Summer School

Exploring the RISC-V Processor Architecture

Presented By:

Engr. Naureen Shaukat

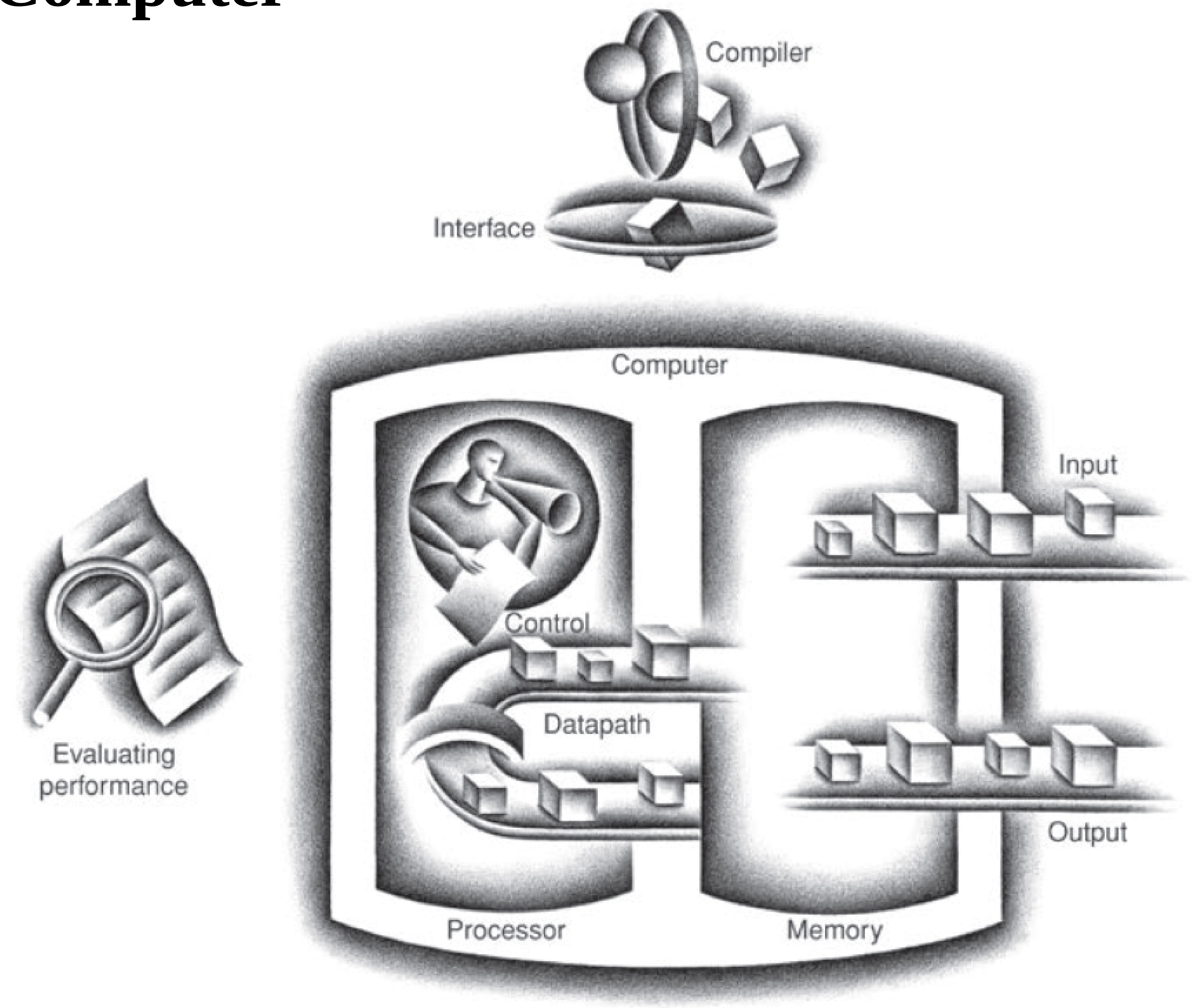
Lecturer - EE Namal University Mianwali

Agenda

- Computer Abstractions and Technology
- RISC V Instructions in the Computer
- The Processor
    - Single Cycle Processor (Datapath with Control Unit)
- Pipelined Datapath and Control
- Pipelining Hazards with Solutions (Data Hazards, Control Hazards)
- Memory Hierarchy
    - Basics of Caches

MODELSIM Hands-on Experience

Organization of a Computer



Traditional Classes of Computer Applications

Personal Computers (PCs)	Servers	Embedded Computers

Personal Computers (PCs)

- Good Performance
- Single User
- Low Cost
- Execute Third Party Software

Servers

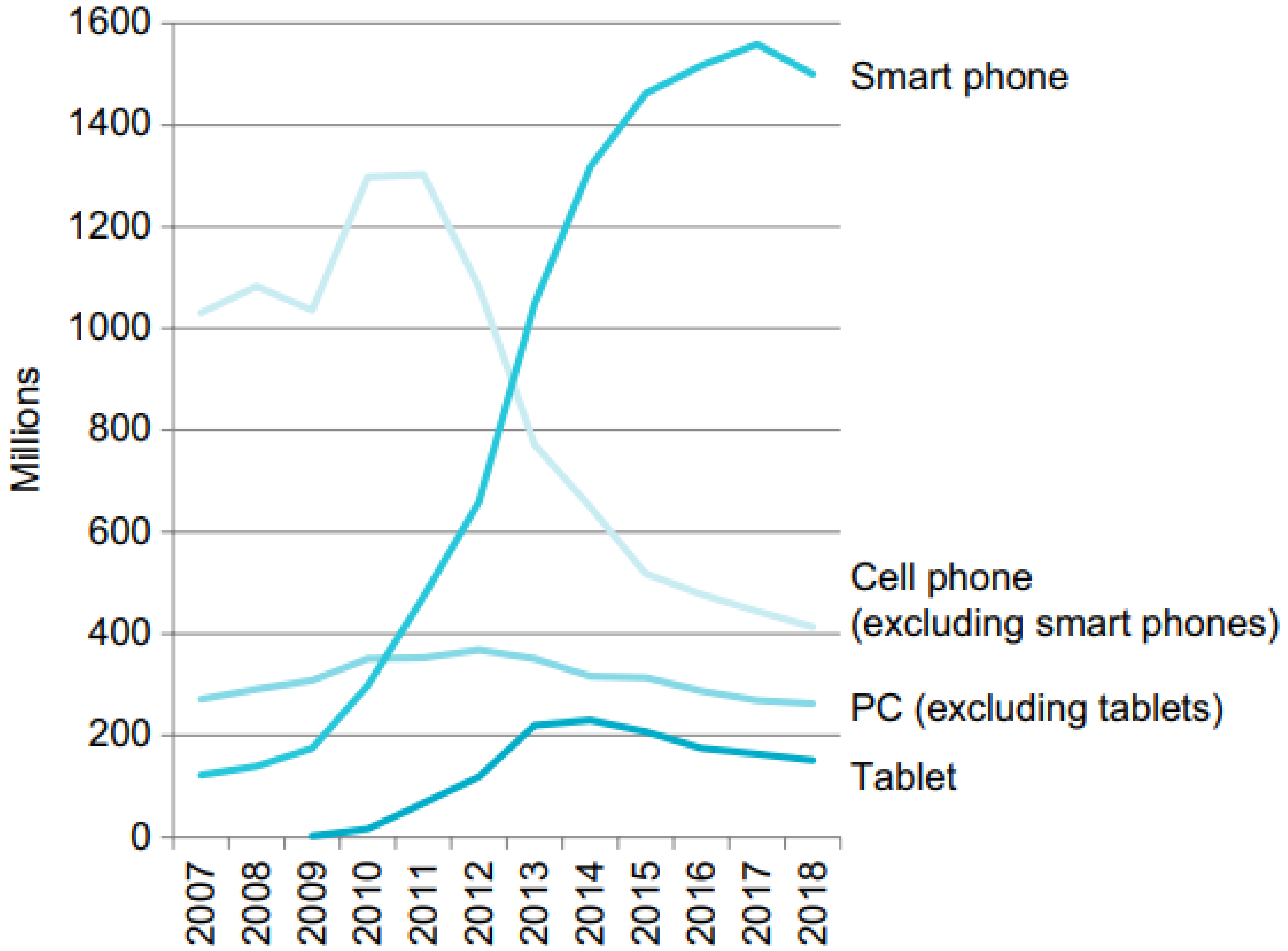
- Run large programs
- Multiple Users
- Can accessed via a network
- Wildest range in cost and capability

Embedded Computers

- A computer inside another device
- Running one predetermined application or software

Post PC Era

Rapid growth over time of tablets and smart phones versus that of PCs and traditional cell phones.

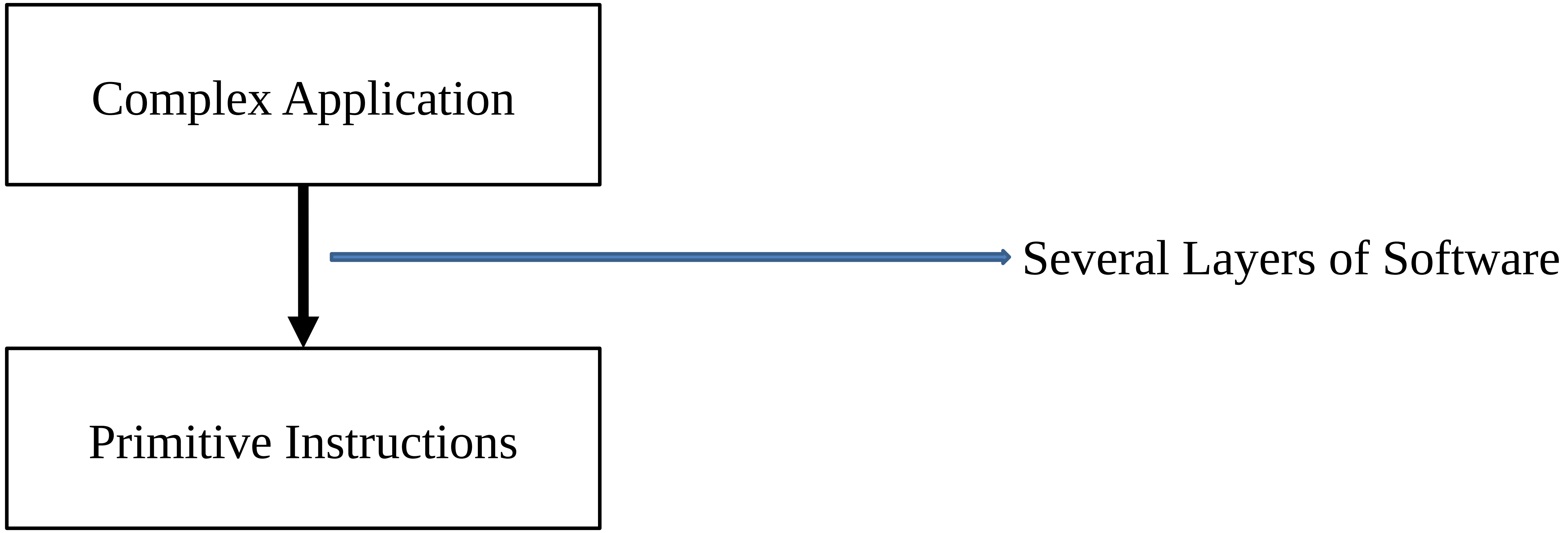


Understanding Program Performance

- Algorithms
- Software systems used to create and translate the program into machine instructions
- Effectiveness of the computer in executing those instructions
- I/O system

Abstraction

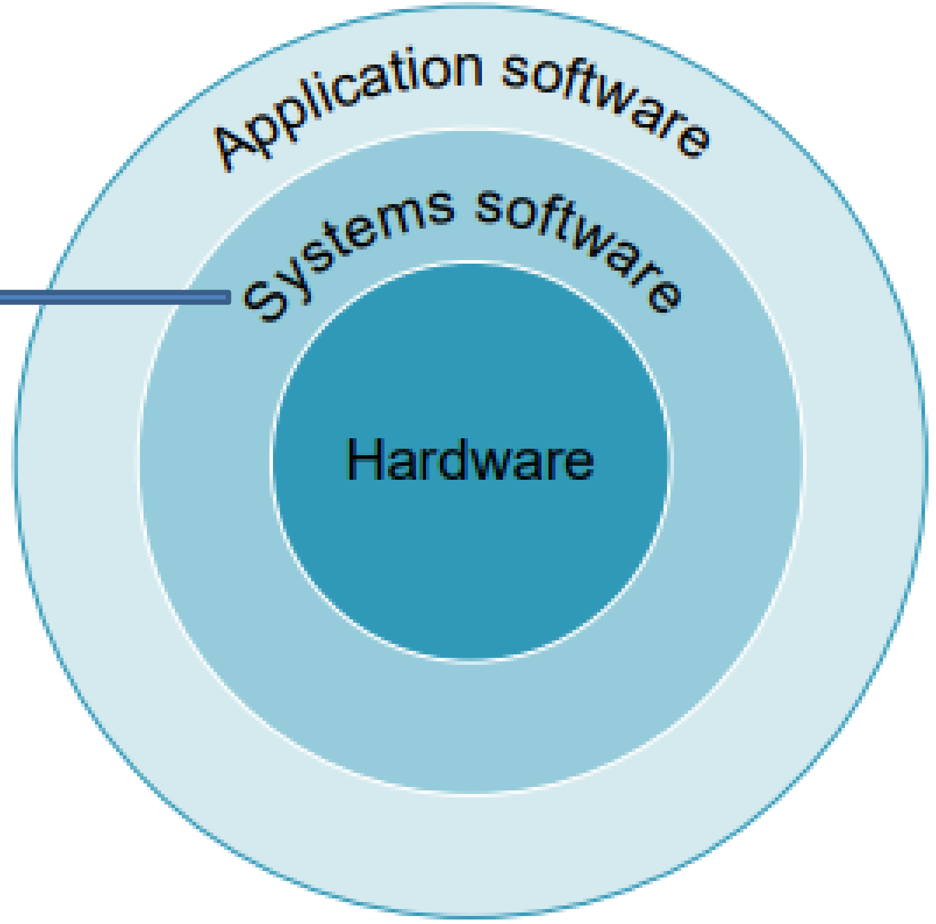
- To go from a complex application to the primitive instructions involves several layers of software that interpret or translate high-level operations into simple computer instructions, an example of the great idea of abstraction.



- Lower level details are hidden to offer a simpler model at higher level

Hierarchical Layers – Hardware and Software

Operating System, Compiler



Operating System

1. Interface between user’s program and hardware (supervisory functions)
2. Handling basic Input and Output Operations
3. Allocating Storage and Memory

Examples: Linux, Android, Windows

Power of Abstraction

High-level language program (in C)

```
swap(size_t v[], size_t k)
{
   size_t temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

↓

Compiler

↓

Assembly language program (for RISC-V)

```
swap:
      slli x6, x11, 3
      add  x6, x10, x6
      lw   x5, 0(x6)
      lw   x7, 4(x6)
      sw   x7, 0(x6)
      sw   x5, 4(x6)
      jalr x0, 0(x1)
```

↓

Assembler

↓

Binary machine language program (for RISC-V)

```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000110011001010000011
00000000100000110011001110000011
00000000011100110011000000100011
00000000010100110011010000100011
00000000000000001000000001100111
```

Interface between Hardware and Software

- Vocabulary
- Instructions
- **Instruction set Architecture**

Performance

- When we say one computer has better performance than another, what do we mean?
- If you were running **a program on two different desktop computers**, you'd say that the faster one is the desktop computer that gets the job done first.
- If you were running **a datacenter** that had several servers running jobs submitted by many users, you'd say that the faster computer was the one that completed the most jobs during a day.
- As **an individual computer user**, you are interested in reducing **response time**.
- **Response time**
- **Throughput**

Performance

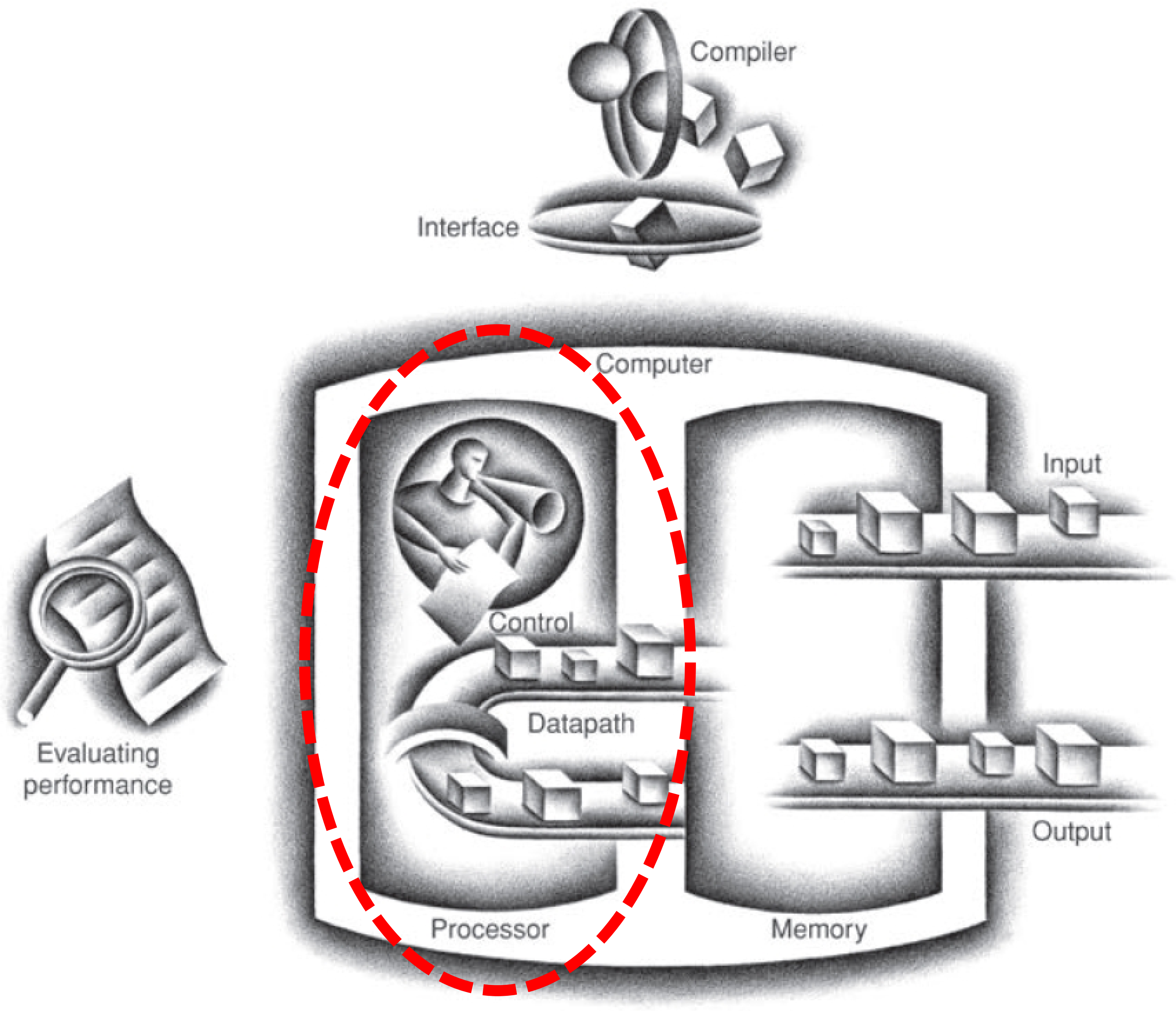
$$Performance = \frac{1}{Execution}$$

$$\begin{matrix}\text{CPU execution time} \\ \text{for a program}\end{matrix} = \begin{matrix}\text{CPU clock cycles} \\ \text{for a program}\end{matrix} \times \text{Clock cycle time}$$

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

- This formula makes it clear that the hardware designer can improve performance by reducing the number of clock cycles required for a program or the length of the clock cycle

Organization of a Computer



Core RISC V Instruction Set	
Memory Reference Instruction :	load word (lw), store word (sw)
Arithmetic Logical Instructions	add, sub, and, or
Conditional branch Instruction	branch if equal (beq)

Compiling a while Loop in C

```
while (save[i] == k)
      i += 1;
```

i	x22
k	x24
Base address of array *save*	x25

: 00000000 00000000 00000000 00001001
For 4 byte addressing we have to multiply i with 4. i*4 is equivalent to shift left by 2
For example: i=9, 9*4=36
If we shift binary 1001 by 2, the binary will be 100100 that is 36.

RISC V Assembly Code

