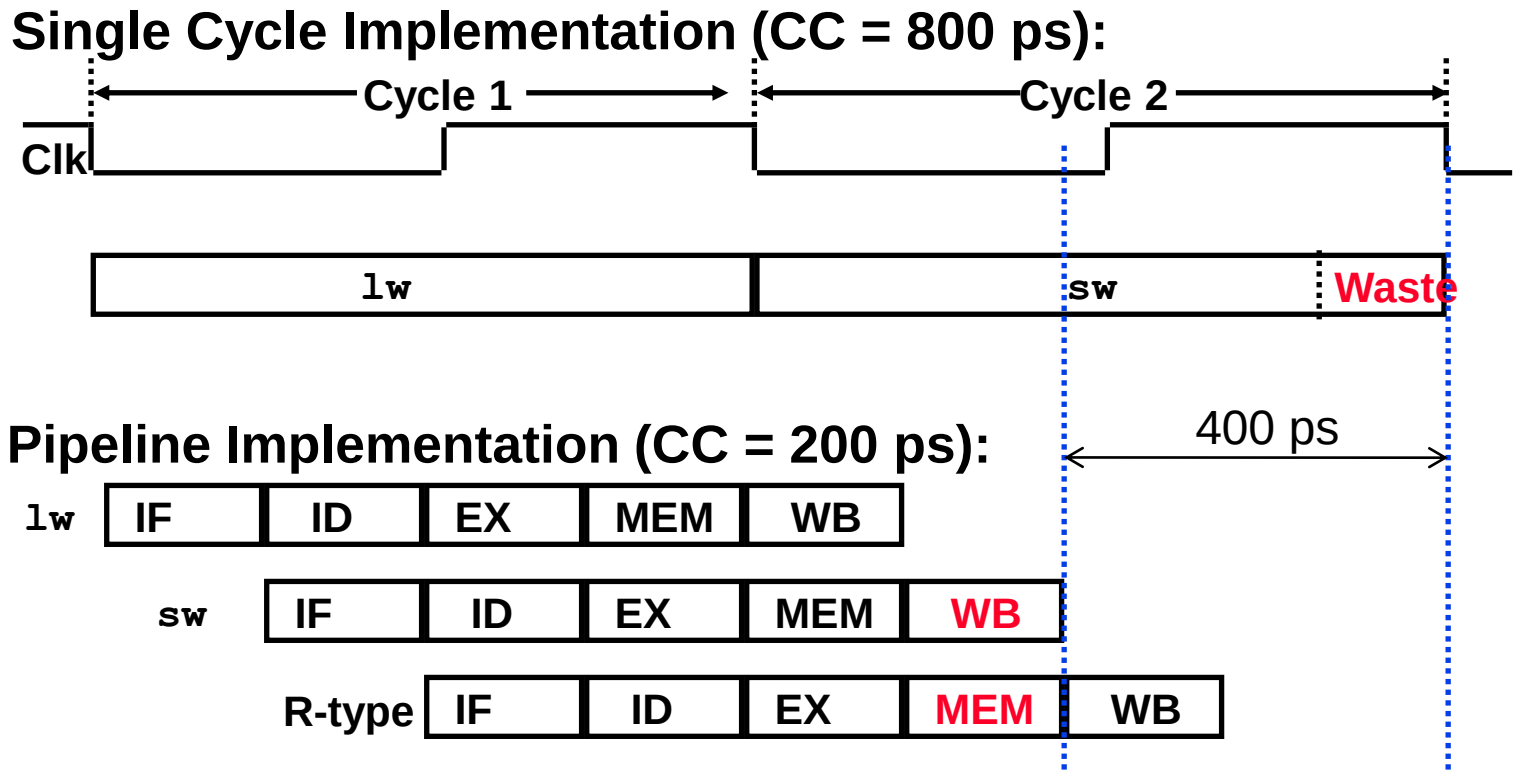
$$\text{CPU time} = \text{CPI} * \text{CC} * \text{IC}$$

- Under *ideal* conditions (balance) and with a large number of instructions:
  - A five-stage pipeline is nearly five times faster because the CC is nearly five times faster

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

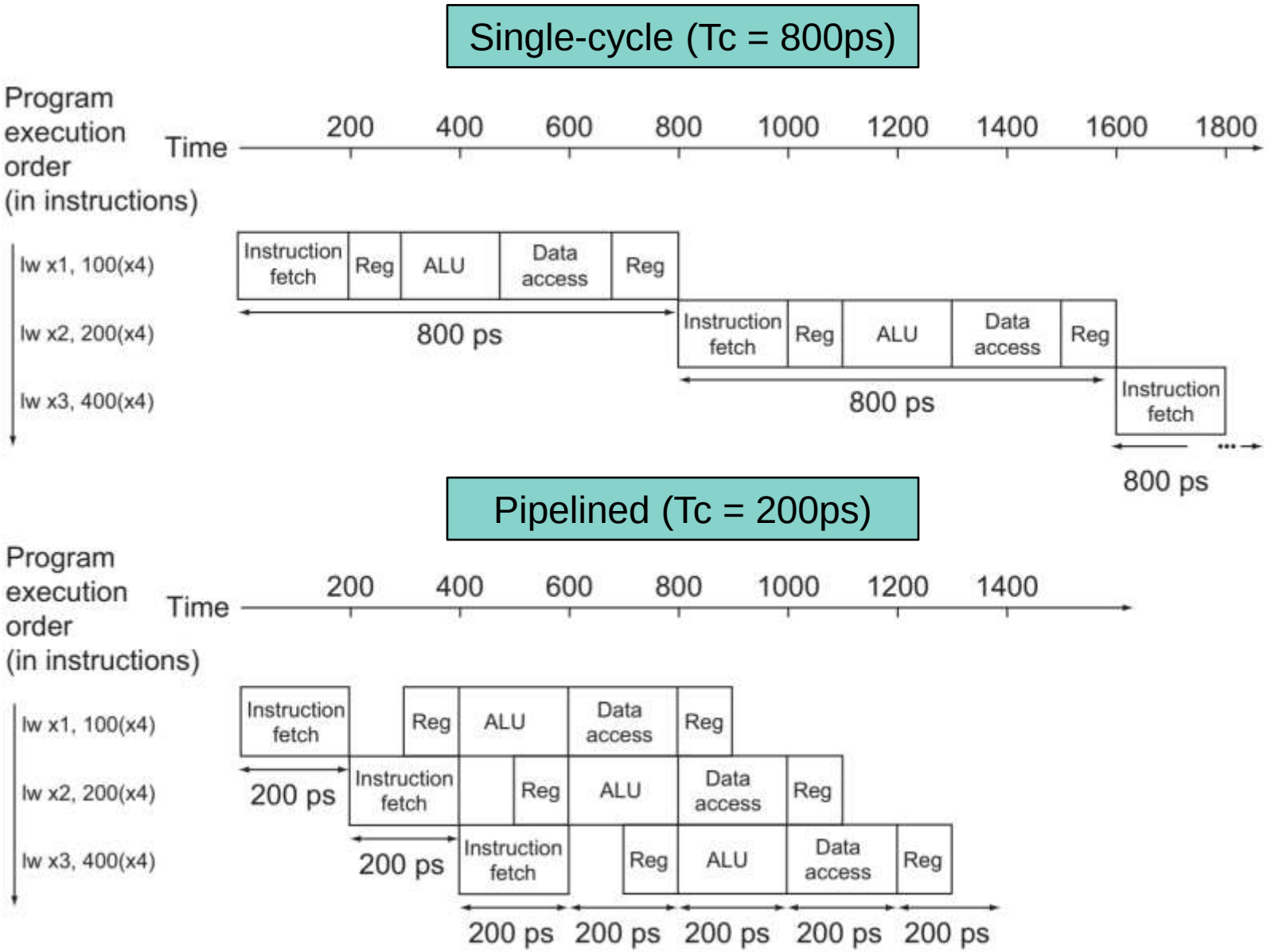
  - improves throughput - total amount of work done in a given time
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced
- In reality, speedup is less because of imbalance and overhead

Single Cycle versus Pipeline



- To complete an entire instruction in the pipelined case takes 1000 ps (as compared to 800 ps for the single cycle case). Why ?
- How long does each take to complete 1,000,000 adds ?

Example with lw instructions



Exercise

Assume that the following instructions are executed in a 5-stage-pipelined RISC-V CPU. Draw the timeline of each instruction

IF ID EX MEM WB

add s0, s1, s2

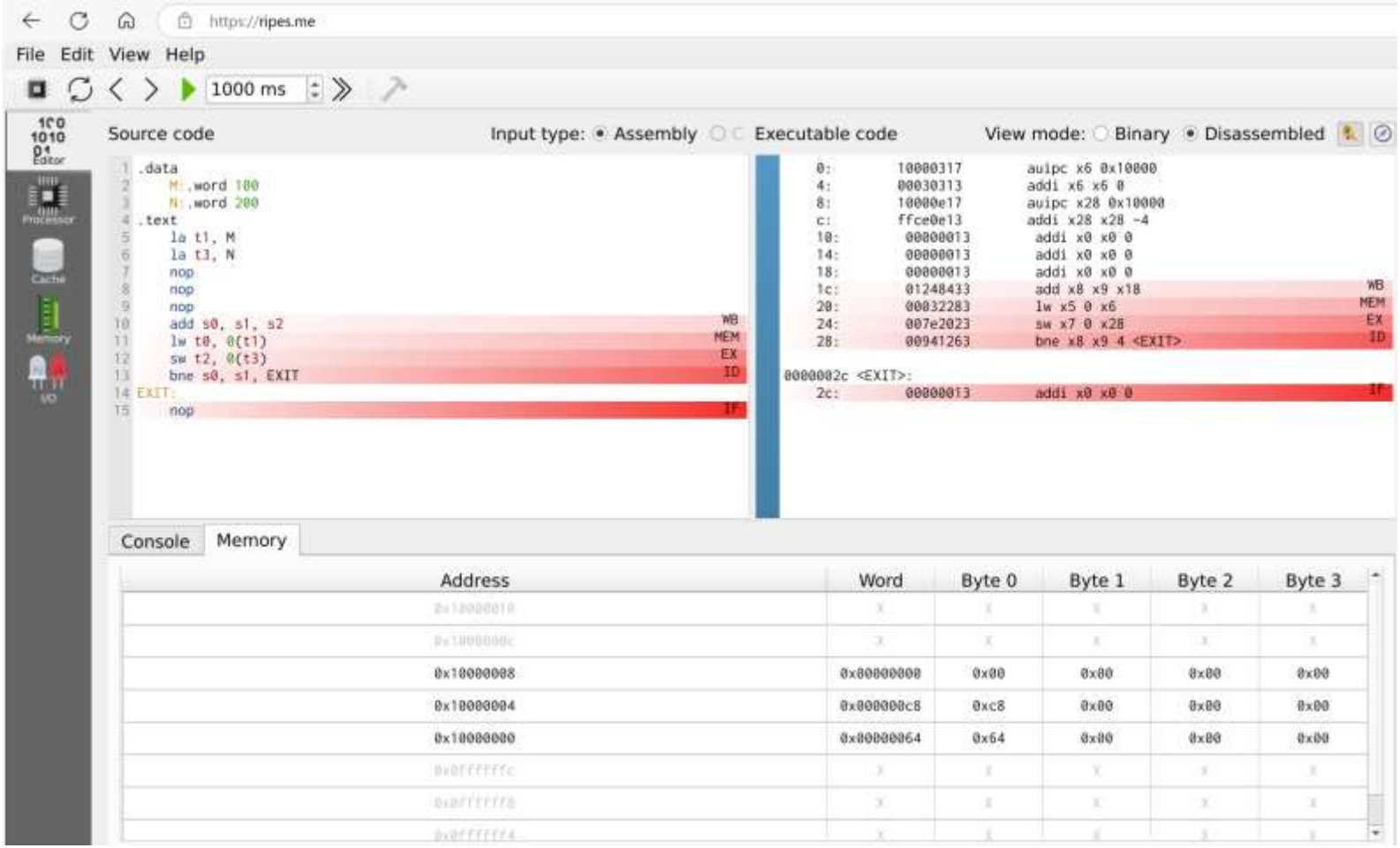
lw t0, 0(t1)

sw t2, 0(t3)

bne s0, s1, EXIT

Simulating the RISC-V pipeline

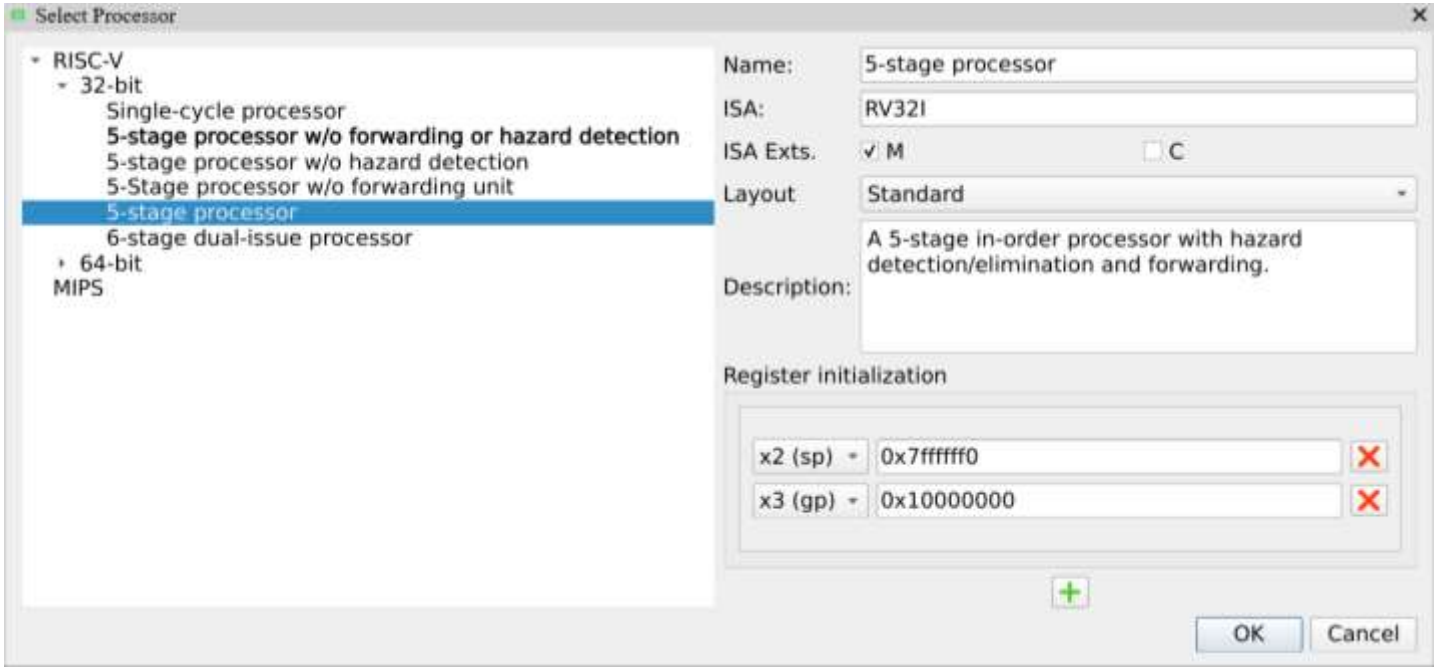
- Very handy tool: https://ripes.me/



Simulating the RISC-V pipeline

- Support several CPU configurations
  - Note: be careful, data hazards happens with la/li pseudo-instructions

```
1 .data
2     M: .word 100
3     N: .word 200
4 .text                                0:        10000317        auipc x6 0x10000
5     la t1, M                         4:        00030313        addi x6 x6 0
```



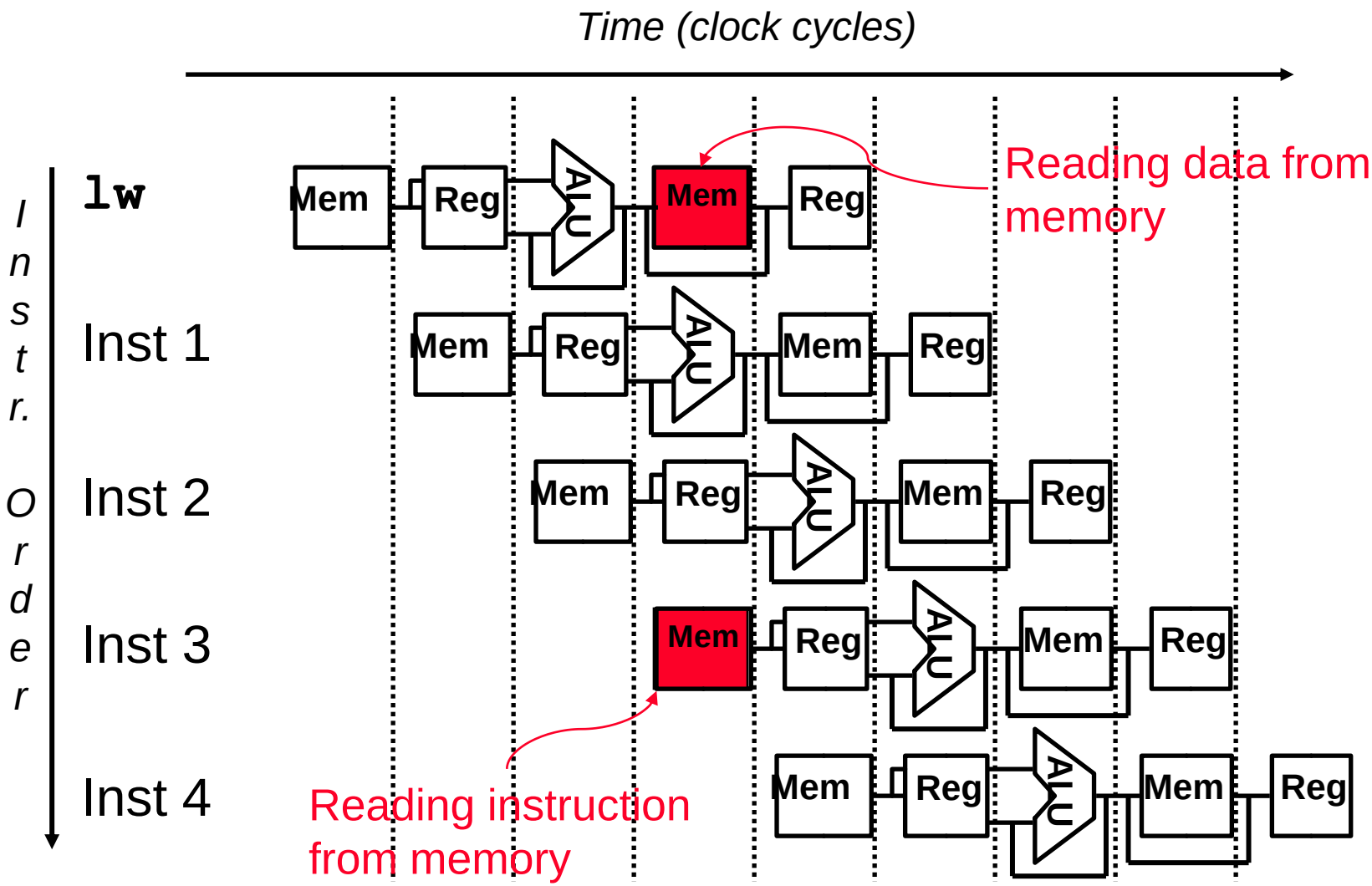
Pipeline Hazards

- Pipeline can lead us into troubles!!!
- Hazards: situations that prevent starting the next instruction in the next cycle
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch and jump instructions, exceptions
- In most cases, hazard can be solved simply by waiting
  - but we need better solutions to take advantages of pipeline

