Chapter 4

Dr. Md Abu Sayeed
EET 340

Creating a Datapath

Datapath element: A unit used to operate on or hold data within a processor. In the LEGv8 implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders

Program counter (PC): The register containing the address of the instruction in the program being executed

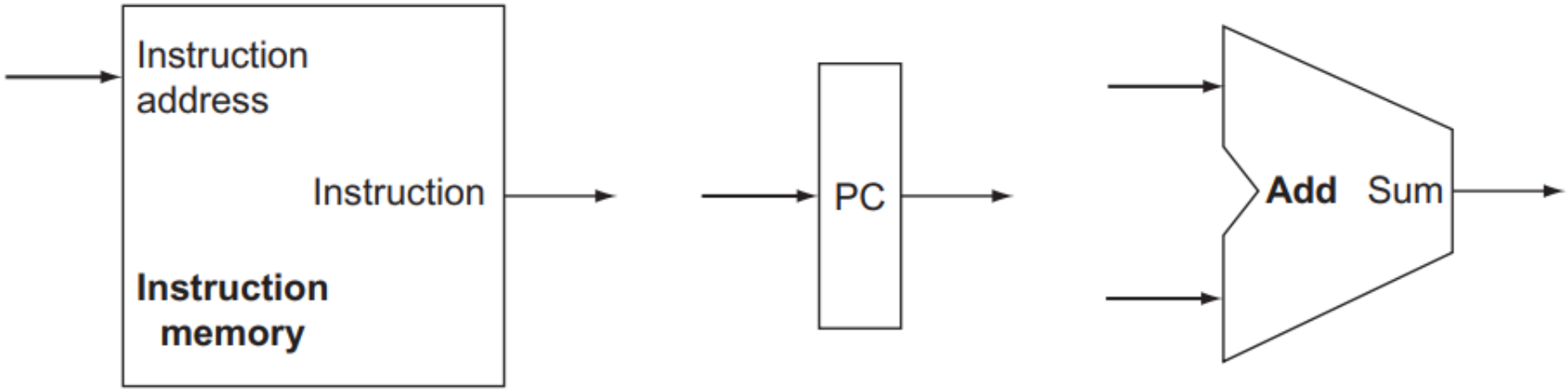
Register file: A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.

Creating a Datapath

Program counter (PC): A register that contains the address of the current instruction.

Instruction memory: It is the memory that instructions are fetched from, and data memory is the memory where the data is written to and read from.

Adder: An adder is a digital circuit that performs addition of numbers.



a. Instruction memory b. Program counter c. Adder

Instruction Fetch and incrementing Program Counter

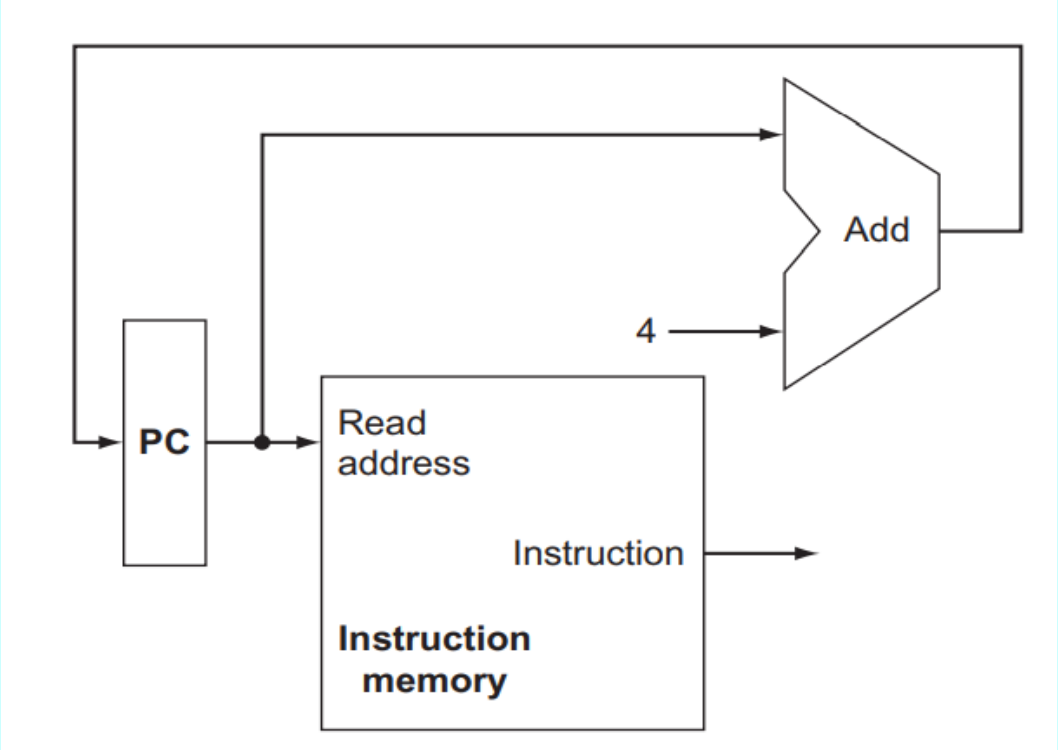
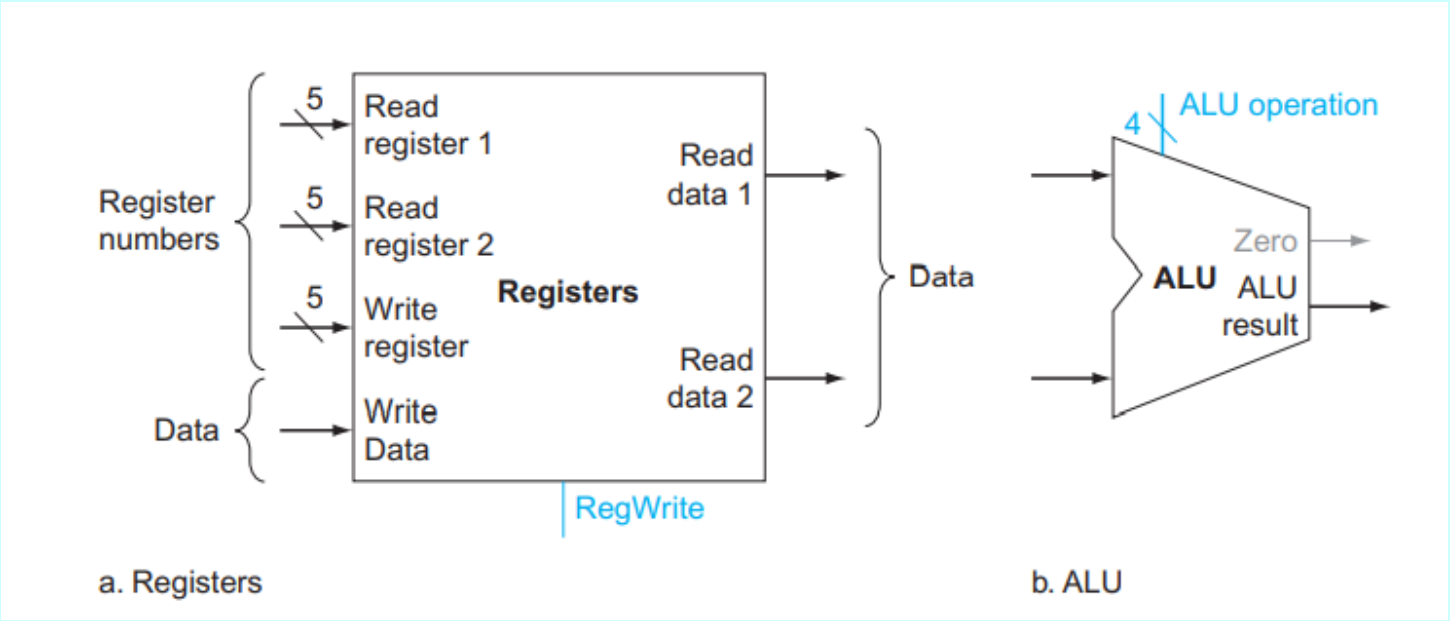