```
Loop: slli x10,x22,2

add x10, x10, x25
```

Base address + x10

```
lw x9, 0(x10)

bne x9,x24, Exit
```

If condition (save[i]==k) is not true, control goes to Exit

```
addi x22, x22, 1
```

else i=i+1

```
beq x0, x0, Loop
```

Register x0 always contain 0

```
Exit:
```

RISC V Instruction Format

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (add)	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
sub (sub)	0100000	00011	00010	000	00001	0110011	sub x1, x2, x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
addi (add immediate)	001111101000		00010	000	00001	0010011	addi x1, x2, 1000
lw (load word)	001111101000		00010	010	00001	0000011	lw x1, 1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
sw (store word)	0011111	00001	00010	010	01000	0100011	sw x1, 1000(x2)

C Code

```
while (save[i] == k)
      i += 1;
```

Assembly Code

```
Loop:slli x10, x22, 2     // Temp reg x10 = i * 4
     add  x10, x10, x25   // x10 = address of save[i]
     lw   x9, 0(x10)      // Temp reg x9 = save[i]
     bne  x9, x24, Exit   // go to Exit if save[i] != k
     addi x22, x22, 1     // i = i + 1
     beq  x0, x0, Loop    // go to Loop
Exit:
```

Binary Code

Address	Instruction					
80000	0000000	00010	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

```
add   x10, x10, x25
```

Instruction Format

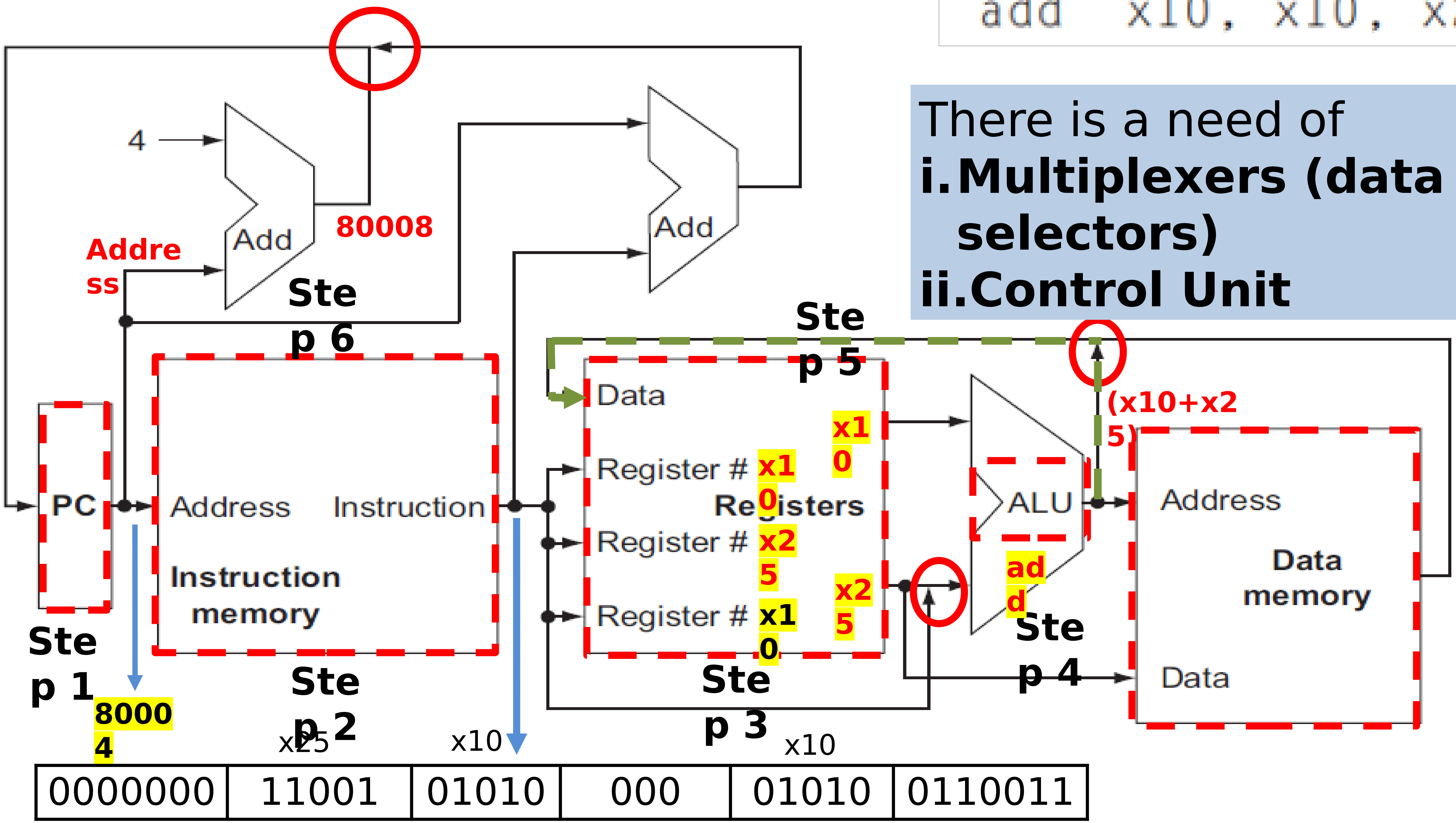
Address	funct7	rs2	rs1	funct3	rd	opcode
80004	0000000	11001	01010	000	01010	0110011
		x25	x10		x10	

High Level View of RISC V Implementation

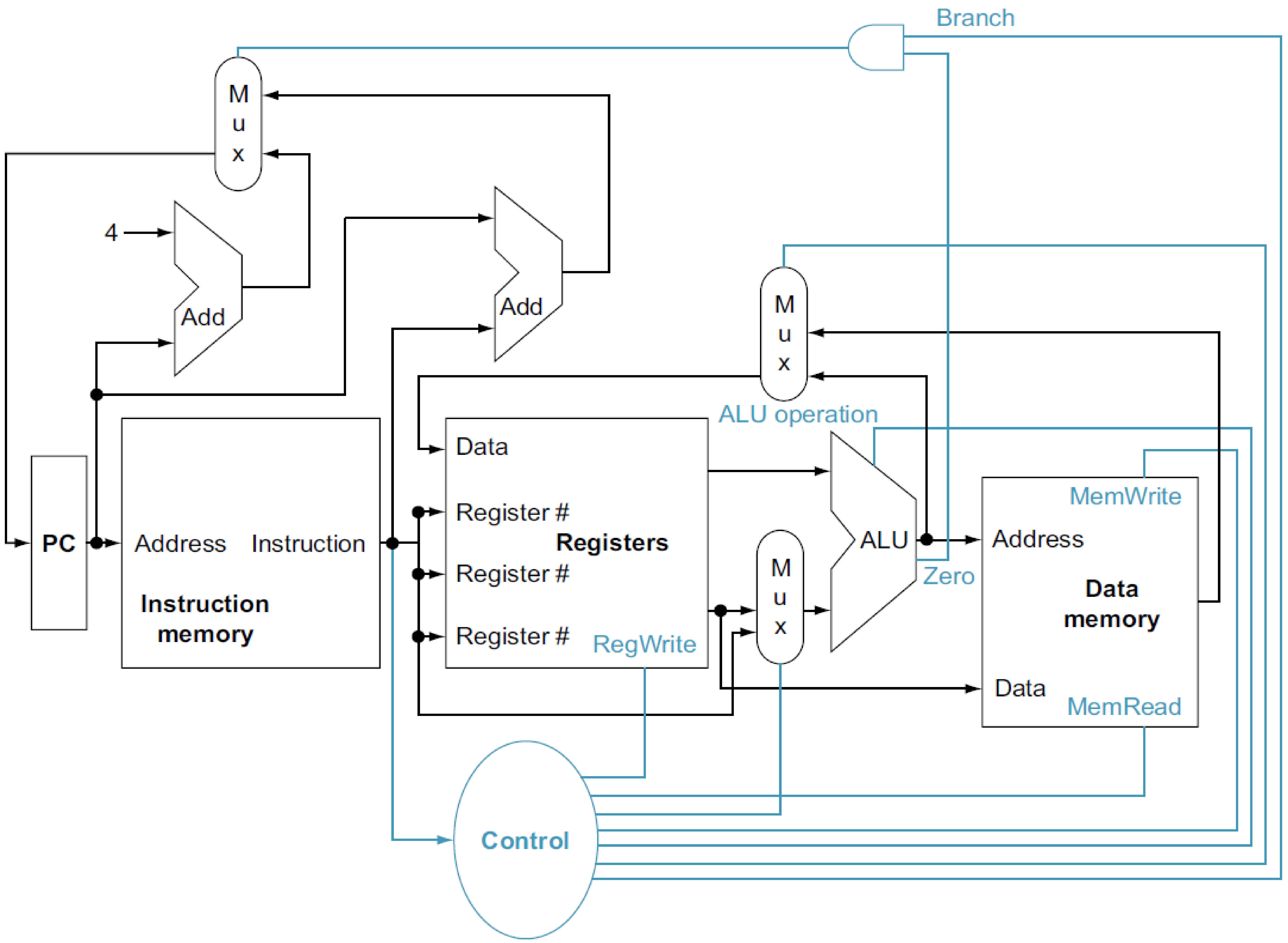
```
add  x10, x10, x25
```

There is a need of

i. **Multiplexers (data selectors)**
ii. **Control Unit**

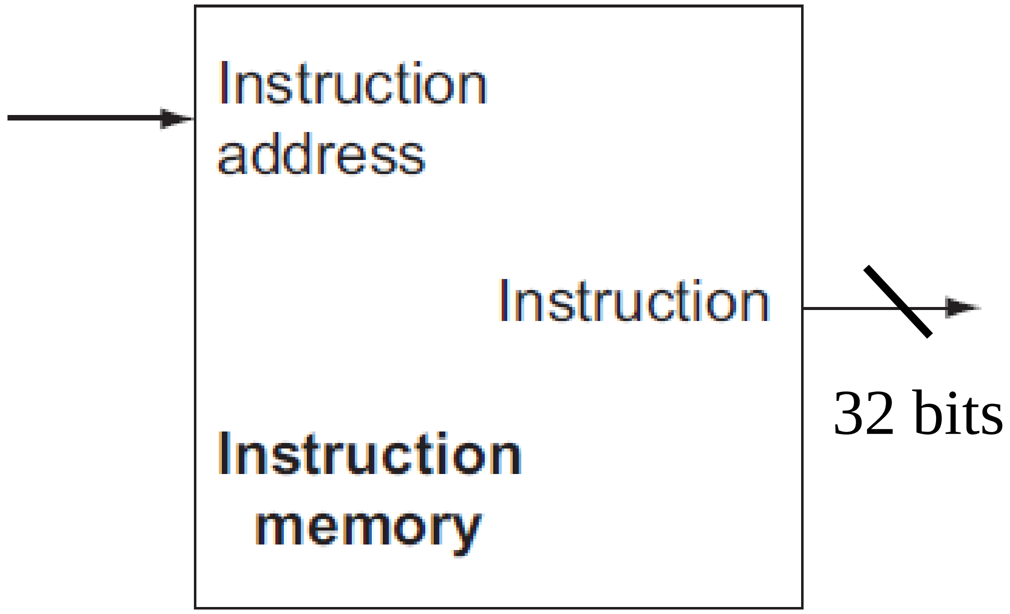


High Level View of RISC V Implementation

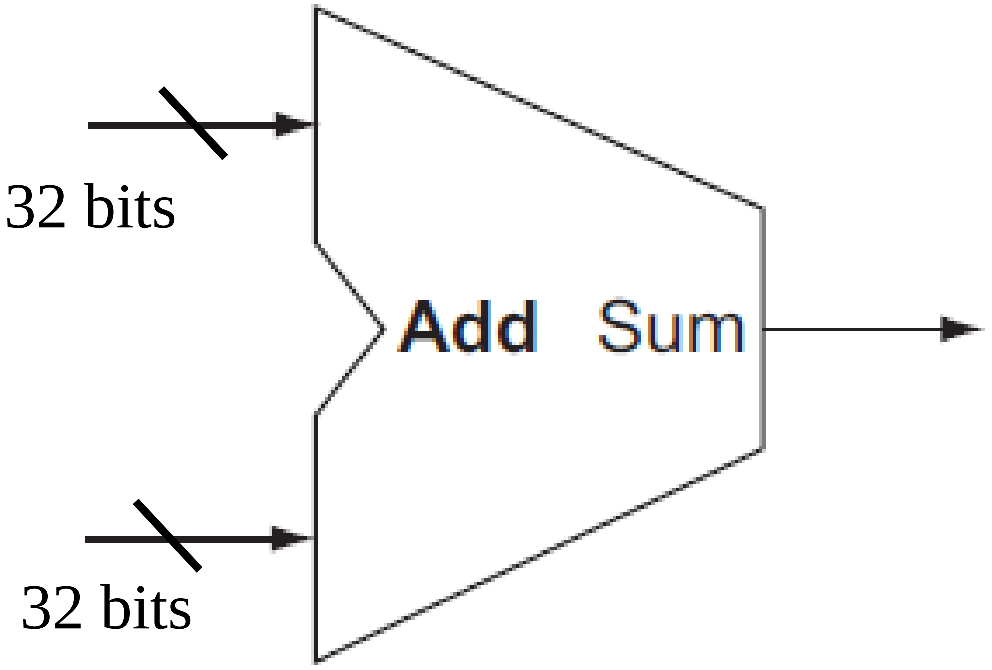


Building a Datapath - Elements

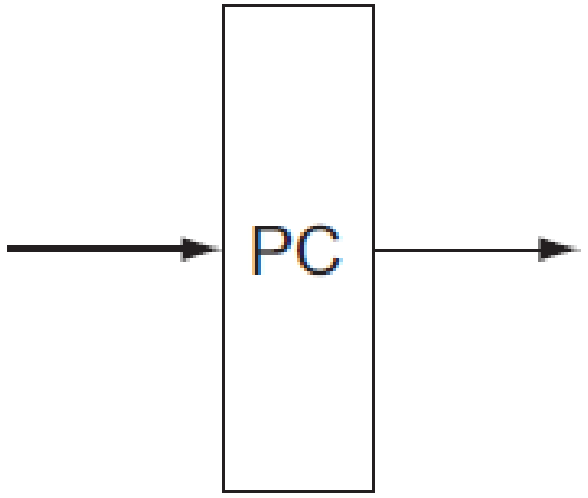
a) **Instruction Memory**



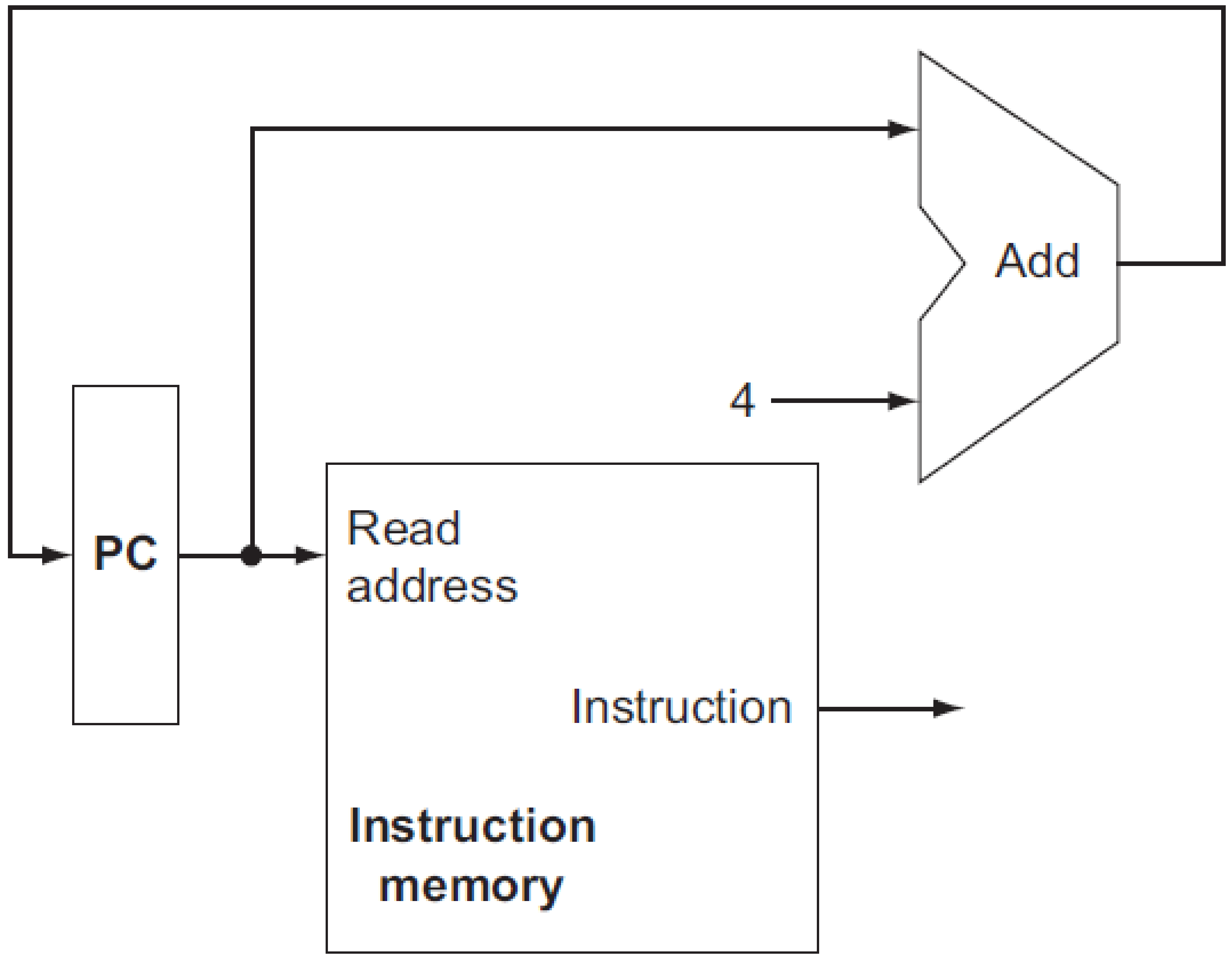
c) **Adder**



b) **Program Counter (PC) –** 32 bit register

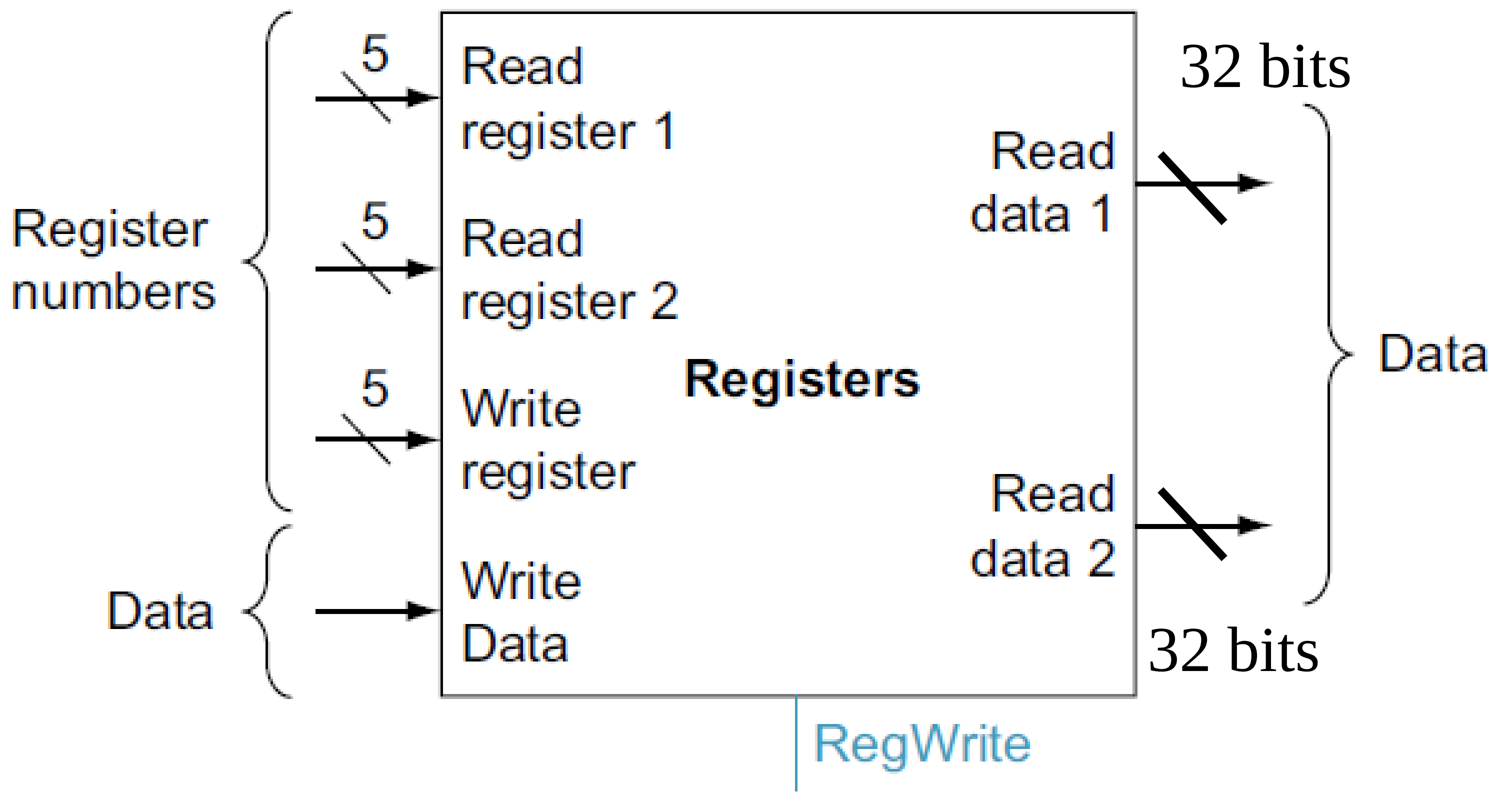


Portion of the datapath – Fetching instructions and incrementing program counter

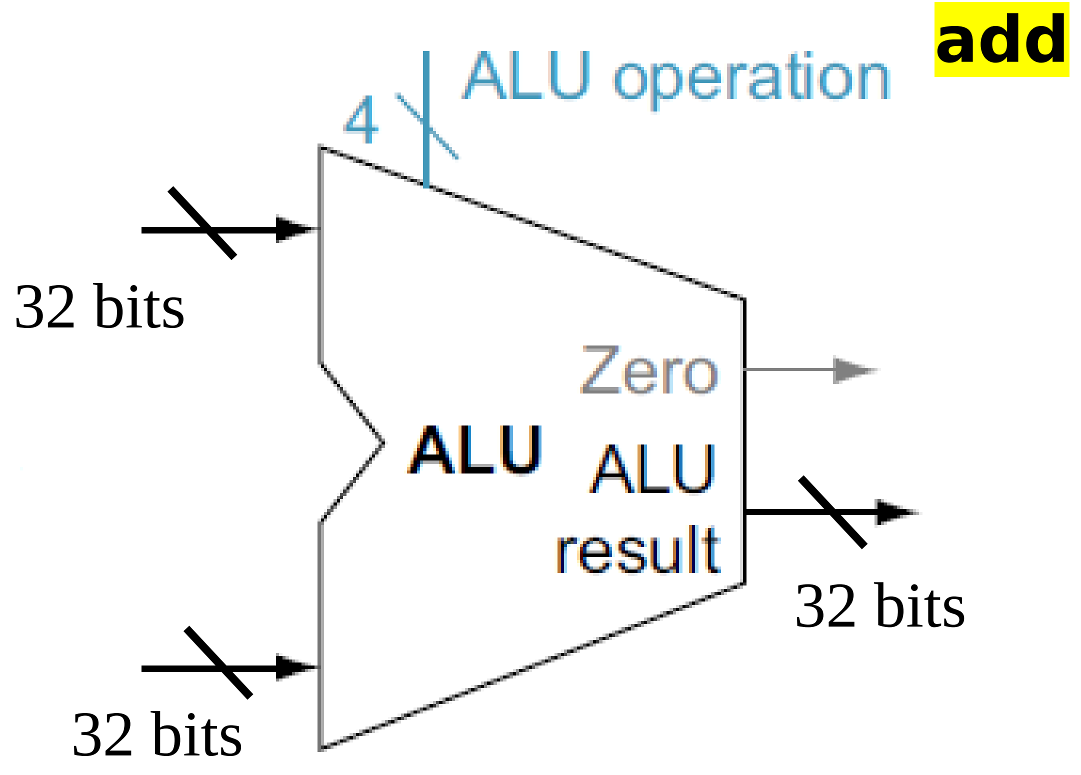


Building a Datapath - Elements

d) Registers (Register File)



e) ALU



rd **rs1** **rs2**

