Pipelined Datapaths

The single-cycle datapath has some problems

- Mainly comprised of combinatorial logic
  - This means each element has a propagation delay.
  - The addition of propagation delays reduces the clock rate possible.
- The clock rate is determined by the **worst case delay**.
  - This means the single-cycle datapath violates the rule "Make the common case fast"
- Can we improve the throughput of the datapath?
- Can we increase the clock rate possible at the same time?
- Can we make the common cast fast?

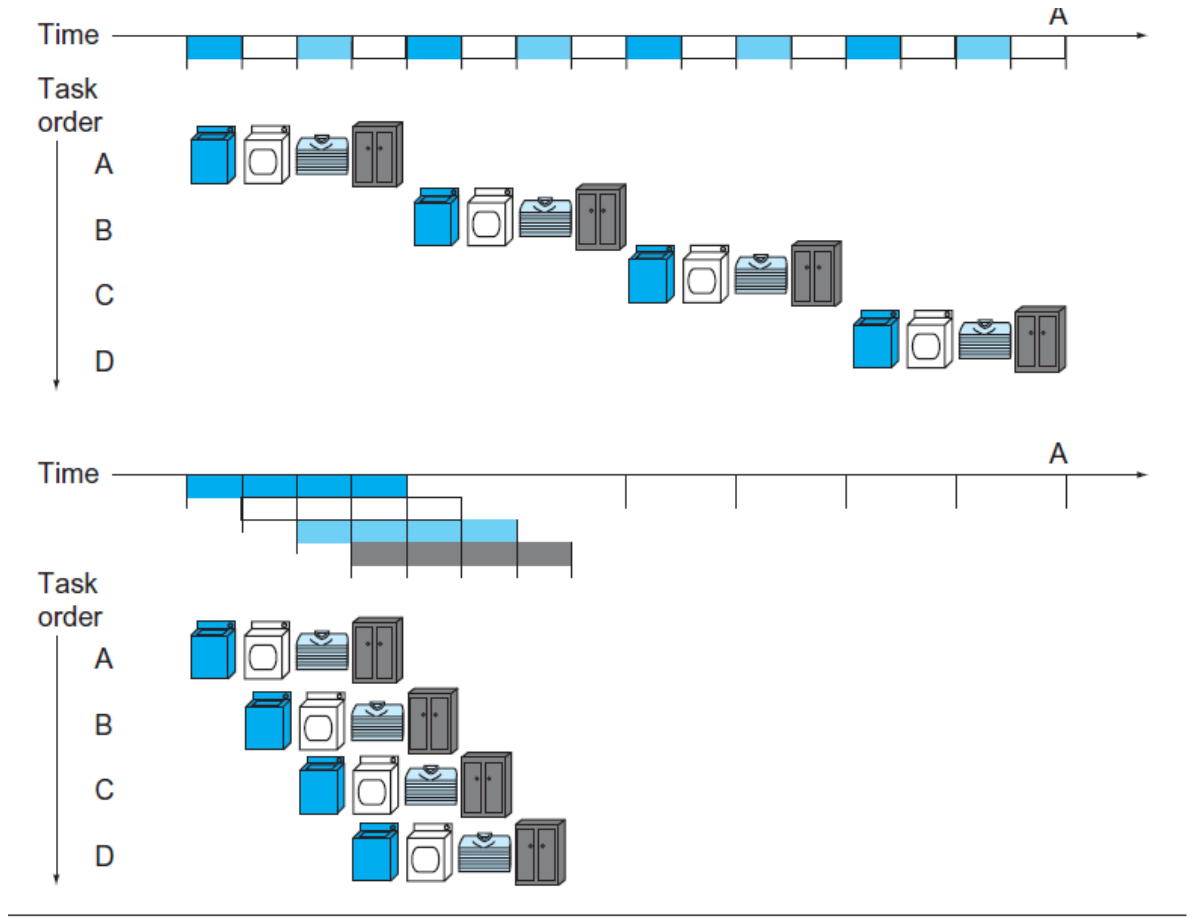
A pipeline

- The single cycle datapath

Must complete each task before another begins!

- A pipelined datapath

As soon as one element of the four datapath elements is complete we can reuse it on a new instruction.