Figure. A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

- To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later.

Register file: The processor's 32 general-purpose registers are stored in a structure called a register file. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the computer

ALU: In computing, an arithmetic logic unit (**ALU**) is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers.



R Format Instructions.

- We will need to read two data words from the register file and write one data word into the register file for each instruction.
- For each data word to be read from the registers, we need an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers.
- To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the data to be written into the register. The register file always The register file always outputs the contents of whatever register numbers are on the Read register inputs.

ALUSrc and ALUop

How ALU Control bits are set depends on the ALUop control bits

Instruction	ALUOp	Instruction operation	Opcode field	Desired ALU action	ALU control input
LDUR	00	load register	XXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXX	pass input b	0111
R-type	10	ADD	10001011000	add	0010
R-type	10	SUB	11001011000	subtract	0110
R-type	10	AND	10001010000	AND	0000
R-type	10	ORR	10101010000	OR	0001

Control Signal

Signal name	Effect when deasserted	Effect when asserted
Reg2Loc	The register number for Read register 2 comes from the Rm field (bits 20:16).	The register number for Read register 2 comes from the Rt field (bits 4:0).
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, lower 16 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.

Reg2Loc =

Unconditional Branch =

MemRead =

MemtoReg =

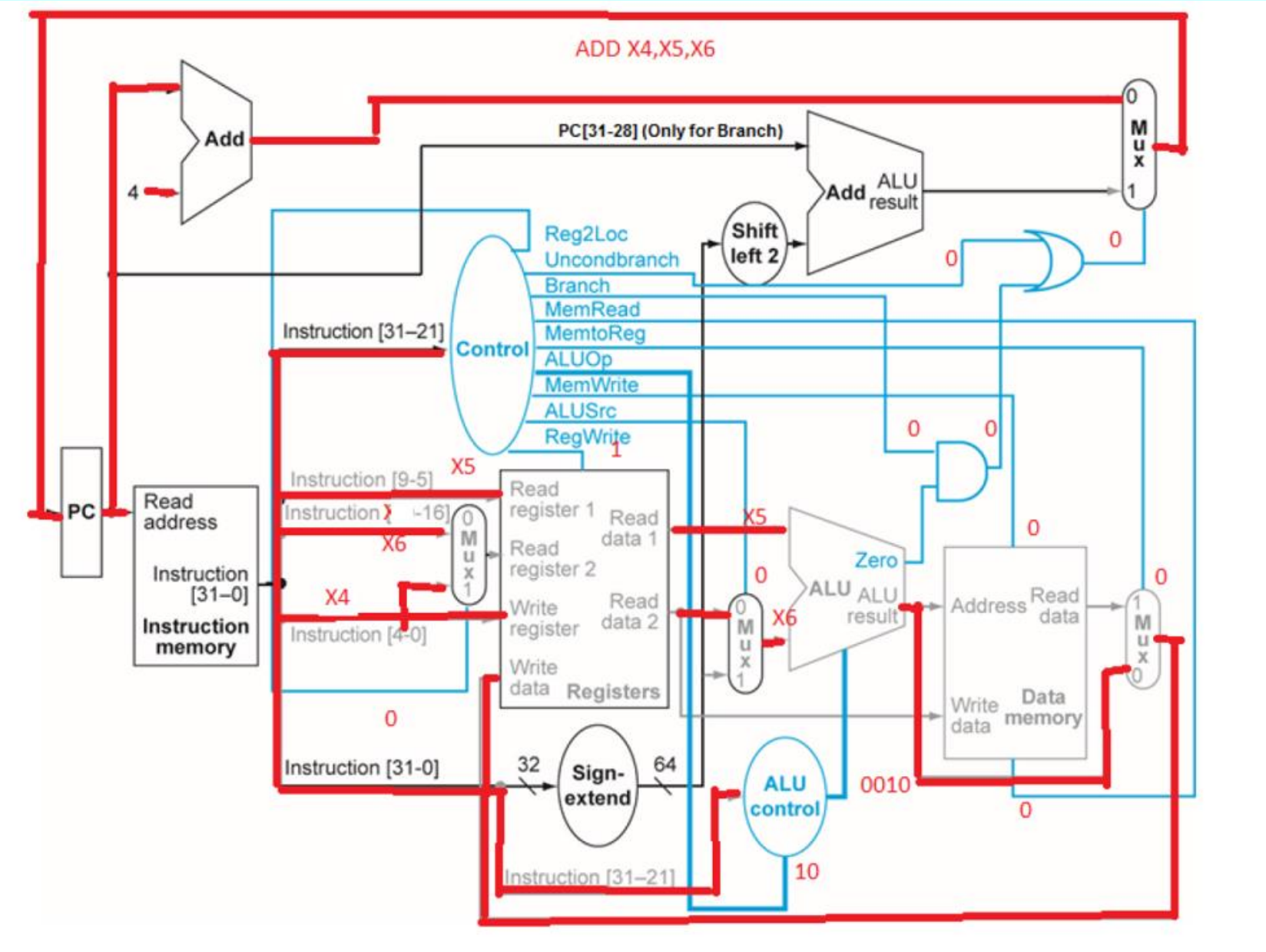
ALUOp =

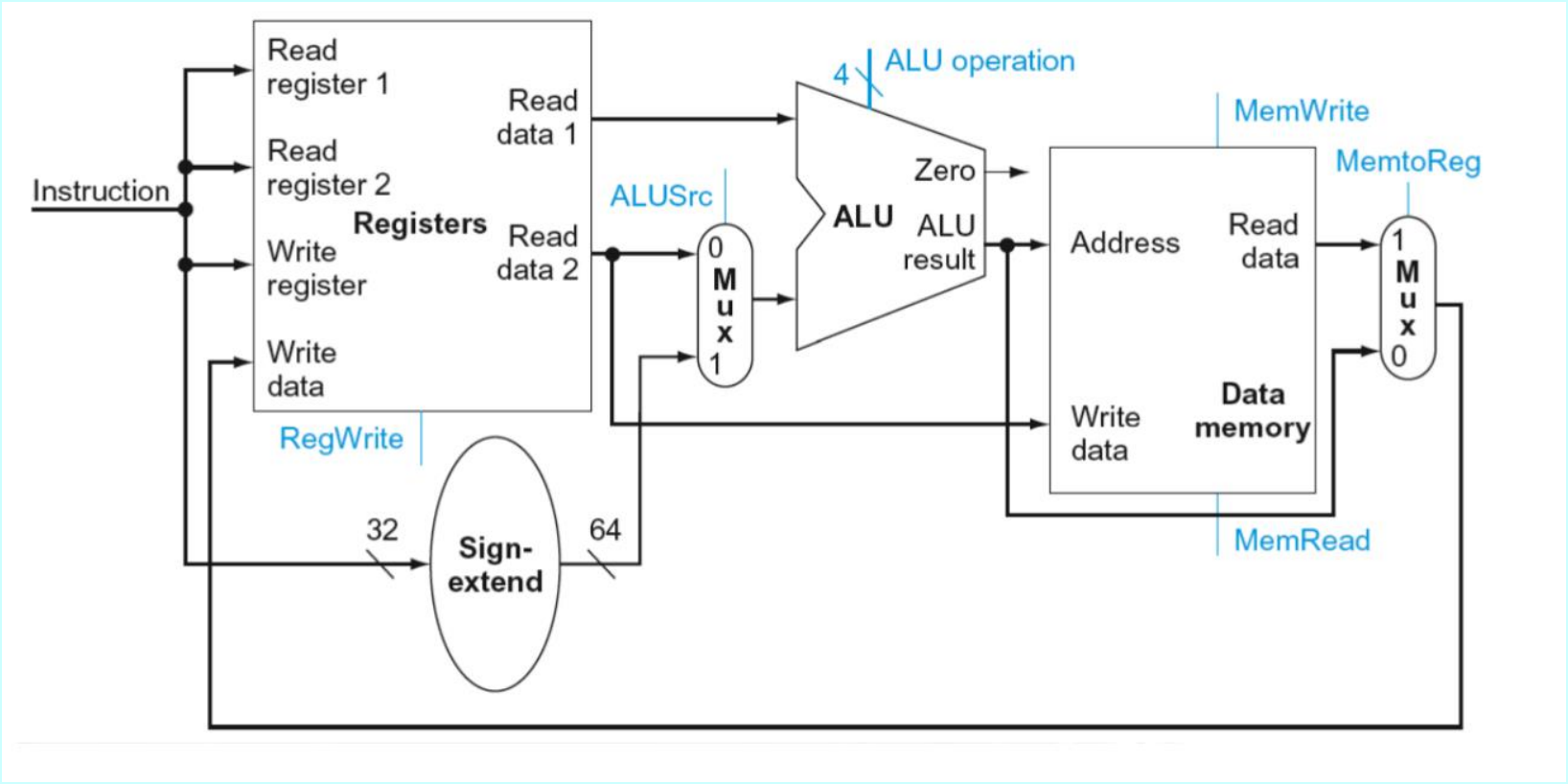
MemWrite =

ALUSrc =

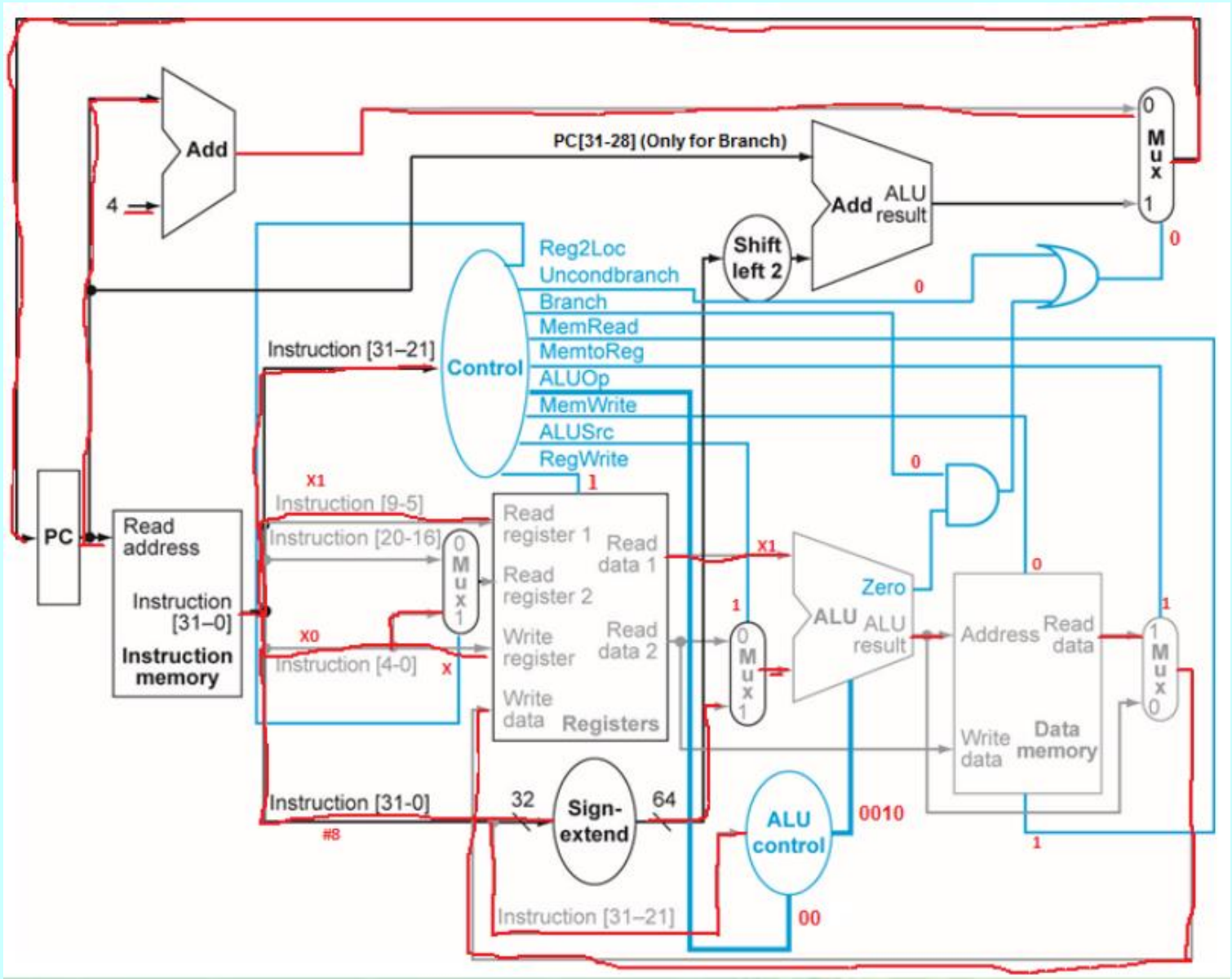
RegWrite =

Tracing Datapath: ADD X4, X5, X6




Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
RegWrite
Instruction
ALUSrc
0
M
u
x
1
4
ALU operation
ALU
Zero
ALU result
MemWrite
MemtoReg
Address
Read data
Write data
Data memory
MemRead
1
M
u
x
0
32
Sign-extend
64

Tracing Datapath: LDUR X0, [X1, #8]



Pipelined Datapath

Pipelining is an implementation technique in which multiple instructions are overlapped in execution.

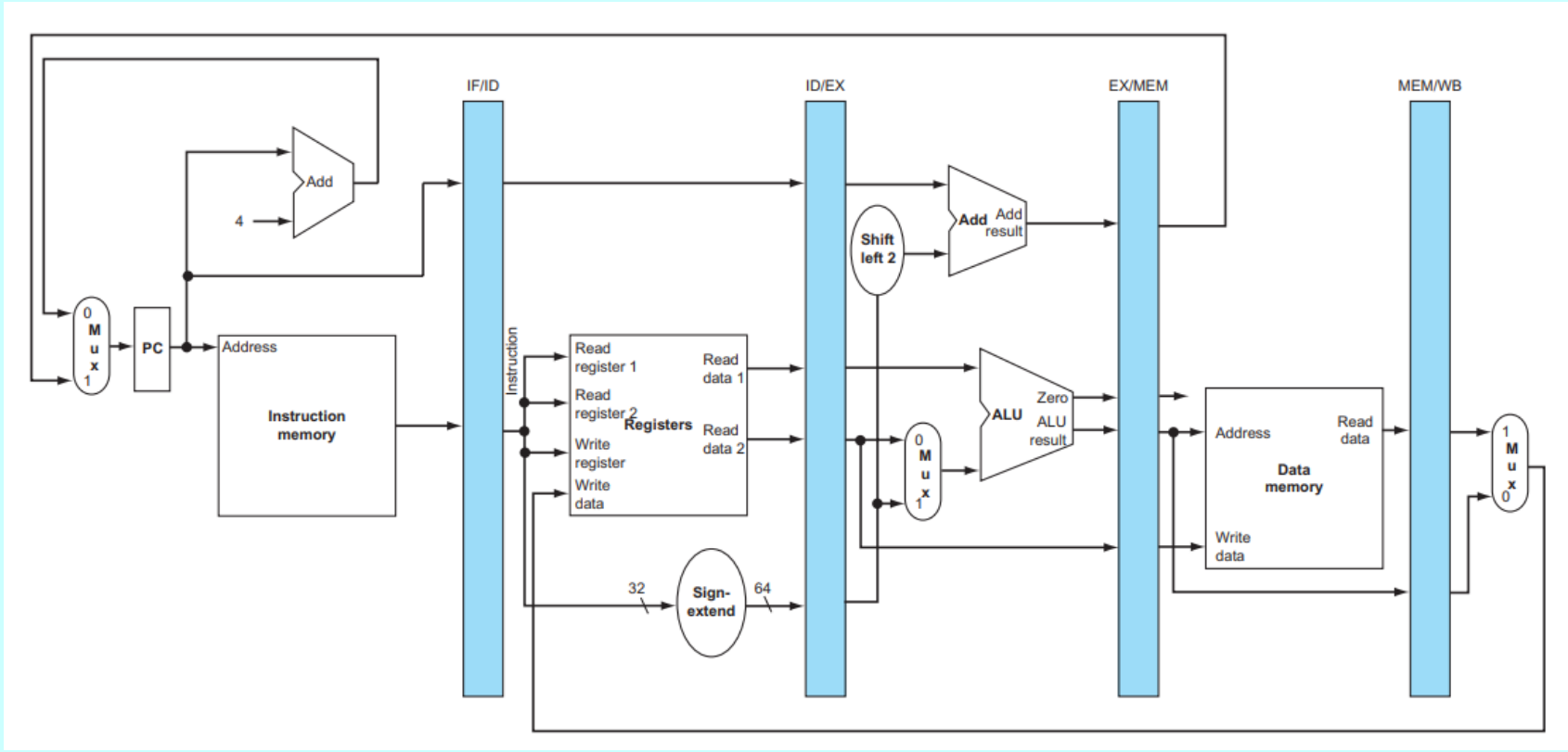


Figure. The pipelined version of the datapath

The Division of instruction into five stages means five five-stage pipeline.