Structure Hazards

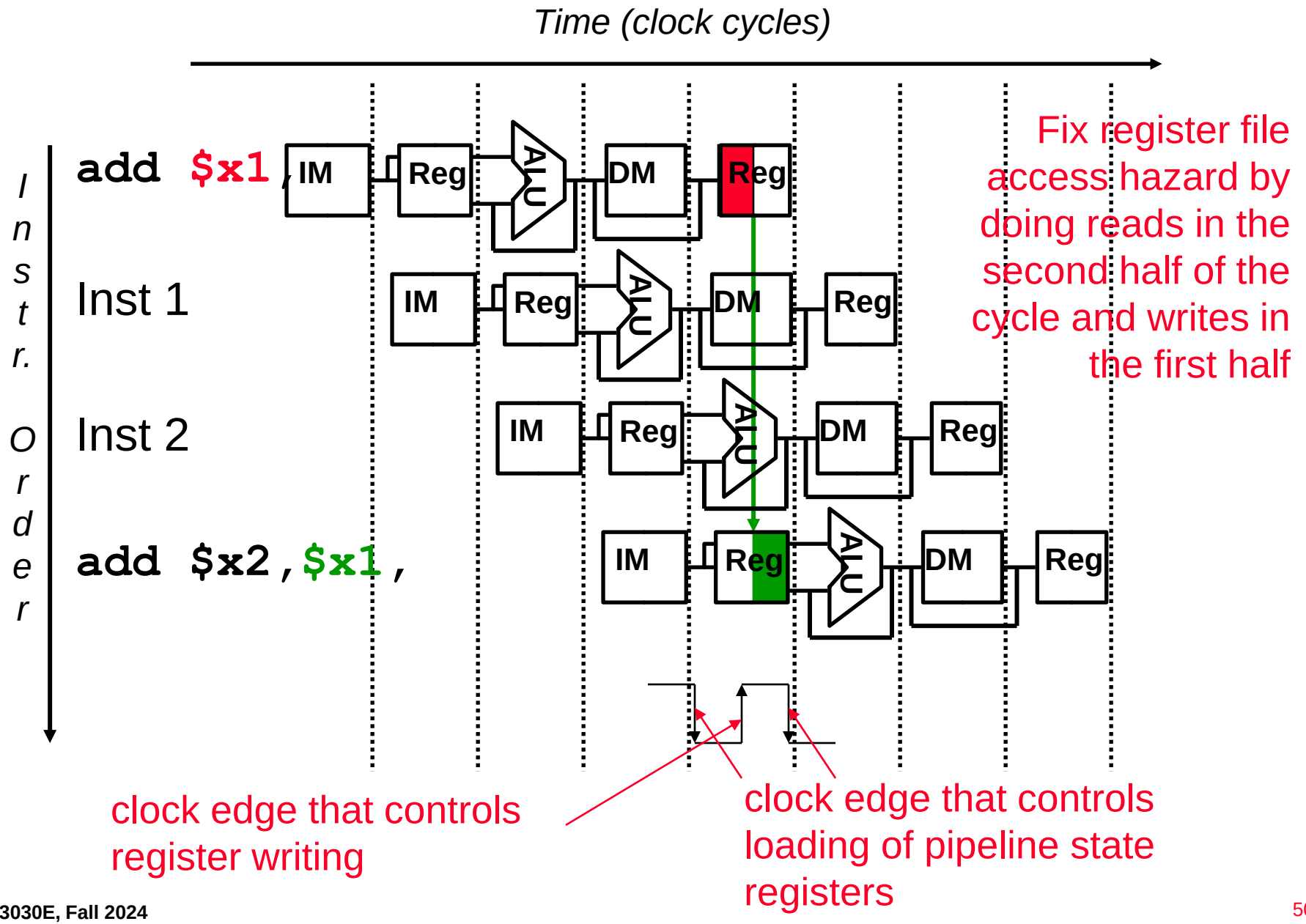
- Conflict for use of a resource
- In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

A Single Memory Would Be a Structural Hazard



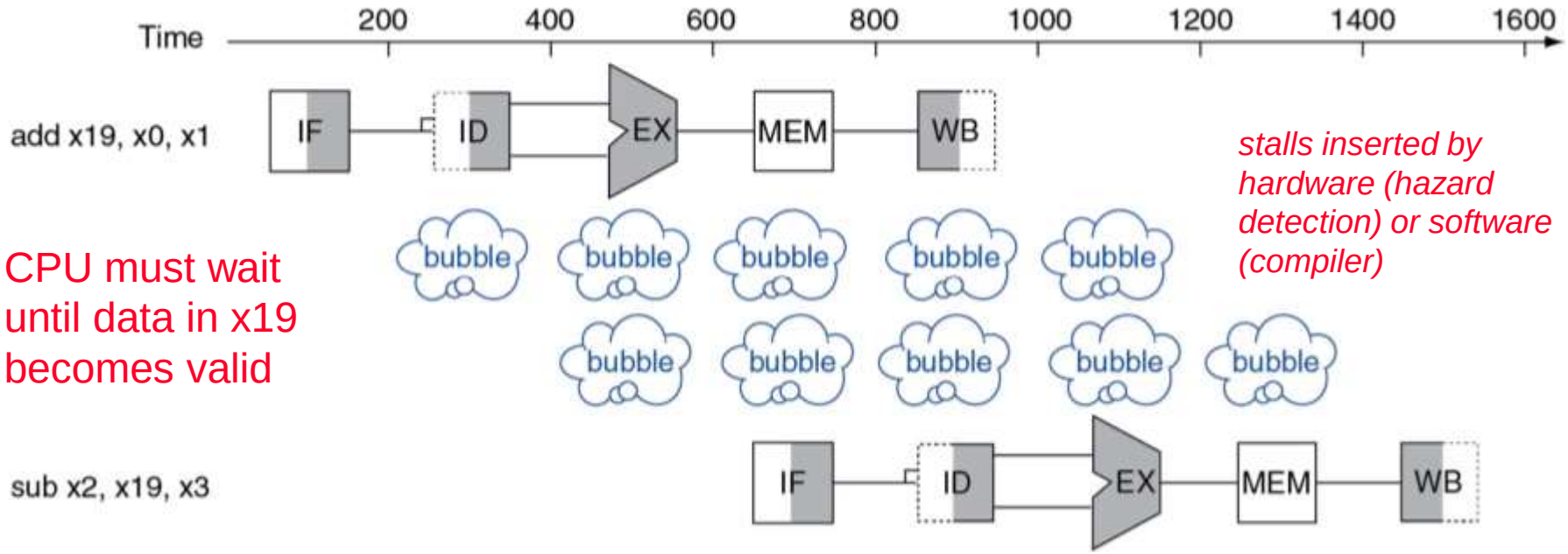
- Fix with separate instr and data memories (I$ and D$)

How About Register File Access?



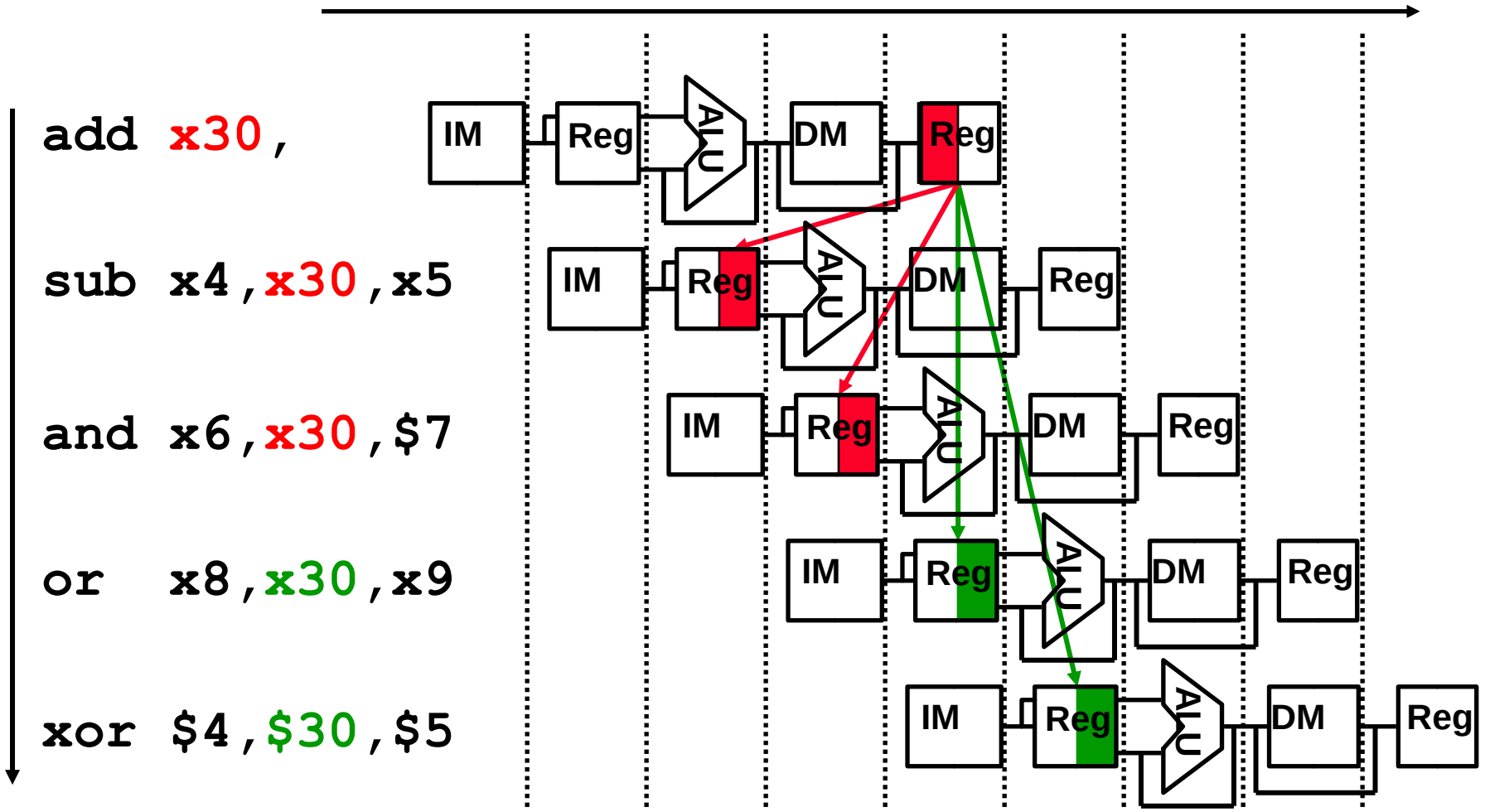
Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - `add  x19, x0, x1`
    `sub  x2, x19, x3`



Example

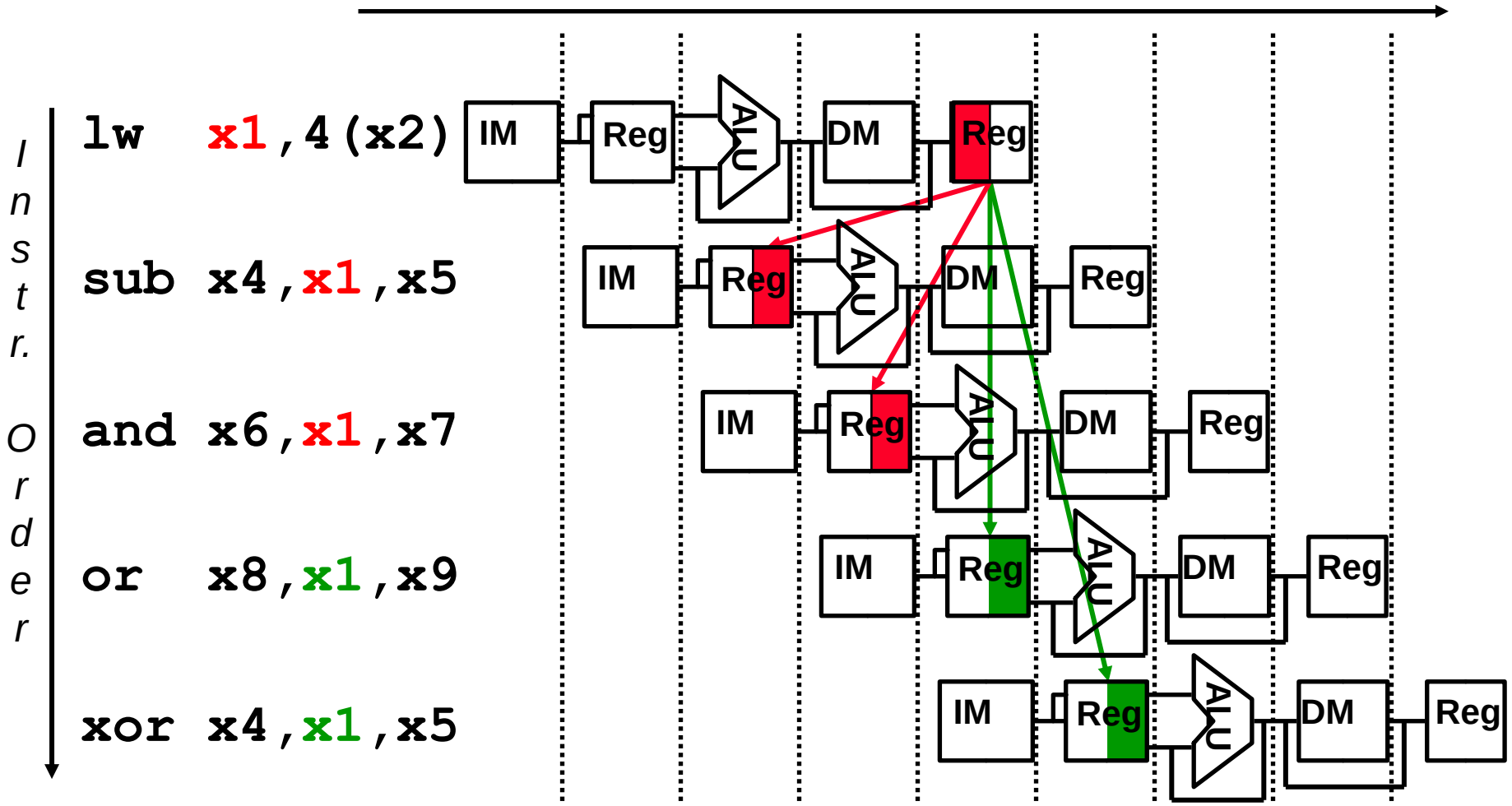
- Dependencies backward in time cause hazards



- Read before write data hazard

Example

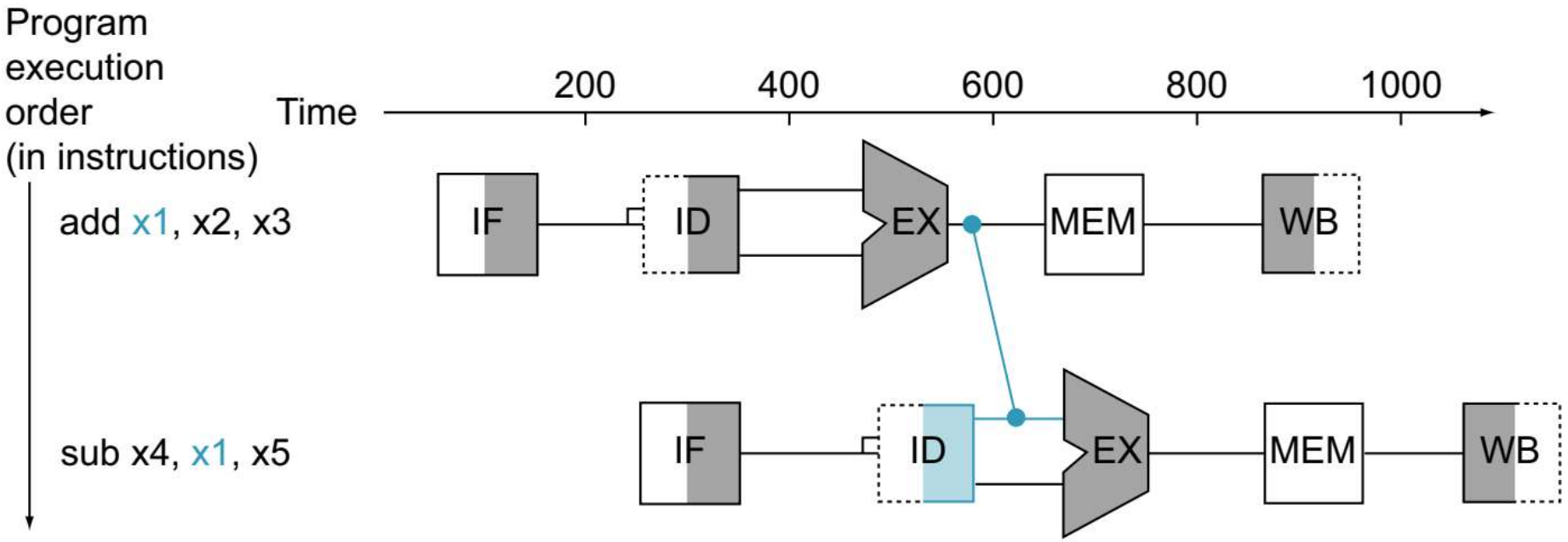
- Dependencies backward in time cause hazards



- Load-use data hazard

Solving hazard with Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath
- Forward from EX to EX (output to input)



Exercise

- What is the value of t2 after the below code is executed in the following CPU
  - CPU without hazard detection or forwarding
  - CPU with hazard detection but no forwarding
  - CPU with forwarding

```
addi t0, zero, 100
addi t1, zero, 200
add t2, t0, t1
nop
nop
nop
nop
```

Exercise

- CPU without hazard detection or forwarding
  - Incorrect value in t2 because of data hazard

Source code — Input type: Assembly

```
1  .data
2      M:.word 100
3      N:.word 200
4  .text
5      addi t0, zero, 100
6      addi t1, zero, 200
7      add t2, t0, t1
8      nop                    WB
9      nop                    MEM
10     nop                    EX
11     nop                    ID
12     nop                    IF
13     nop
14     nop
```

gpr

Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x00000000
x2	sp	0x7ffffff0
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000064
x6	t1	0x000000c8
x7	t2	0x00000000
x8	s0	0x00000000
x9	s1	0x00000000

Exercise

- CPU with hazard detection but no forwarding
  - Correct value in t2 but with additional 2 stalls

```
4  .text
5      addi t0, zero, 100                    MEM
6      addi t1, zero, 200                    EX
7      add t2, t0, t1                        ID
8      nop                                   IF
9      nop
10     nop
```

```
4  .text
5      addi t0, zero, 100                    WB
6      addi t1, zero, 200                    MEM
7      add t2, t0, t1                        ID
8      nop                                   IF
9      nop
10     nop
```

```
4  .text
5      addi t0, zero, 100
6      addi t1, zero, 200                    WB
7      add t2, t0, t1                        ID
8      nop                                   IF
9      nop
10     nop
```

```
4  .text
5      addi t0, zero, 100
6      addi t1, zero, 200
7      add t2, t0, t1                        EX
8      nop                                   ID
9      nop                                   IF
10     nop
```

gpr

Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x00000000
x2	sp	0x7ffffff0
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000064
x6	t1	0x000000c8
x7	t2	0x0000012c
x8	s0	0x00000000
x9	s1	0x00000000

Exercise

- CPU with forwarding
  - Correct value in t2 with no additional stalls
  - Is hazard detection required in this case?