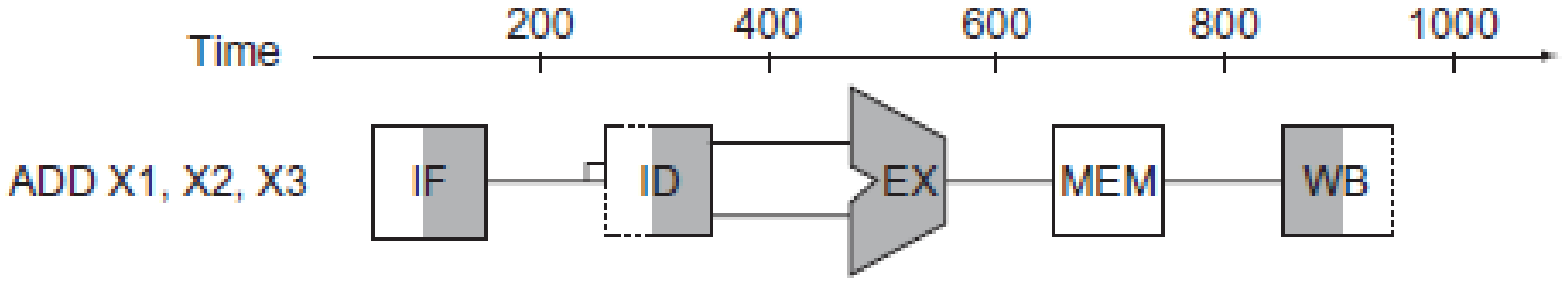


A LEGv8 pipelined datapath

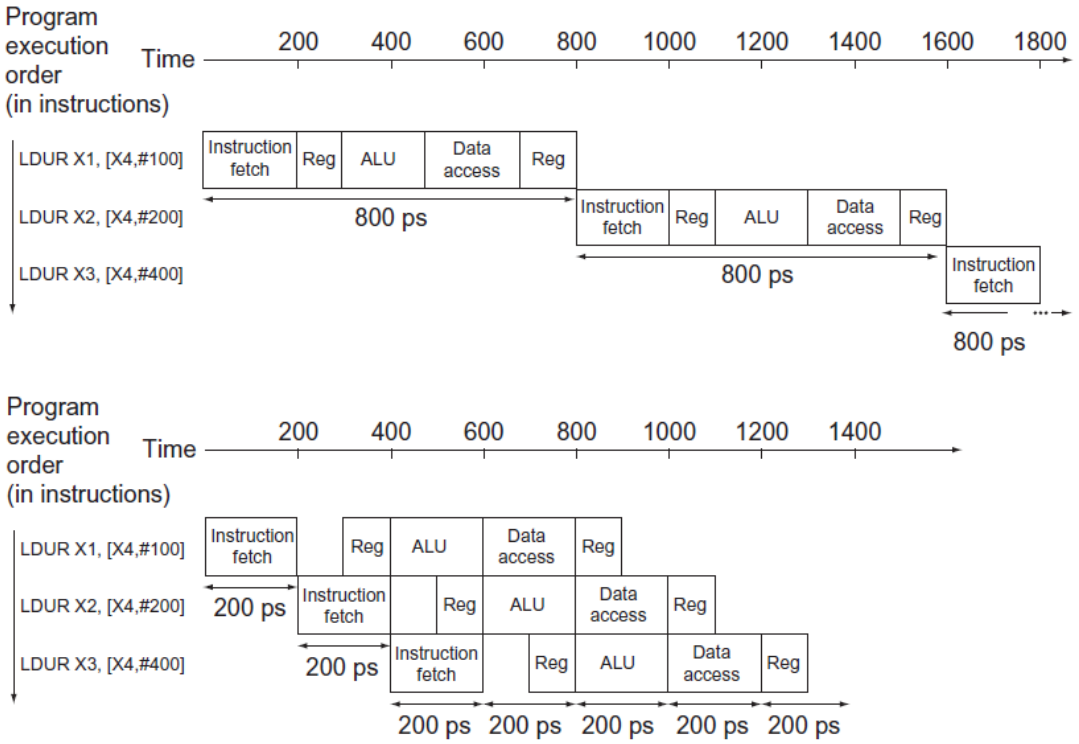
LEGv8 instructions we have looked at previously take 5 steps.

- Fetch instructions from memory.
- Read registers and decode the instruction
- Execute the operation or calculate an address
- Access an operand in data memory
- Write the result into a register

Let's split up the single-cycle datapath into **five stages** to speed up execution.