IF: Instruction Fetch

ID: Instruction Decode

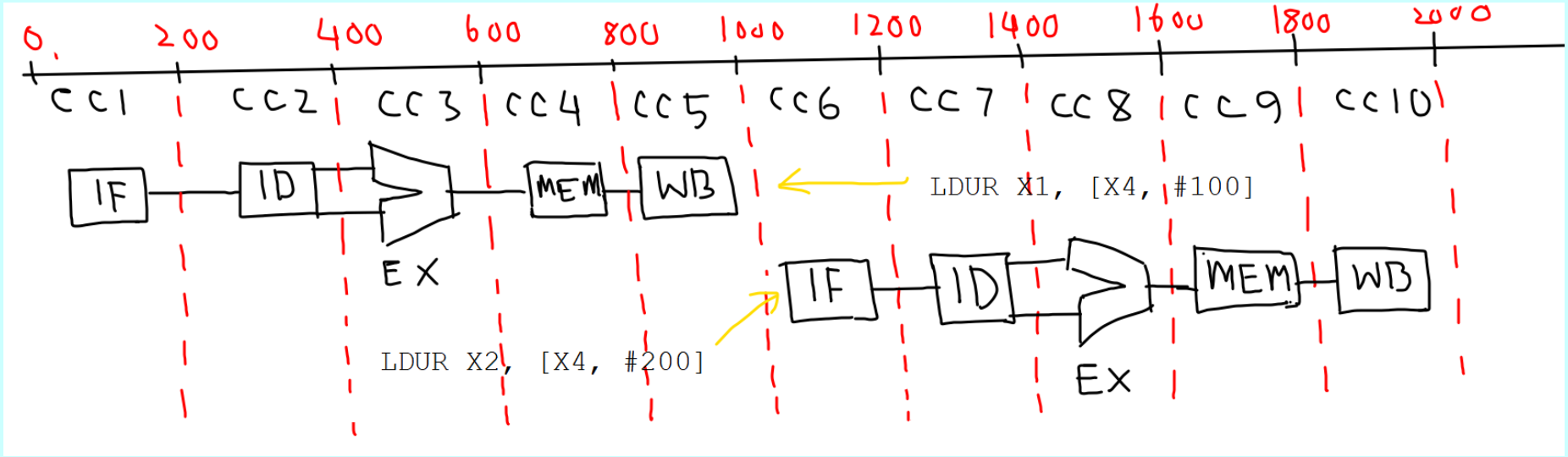
EX: Execute/Address calculation

MEM: Memory Access

WB: Write-back

Example: Draw the multicycle Non-pipelined diagram for the following instructions. Calculate total execution time. Consider each stage takes one clock cycle which is 200ps.

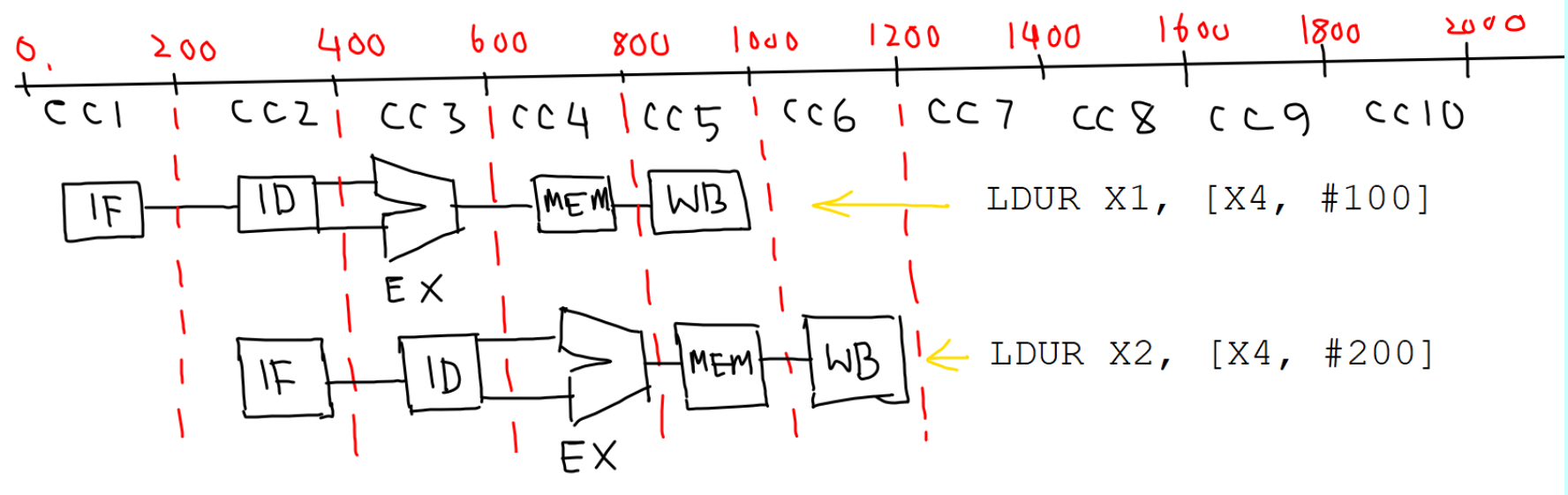
```
LDUR X1, [X4, #100]
LDUR X2, [X4, #200]
```



Total Execution Time = 200 x 10 = 2000 ps

Example: Draw the multicycle pipelined diagram for the following instructions. Calculate total execution time. Consider each stage takes one clock cycle which is 200ps.

```
LDUR X1, [X4, #100]
LDUR X2, [X4, #200]
```



Total clock cycle required = 2 x 200 = 1200 PS

Pipeline Hazards: There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards, and there are three different types.

Data Hazards: When a planned instruction cannot execute in the proper clock cycle because data that are needed to execute the instruction are not yet available.