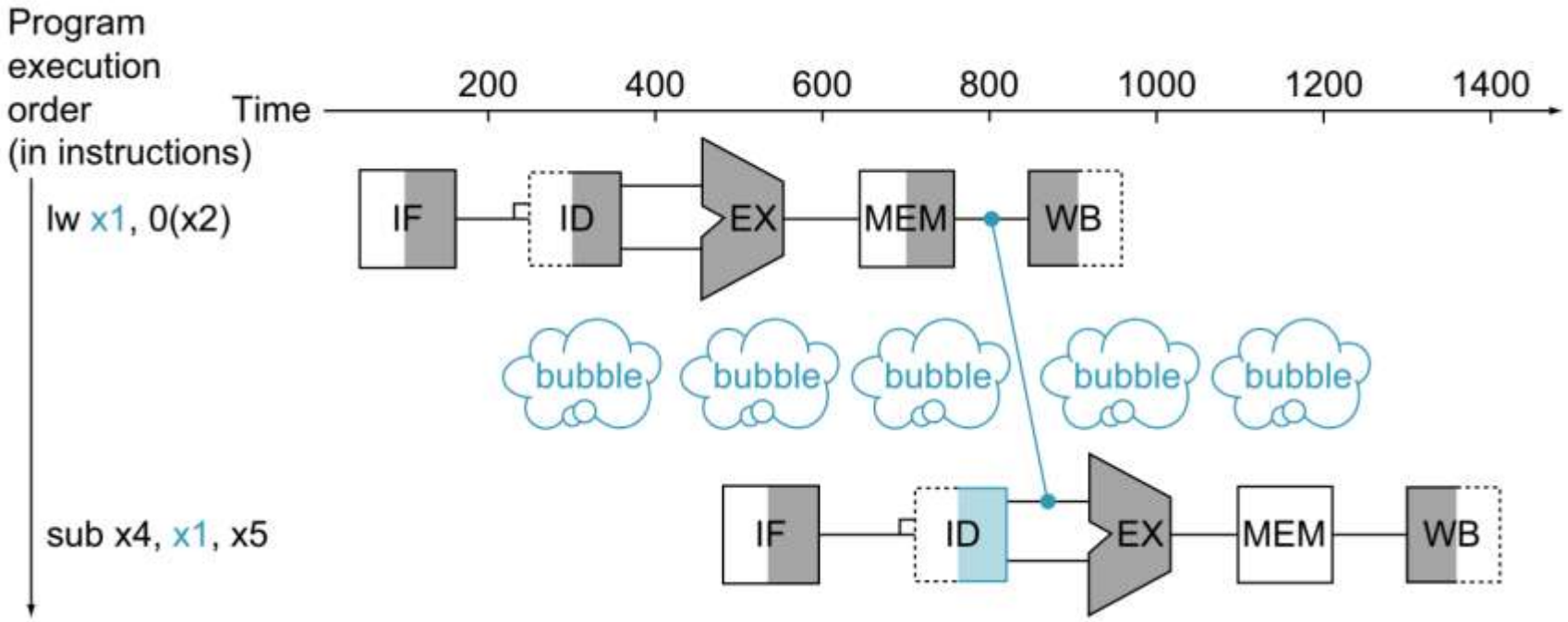
```
1  .data
2      M: .word 100
3      N: .word 200
4  .text
5      addi t0, zero, 100      WB
6      addi t1, zero, 200      MEM
7      add t2, t0, t1          EX
8      nop                     ID
9      nop                     IF
10     nop
11     nop
12     nop
```

gpr

Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x00000000
x2	sp	0x7ffffff0
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000064
x6	t1	0x000000c8
x7	t2	0x0000012c
x8	s0	0x00000000
x9	s1	0x00000000

Solving Load-Use Data Hazard

- Forward from MEM (output) to EX (input)
- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!
  - One cycle stall is necessary → handle by software, or by hardware hazard detection



Exercise

- What is the value of t2 after the below code is executed in the following CPU configuration
  - Without hazard detection or forwarding
  - With forwarding but no hazard detection
  - With hazard detection but no forwarding
  - With both hazard detection and forwarding

```
.data
M:.word 100
.text
auipc t1, 0x10000 #la t1, M
nop
nop
nop
nop
lw t0, 0(t1)
addi t2, t0, 100
nop
nop
nop
nop
```

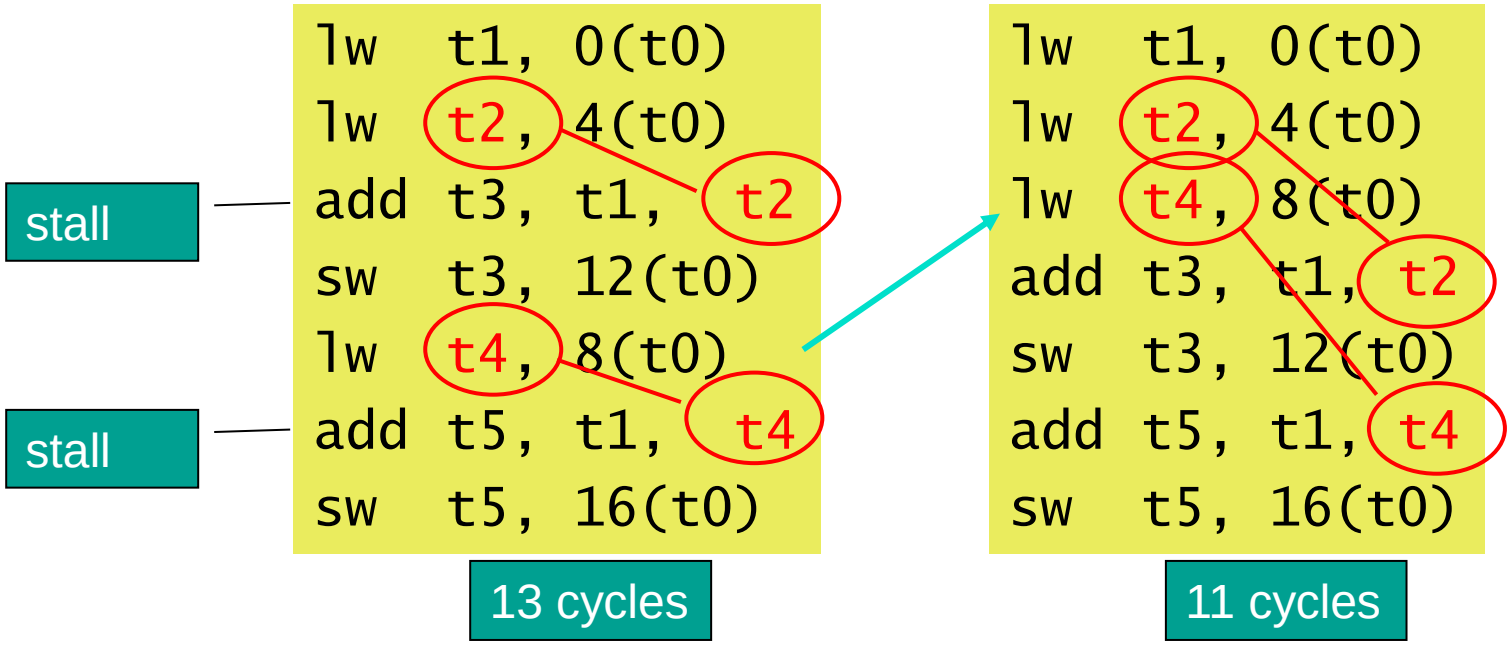
Exercise

- Without hazard detection or forwarding
  - t2 = 0x64
- With forwarding but no hazard detection
  - t2 = 0x64
- With hazard detection but no forwarding
  - t2 = 0xc8, with additional 2 stalls
- With both hazard detection and forwarding
  - t2 = 0xc8, with additional 1 stall

Code scheduling to avoid stalls

- Reorder code to avoid use of load result in the next instruction
- C code:

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

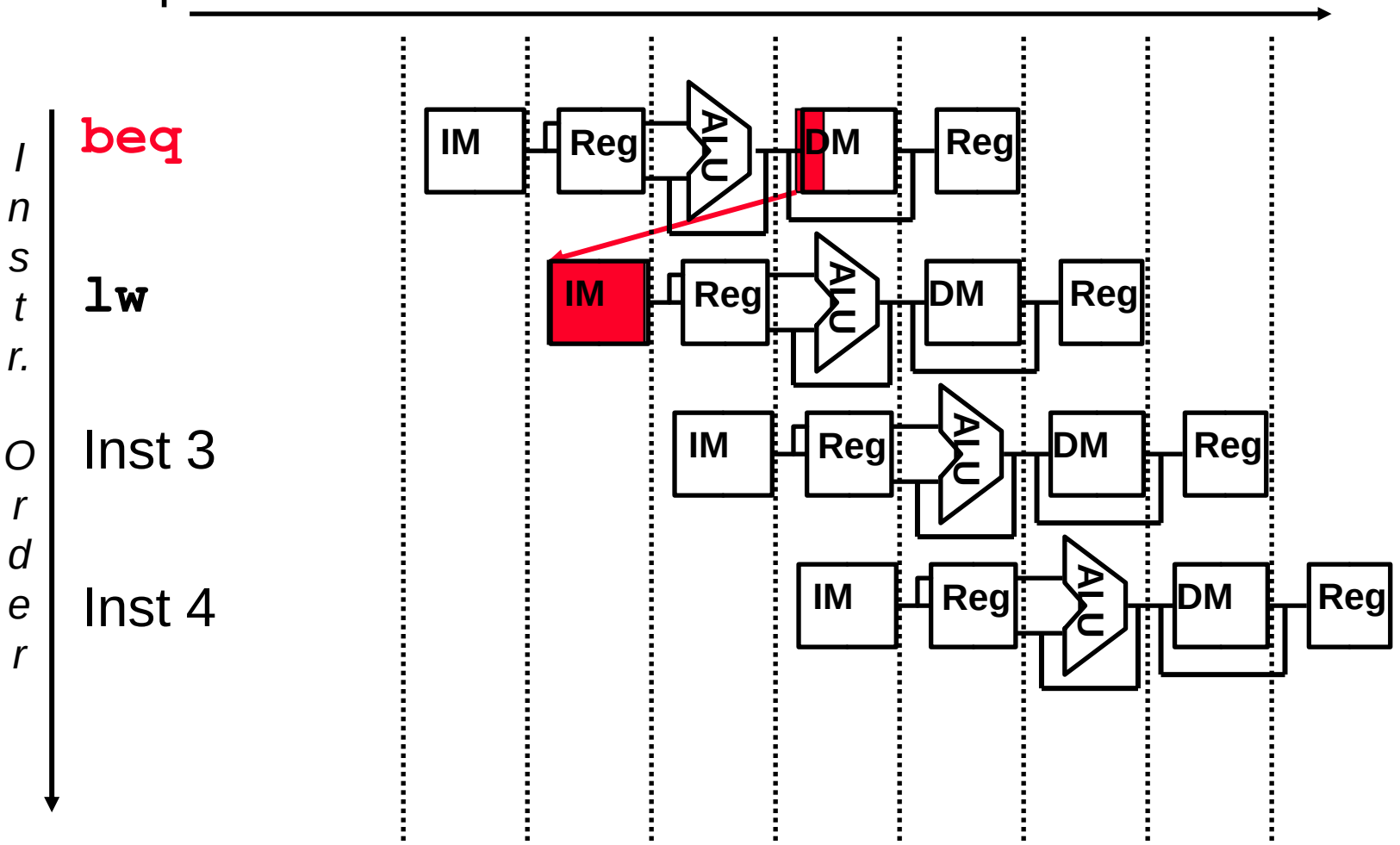


Control hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch
- In RISC-V pipeline
  - Need to **compare registers and compute target early** in the pipeline
  - Add **hardware to do it in ID stage**

Branch instructions cause control hazards

- Dependencies backward in time cause hazards

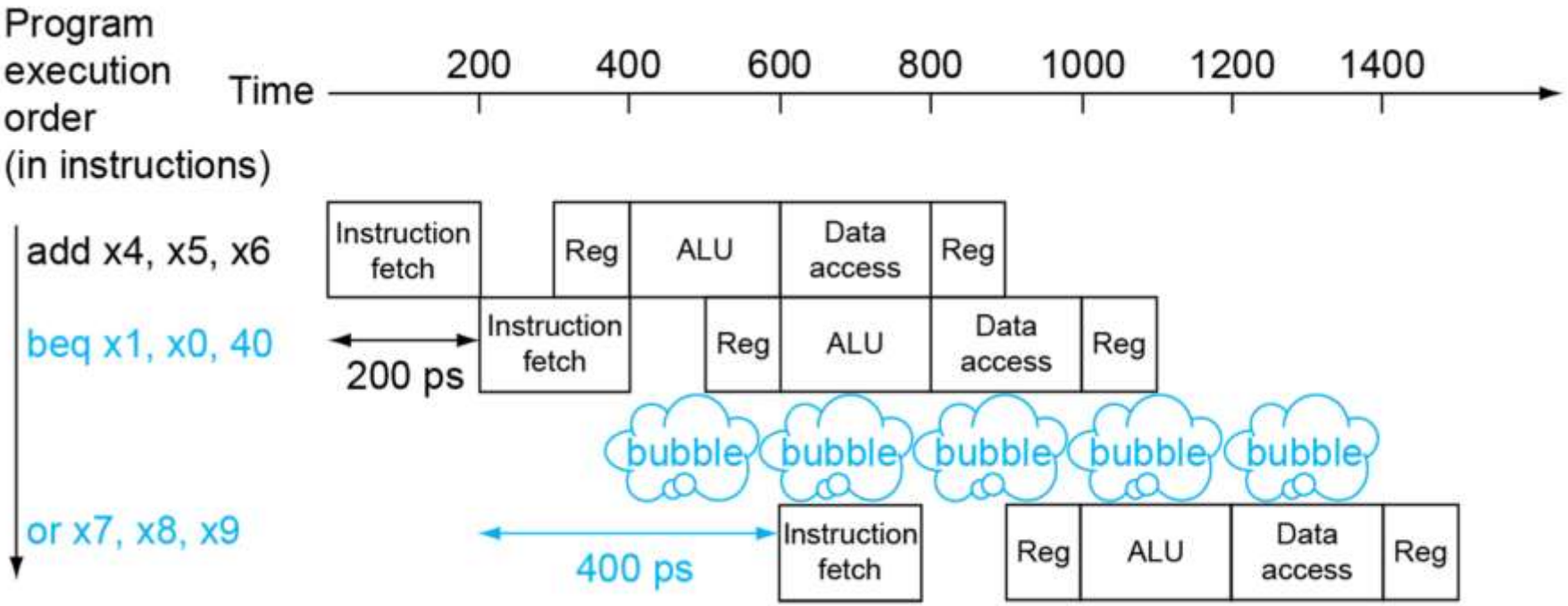


Solving control hazards

- Delayed branch
- Compute target earlier
  - Reduce number of stall cycles per branch instr.
  - Need to **compare registers and compute target early** in the pipeline.
  - Add **hardware to do it in ID stage.**
  - May cause additional stall in case of data hazards.
- Branch prediction
  - Heuristically improve overall performance.
  - Complicated datapath with additional component.

Stall on Branch (Delayed branch)

- Wait until branch outcome determined before fetching next instruction

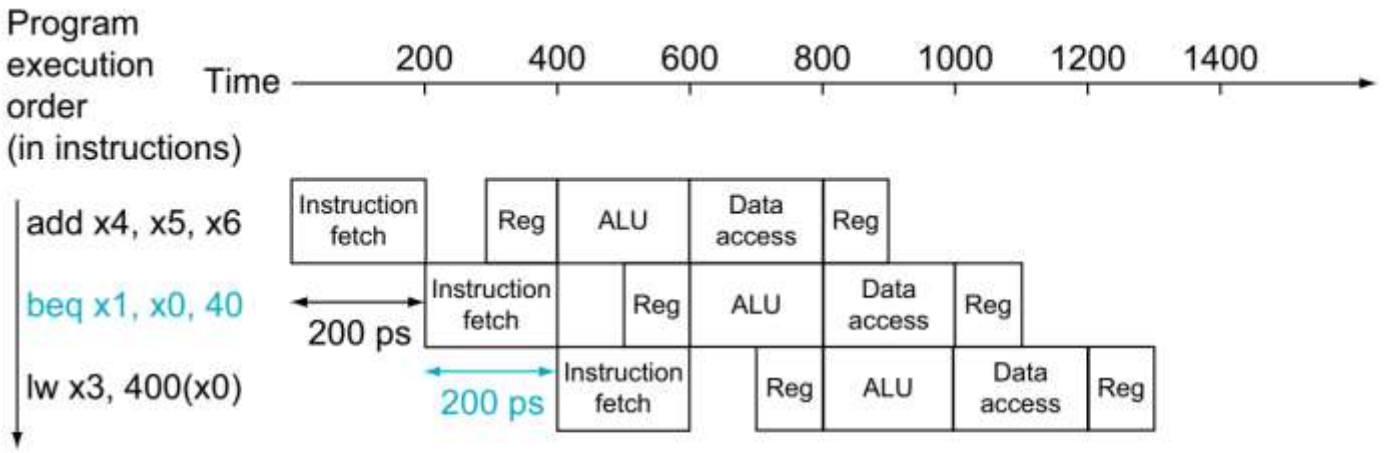


Branch prediction

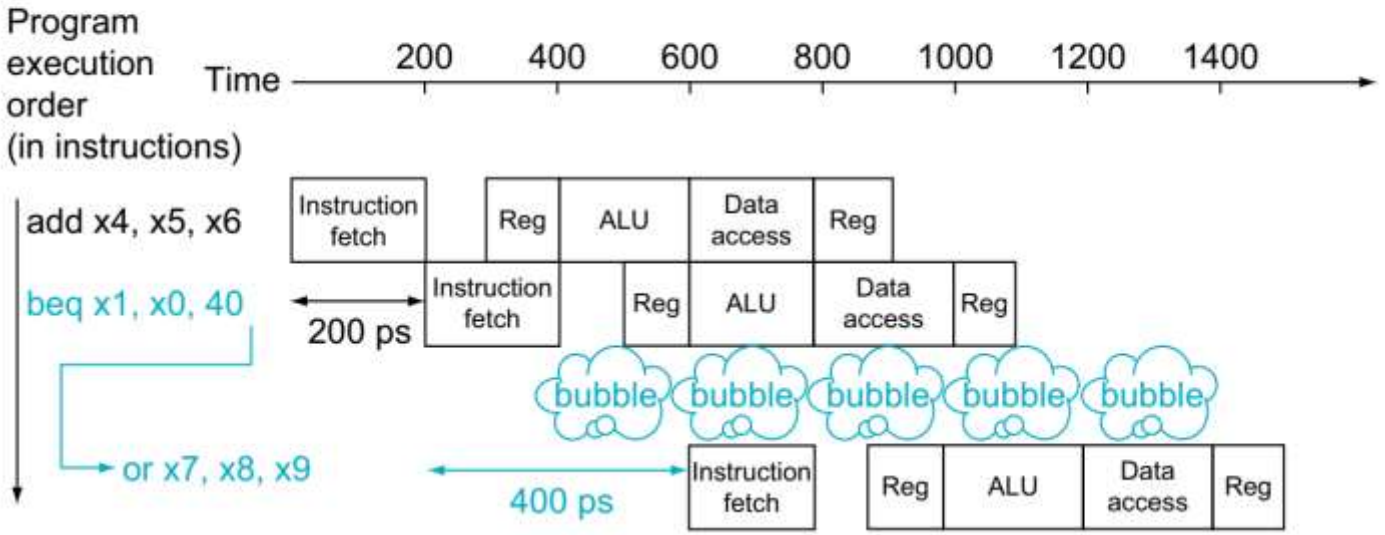
- Predict outcome of branch
- Only stall if prediction is wrong
- In RISC-V pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

RISC-V with Predict Not Taken

Prediction correct



Prediction incorrect



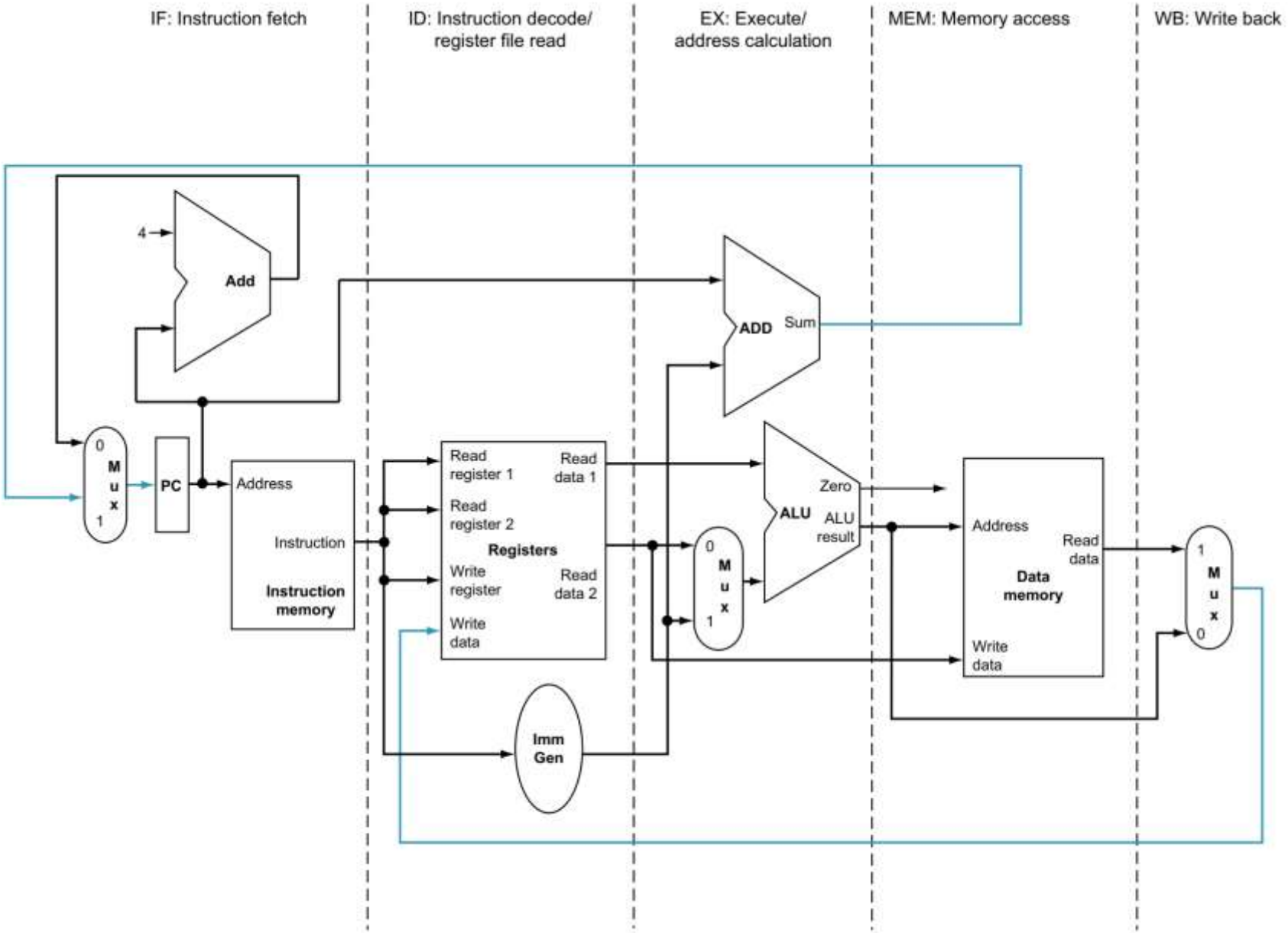
More-realistic branch prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history
  - Accuracy can reach >90% with SPectInt