What does this gain us?



Pipelining improves performance by increasing instruction **throughput**

A program can execute billions of instructions; therefore, **throughput** is an important metric.

What are the disadvantages (trade-offs)

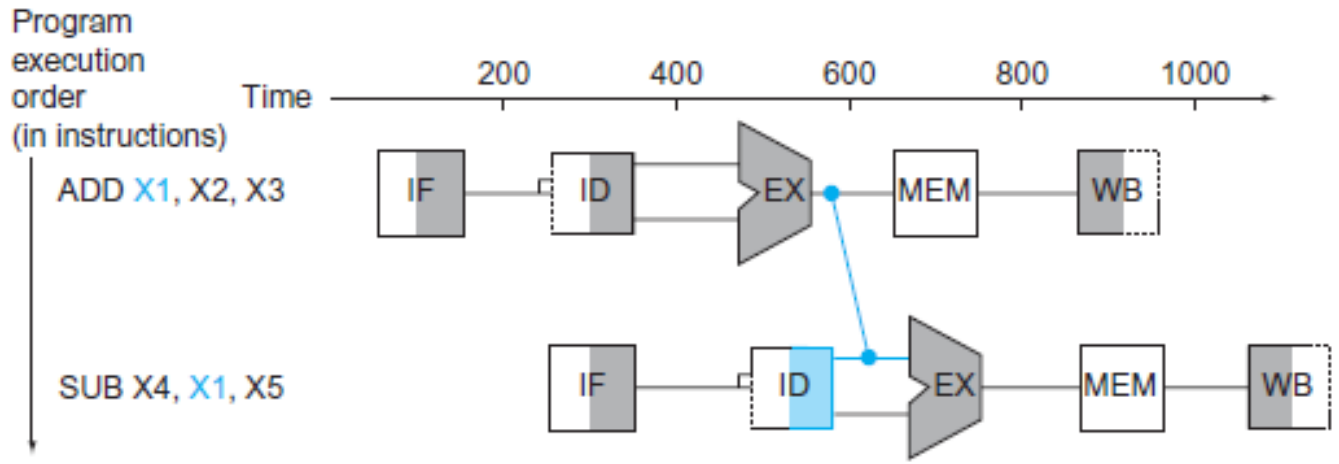
Pipeline Hazards

- Structural Hazard
  The datapath design doesn’t allow simultaneous execution of different instructions.
- Data Hazard
  The pipeline must **stall**(wait) for a step in the previous instruction to complete (It requires that data).
- Control Hazard
  The pipeline must decide what instruction to execute based on a previous instruction (A conditional branch).

Resolving data hazards

Forwarding

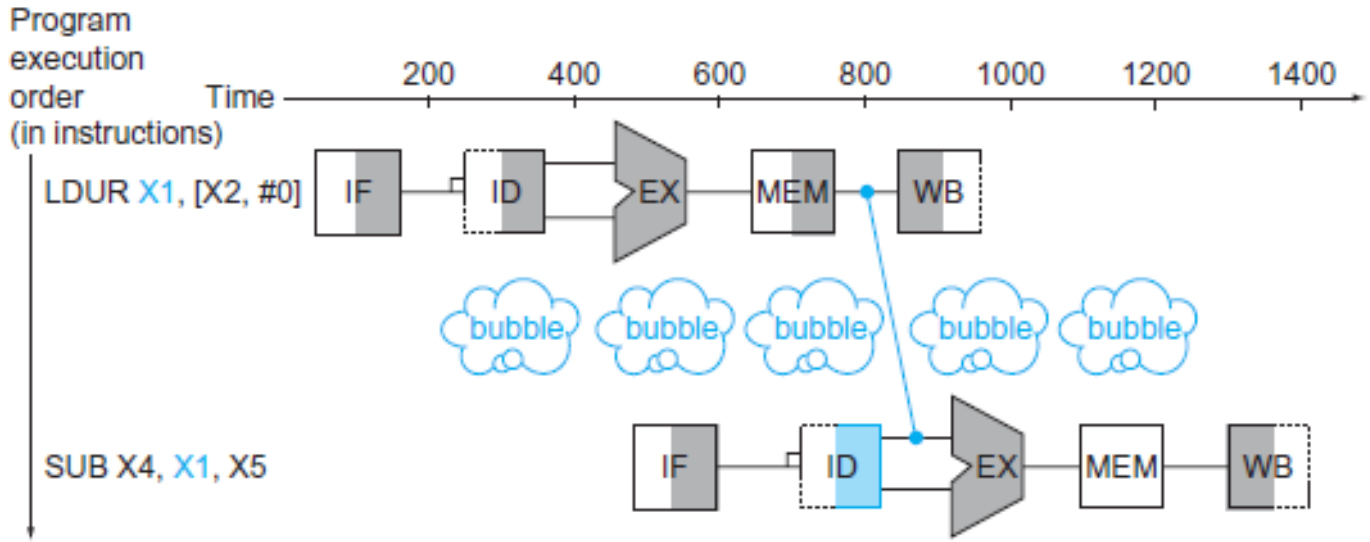
Without intervention if a register is required in the subsequent instruction we must wait for the previous instruction to complete.