```
add x2,x3,x4
```

funct7	rs2	rs1	funct3	rd	opcode
0000000	00100	00011	000	00010	0110011

<u>Memory Reference Instructions</u>: **load** word (lw) and **store** word (sw)

```
lw x1, offset(x2)
```

Memory address = [x2+offset]

12 bit signed offset

```
sw x1, offset(x2)
```

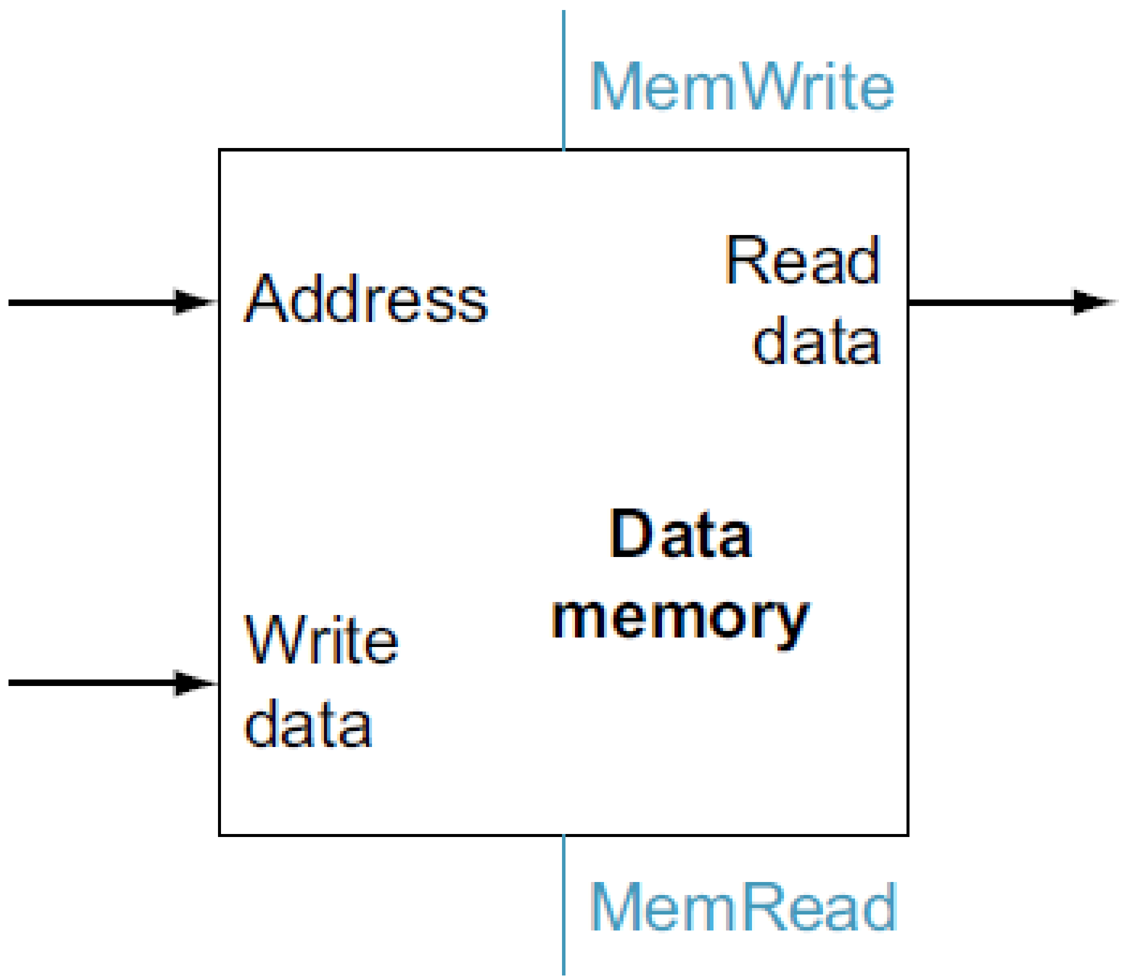
We will need a

i. Unit to **sign extend** the 12 bit offset field in the instruction to a 32-bit signed value

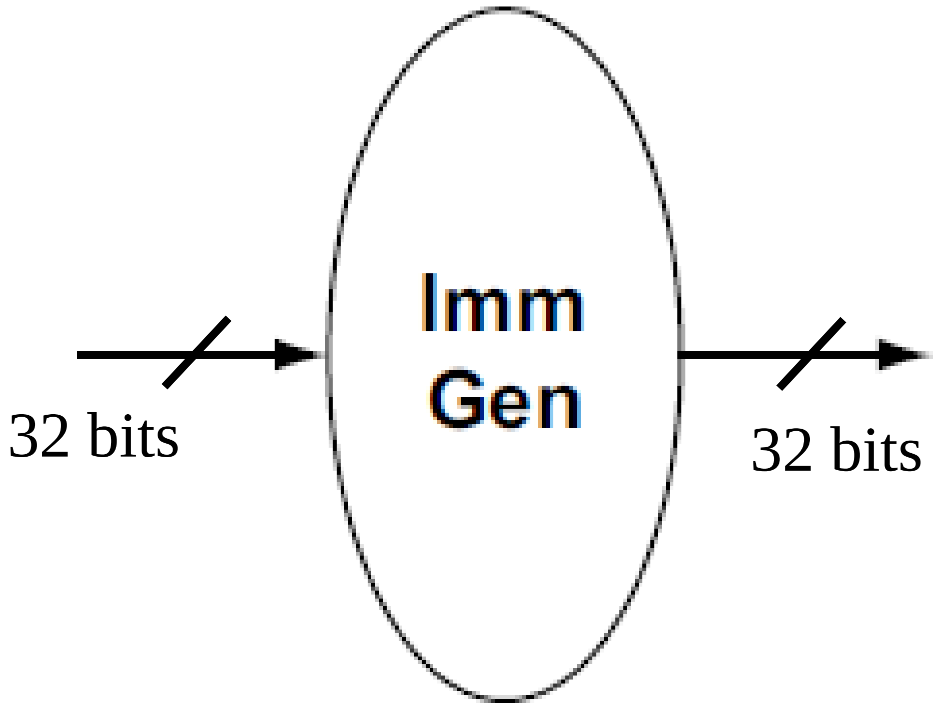
ii. A **data memory unit** to read from or write too.

Building a Datapath - Elements

f) Data Memory Unit



g) Immediate Generation Unit

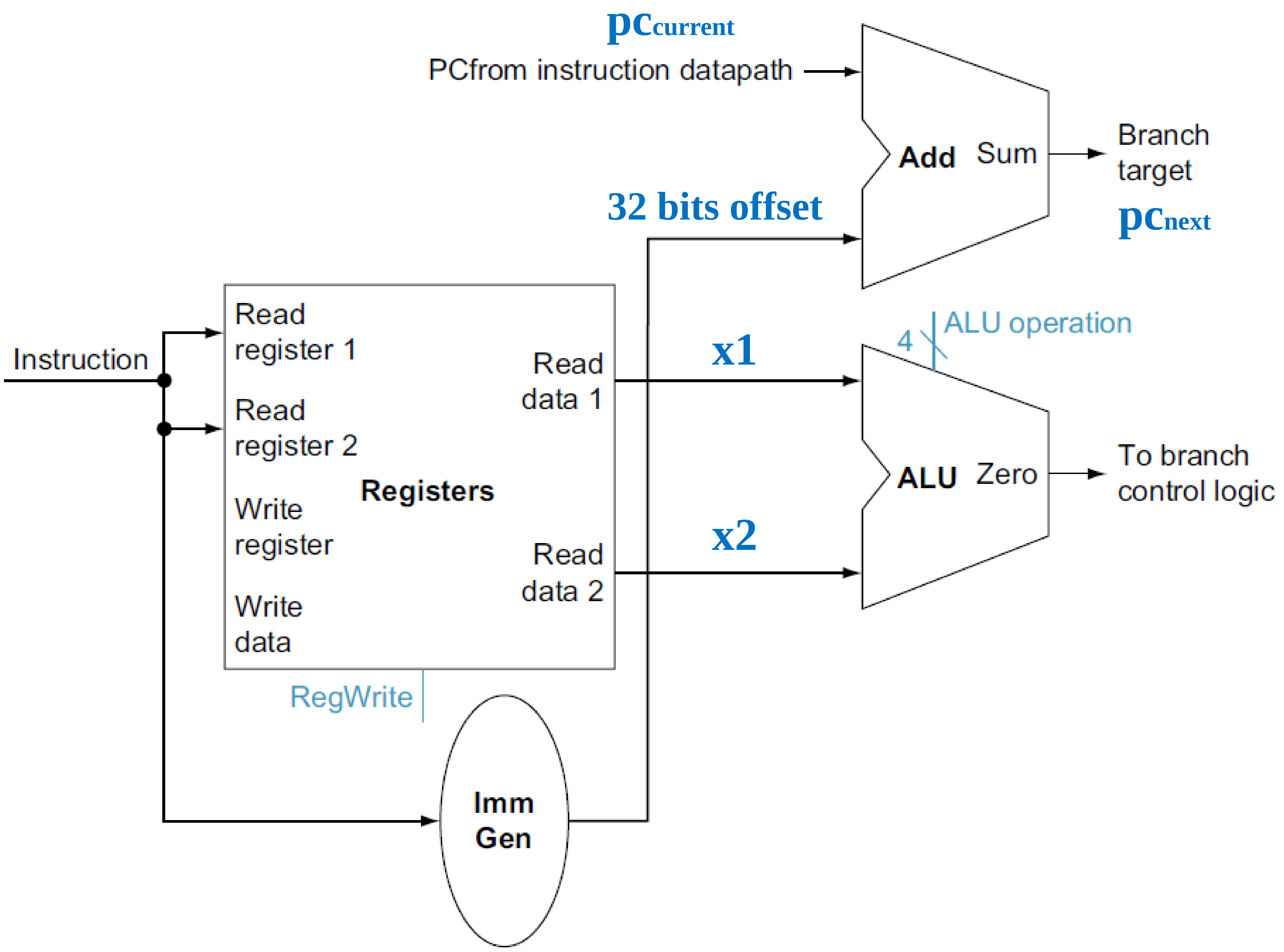


Portion of a datapath – Branch Instruction

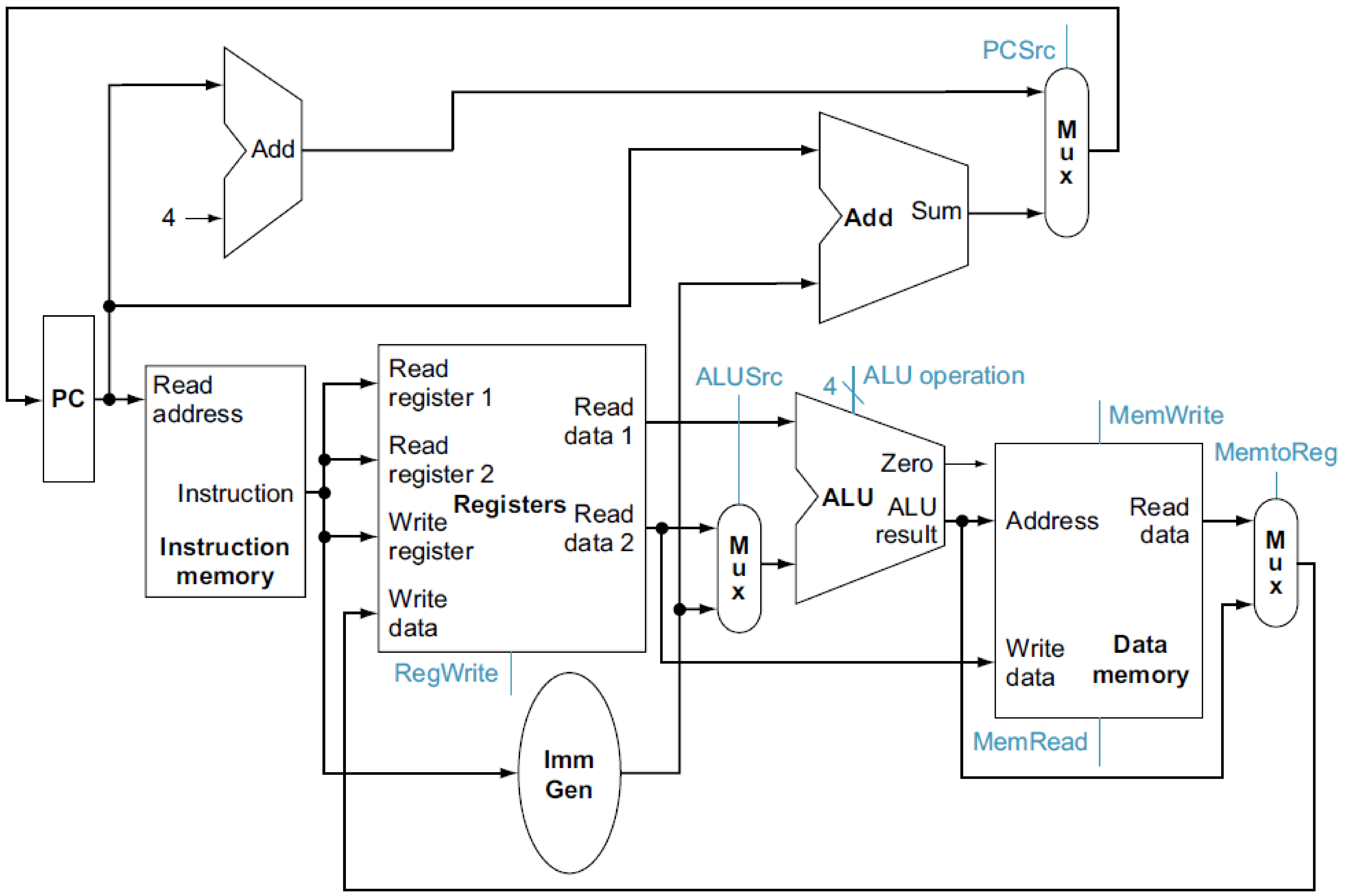
`beq x1, x2, offset`

If branch is taken

$pc_{next} = pc_{current} + offset$



Simple Datapath for core RISC V Architecture by different instruction classes

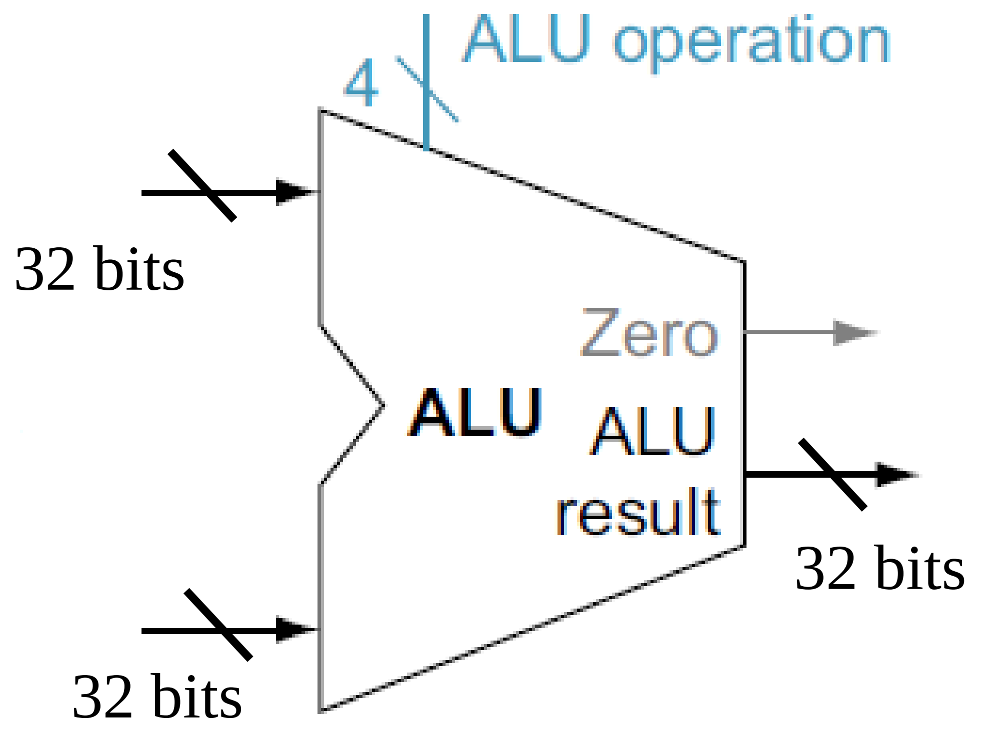


The ALU Control

Simple Implementation:

i. load word (lw),
ii. store word (sw),
iii. branch if equal (beq),
iv. arithmetic logical instructions add, sub, and, or

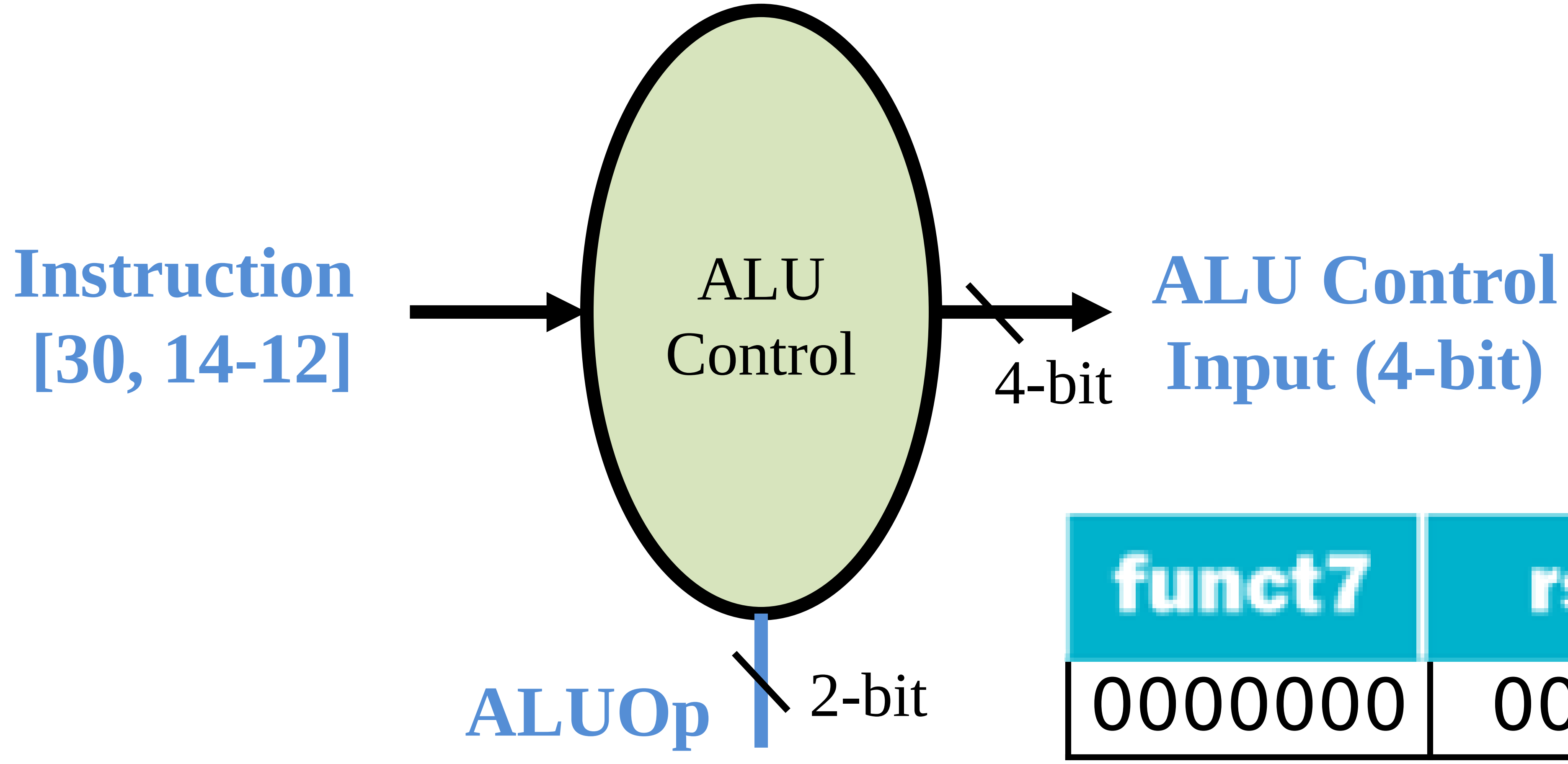
ALU



To generate these control lines, we need a small **ALU Control block.**

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

The ALU Control Block



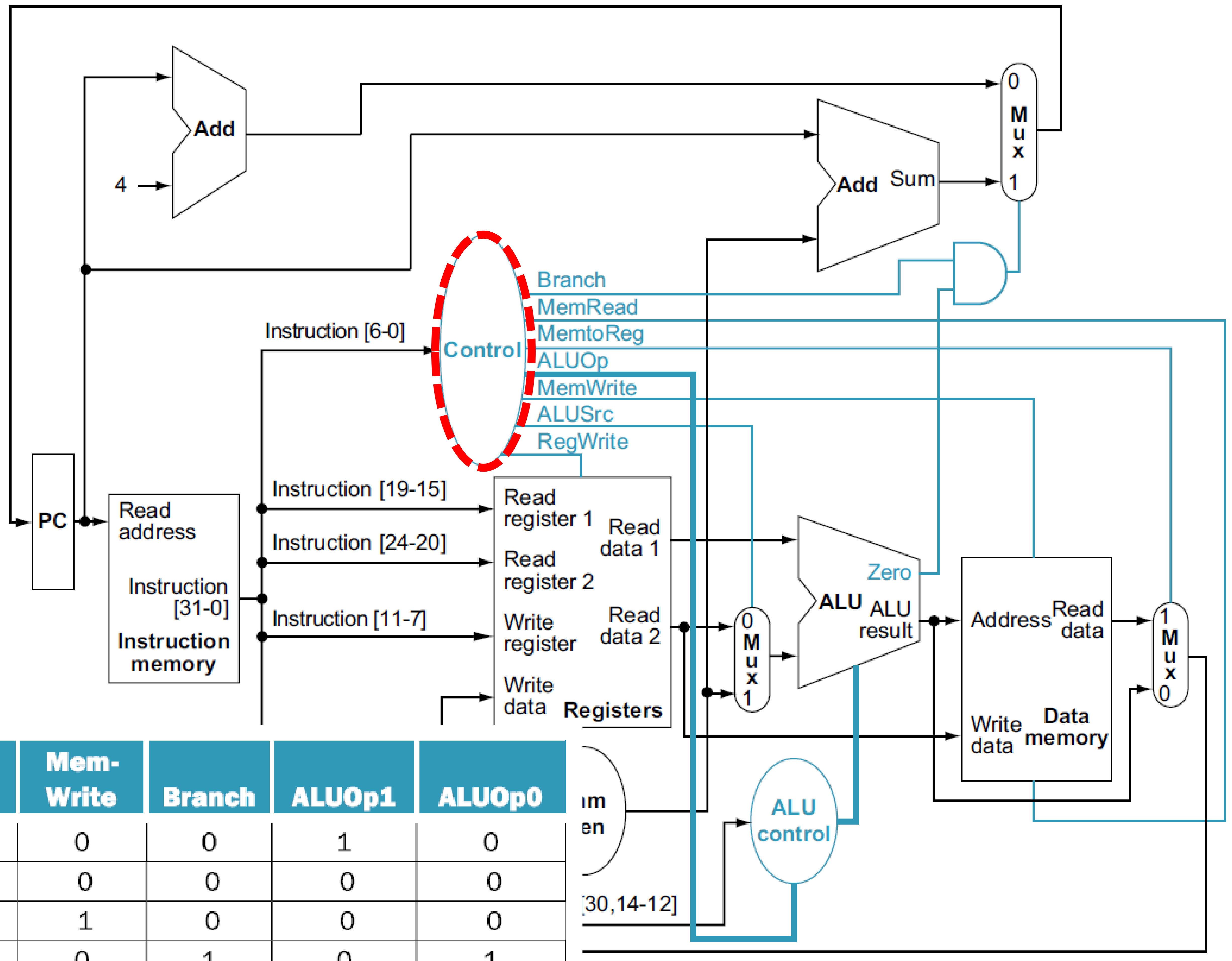
ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

add

funct7	rs2	rs1	funct3	rd	opcode
0000000	00100	00011	000	00010	0110011

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

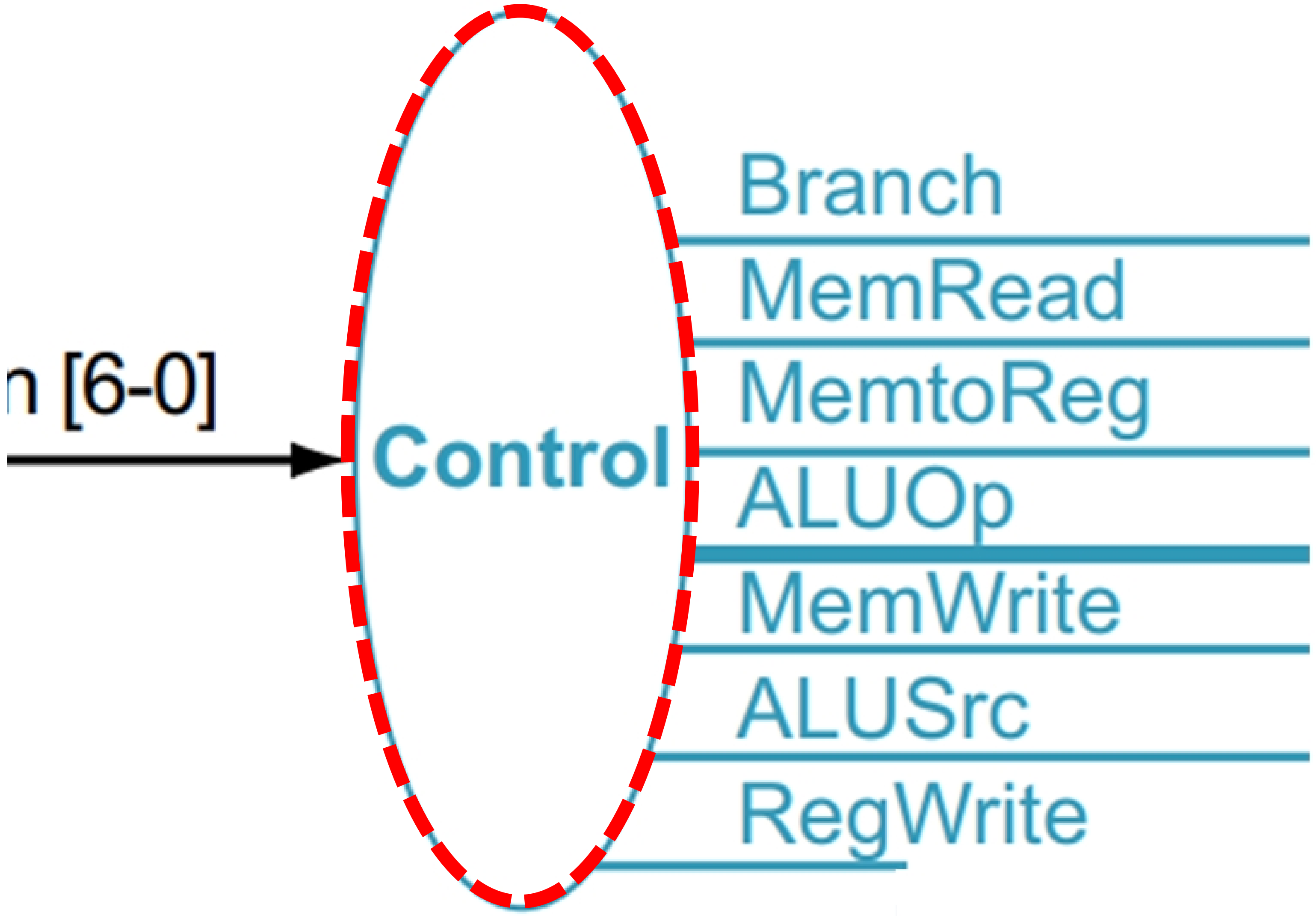
Datapath with Control Unit



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

Designing the Control Unit

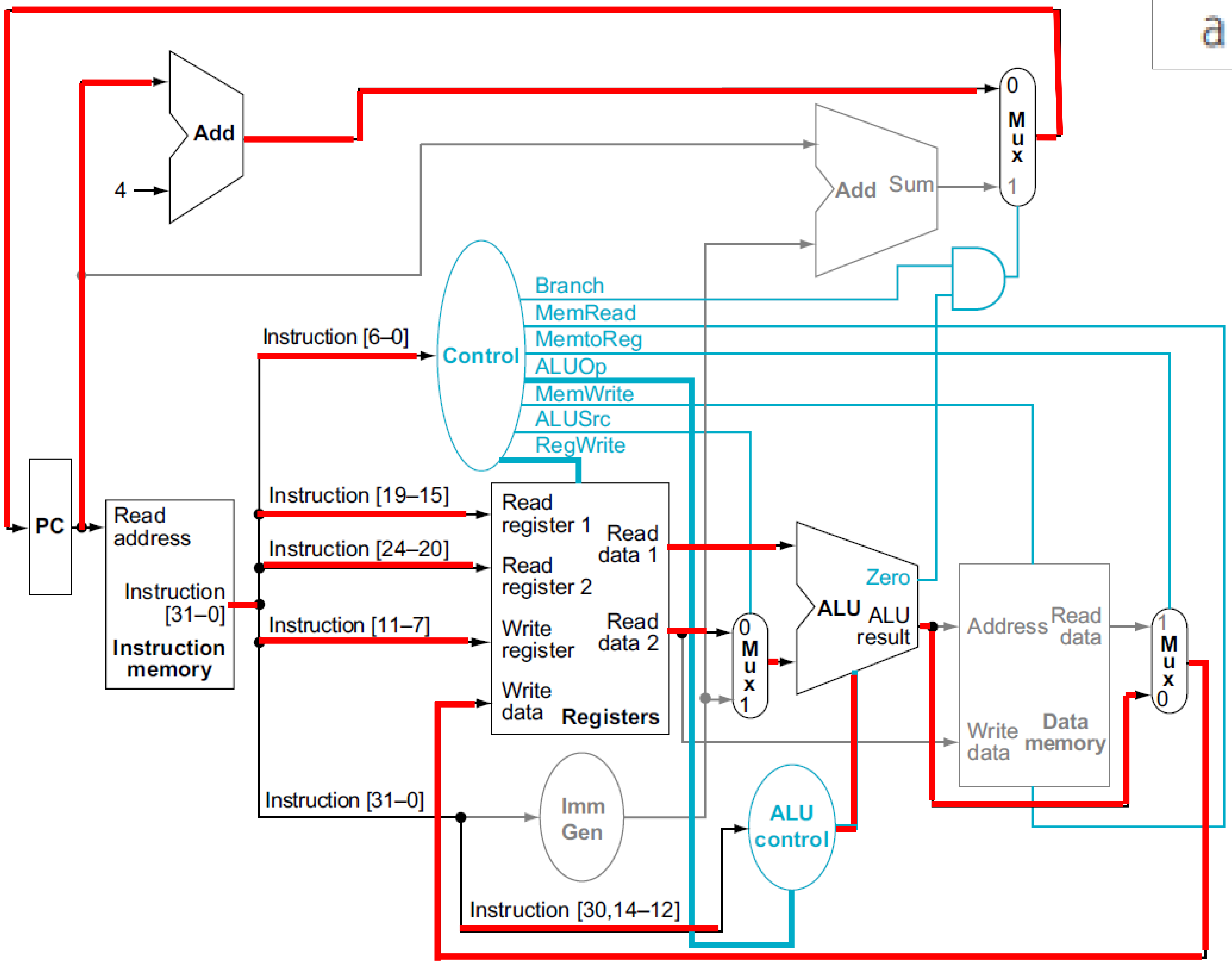
Input or output	Signal name	R-format	lw	sw	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1



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

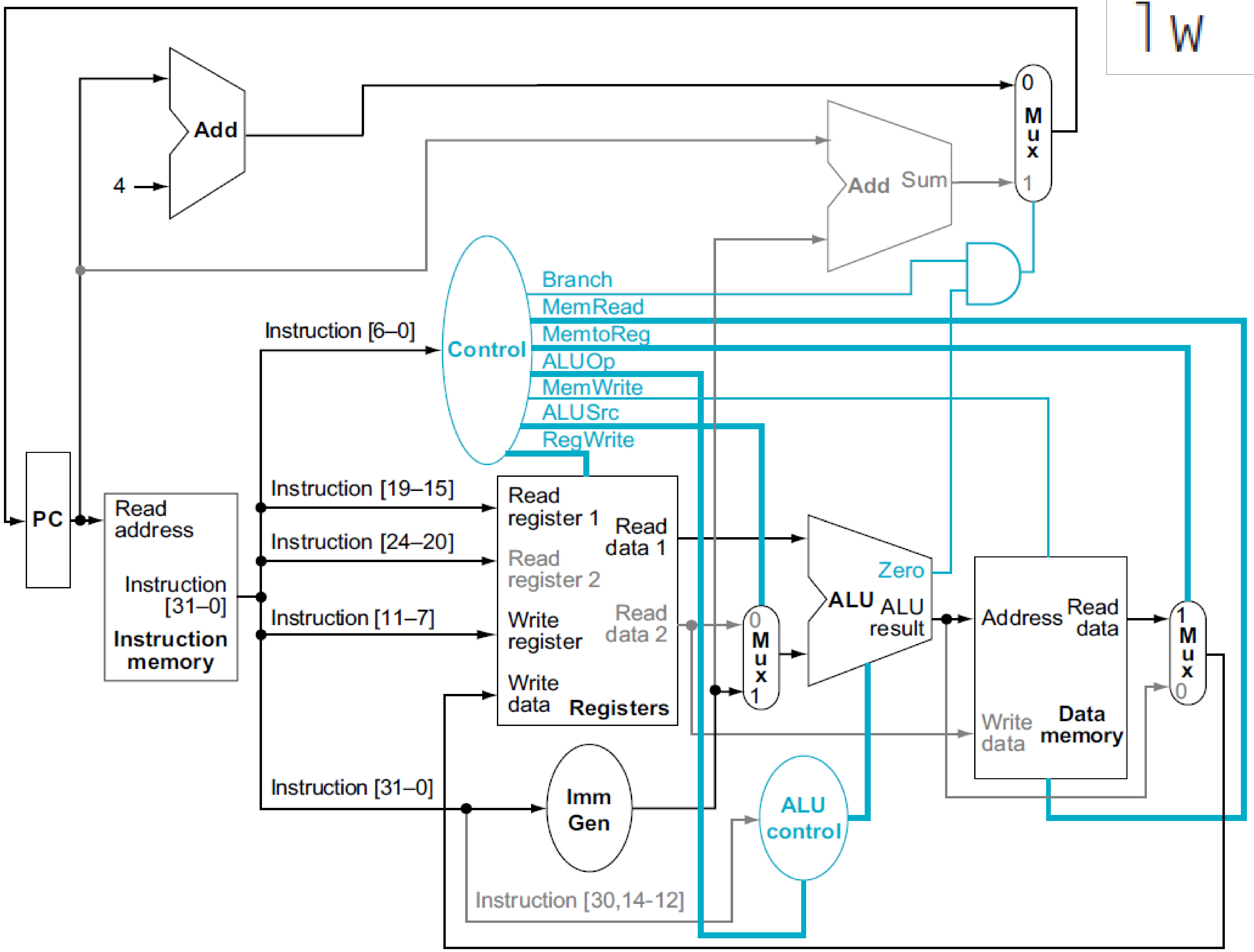
Datapath in an Operation for an R-type instruction