Designing RISC-V Pipelined Datapath

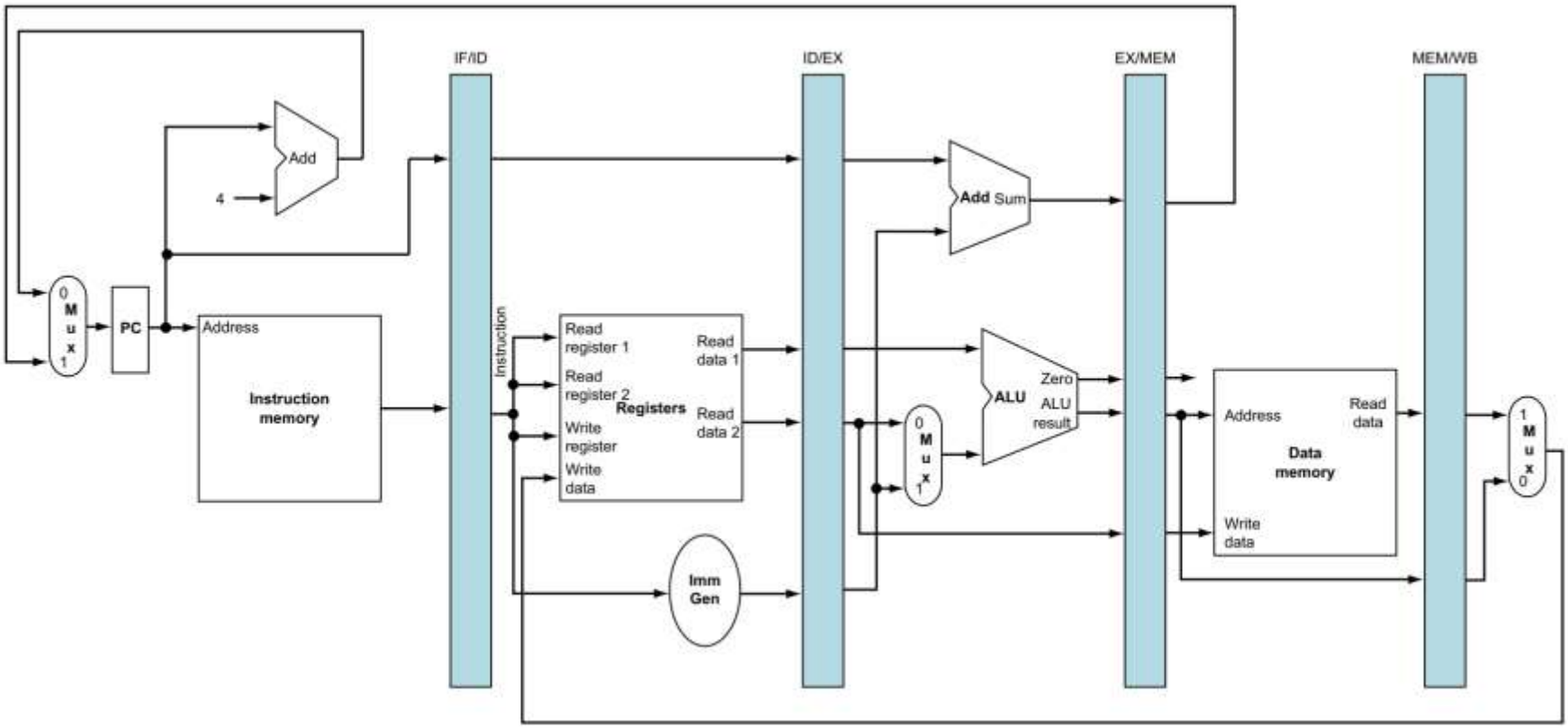
- Let's see how pipelined datapath works
  - All stages can work simultaneously.
  - Output from previous stage/cycle is input of next stage/cycle.
- And how it is constructed
  - Pipeline diagrams for load & store instructions.
  - Adding supports for handling hazards.

Designing RISC-V pipelined datapath

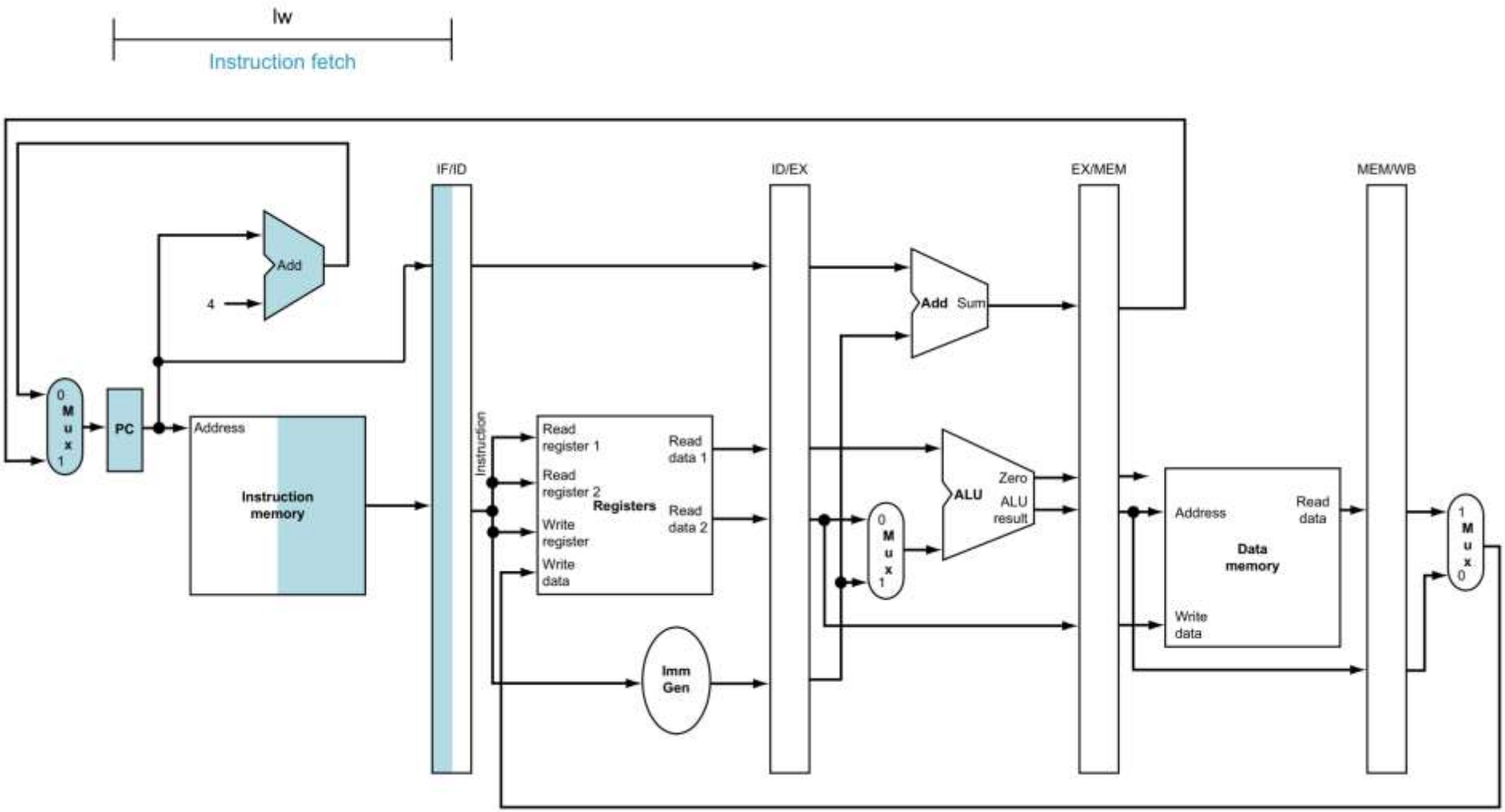


Pipeline registers

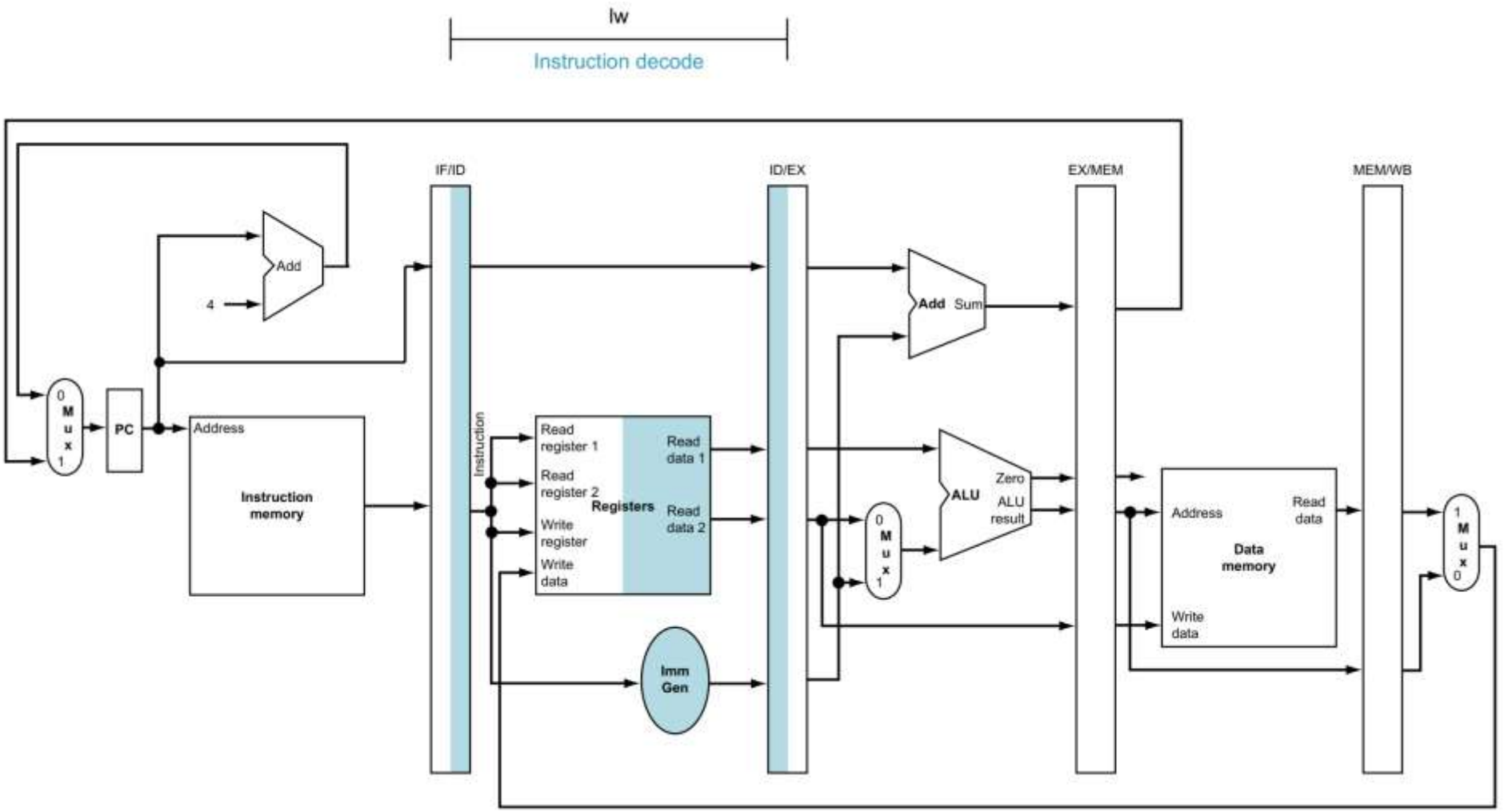
- Need registers between stages
  - To hold information produced in previous cycle



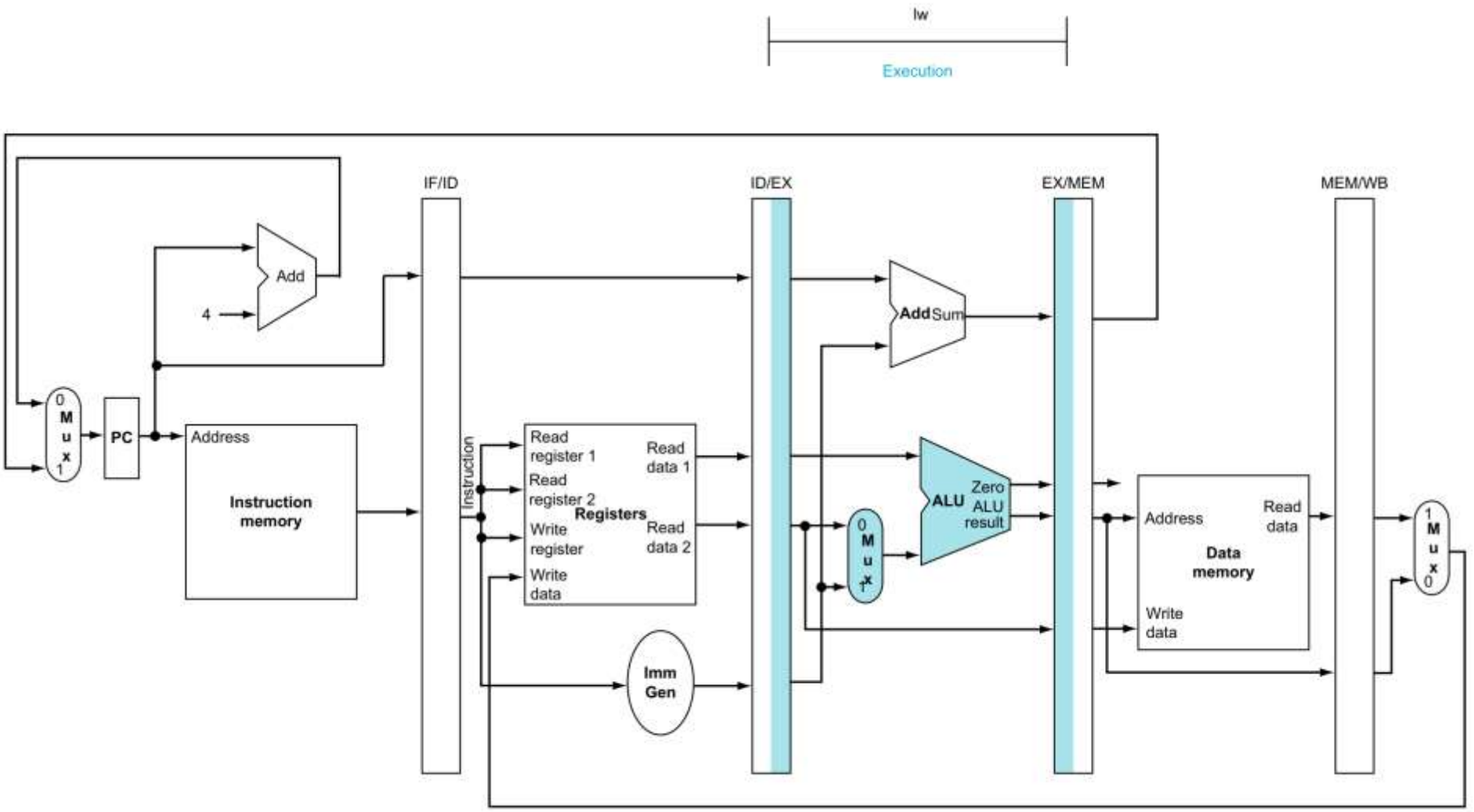
IF stage


lw
Instruction fetch
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
Imm Gen
Add Sum
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
1
M
u
x
0