Forwarding is the addition of extra hardware allowing us to bypass some stages and access the interim result.

Resolving data hazards

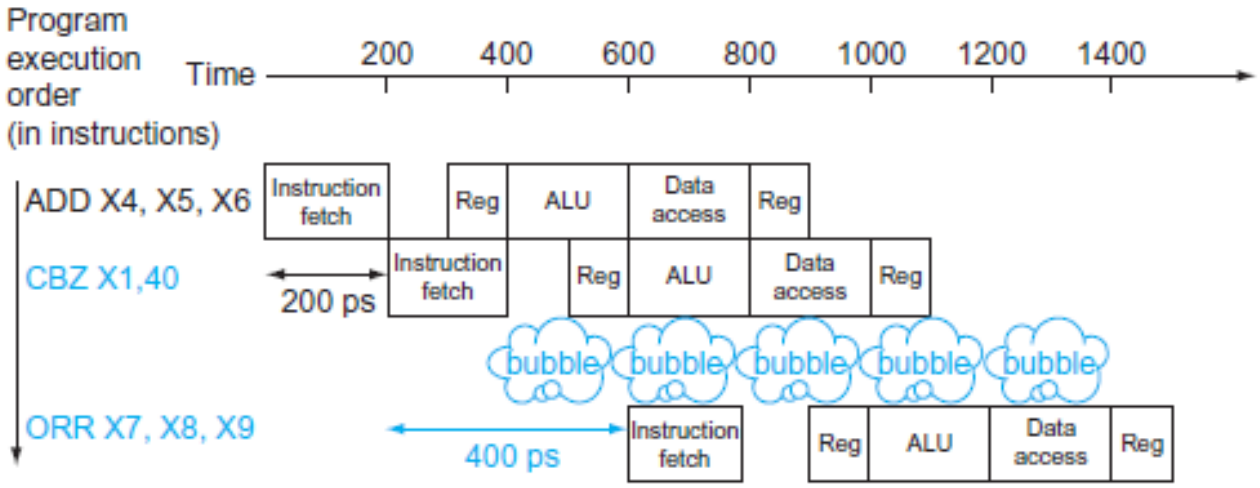
Forwarding
Forwarding isn't perfect, sometimes even the interim result occurs after the subsequent instruction requires it.



Here we must insert a **stall**, to allow X1 to be used in the SUB instruction.