```
add  x10, x10, x25
```



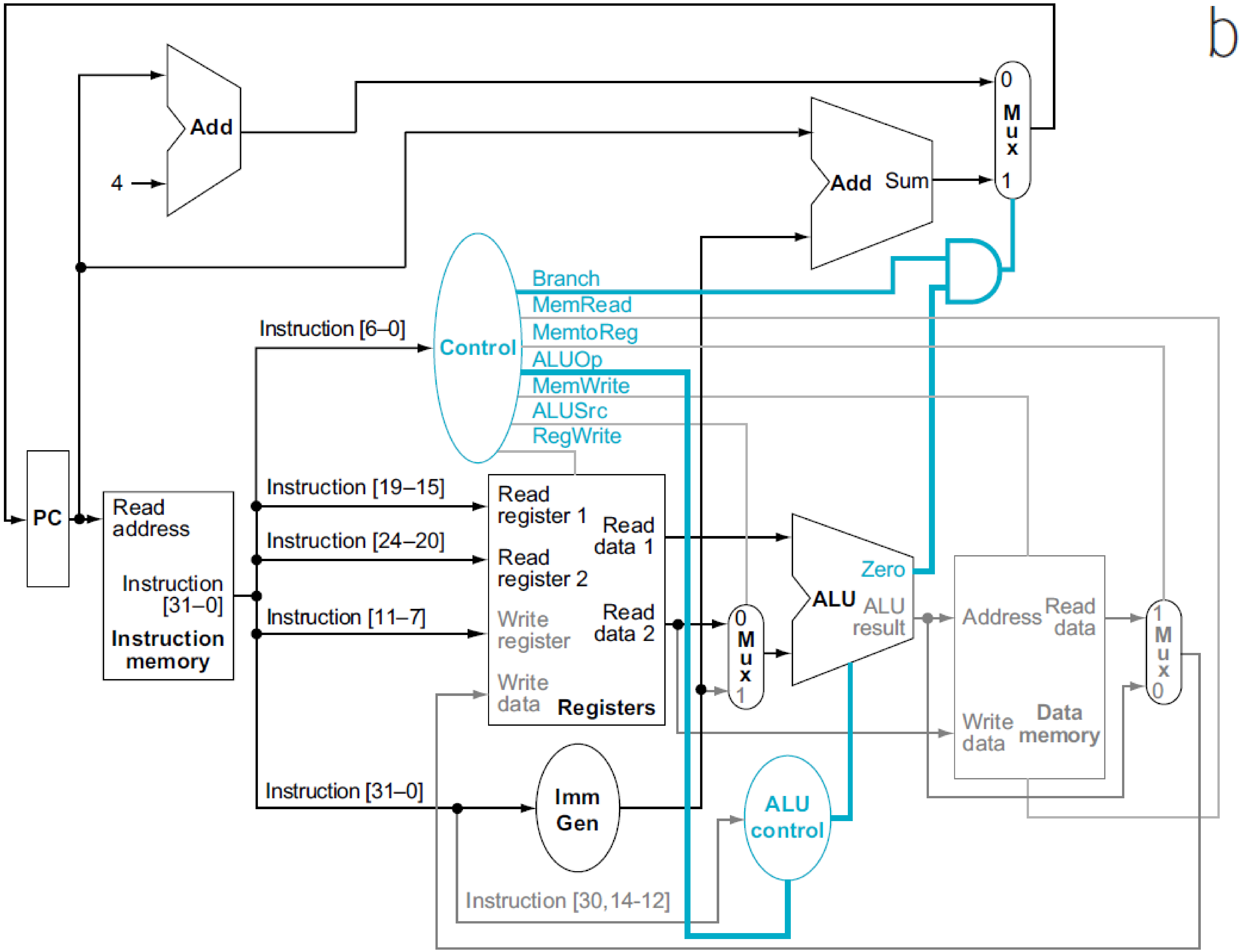
Datapath in an Operation for a load instruction