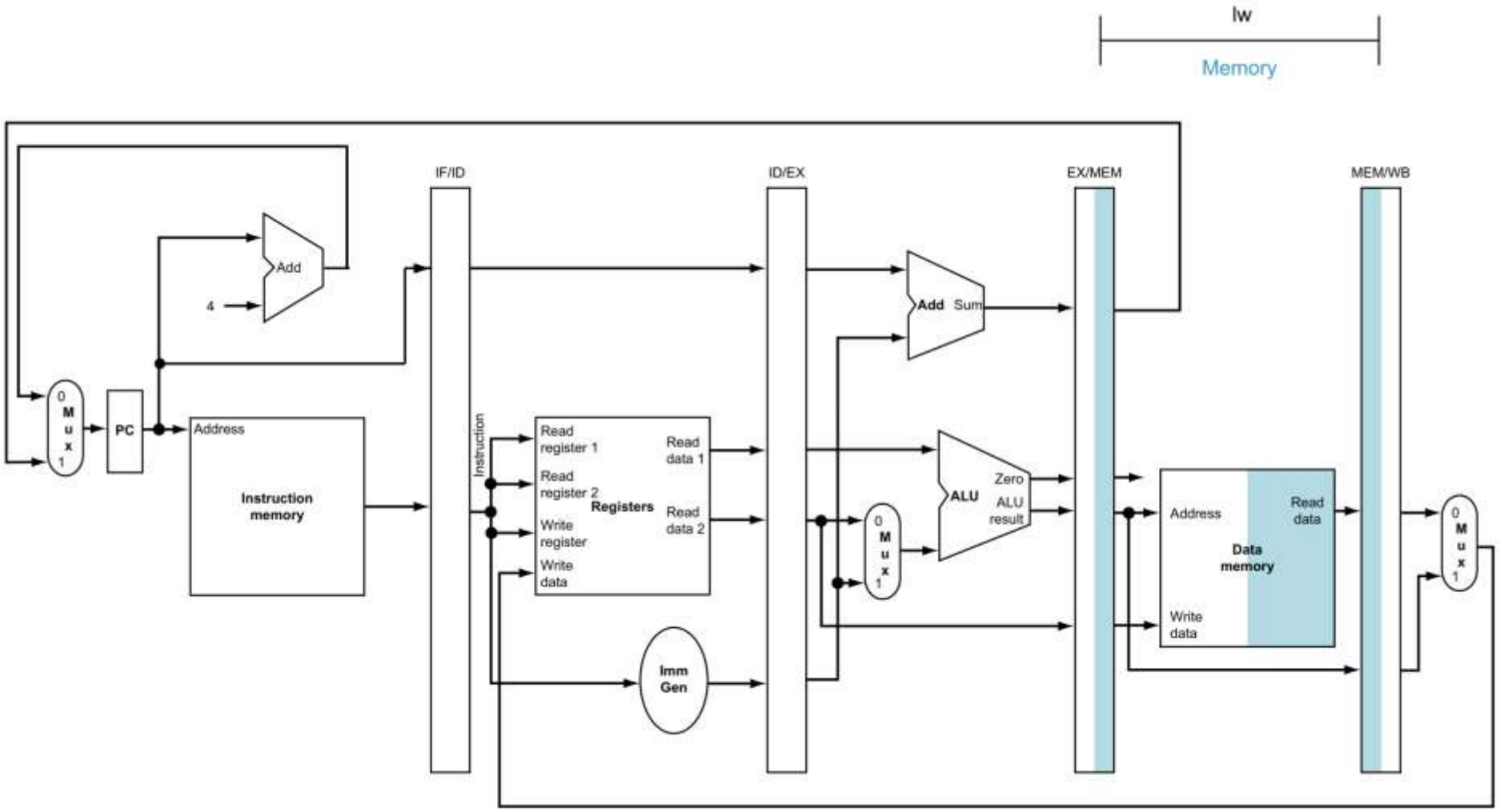
ID stage


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

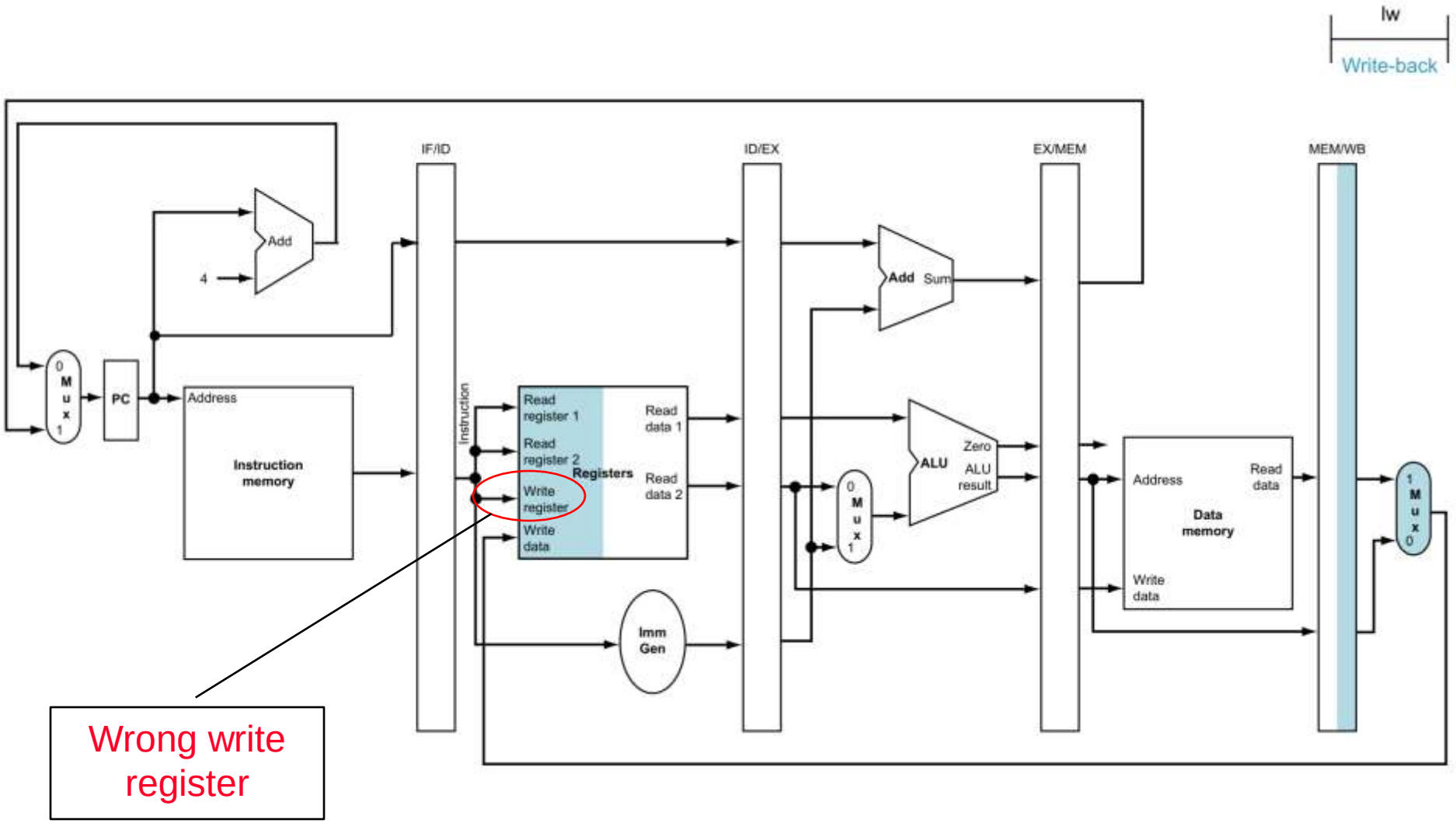
EX stage


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

MEM stage for the lw instruction

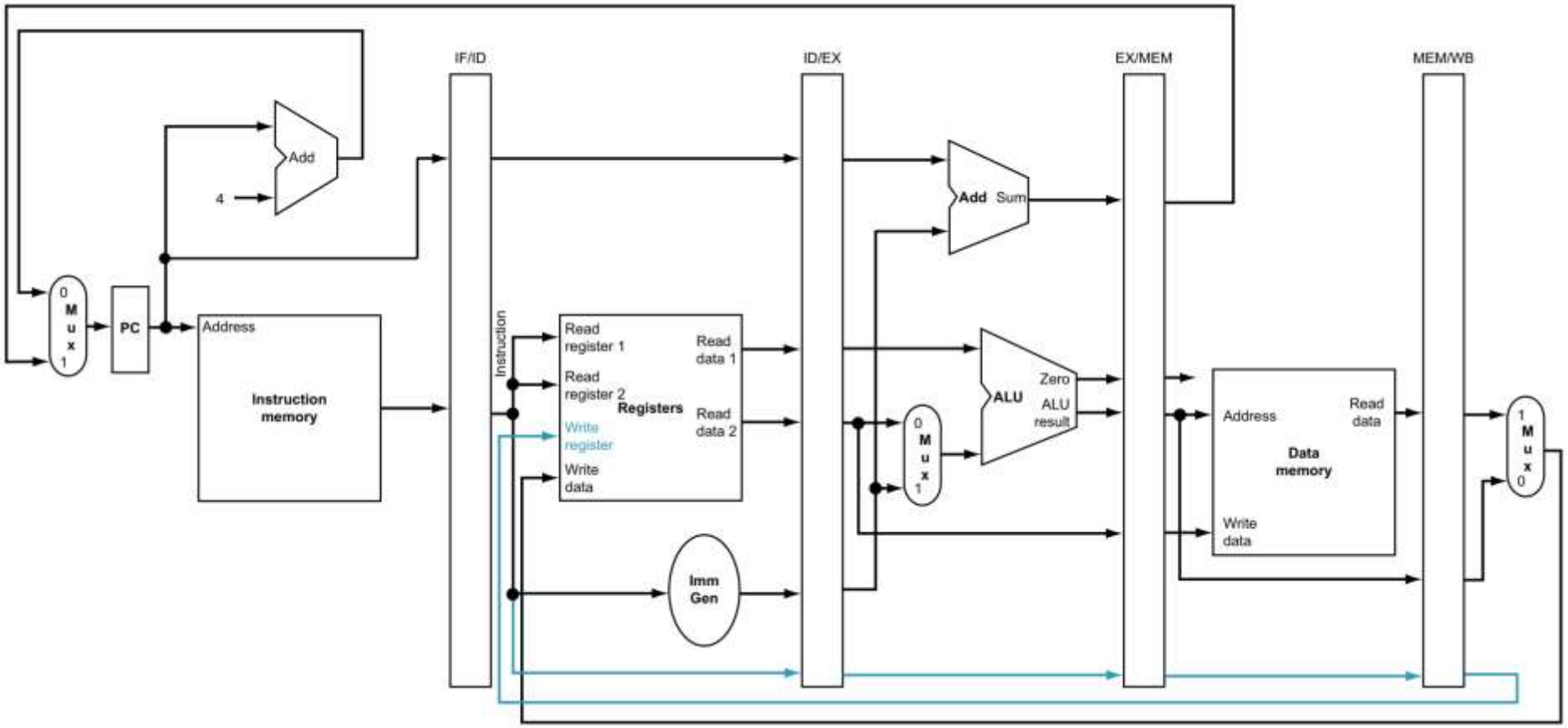


WB for the lw instruction



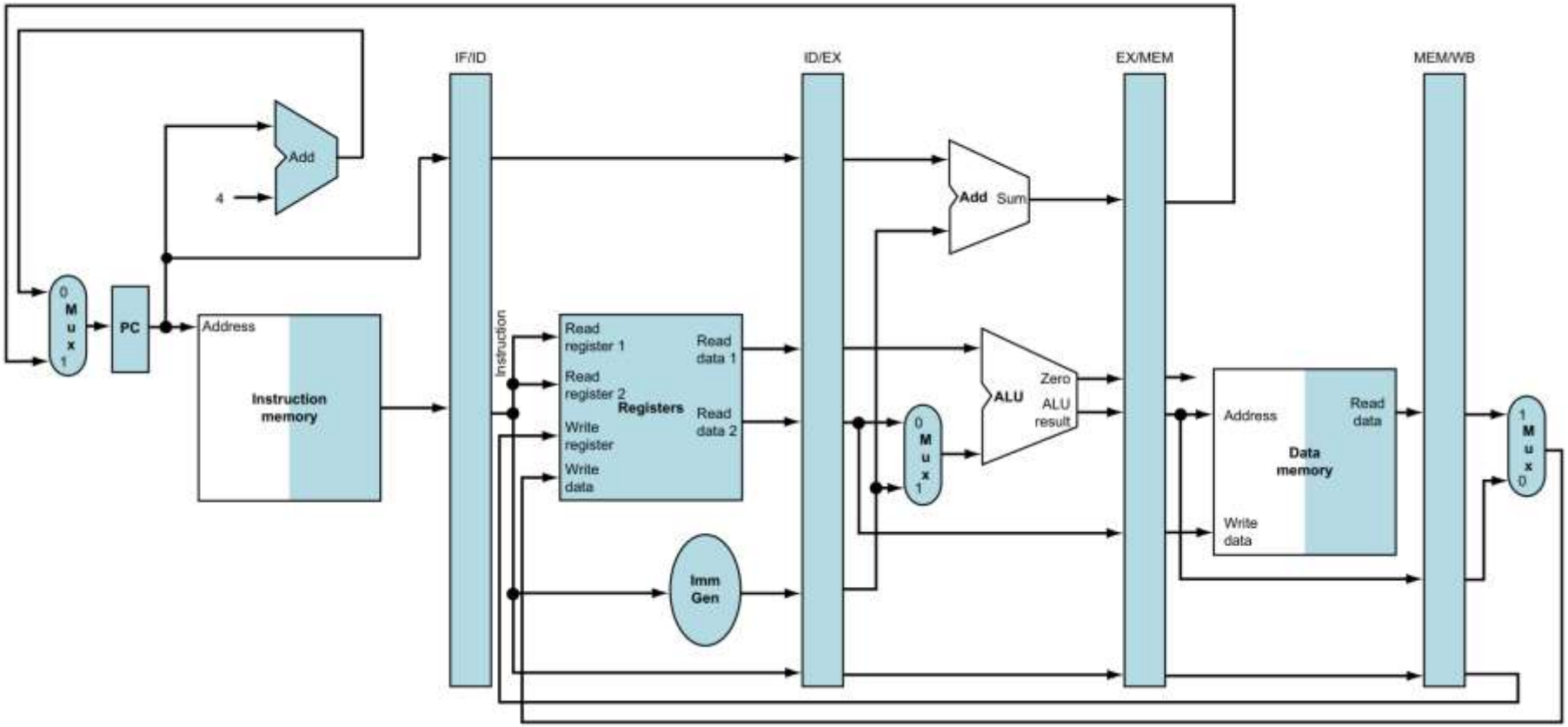
Correction to support Load instruction

- Correct the write register

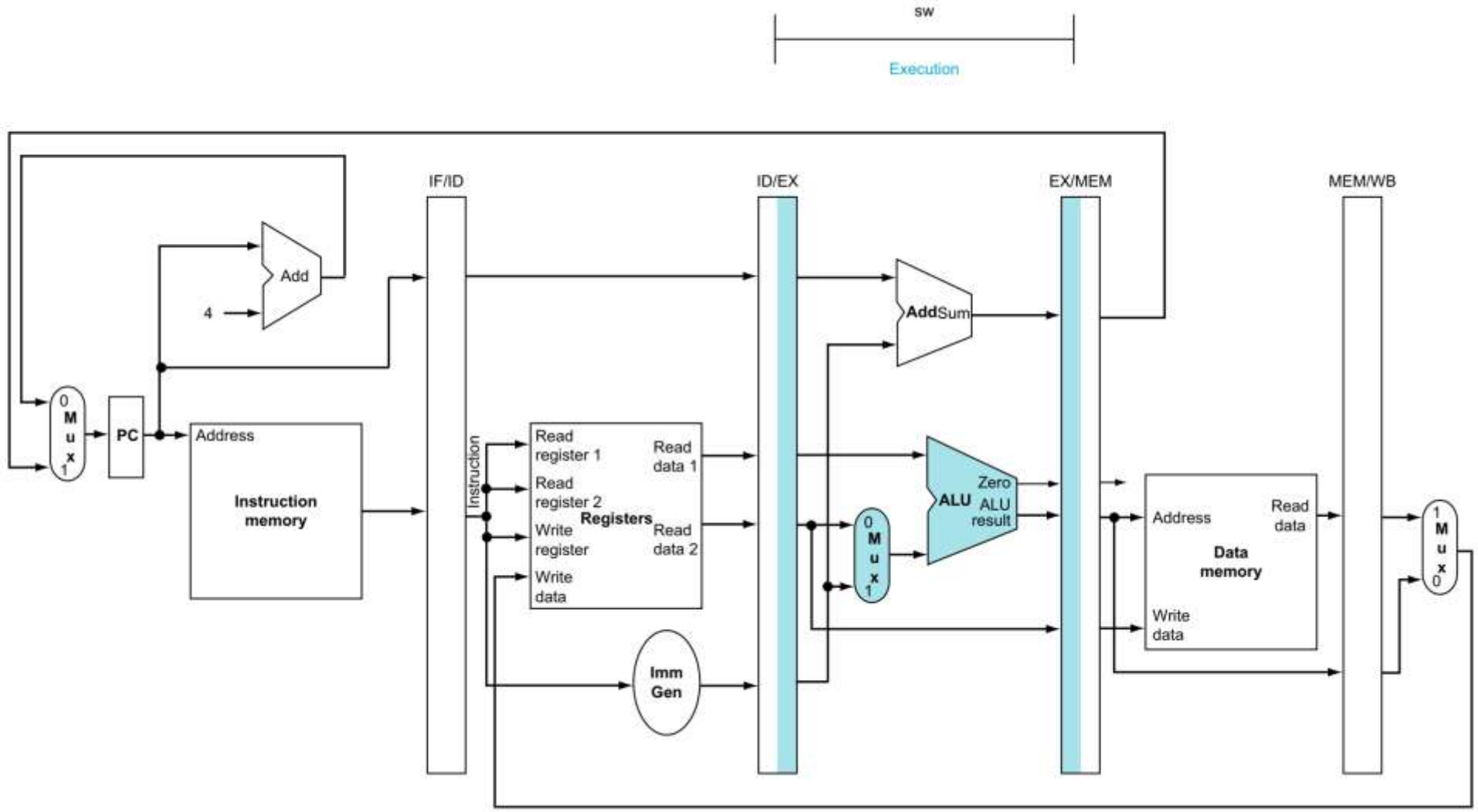


Datapath in all five stages of a load instruction

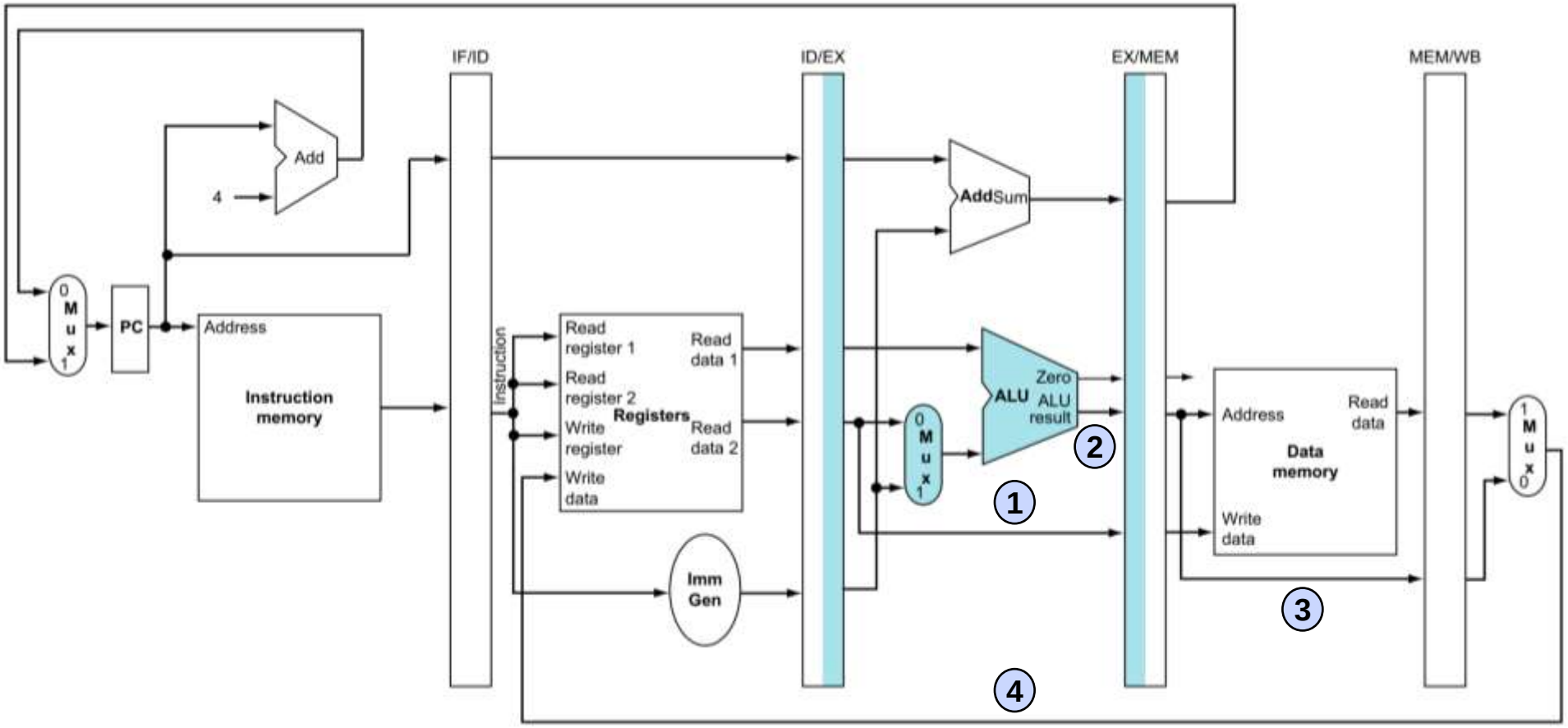
- lw is the “longest” instruction, all stages are utilized.