Resolving control hazards

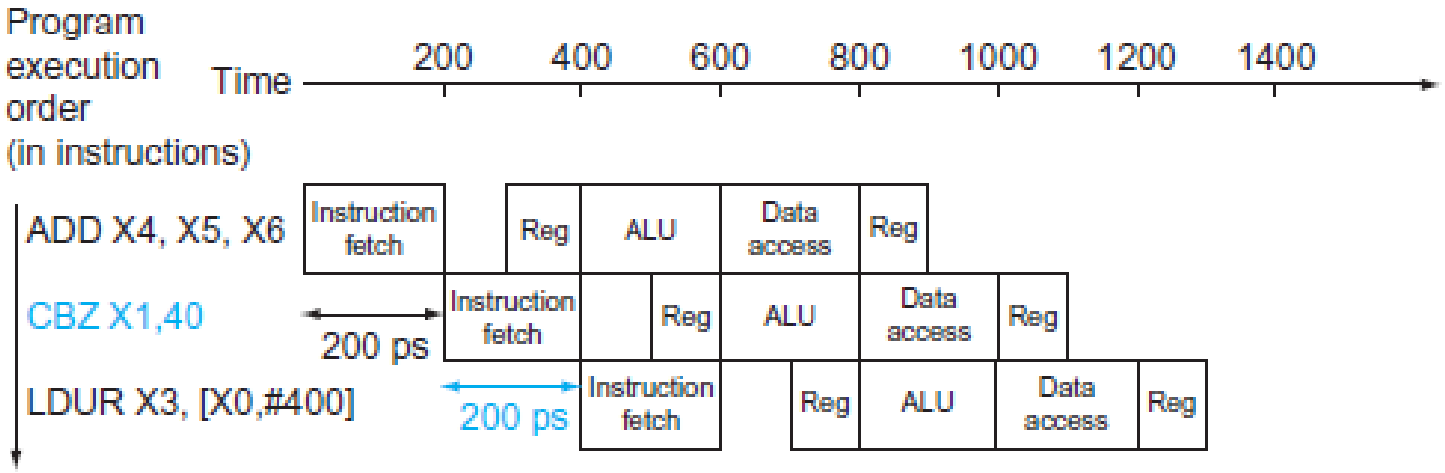
There are two solutions to control hazards.
1) Stall on Branch
As we don't know what the next instruction will be after a branch, we could just **stall** until we determine the outcome of the branch.



This however introduces additional delay into the pipeline.

Resolving control hazards

2) Branch Prediction
Another method of resolving control hazards is to predict which branch will be taken. One method is to just **predict** that conditional branches will always be ignored.

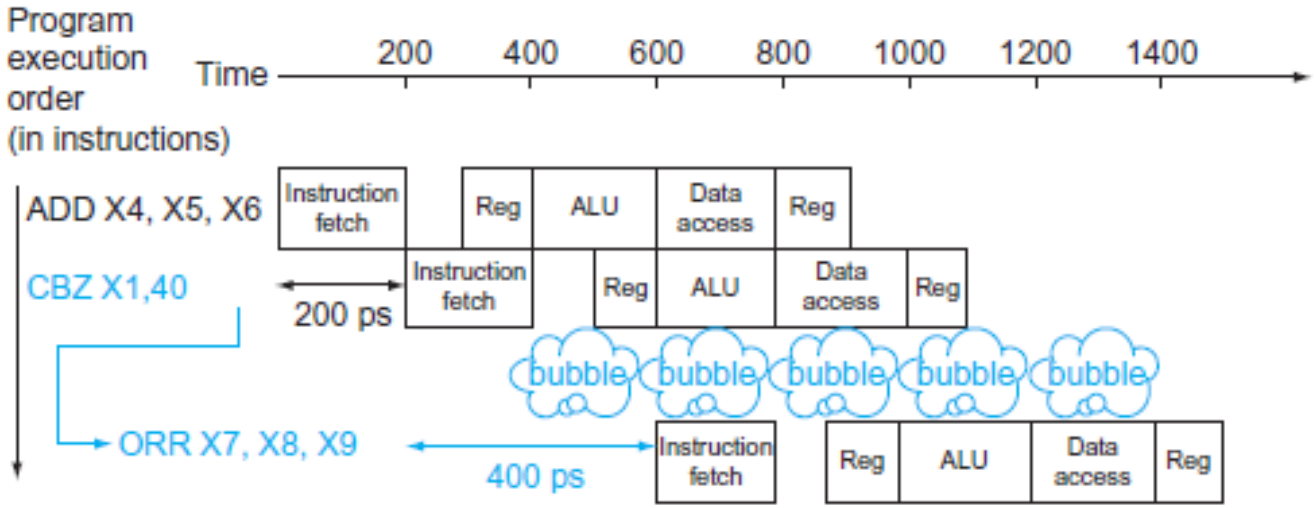


If the branch isn't taken, then we just continue with our pipeline with no delay.

Resolving control hazards

2) Branch Prediction

However, if the branch is taken…



We now need to abort the predicted instruction and begin executing the branched instruction.

Summary

Pipelining exploits **parallelism** between instructions to improve the **throughput** of a sequential instruction stream.

For each code sequence below, will they stall? Can we avoid stalling using forwarding?