```
lw  x1,  offset(x2)
```



Datapath in an Operation for a branch if equal instruction

beq x1, x2, offset



Single Cycle Processor

- A **single cycle processor** is a processor that carries out **one instruction** in a **single clock cycle**.

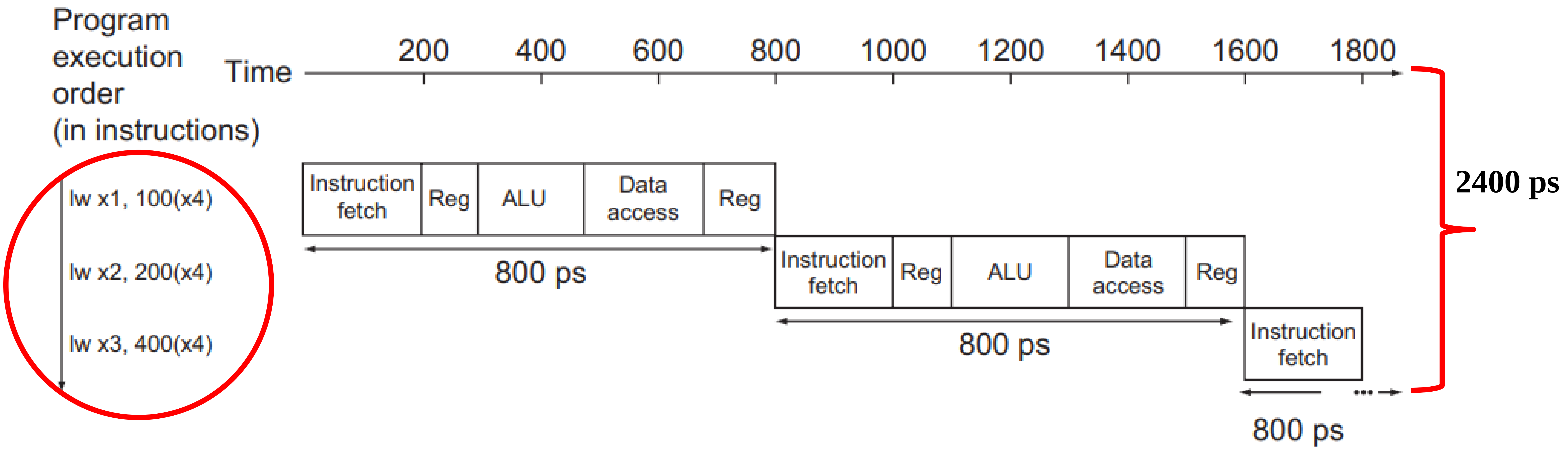
- Clock cycle is a fundamental unit of time that regulates the operation of the processor and synchronizes its various components.

- A processor has a clock oscillator that generates a continuous stream of electrical pulses known as the clock signal. Each pulse represents **one clock cycle.**

- All the components of the processor, including the arithmetic logic unit (ALU), memory, and control unit, are **synchronized** to this clock signal. This means that every action and operation within the processor happens in sync with the clock cycles.

Single Cycle Processor

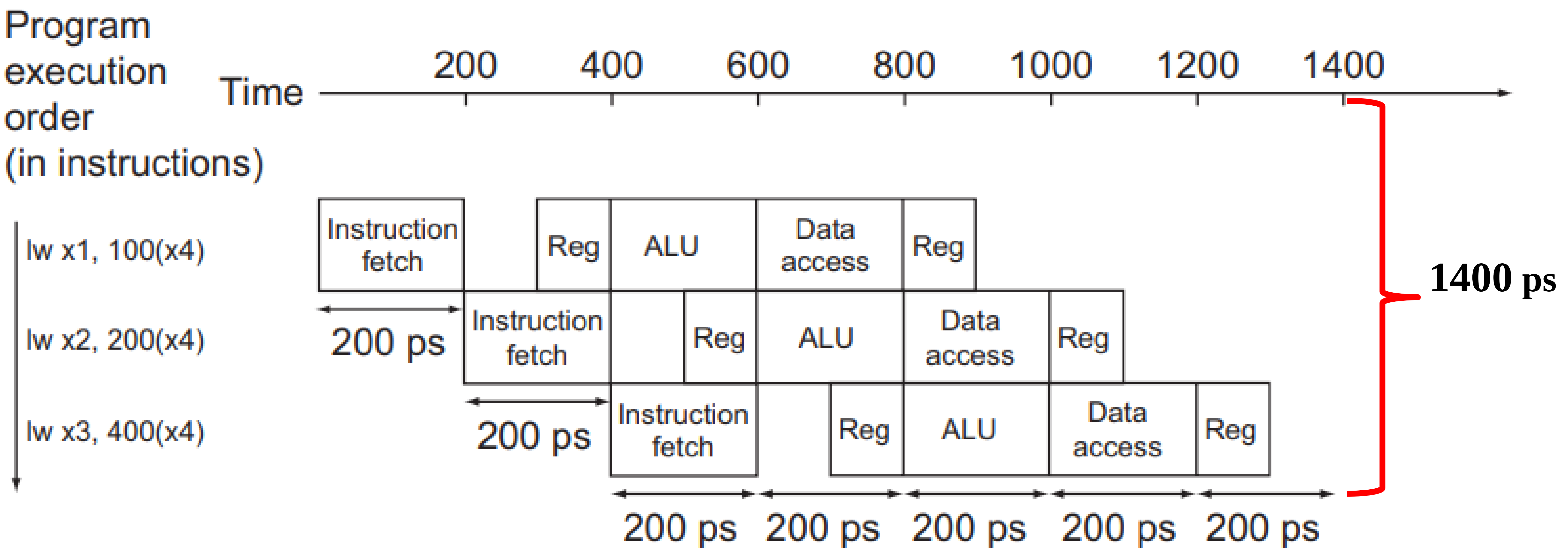
Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps



Pipelining

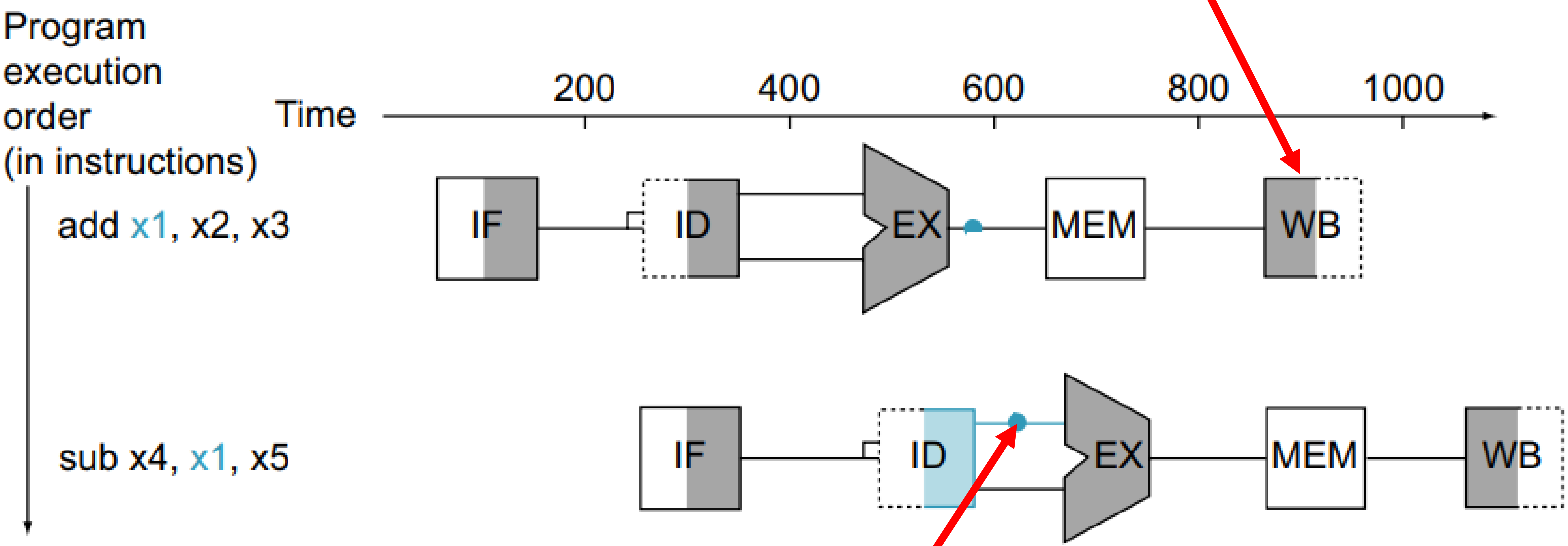
Implementation technique in which **multiple instructions** are **overlapped** in execution.

Pipelining is a technique used in computer architecture to improve the **throughput** and **efficiency** of a processor by overlapping the execution of multiple instructions.



Pipelining

WB stage will update the value of x1 from the result of x2+x3 in the register file



But the updated value of x1 is required here.

Pipelining Hazard – Structural Hazards, Data Hazards, Control Hazards

Pipelining Hazards

Structural Hazard

Structural hazard occurs when two instructions need same hardware resource at same time.

Data Hazard

Data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline.

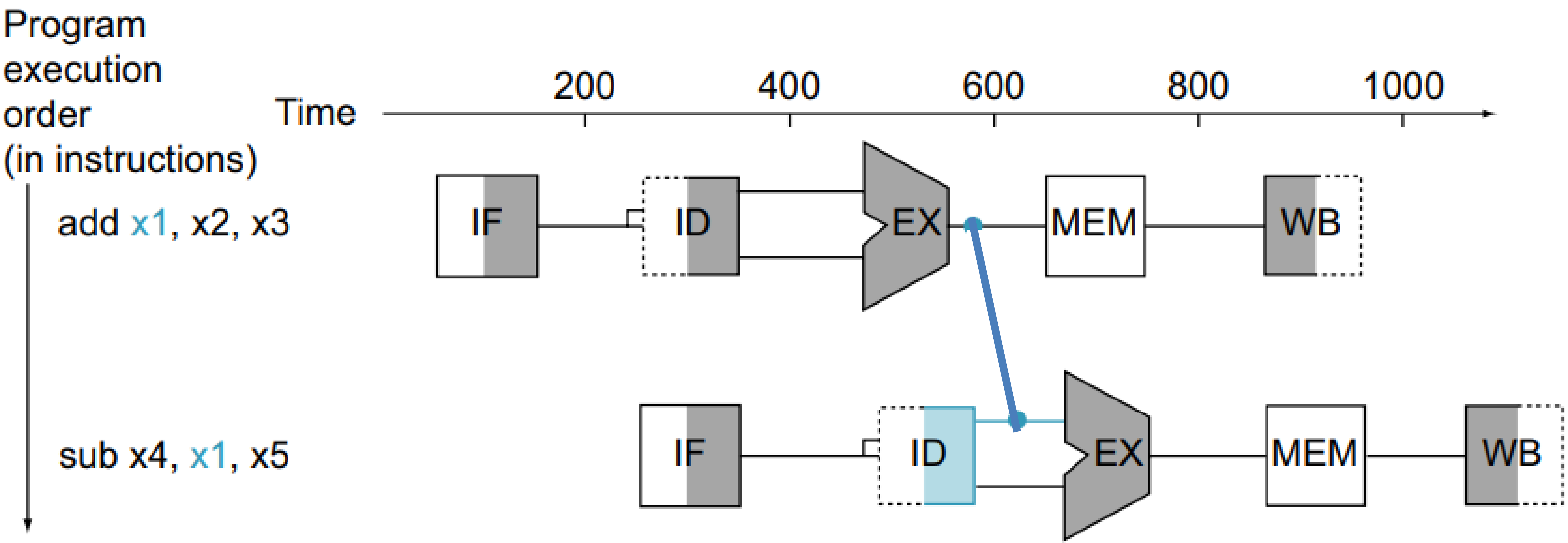
```
add  x19, x0, x1
sub  x2, x19, x3
```

Pipelining Hazards