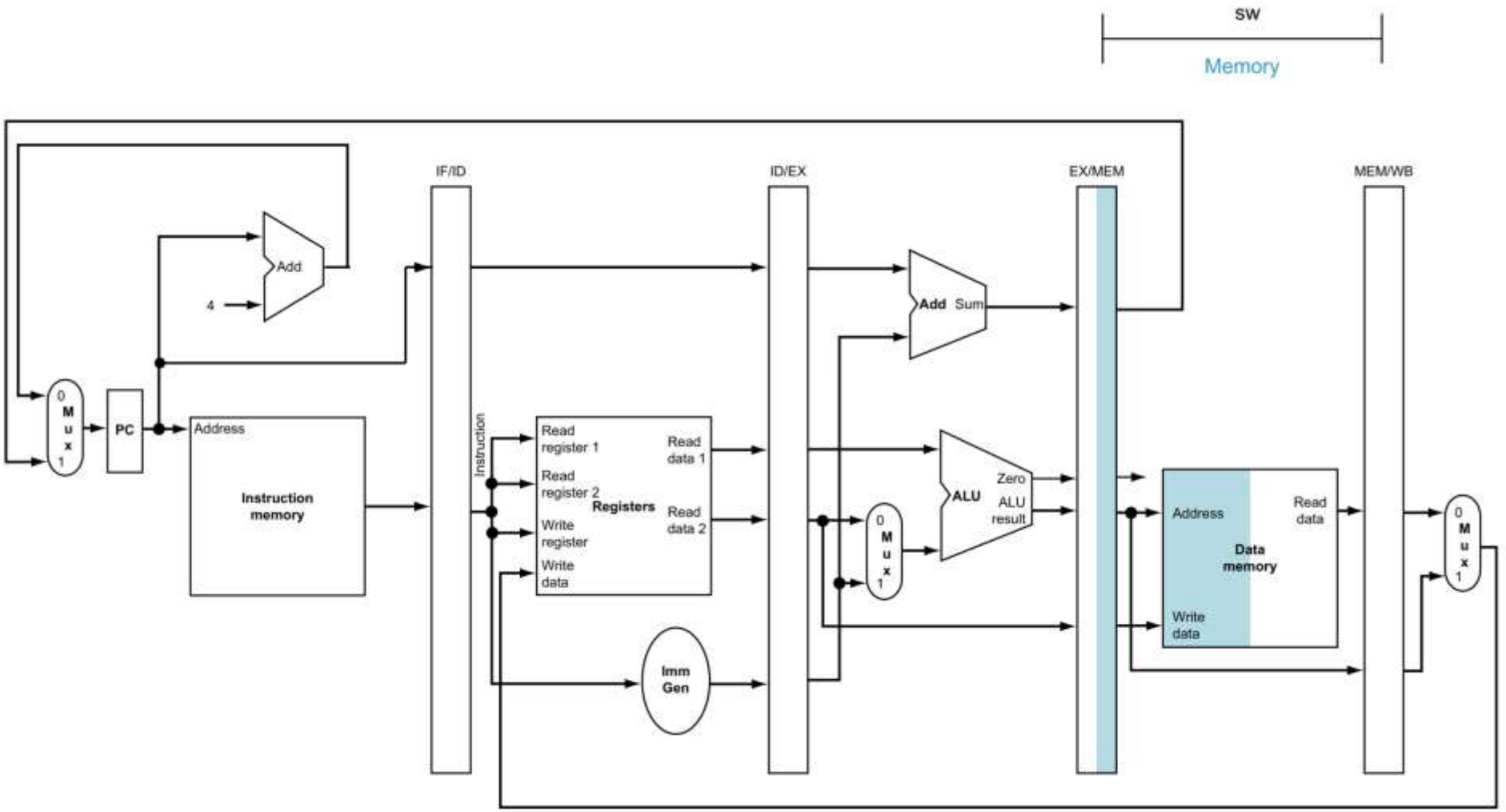
EX for the sw instruction



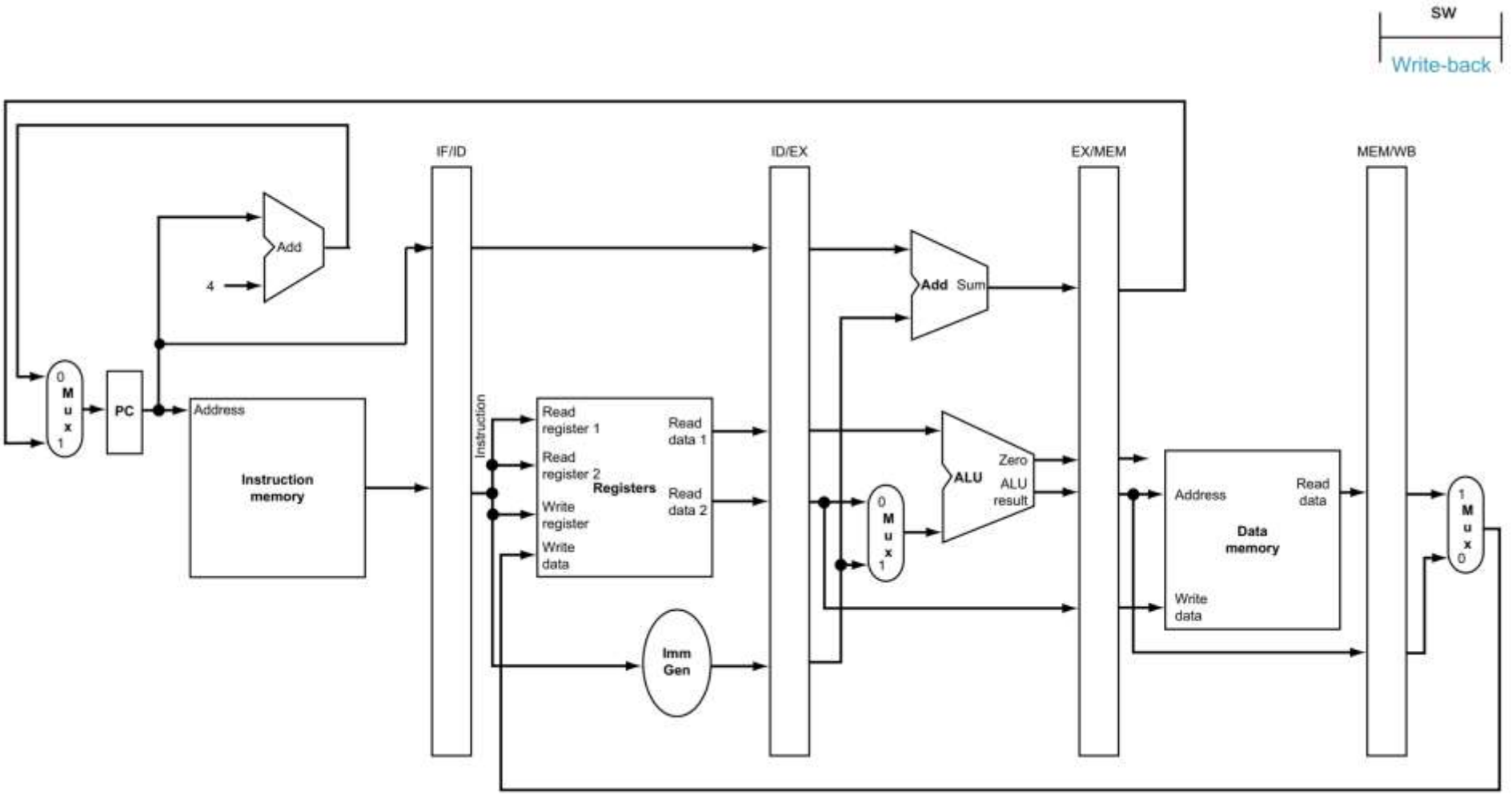
EX for the sw instruction



MEM for the sw instruction



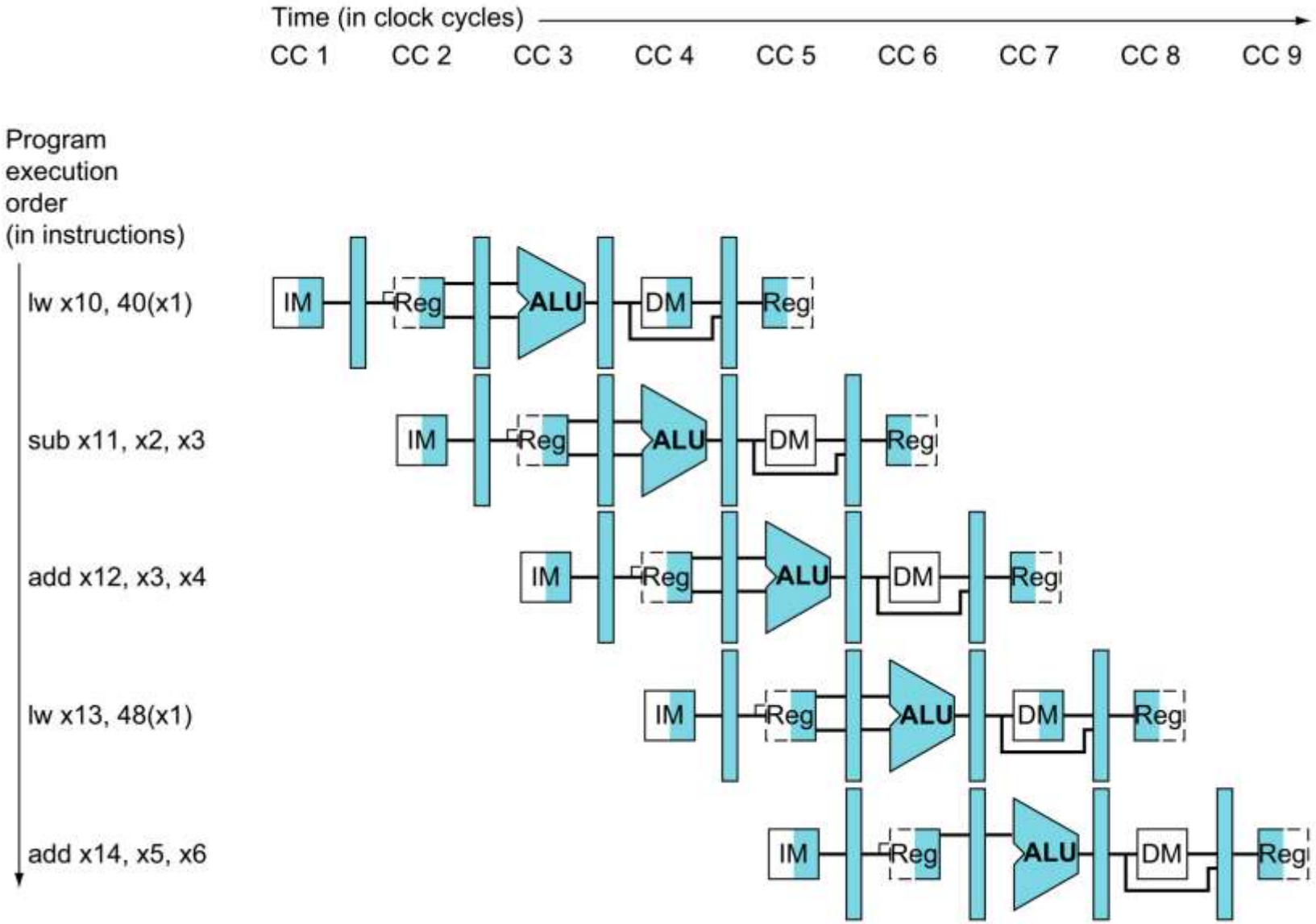
WB for the sw instruction



Graphically representing pipelines

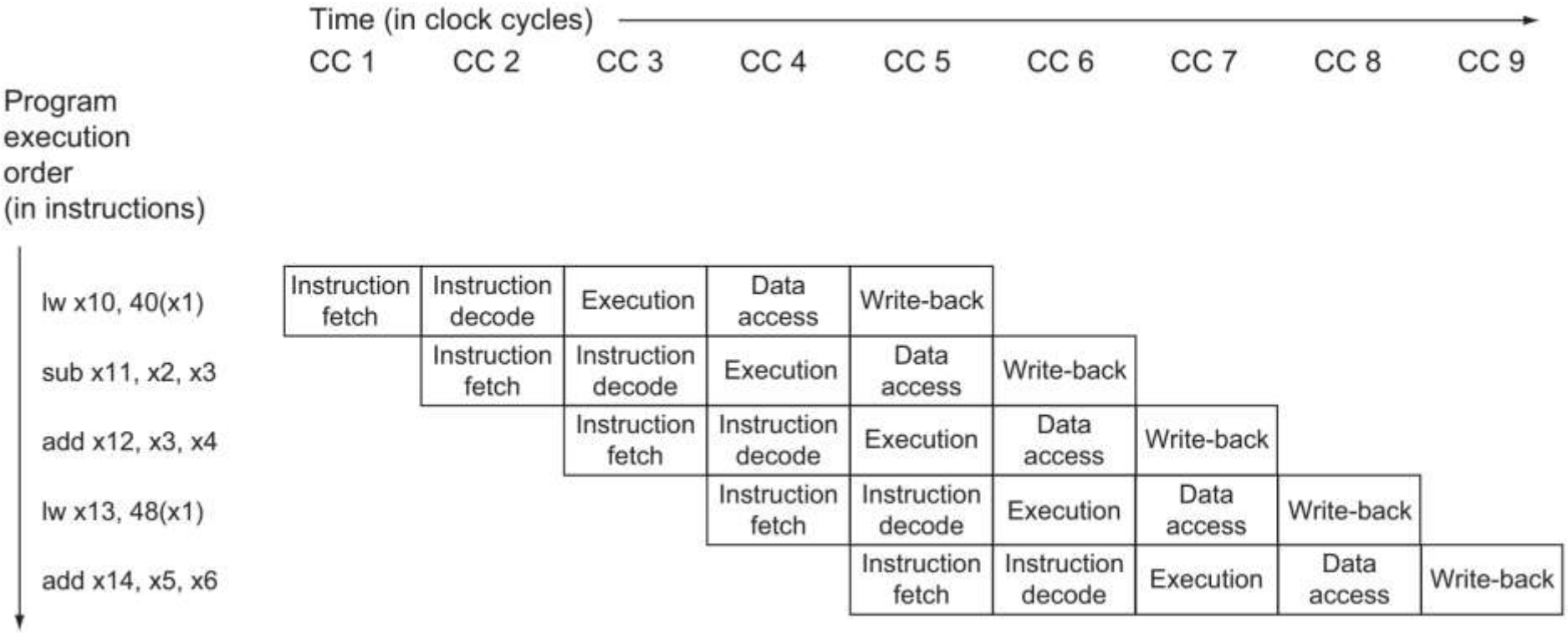
- Two ways to represent pipeline graphically
  - Multi-cycle pipeline diagram
    - Represent the pipeline states through several clock cycles.
    - Simpler, see the whole picture.
    - Do not contain all the details of each stage.
  - Single cycle pipeline diagram
    - Show the state of the entire datapath in a single clock cycle.
    - Focus on the details but not the whole picture.

Multi-cycle pipeline diagram



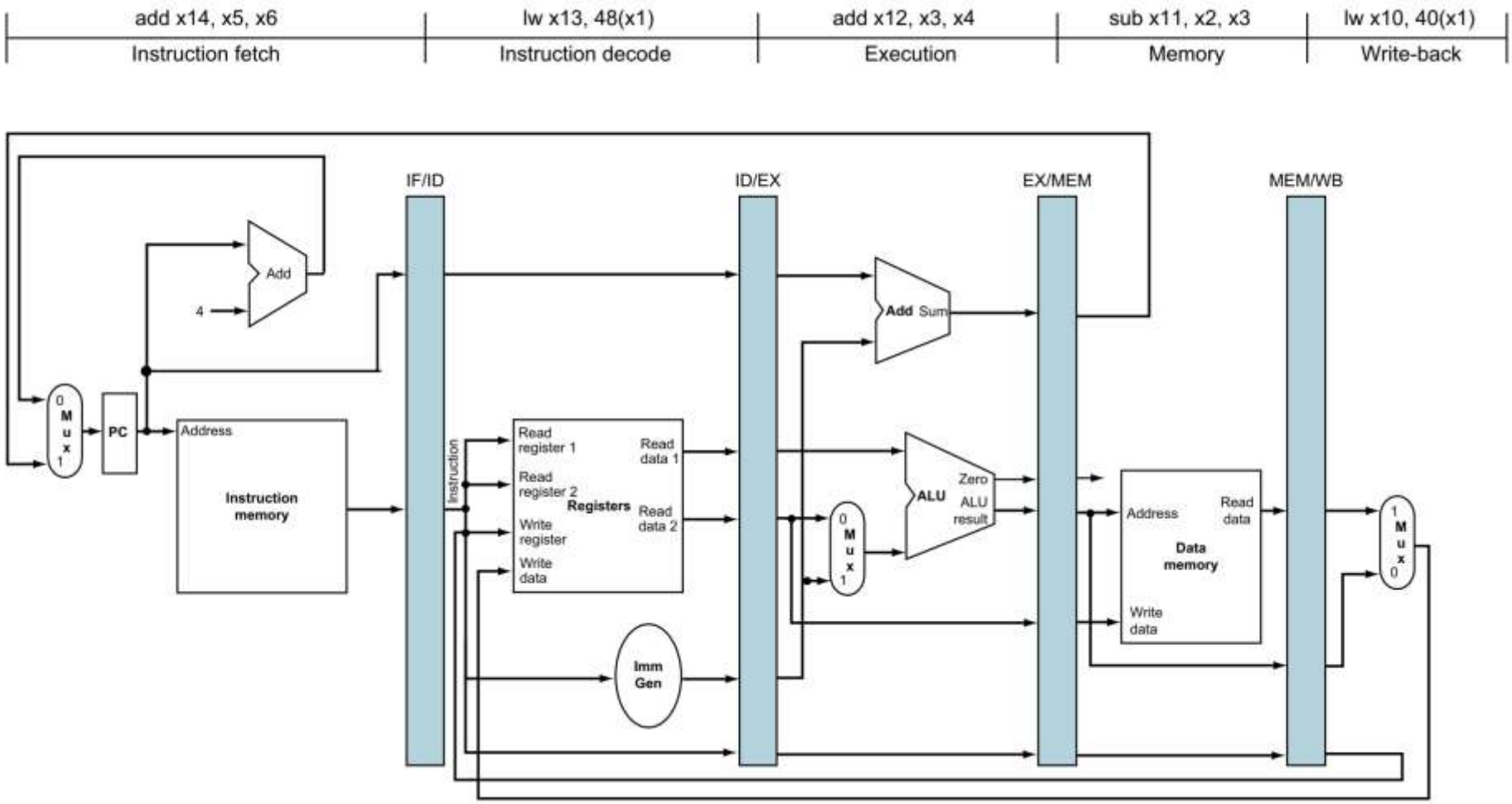
Multi-cycle pipeline diagram

- Traditional form



Single-cycle pipeline diagram

- Snapshot of pipeline status in a given cycle



Example

- See how the lw instruction is executed through 5 pipeline stages
  - The nop instructions are to flush the pipeline

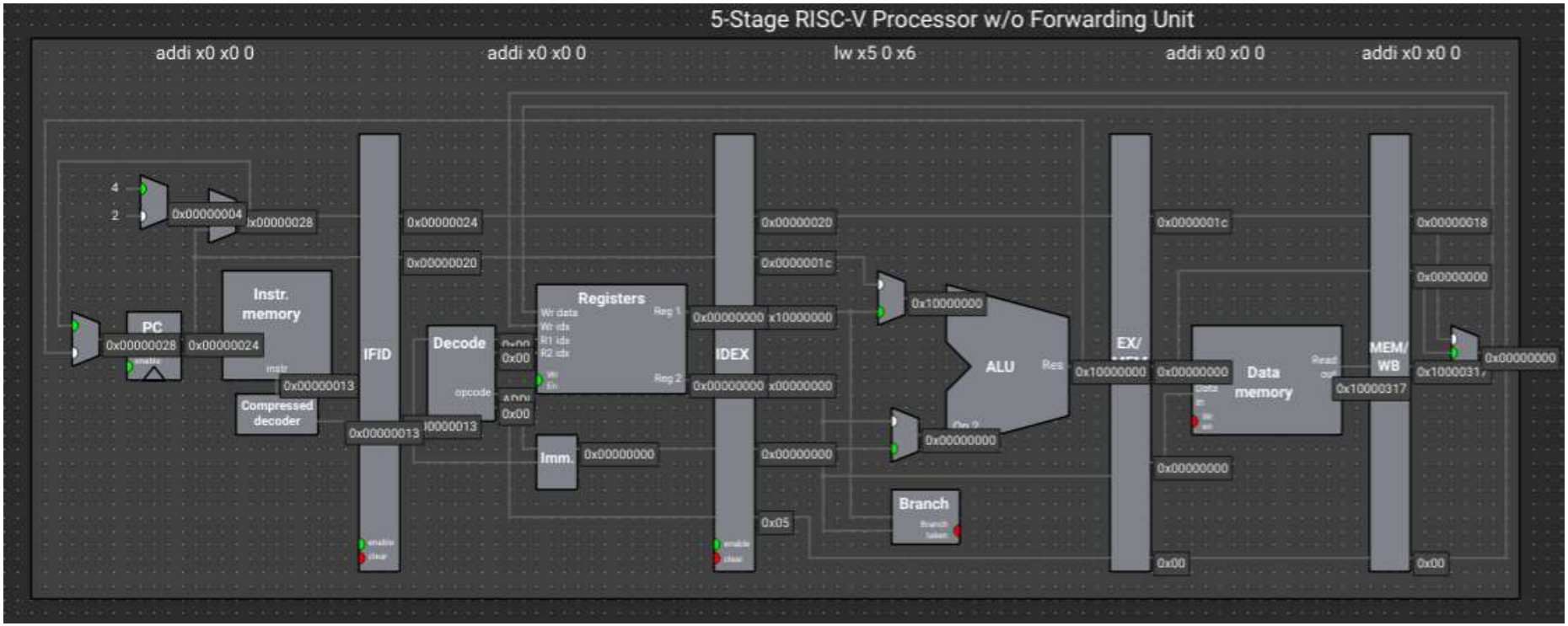
Source code — Input type: (•) Assembly ( ) C

```
1  .data
2      M:.word 100
3  .text
4      la s1, M
5      nop
6      nop
7      nop
8      lw t1, 0(s1)
9      nop
10     nop
11     nop
12     nop
13     nop
```