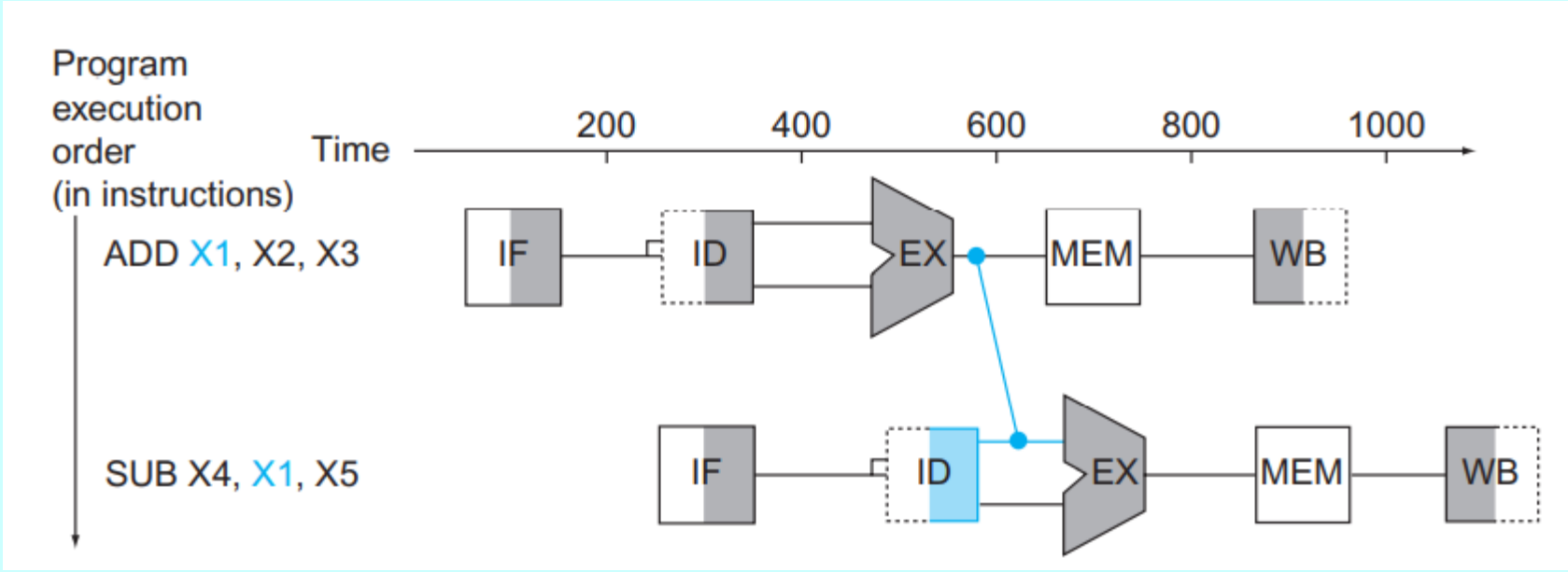
```
ADD  X19, X0, X1
SUB  X2, X19, X3
```

Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.

Solution for Hazards:

Forwarding: A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer visible registers or memory.

Forwarding paths are valid only if the destination stage is later in time than the source stage. For example, there cannot be a valid forwarding path from the output of the memory access stage in the first instruction to the input of the execution stage of the following,



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 pipeline stall is a delay in execution of an instruction in order to resolve a hazard. A stall initiated in order to resolve a hazard.

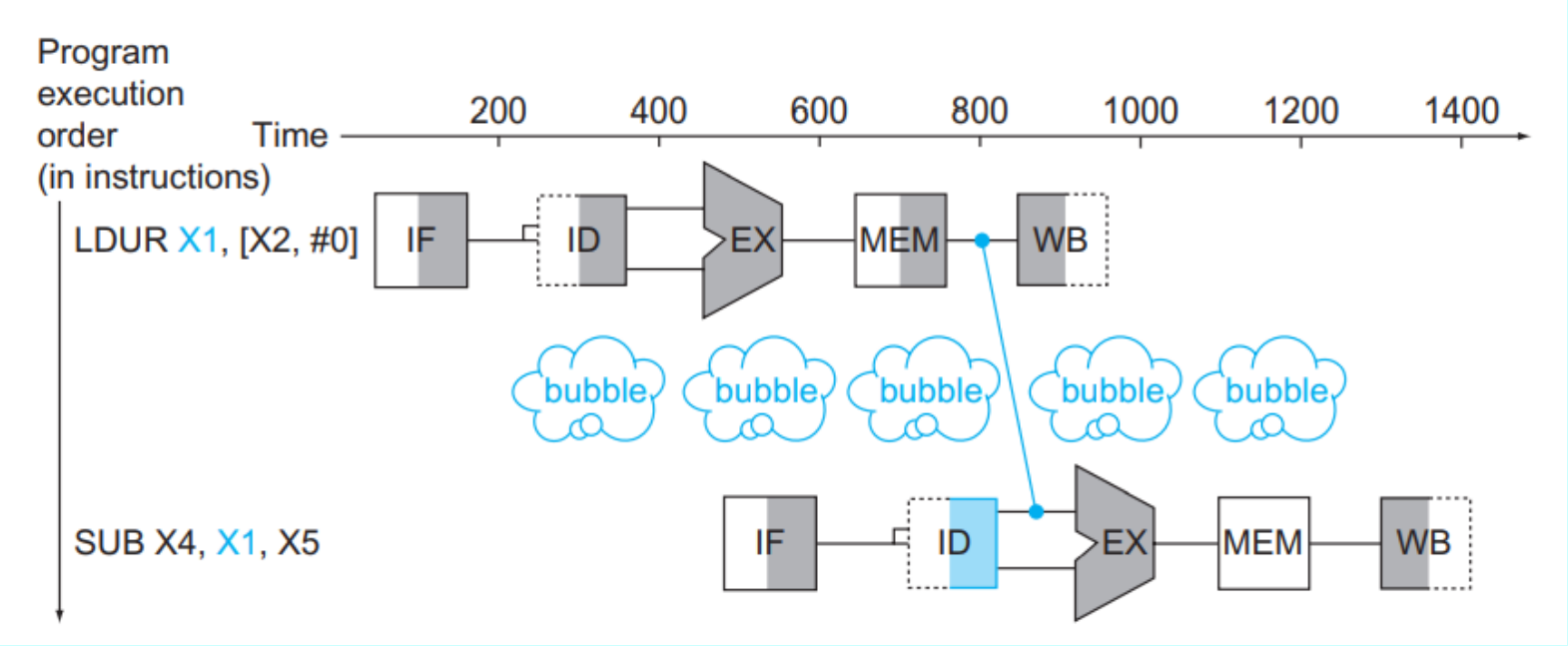


FIGURE. We need a stall even with forwarding when an R-format instruction following a load tries to use the data.