Control Hazard **Delay** in determining the proper instruction to fetch

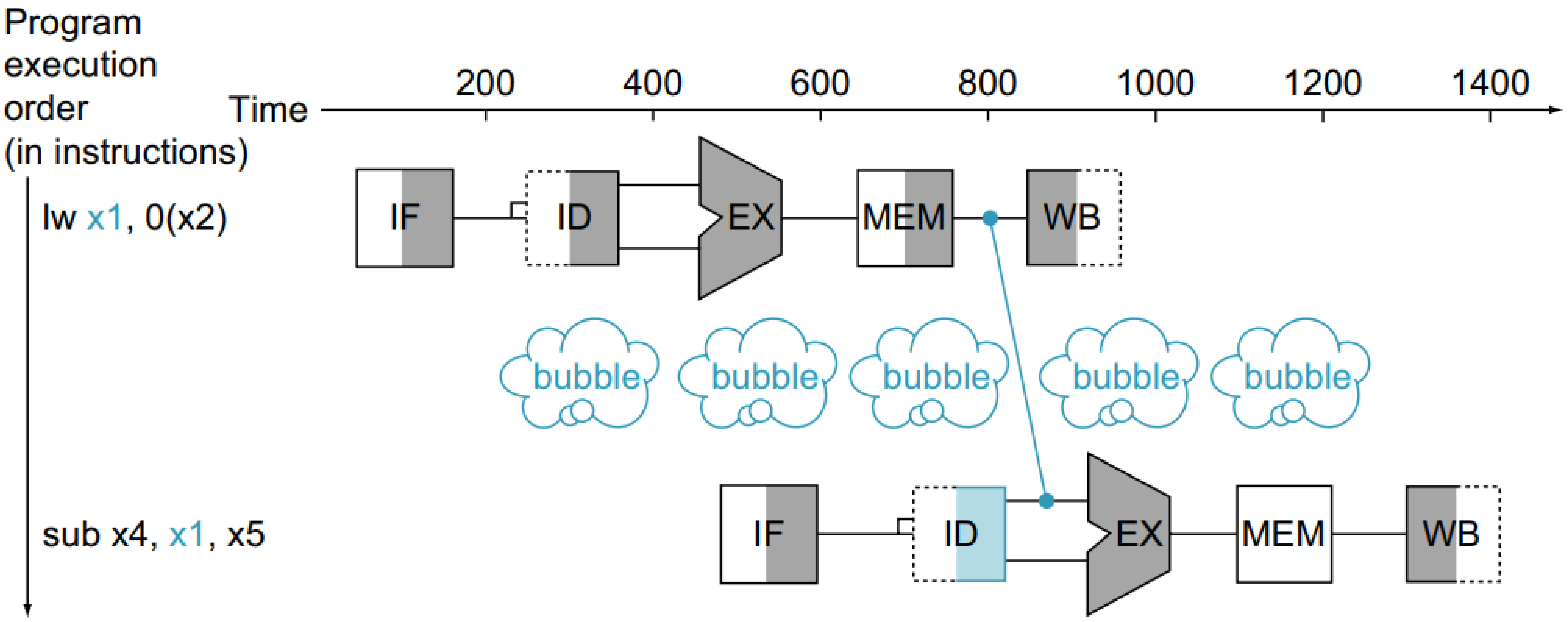
```
add x4, x5, x6
beq x1, x0, 40
lw x3, 400(x0)
```

Pipelining Hazard - Solution



Forwarding

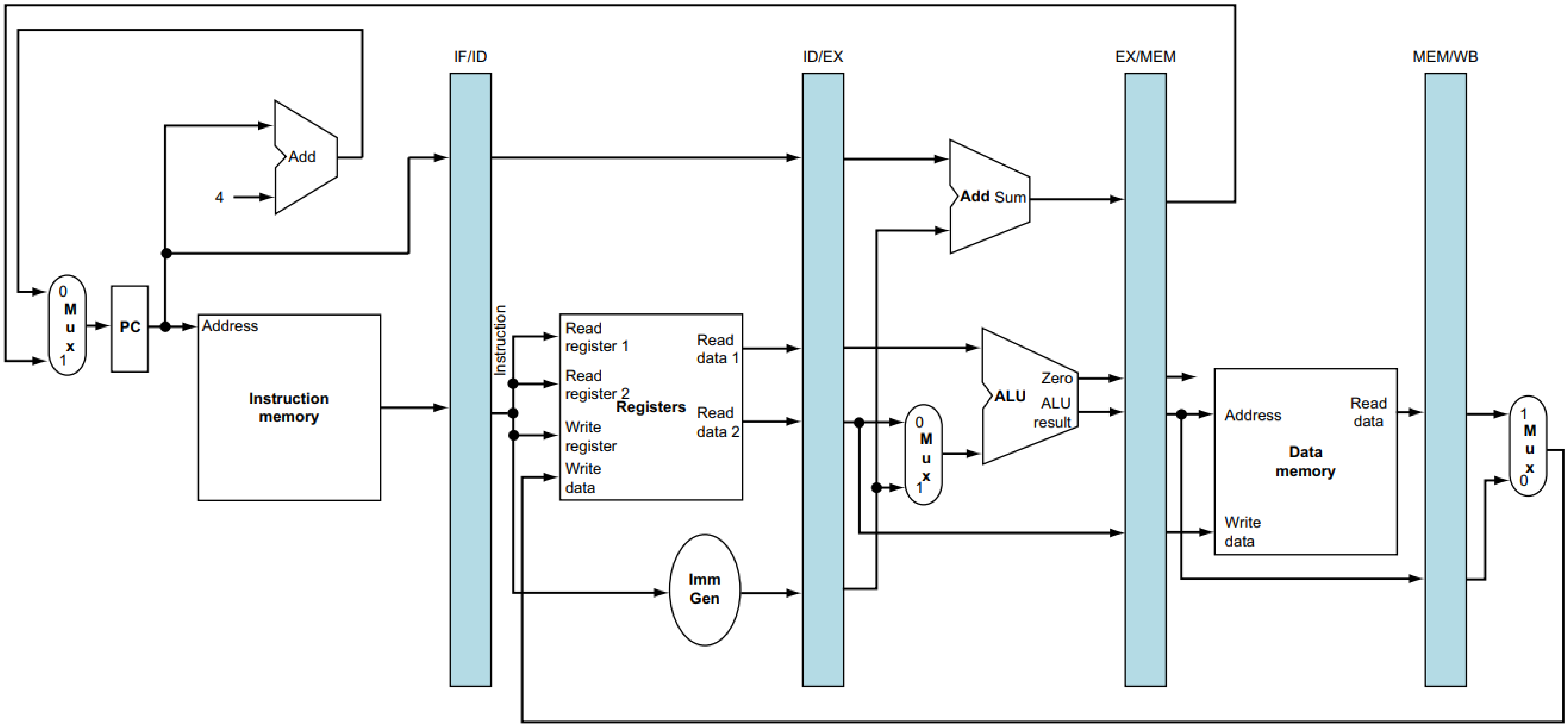
Pipelining Hazard - Solution



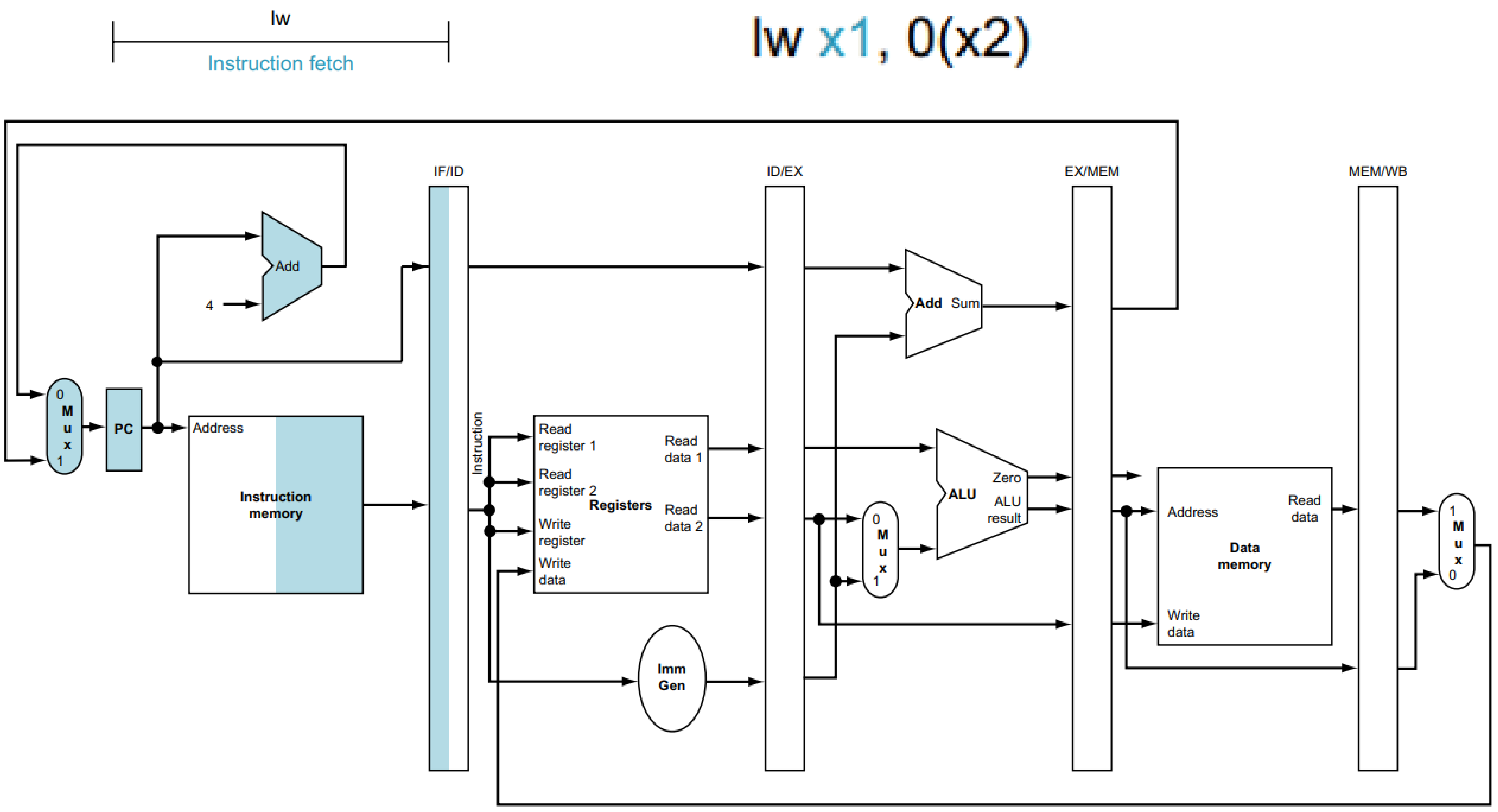
Load-use Data Hazard A specific form of data hazard in which the data being loaded by a load instruction have not yet become available when they are needed by another instruction.

Pipeline Stall (Bubble). A stall initiated in order to resolve a hazard.

Pipelined Version of Datapath

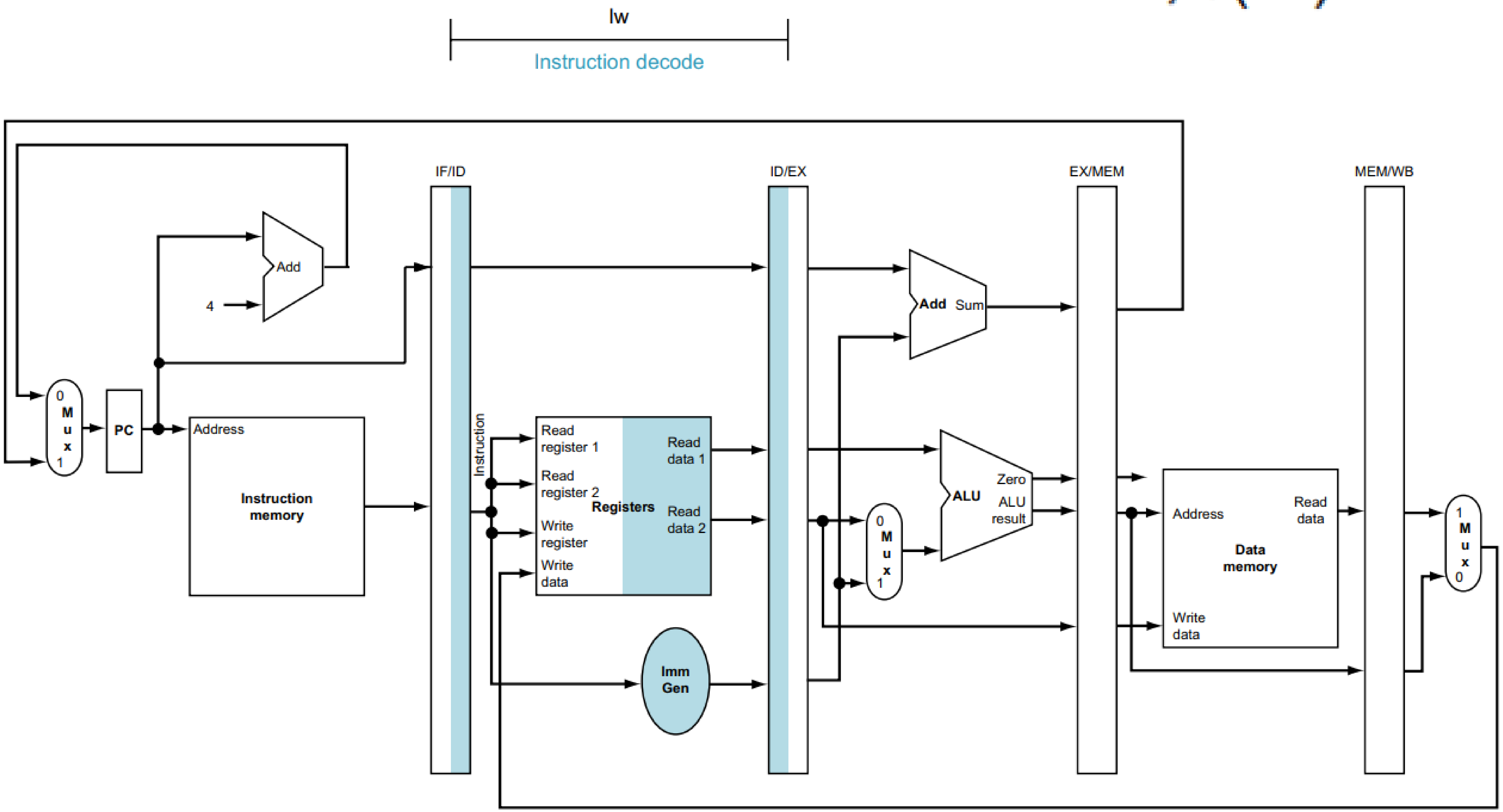


Five Pipe stages – Load instruction – Stage 1: Instruction Fetch



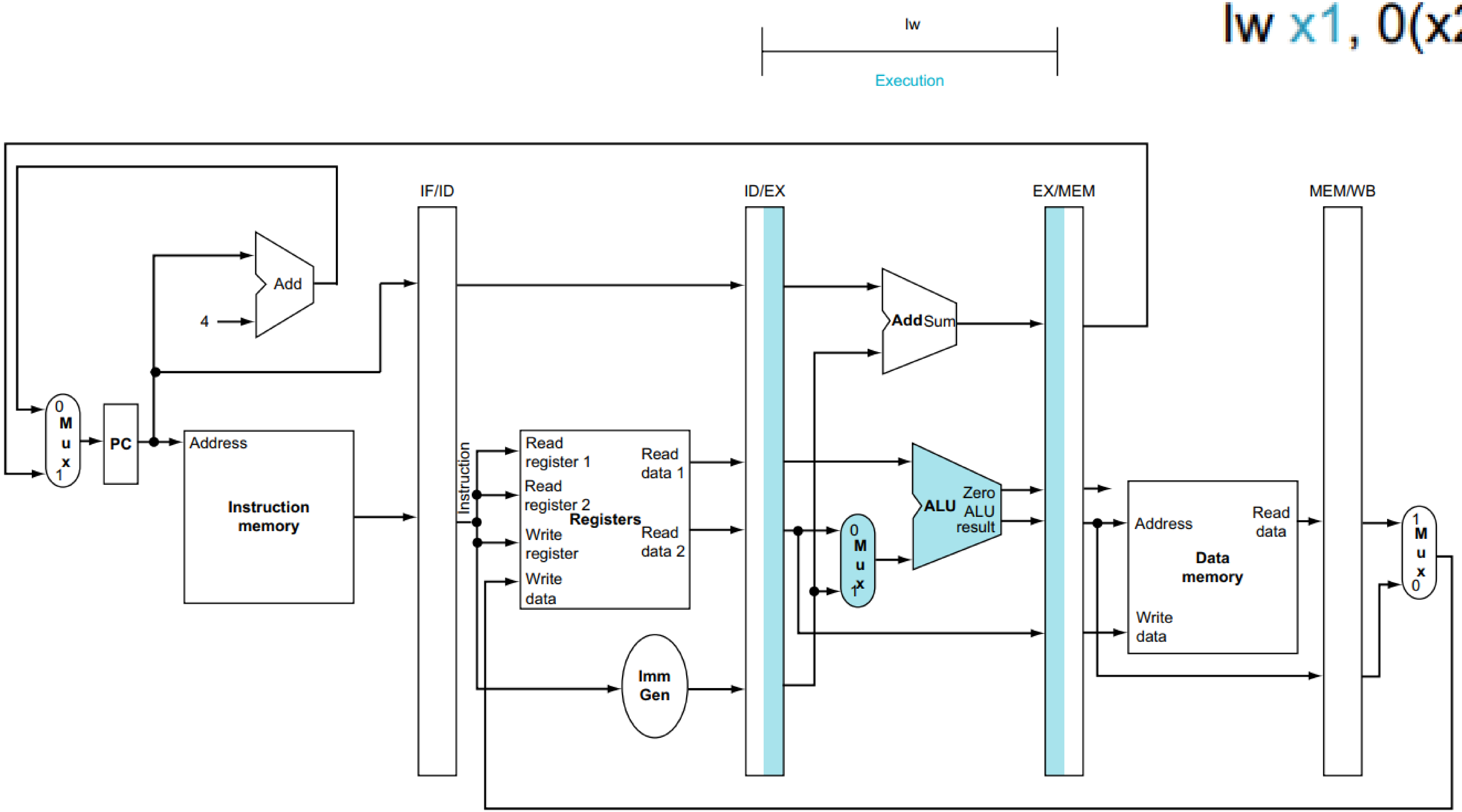
Five Pipe stages – Load instruction – Stage 2: Instruction decode and register file read

lw x1, 0(x2)



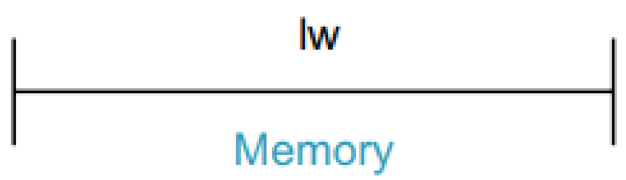
Five Pipe stages – Load instruction – Stage 3: Execution

lw x1, 0(x2)

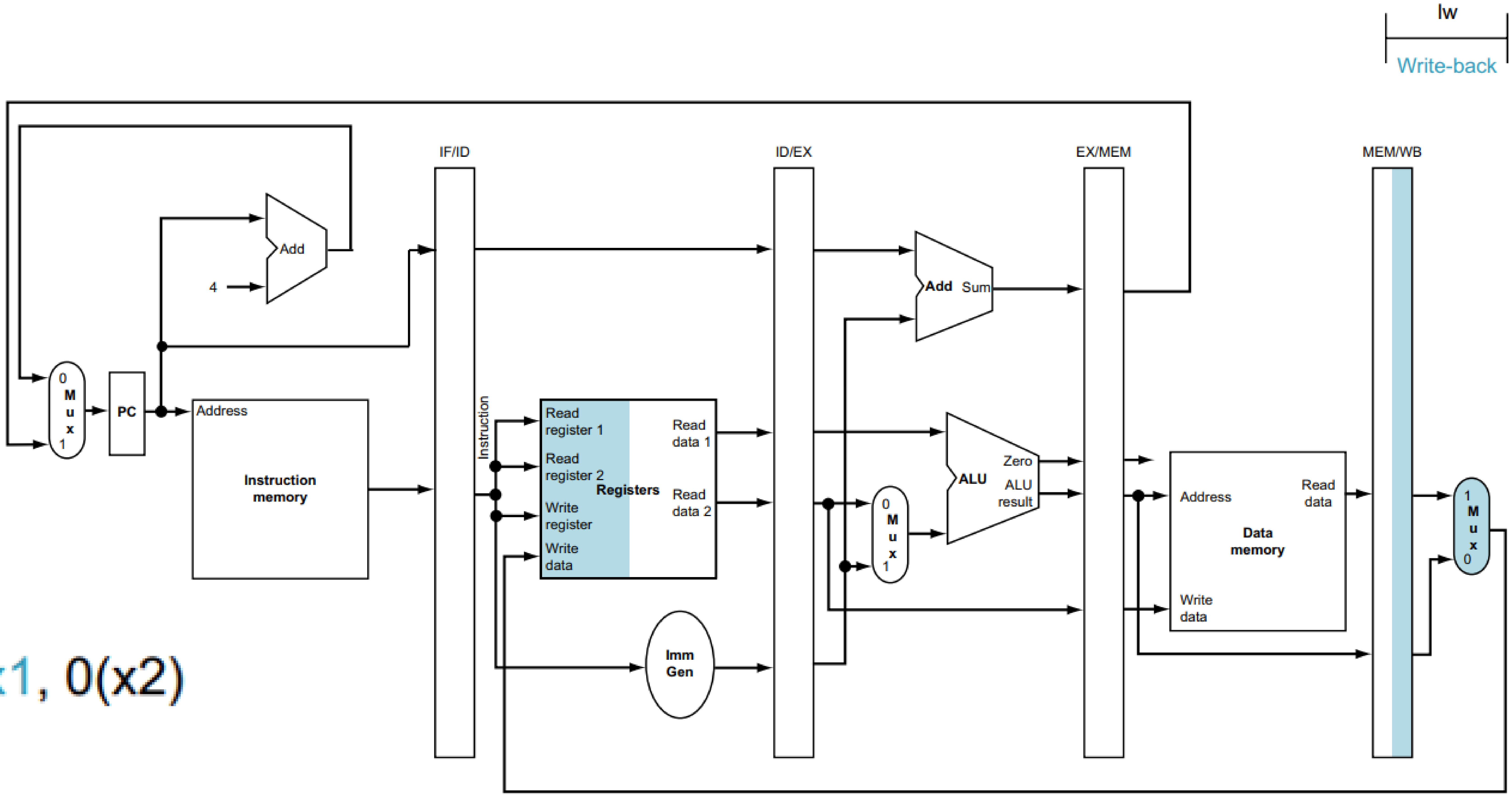


Five Pipe stages – Load instruction – Stage 4: Memory

lw x1, 0(x2)

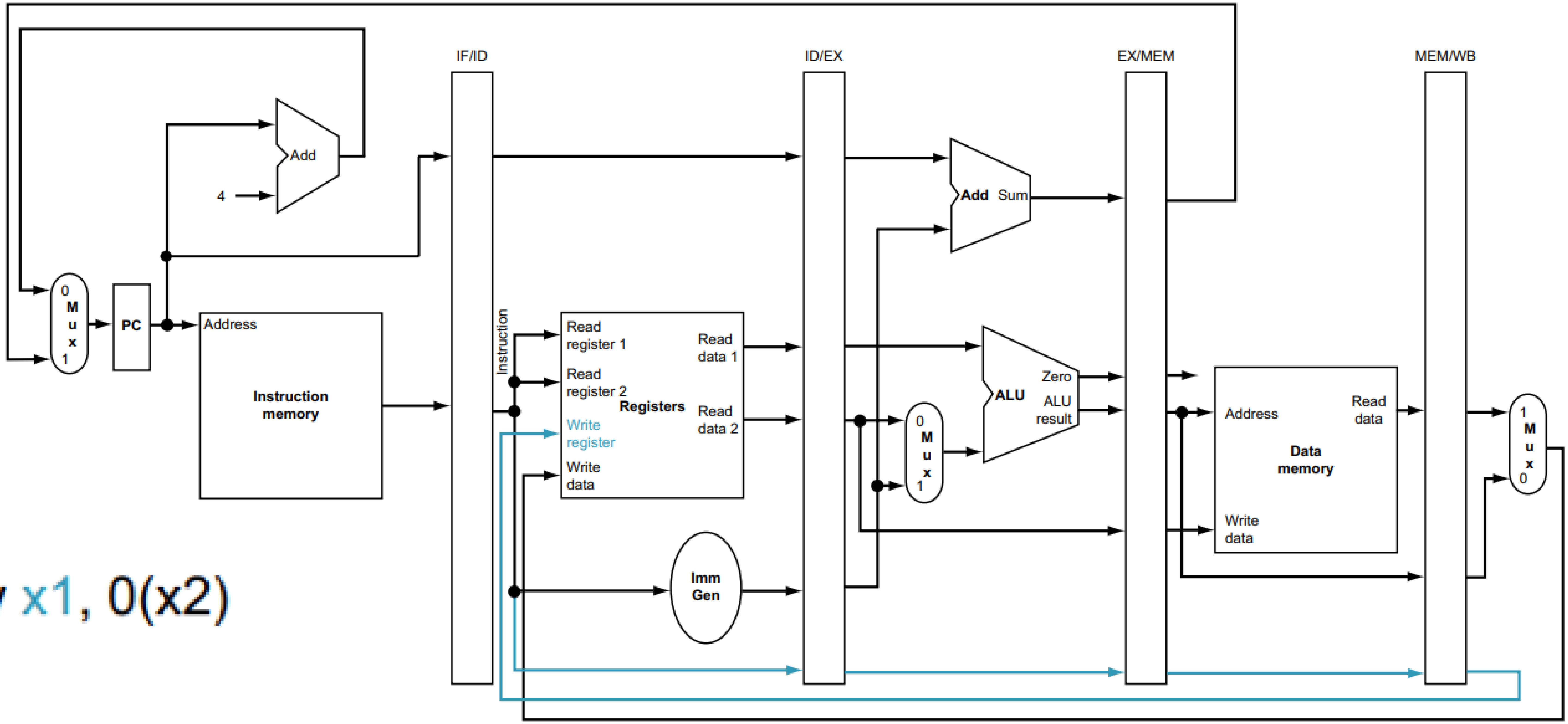


Five Pipe stages – Load instruction – Stage 5: Write Back



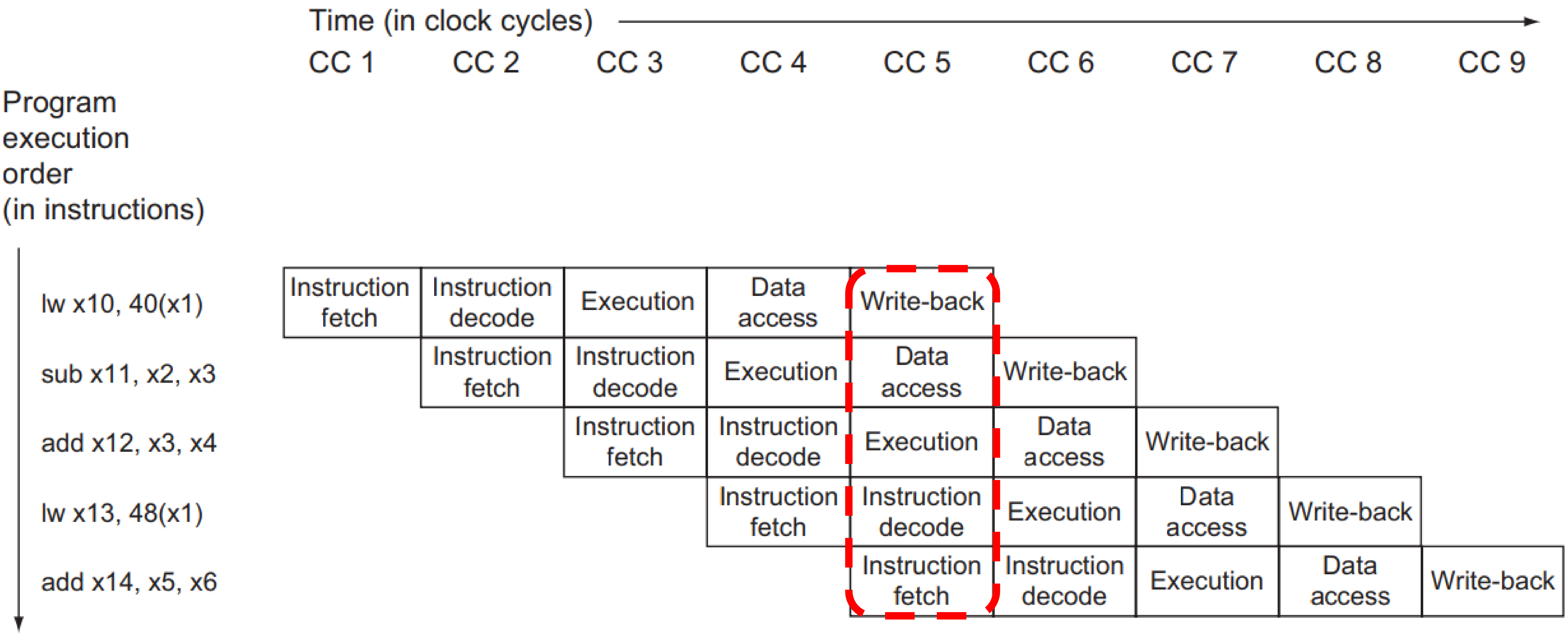
lw x1, 0(x2)

Five Pipe stages – Load instruction

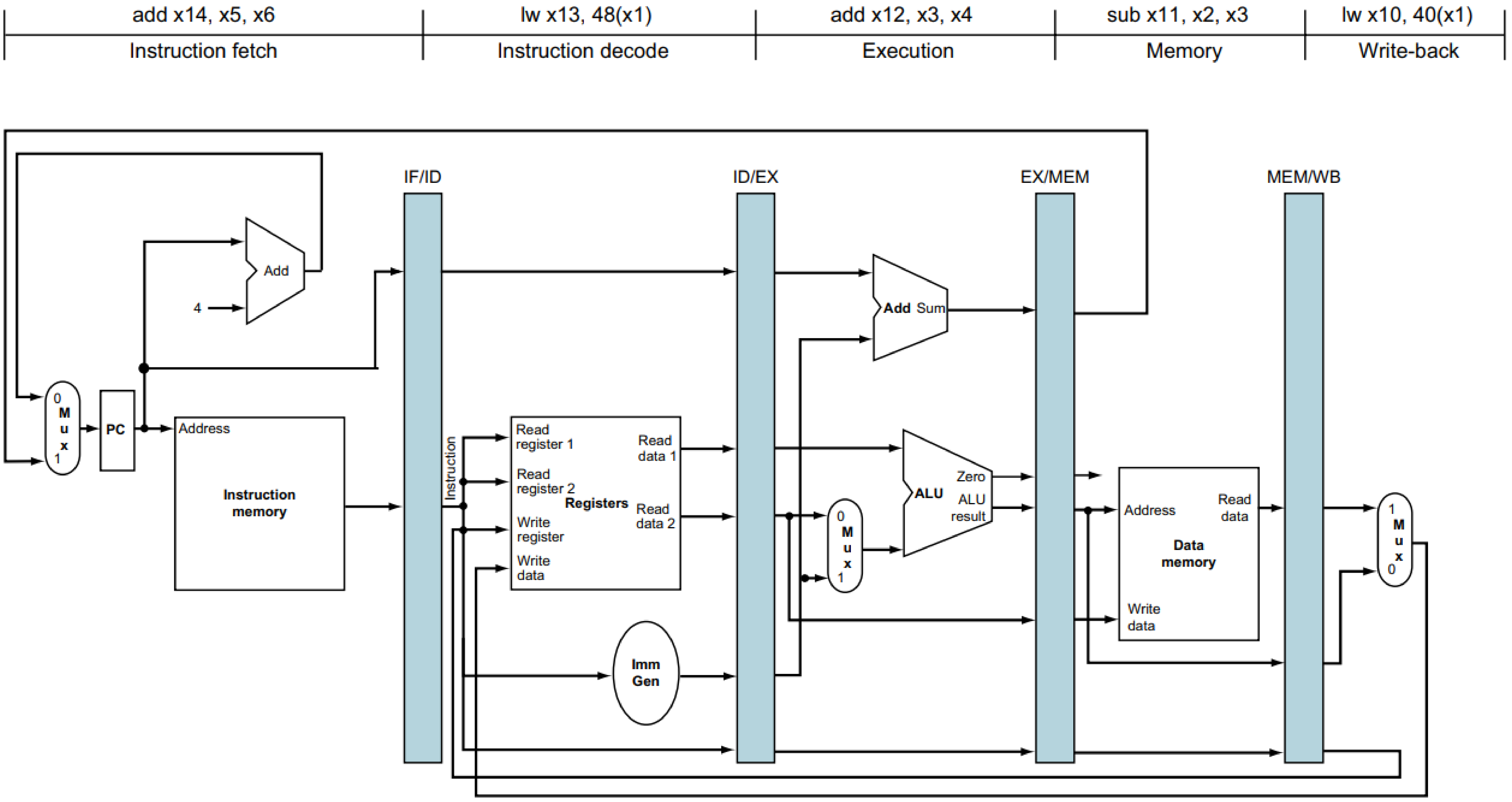


lw x1, 0(x2)

Traditional multiple-clock-cycle pipeline diagram of five instructions



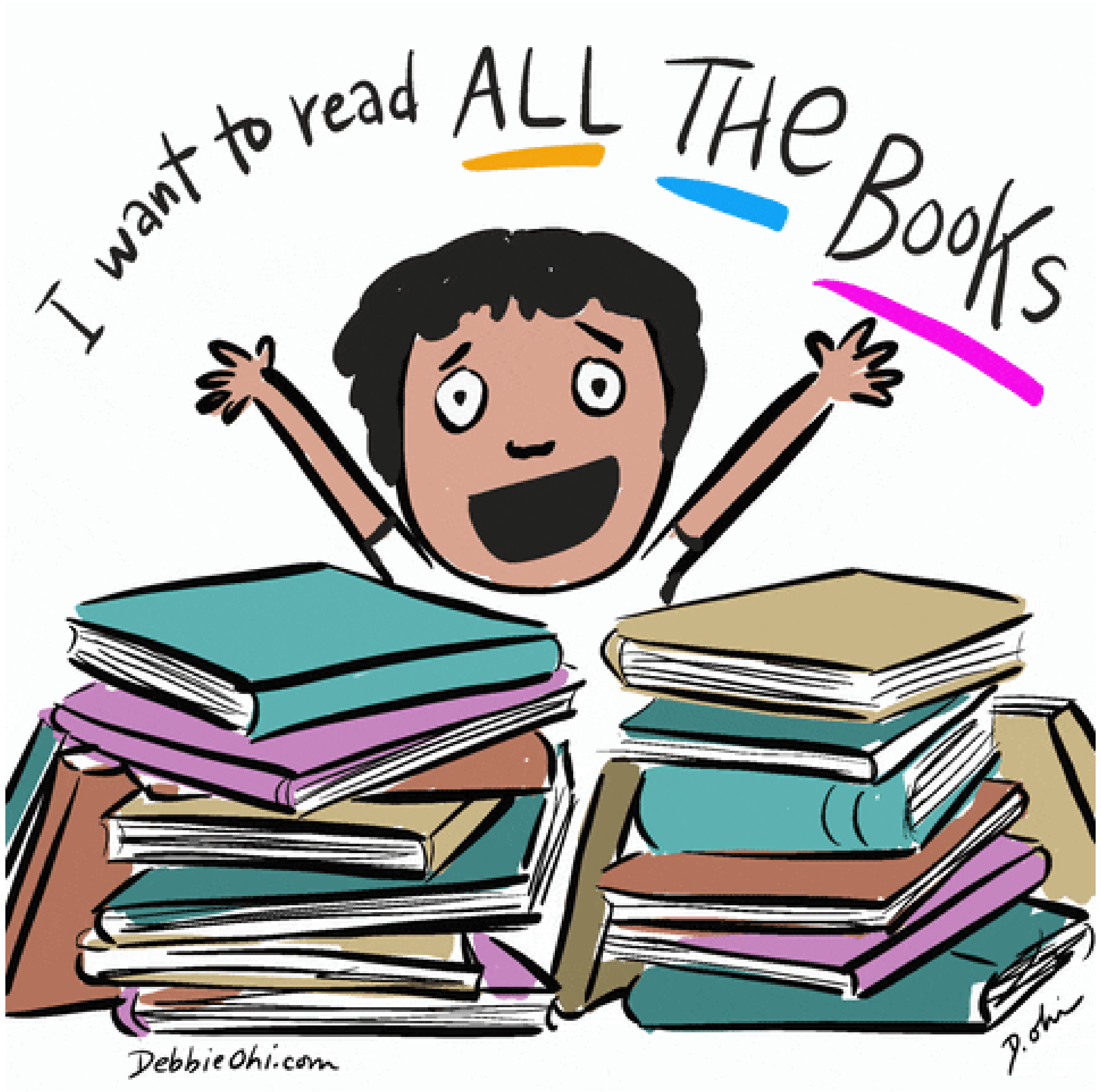
CC5


add x14, x5, x6
Instruction fetch
lw x13, 48(x1)
Instruction decode
add x12, x3, x4
Execution
sub x11, x2, x3
Memory
lw x10, 40(x1)
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Imm Gen
Add Sum
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

Pipelining – Conclusion