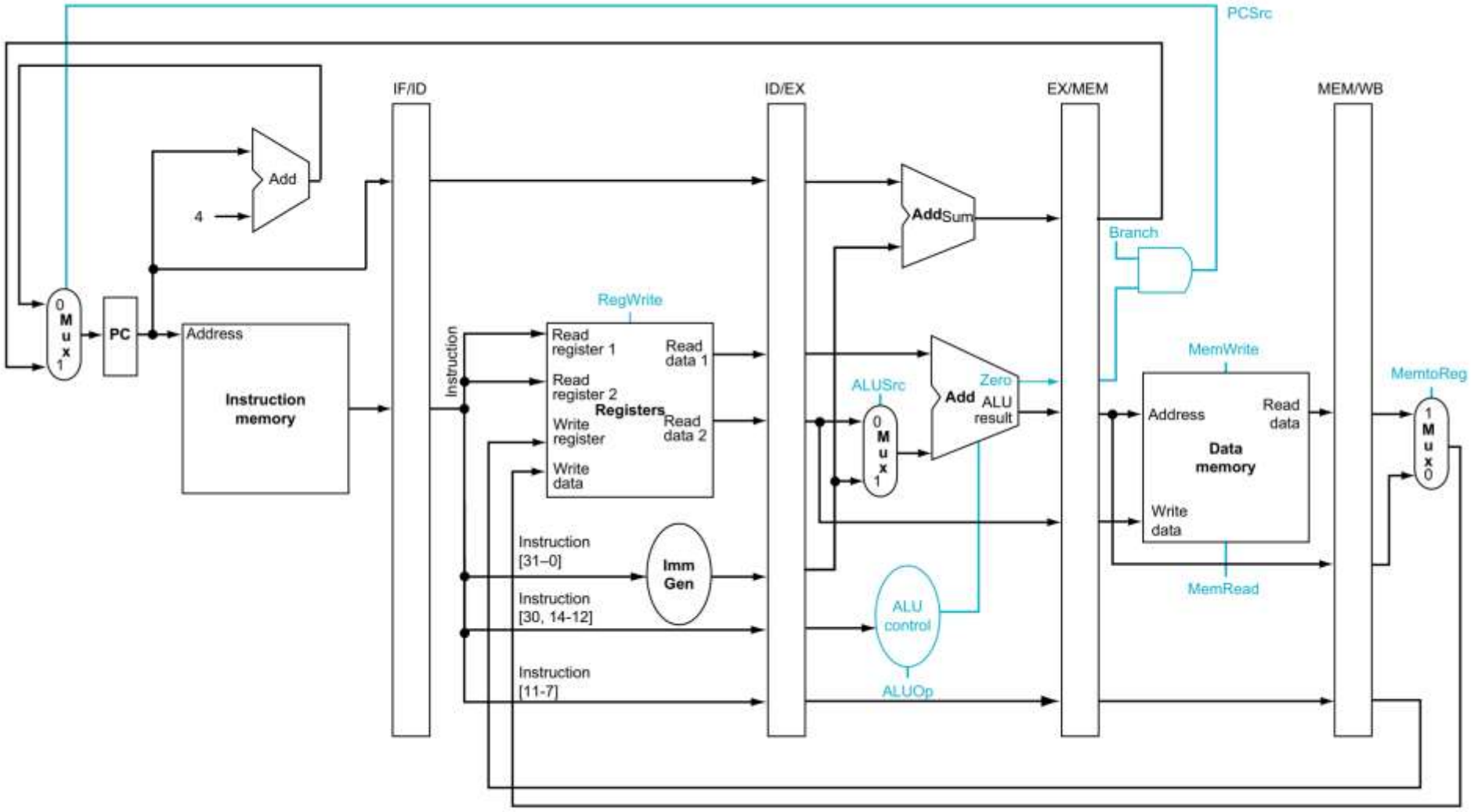
WB
MEM
EX
ID
IF

Simulating the RISC-V pipeline

- Single cycle pipeline diagram showing execution of lw

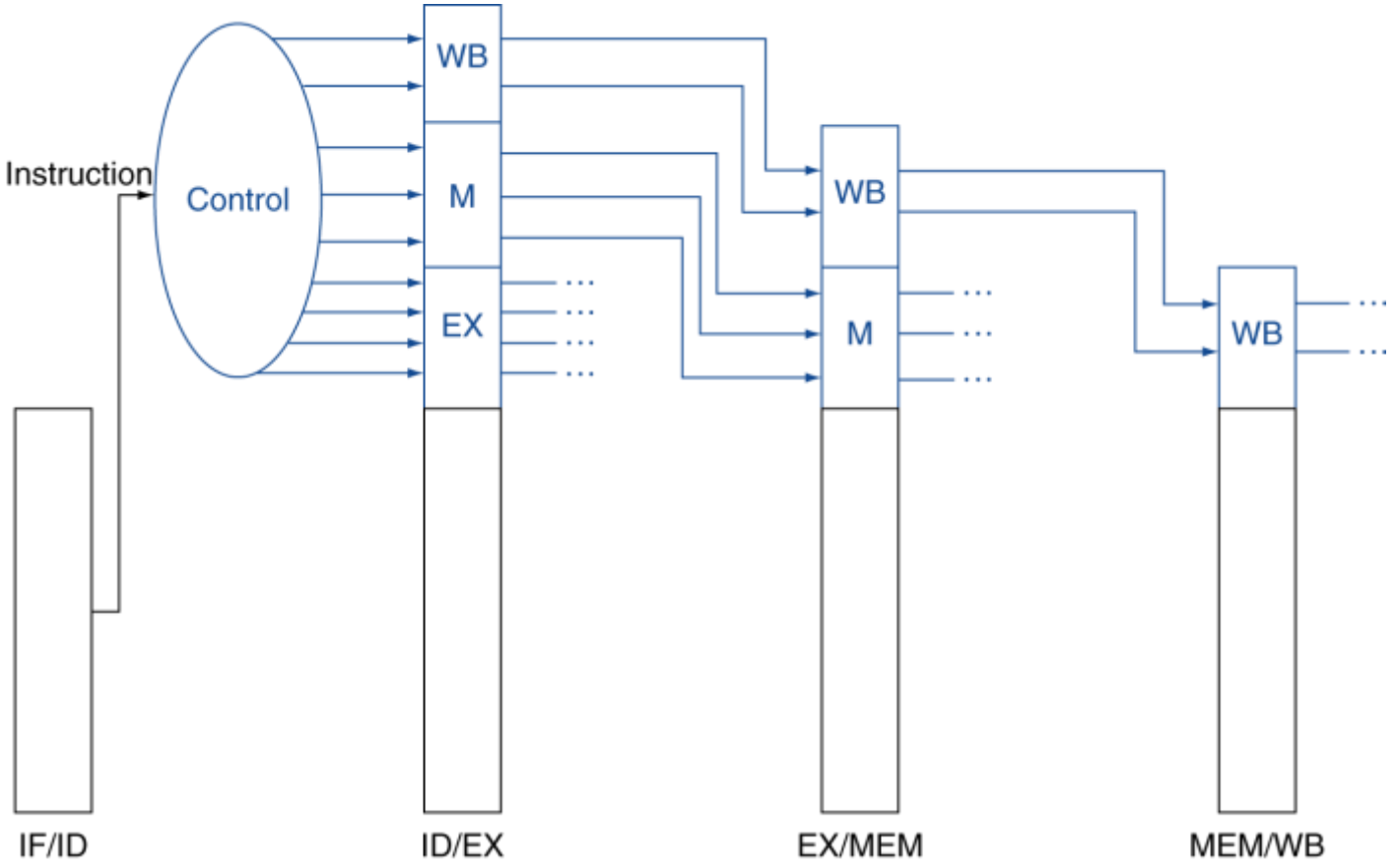


Pipelined datapath with control signal (simplified)

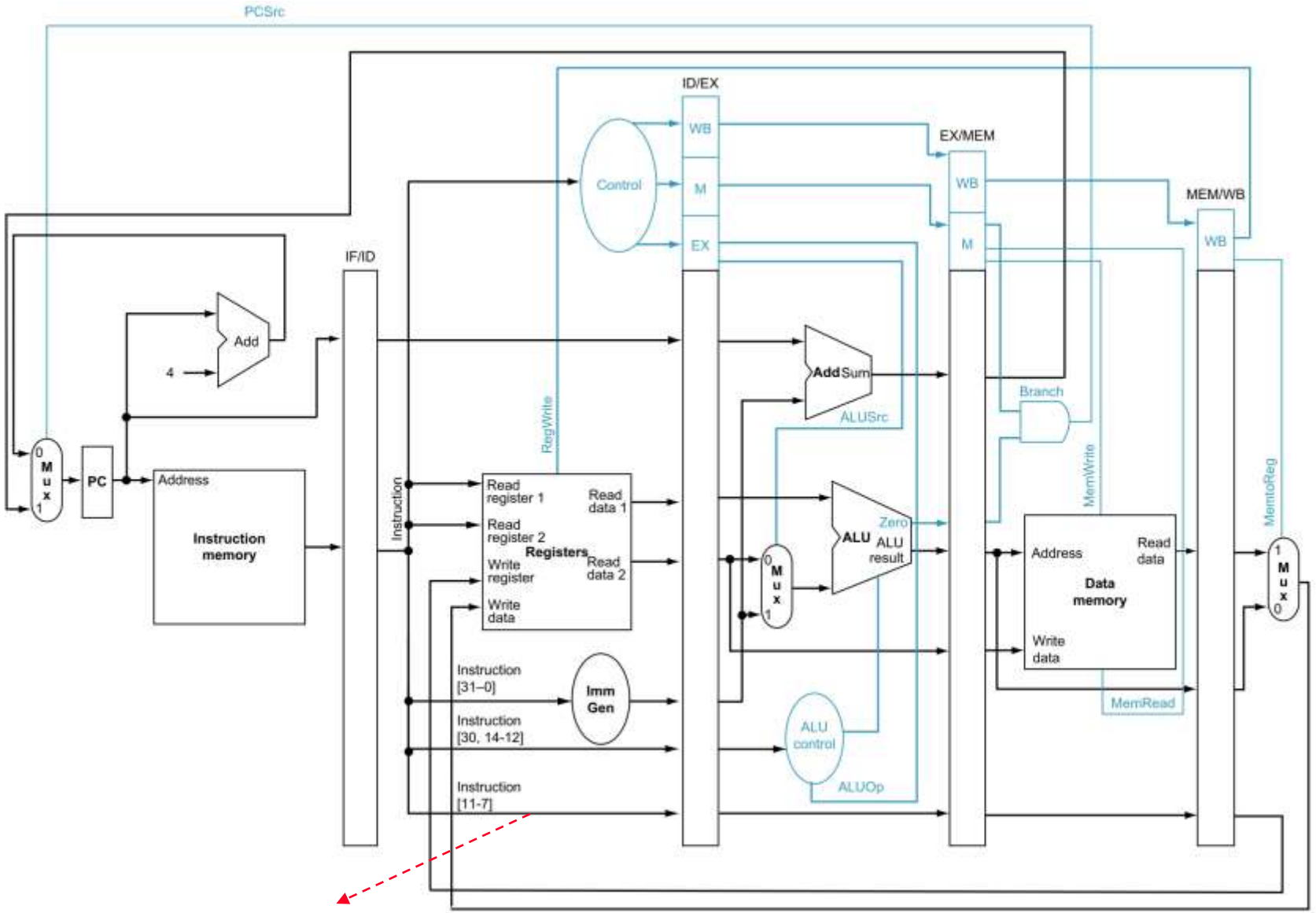


Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



Pipelined datapath with Control

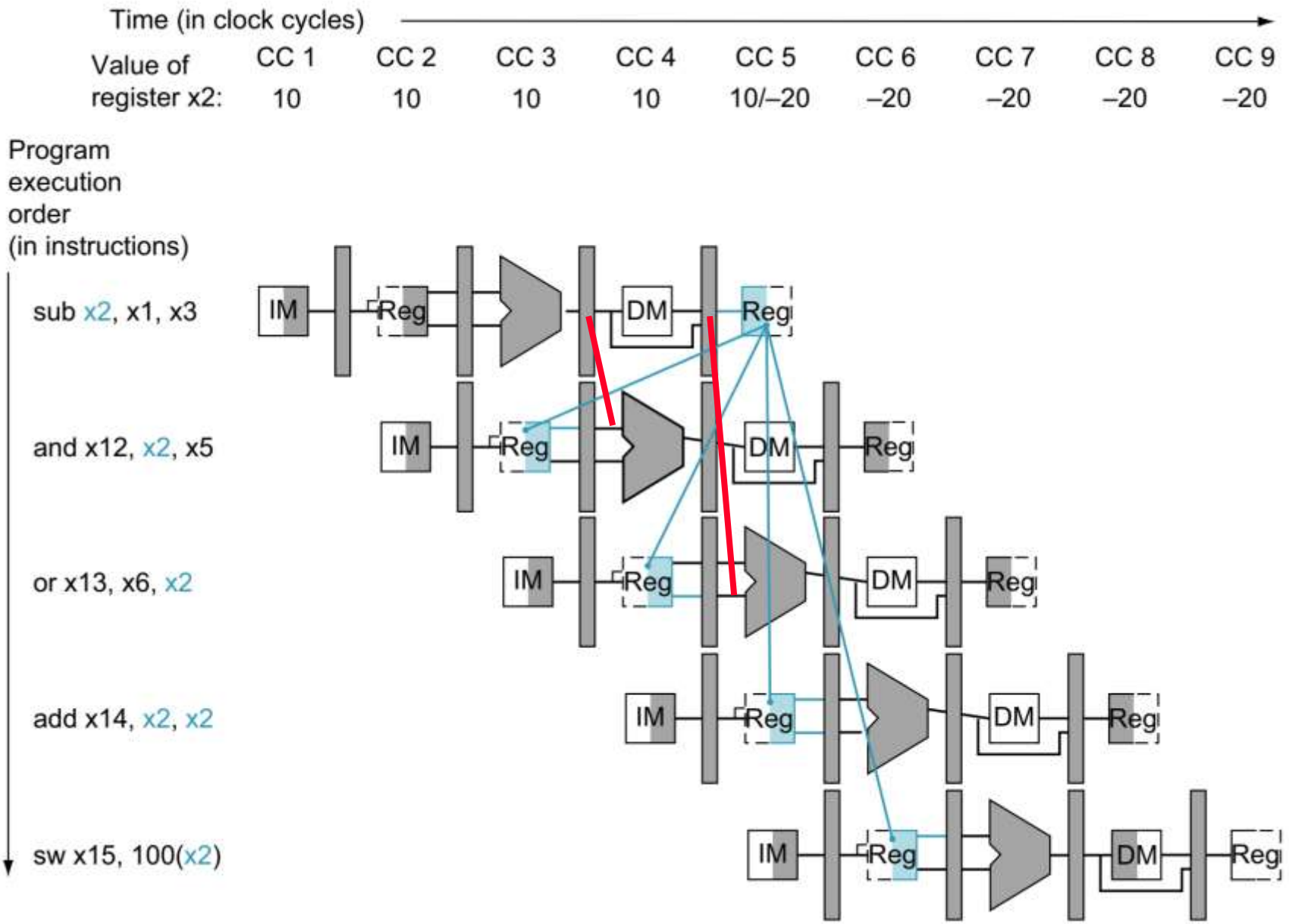


Rd is sent through the pipeline
for use in apppropriate time in WB

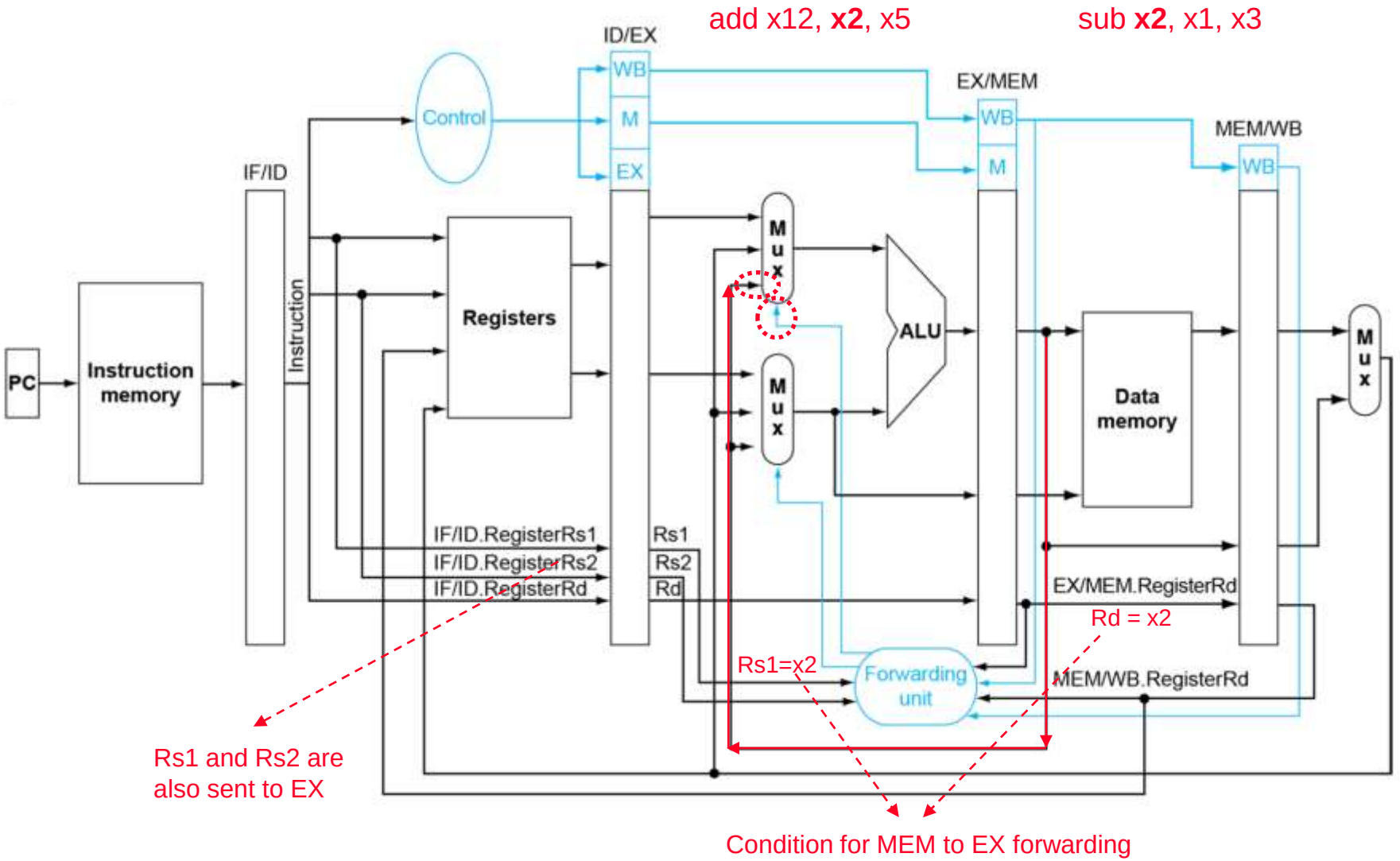
Pipeline implementation for solving hazard

- What we have built sofa is for pipeline without hazard
- Now, adding more hardware to solve the hazard
  - Read before write data hazard
  - Load – use data hazard
  - Control hazard