Problem:

a. Identify Data hazards and determine total number of stalls in the assembly code. Assume there is no forwarding. Rewrite the assembly code and show stalls in the code if any. Draw the multicycle pipeline diagram.

b. Apply forwarding to reduce the stalls without changing the functionality and order. Rewrite the assembly code.

c. Draw the multi-cycle pipeline diagram (one loop) for the optimized assembly code from section b.

```
ADD X1, X2, X3
ADD X4, X1, X5
```

(a) Identification of Data Hazard/ Stalls

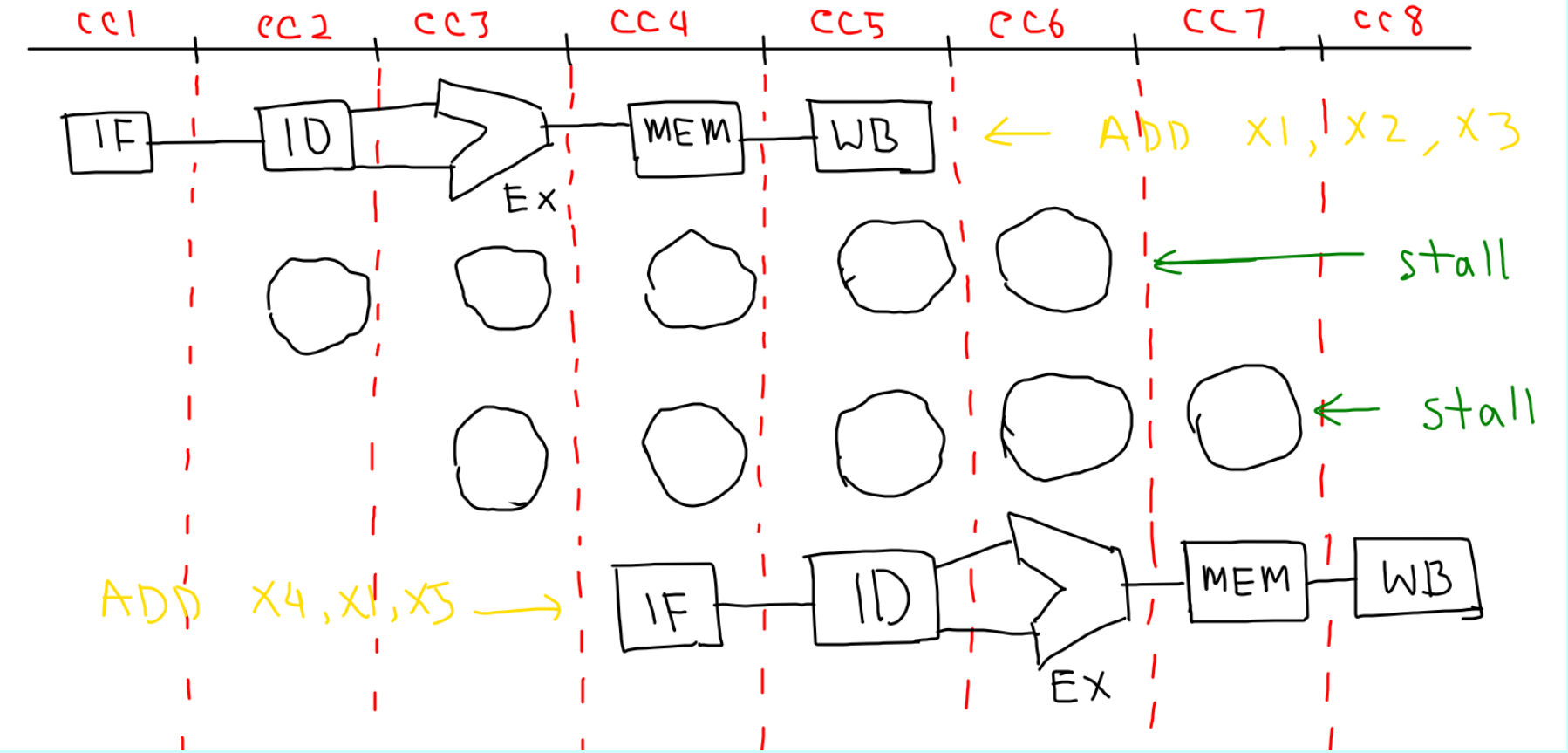
```
ADD X1, X2, X3
ADD X4, X1, X5   → Data Hazard
```

Two stalls are required to resolve a data hazard. Total stalls= 2

Rewriting the assembly code and showing the stalls (No forwarding)

```
ADD X1, X2, X3
Stall
stall
ADD X4, X1, X5
```

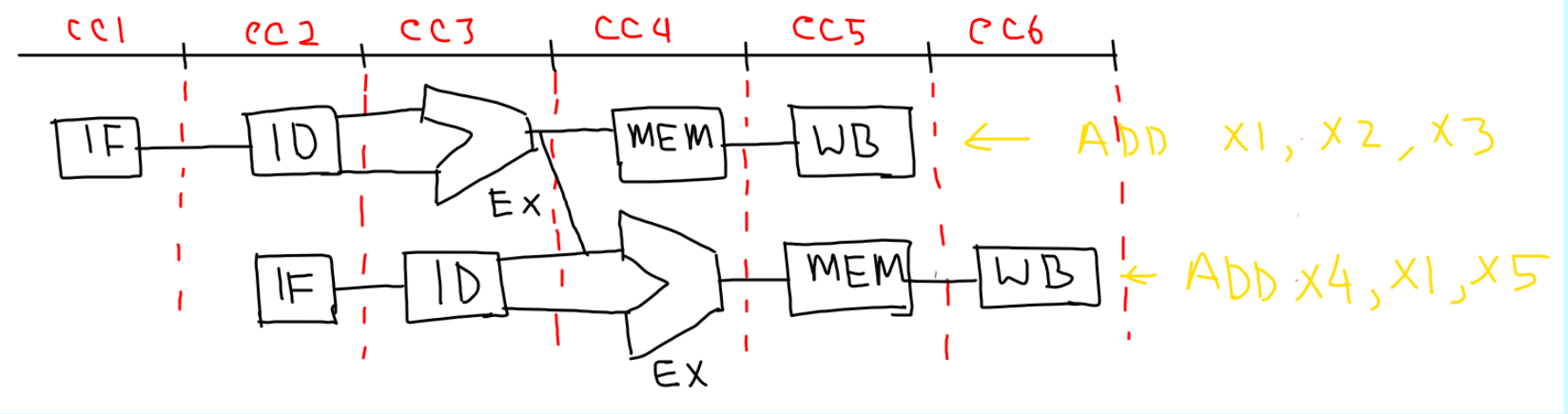
Multicycle pipeline Diagram (No forwarding)