Pipelining is a technique used in computer processors to **improve overall throughput and efficiency** by allowing multiple instructions to be in various stages of execution simultaneously. It's similar to how an assembly line in a factory can increase the rate of production by having different stages of manufacturing happening concurrently. However, pipelining does not reduce the time it takes for an individual instruction to complete, which is referred to as the instruction's latency.

Memory



Memory – Principle of Locality

- Program access a relatively small portion of their address space at any instant of time.
- Two types

1. **Temporal Locality** – Locality in time

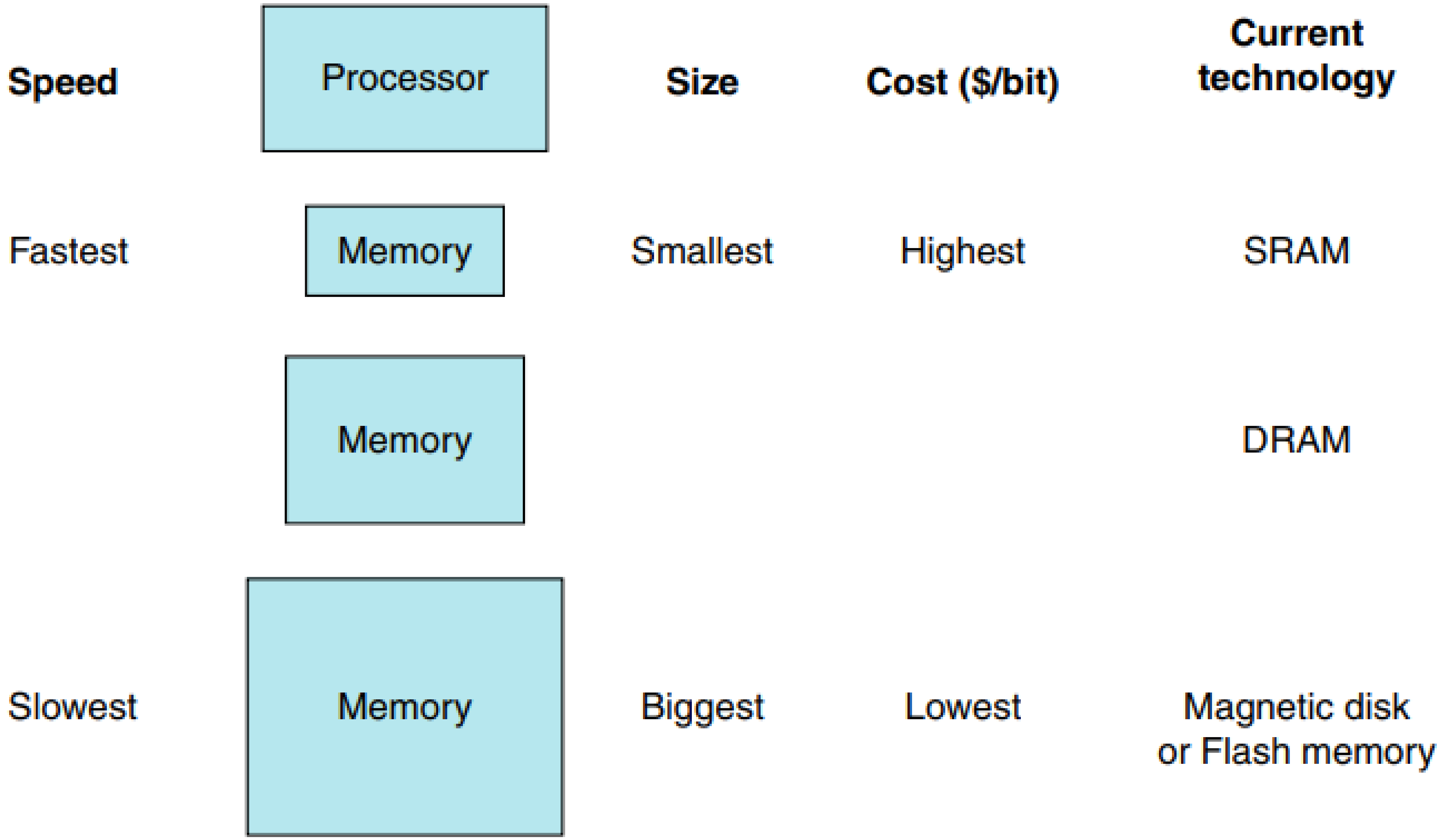
If an item is referenced, it will tend to be referenced again soon.

2. **Spatial Locality** – Locality in space

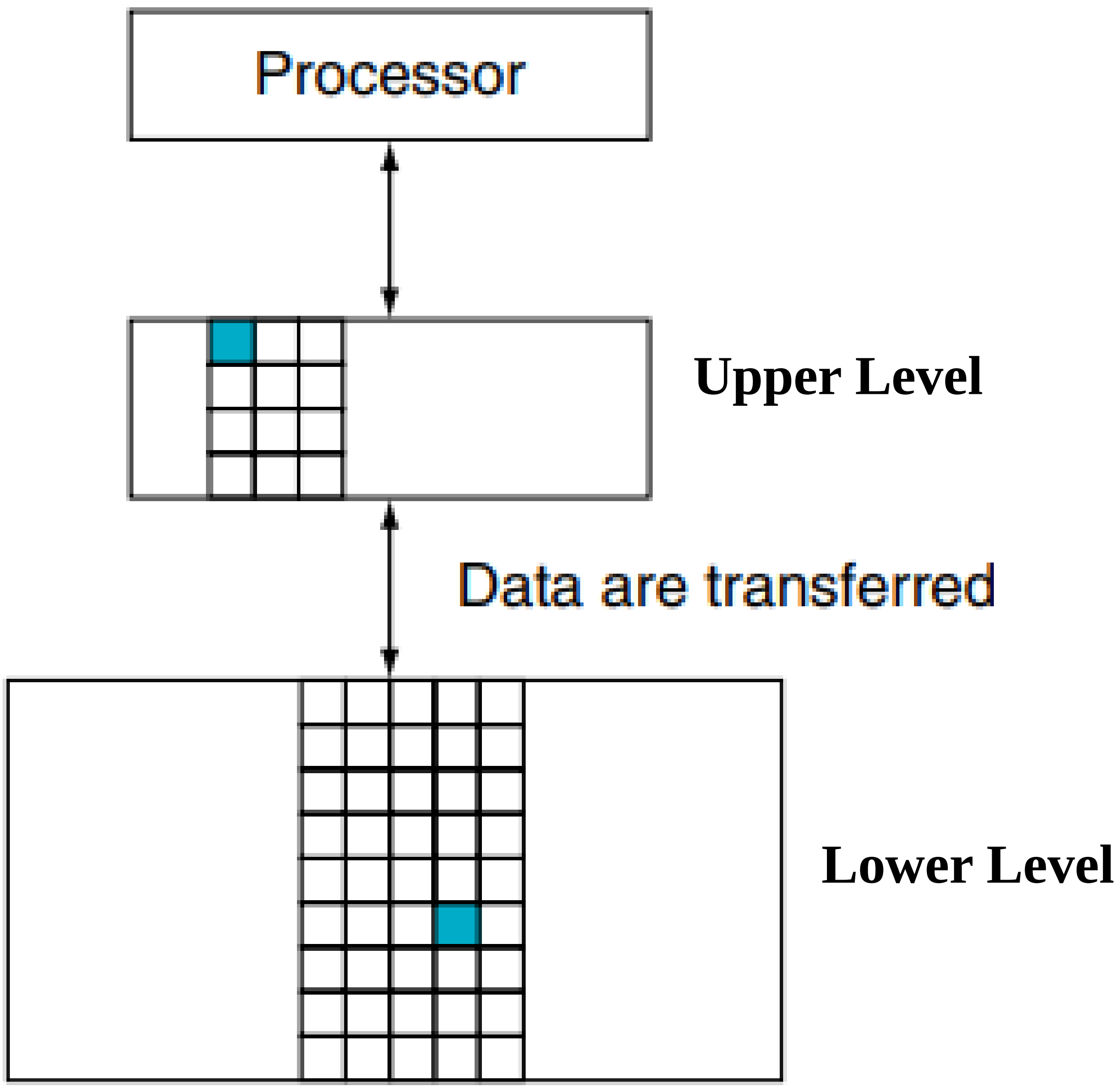
If an item is referenced, items whose addresses are close by will tend to be referenced soon

Memory Hierarchy

A structure that uses **multiple levels of memories**;
as the **distance** from the processor **increases**, the **size** of the memories and the **access time** both **increase** while the cost **decreases**.

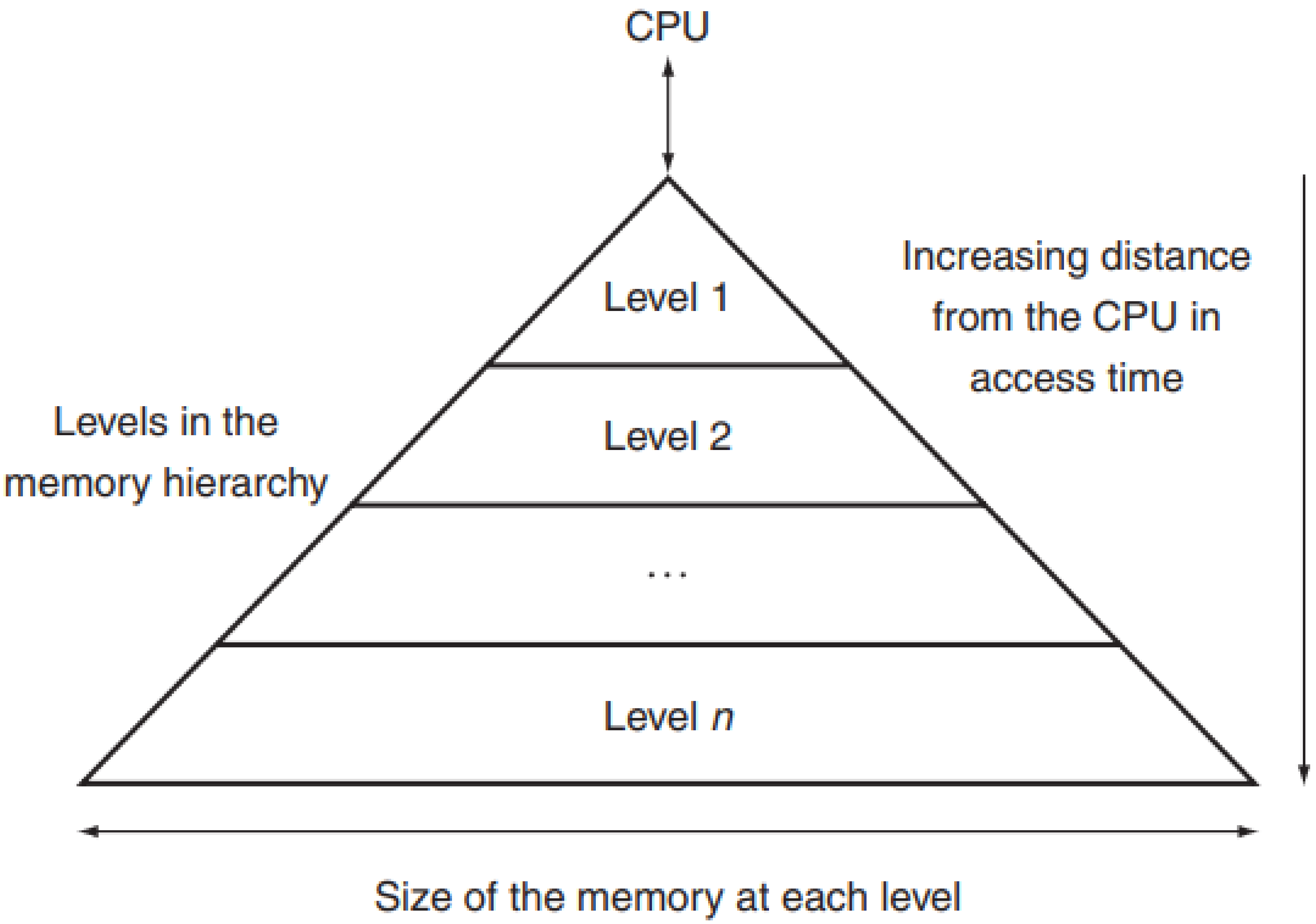


Levels in the Memory Hierarchy



- **Upper level** is the subset of **lower level**.
- Hit rate
- Miss rate
- Miss penalty

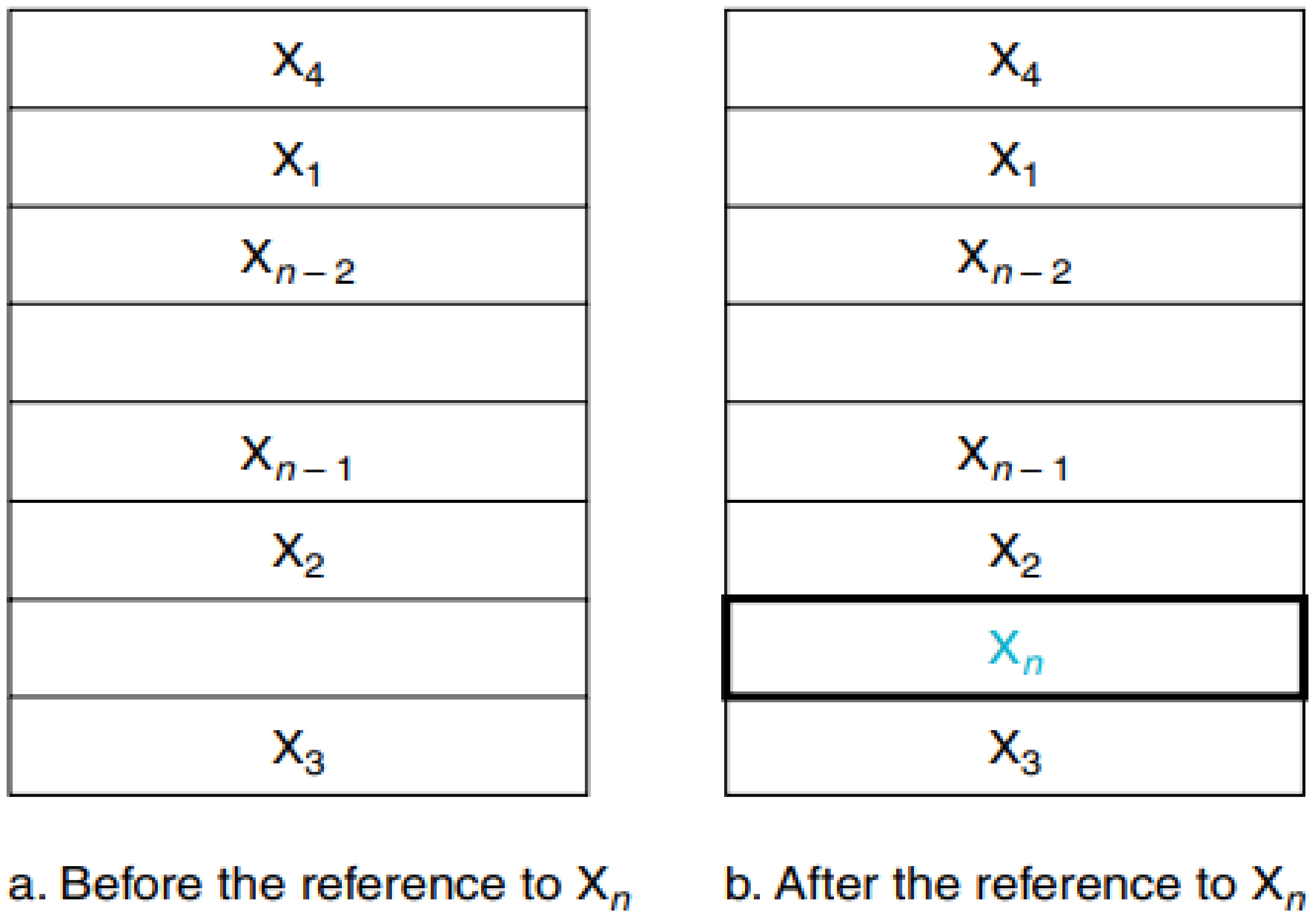
Memory Hierarchy



Memory Technologies

Memory technology	Typical access time	$ per GiB in 2020
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$3–$6
Flash semiconductor memory	5,000–50,000 ns	$0.06–$0.12
Magnetic disk	5,000,000–20,000,000 ns	$0.01–$0.02

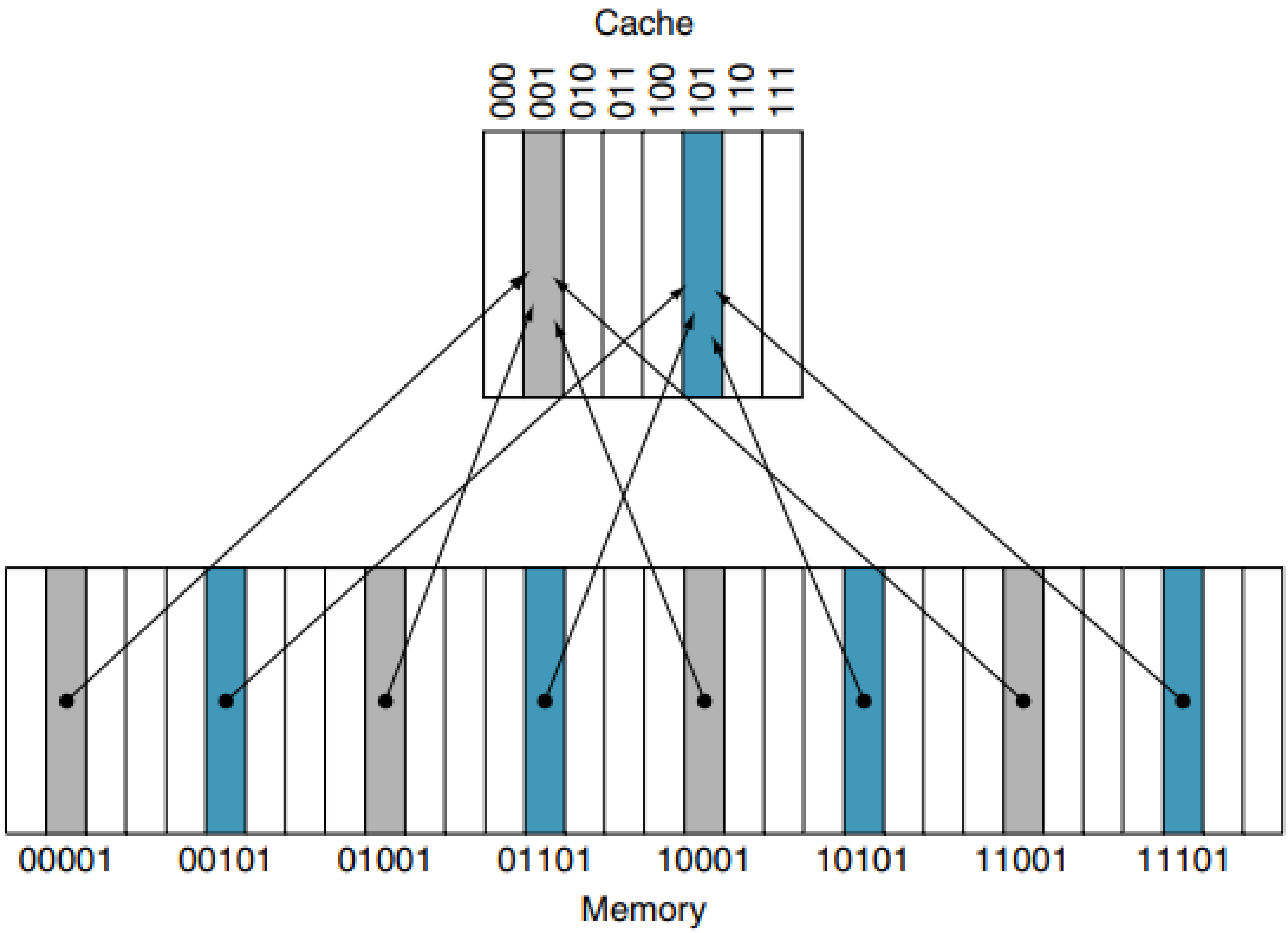
Caches



- How do we know if a data item is in the cache?
- How do we find it?

Direct Mapped Caches

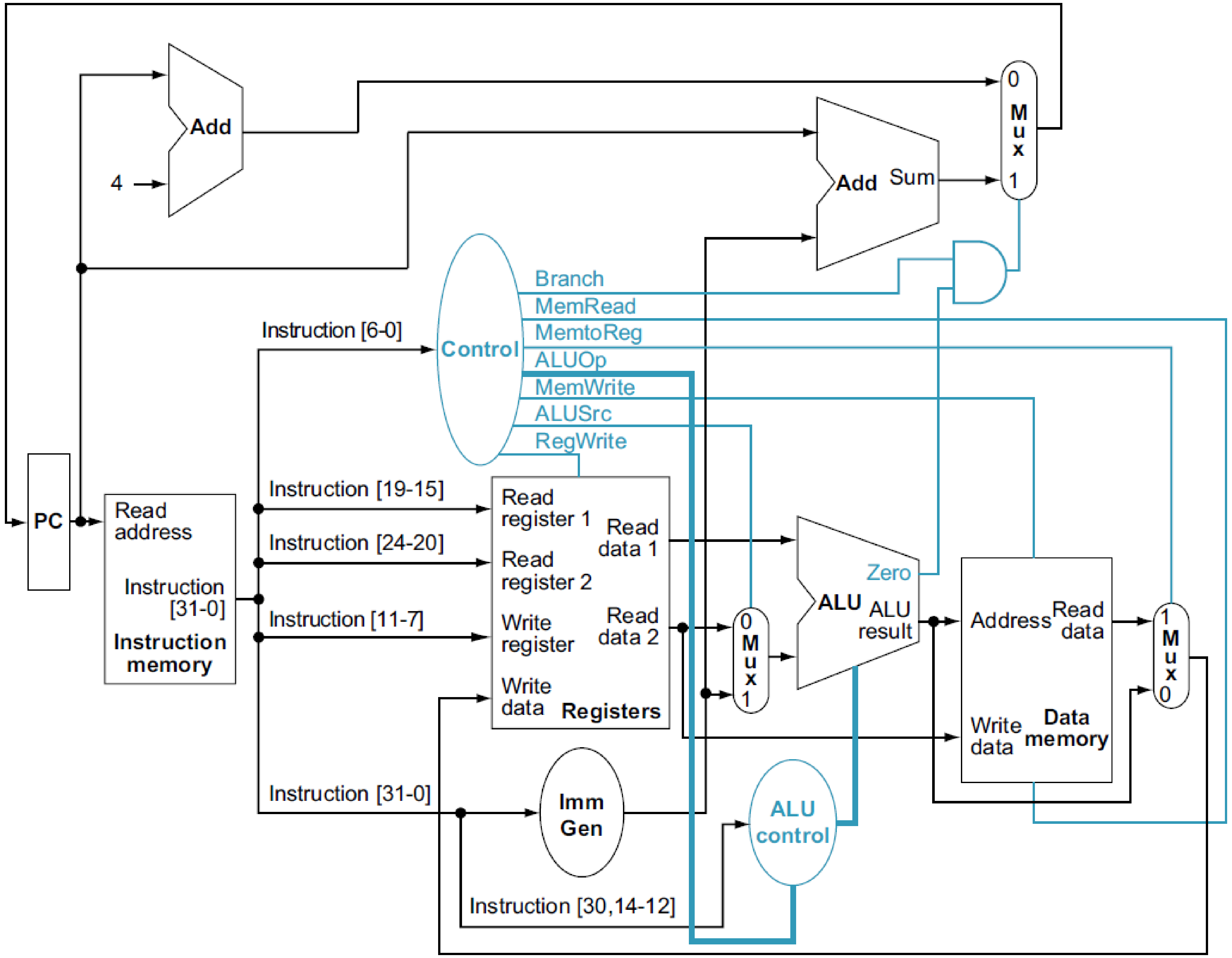
(Block address) modulo (Number of blocks in the cache)



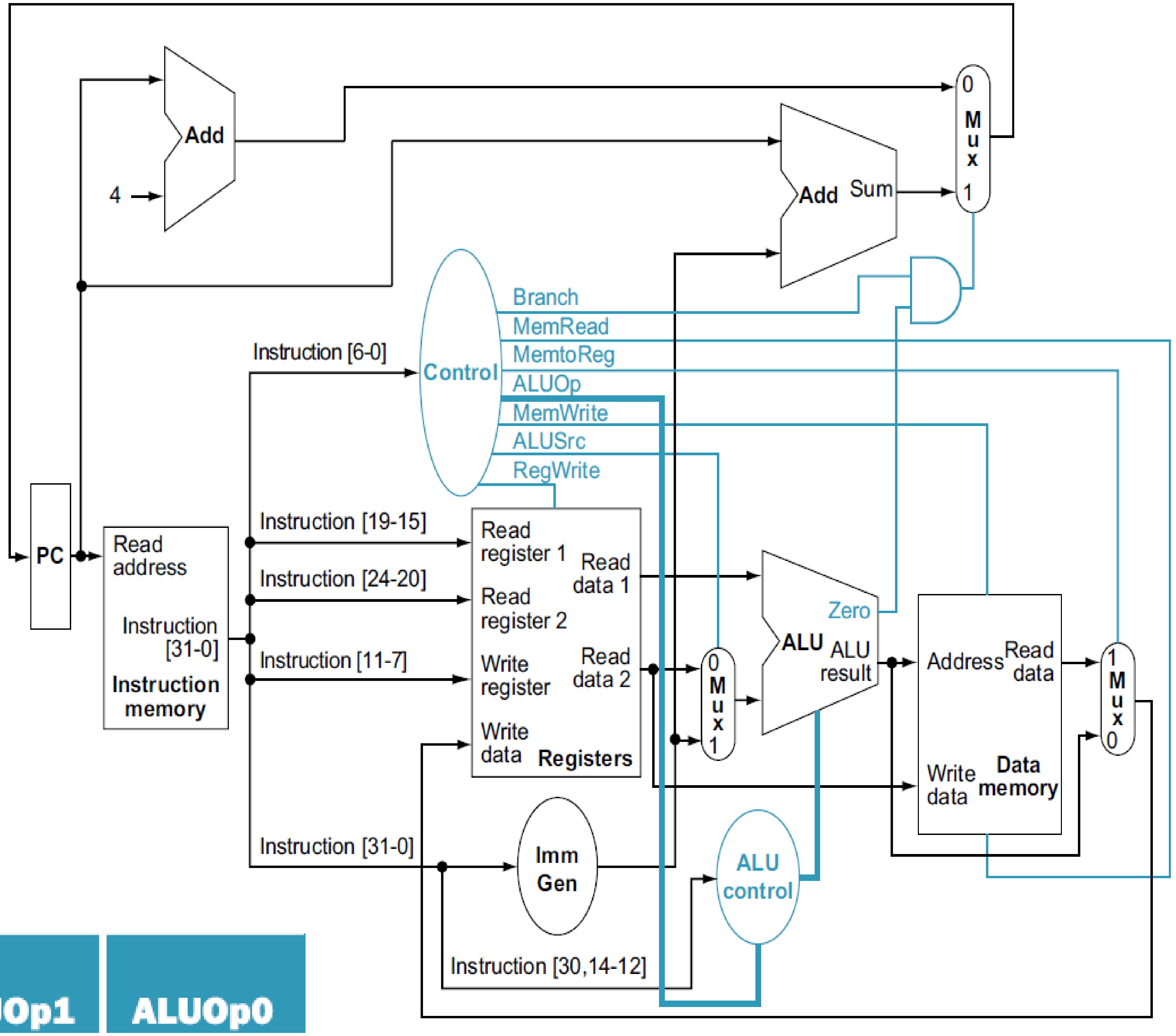
Modelsim – Single Cycle Processor