Solving data hazard with forwarding



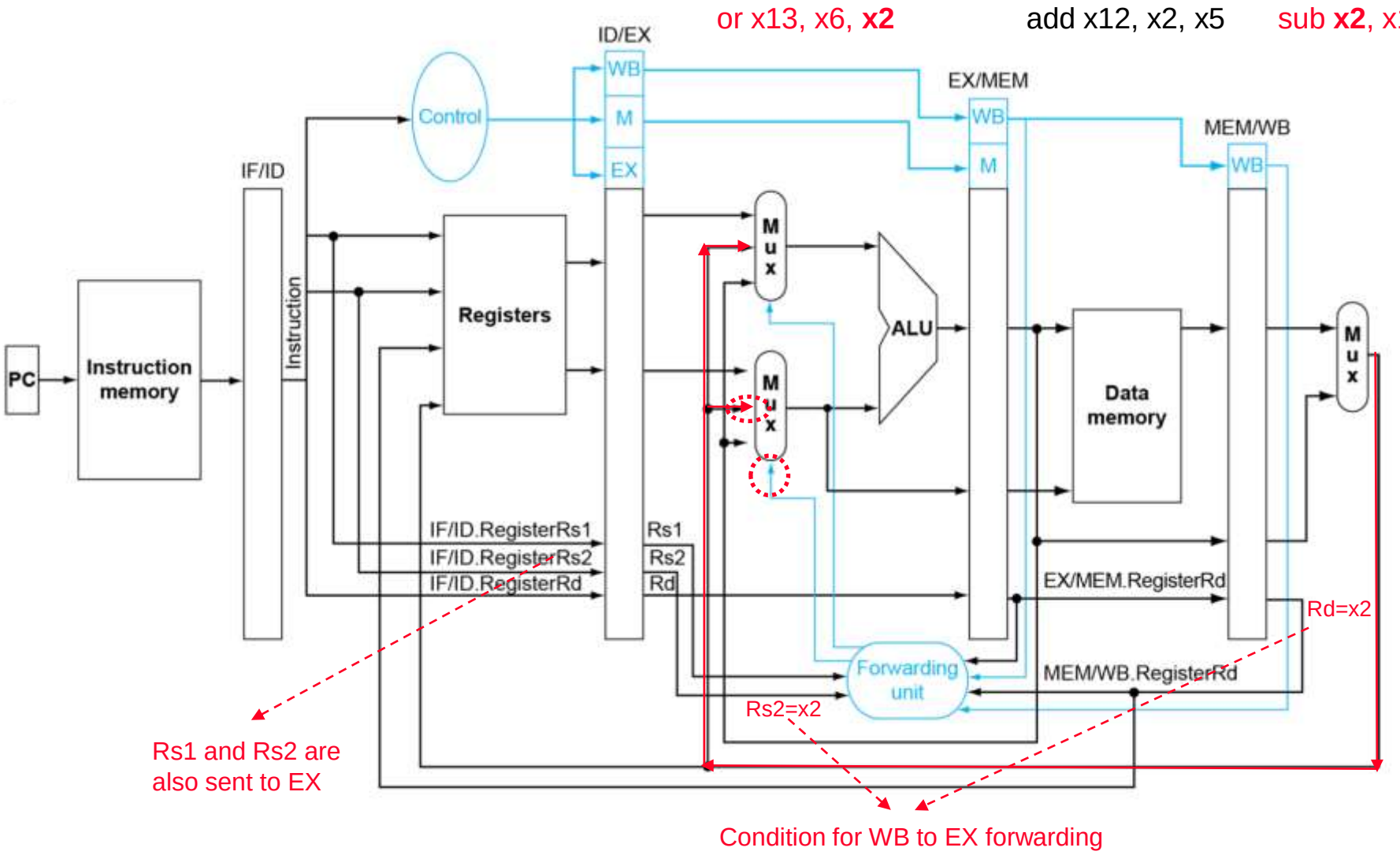
Datapath with Forwarding



The datapath modified to resolve hazards via forwarding

ALU – ALU forwarding

Datapath with Forwarding



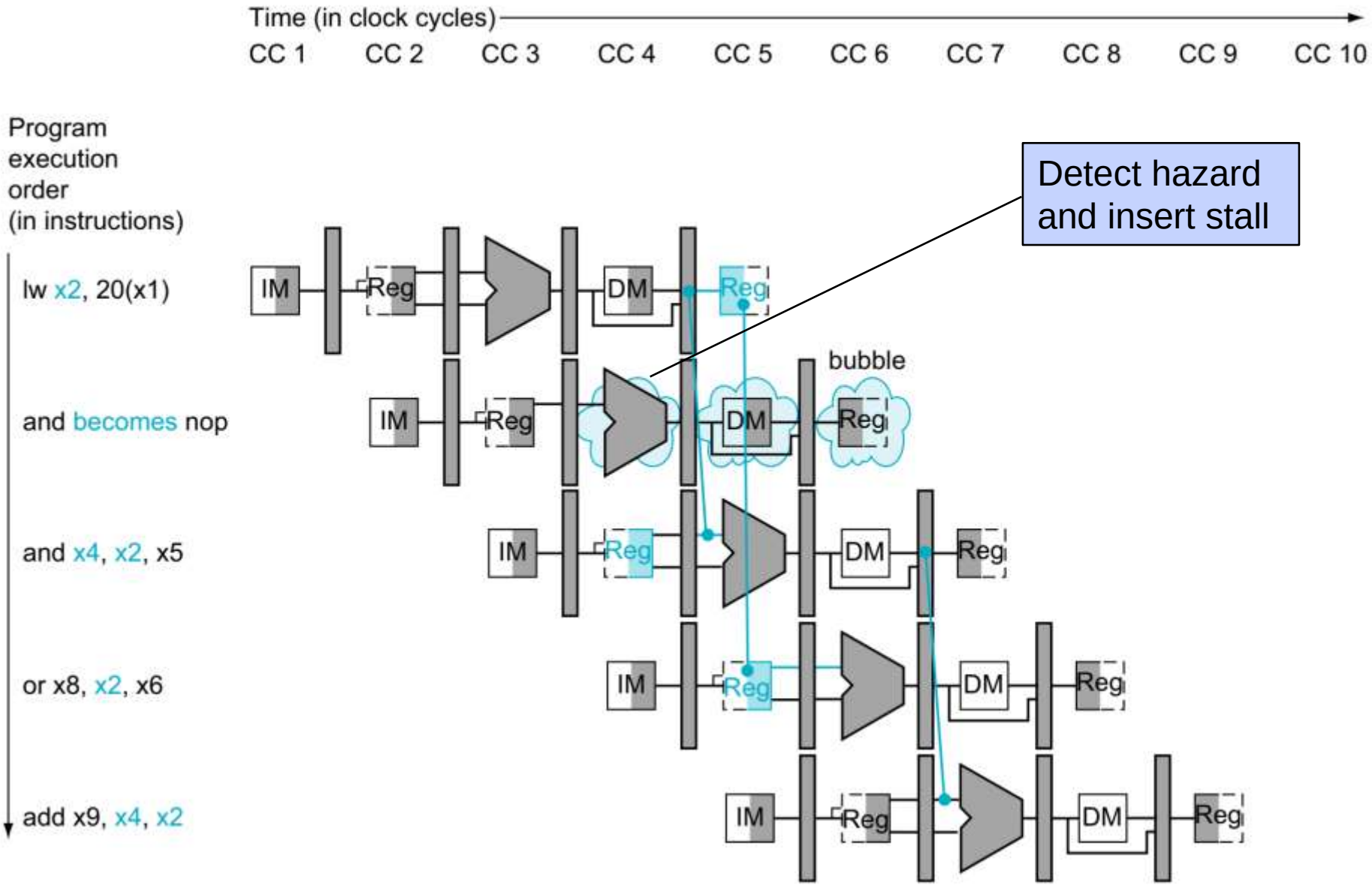
The datapath modified to resolve hazards via forwarding

Full path forwarding

Read before write data hazard

- Require forwarding unit, the forwarding could be
  - ALU – ALU forwarding
  - Full path forwarding (MEM to ALU forwarding)
  - No stall is required

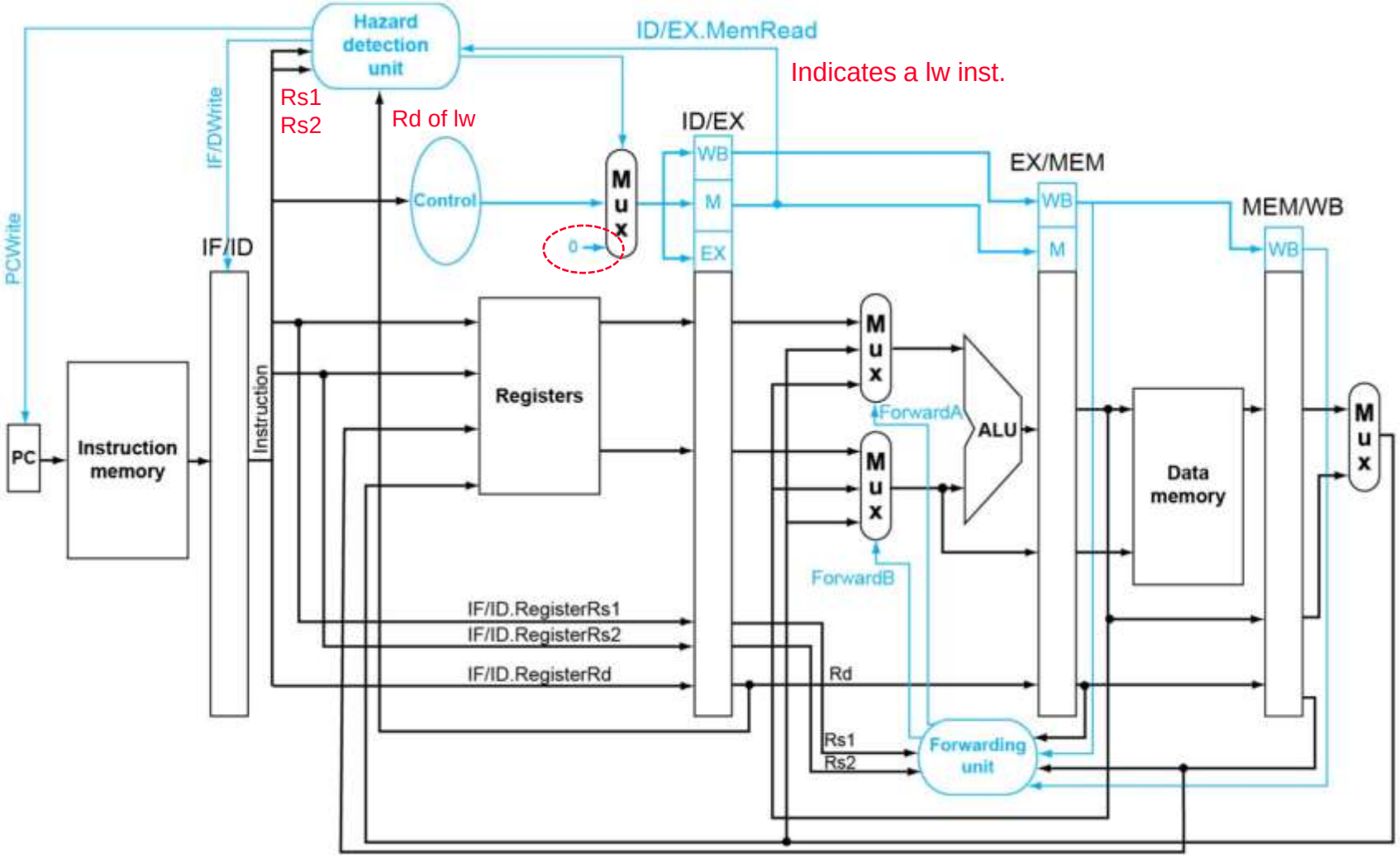
Load-Use Data Hazard



How to stall the pipeline

- Force control values in ID/EX 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 lw
    - Can subsequently forward to EX stage

Datapath with hazard detection



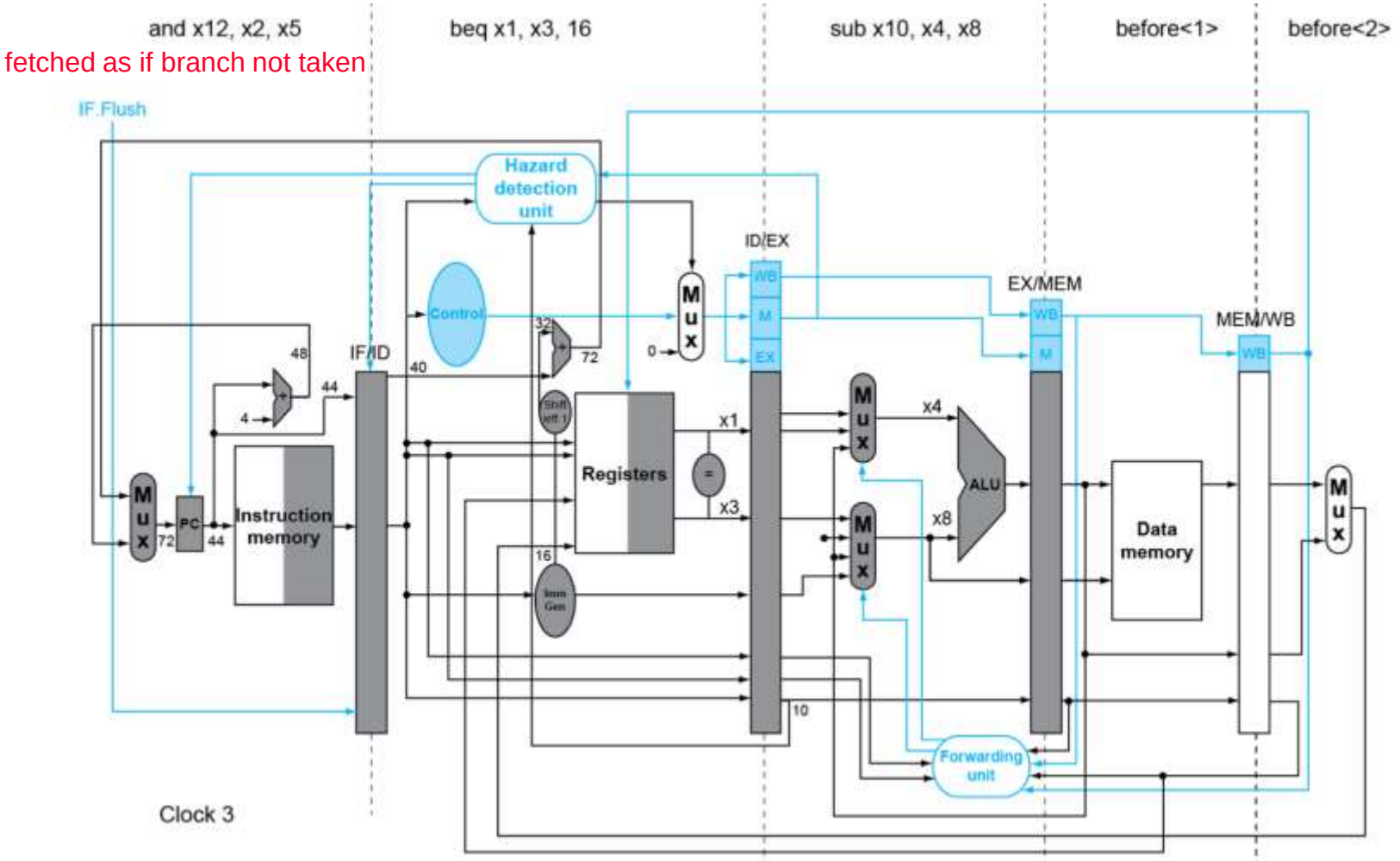
Note: the 0 input, PCWrite, IF/ID write are to stall the pipeline

Load use data hazard

- Require detection unit to detect load-use hazard
- Must have one stall
- Require forwarding unit to forward from MEM stage to ALU stage

Control hazard: early branch calculation +predict not-taken

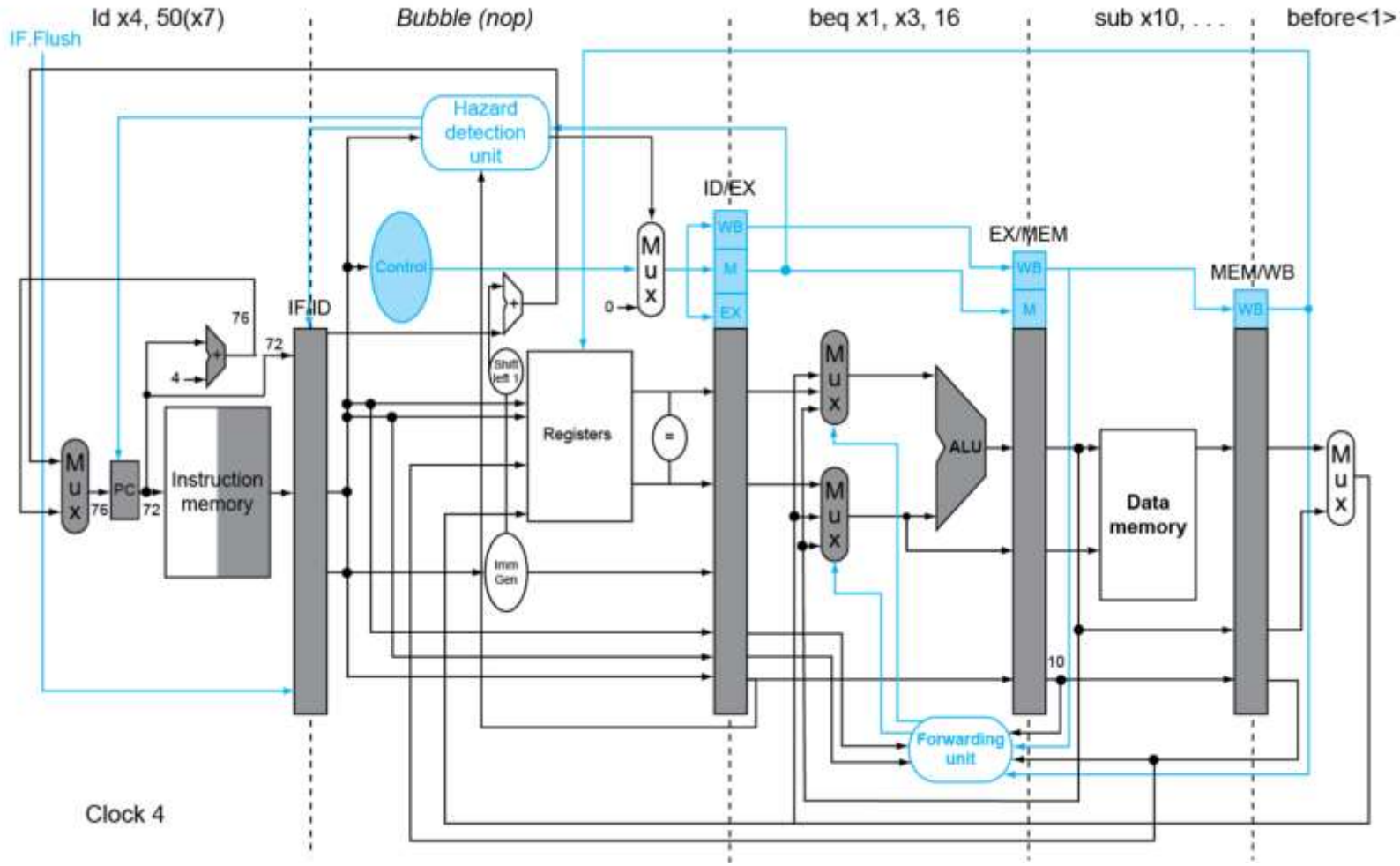
- In case the branch is actually taken



Control hazard: early branch calculation +predict not-taken

- In case the branch is actually taken

and x12, x2, x5 replaced by bubble



Summary

- ISA influences design of datapath and control.
- All modern-day processors use pipelining.
- Pipelining doesn't help latency of a single instruction, it helps throughput of entire workload.
- Potential speedup: a CPI of 1 and a fast CC.
- Must detect and resolve hazards.
  - Structural, data, control.
  - Stalling negatively affects CPI (makes CPI worse than the ideal of 1).

Summary

- Design of datapath
  - Single cycle non-pipelined CPU
  - Multi-cycle pipelined CPU
- Solving hazards
  - Structural hazards: I/D caches and separate register read/write
  - Data hazards:
    - Hazard detection present/not present: stalls are added by software or hardware.
    - Forwarding: no forwarding/partial forwarding/full forwarding.
  - Control hazard:
    - Early target calculation or not.
    - Delayed branch, static/dynamic prediction.