(b) Rewriting the assembly code and showing the stalls (with forwarding)

```
ADD X1, X2, X3
ADD X4, X1, X5
```

(c) Multicycle pipeline diagram for b



Problem:

a. Identify Data hazards and determine total number of stalls in the assembly code. Assume there is no forwarding. Rewrite the assembly code and show stalls in the code if any.

b. Apply forwarding to reduce the stalls without changing the functionality and order. Rewrite the assembly code.

c. Draw the multi-cycle pipeline diagram (one loop) for the optimized assembly code from section b.

```
SUBI X2, X2, #1
LSL X4, X3, #1
SUB X5, X4, X6
LDUR X7, [X0, #0]
ADDI X7, X7, #1
```

(a) Identification of Data Hazard/ Stalls

```
SUBI X2, X2, #1
LSL X4, X3, #1      → Data hazard
SUB X5, X4, X6
LDUR X7, [X0, #0]
ADDI X7, X7, #1     → Data Hazard
```

Two stalls are required to resolve 1 Data hazard. Total stalls= 2+2=4

Rewriting the assembly code and showing the stalls (No forwarding)

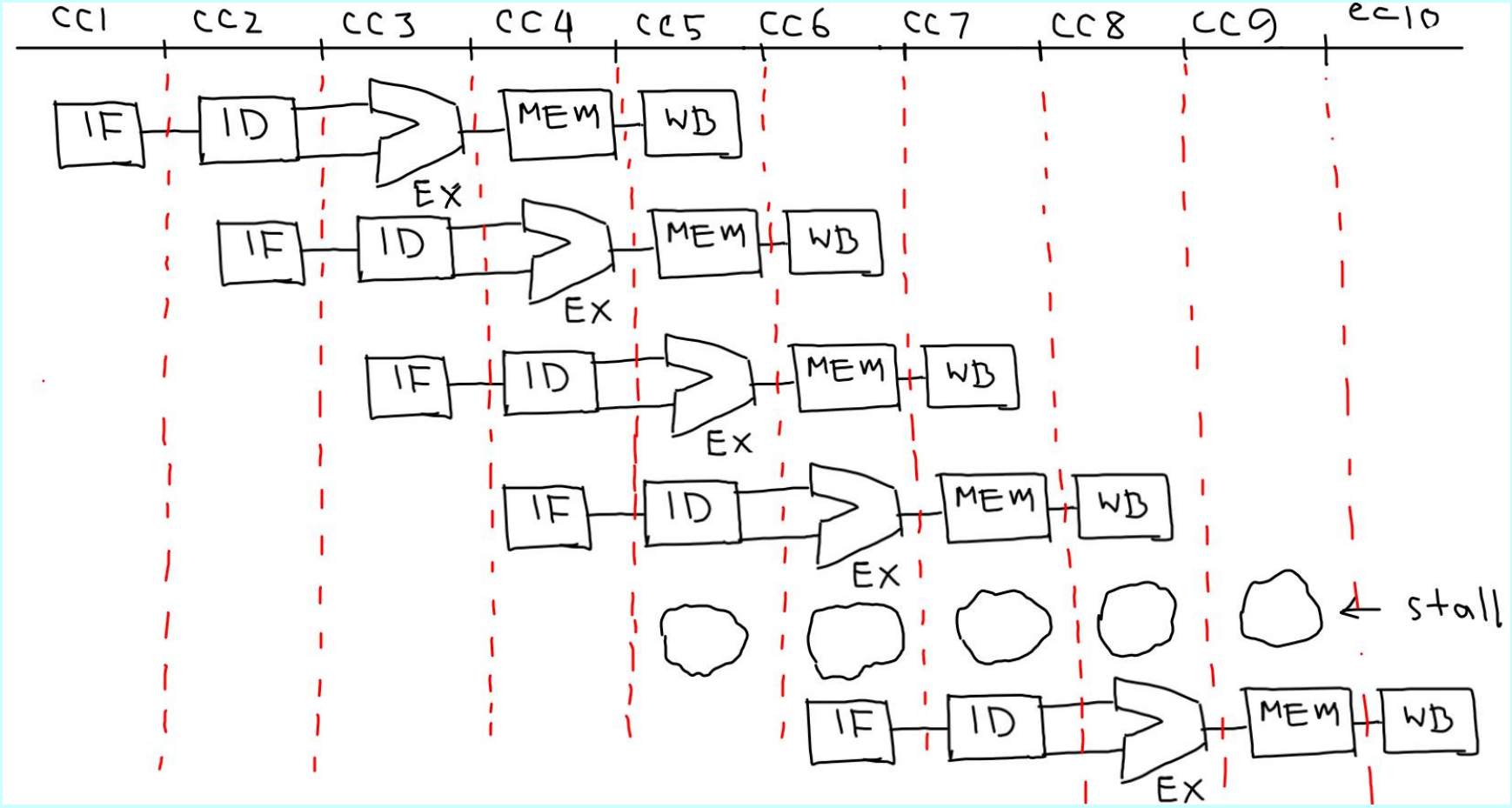
```
SUBI X2, X2, #1
LSL X4, X3, #1
Stall
stall
SUB X5, X4, X6
LDUR X7, [X0, #0]
Stall
stall
ADDI X7, X7, #1
```

(b) Rewriting the assembly code (With forwarding)

```
SUBI X2, X2, #1
LSL X4, X3, #1
SUB X5, X4, X6
LDUR X7, [X0, #0]
stall
ADDI X7, X7, #1
```

Total stall = 1

(c) Multicycle pipeline diagram for section b.



Note: The figures, text etc included in slides are borrowed from books, other sources for academic purpose only. The author does not claim any originality.

Source:

1.Computer Organization and Design (ARM Edition) by David A. Patterson