```
while  (save[i] == k)
        i += 1;
```

RISC V Assembler Code	Address	Instruction					
slli x10,x22,2	80000	*funct7*	*rs2*	*rs1*	*funct3*	*rd*	*opcode*
		0000000	00010	10110	001	01010	0010011
add x10,x10,x25	80004	*funct7*	*rs2*	*rs1*	*funct3*	*rd*	*opcode*
		0000000	11001	01010	000	01010	0110011
lw x9,0(x10)	80008	*imm*		*rs1*	*funct3*	*rd*	*opcode*
		000000000000		01010	011	01001	0000011
bne x9,x24,Exit	80012	*imm 7*	*rs2*	*rs1*	*funct3*	*imm5*	*opcode*
		0000000	11000	01001	001	01100	1100011
addi x22,x22,1	80016	*funct7*	*rs2*	*rs1*	*funct3*	*rd*	*opcode*
		0000000	00001	10110	000	10110	0010011
beq x0,x0,Loop	80020	*imm 7*	*rs2*	*rs1*	*funct3*	*imm5*	*opcode*
		1111111	00000	00000	000	01101	1100011
Exit	80024						



add x10,x10,x25	**80004**	***funct7***	***rs2***	***rs1***	***funct3***	***rd***	***opcode***
		0000000	11001	01010	000	01010	0110011
lw x9,0(x10)	**80008**	***imm***		***rs1***	***funct3***	***rd***	***opcode***
		000000000000		01010	011	01001	0000011
bne x9,x24,Exit	**80012**	***imm 7***	***rs2***	***rs1***	***funct3***	***imm5***	***opcode***
		0000000	11000	01001	001	01100	1100011
addi x22,x22,1	**80016**	***funct7***	***rs2***	***rs1***	***funct3***	***rd***	***opcode***
		0000000	00001	10110	000	10110	0010011



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