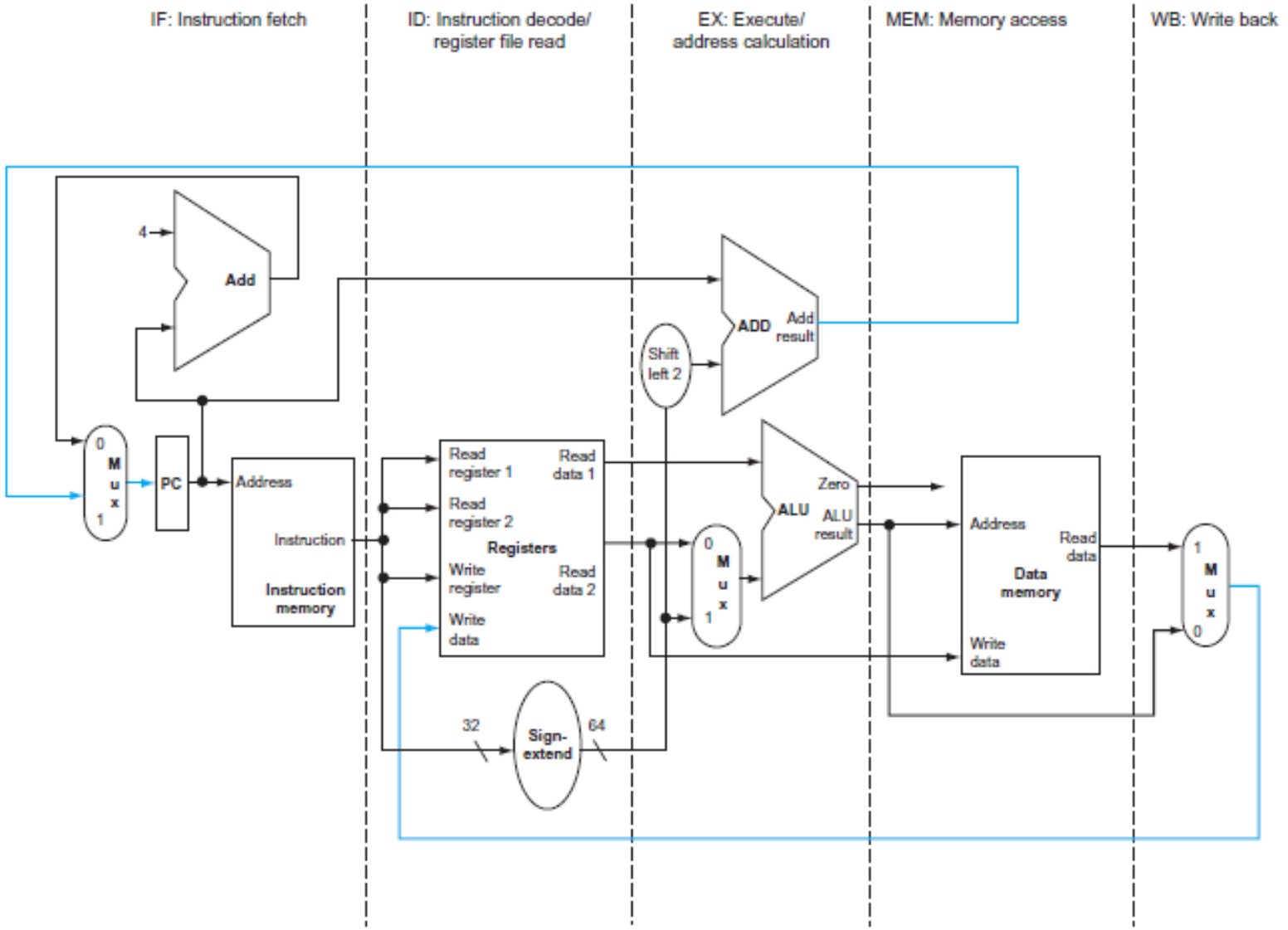
Sequence 1	Sequence 2	Sequence 3
LDUR X0, [X0,#0]	ADD X1, X0, X0	ADDI X1, X0, #1
ADD X1, X0, X0	ADDI X2, X0, #5	ADDI X2, X0, #2
	ADDI X4, X1, #5	ADDI X3, X0, #3
		ADDI X4, X0, #4
		ADDI X5, X0, #5

The pipelined datapath

Let’s split the single cycle datapath into five stages and give each stage a name.

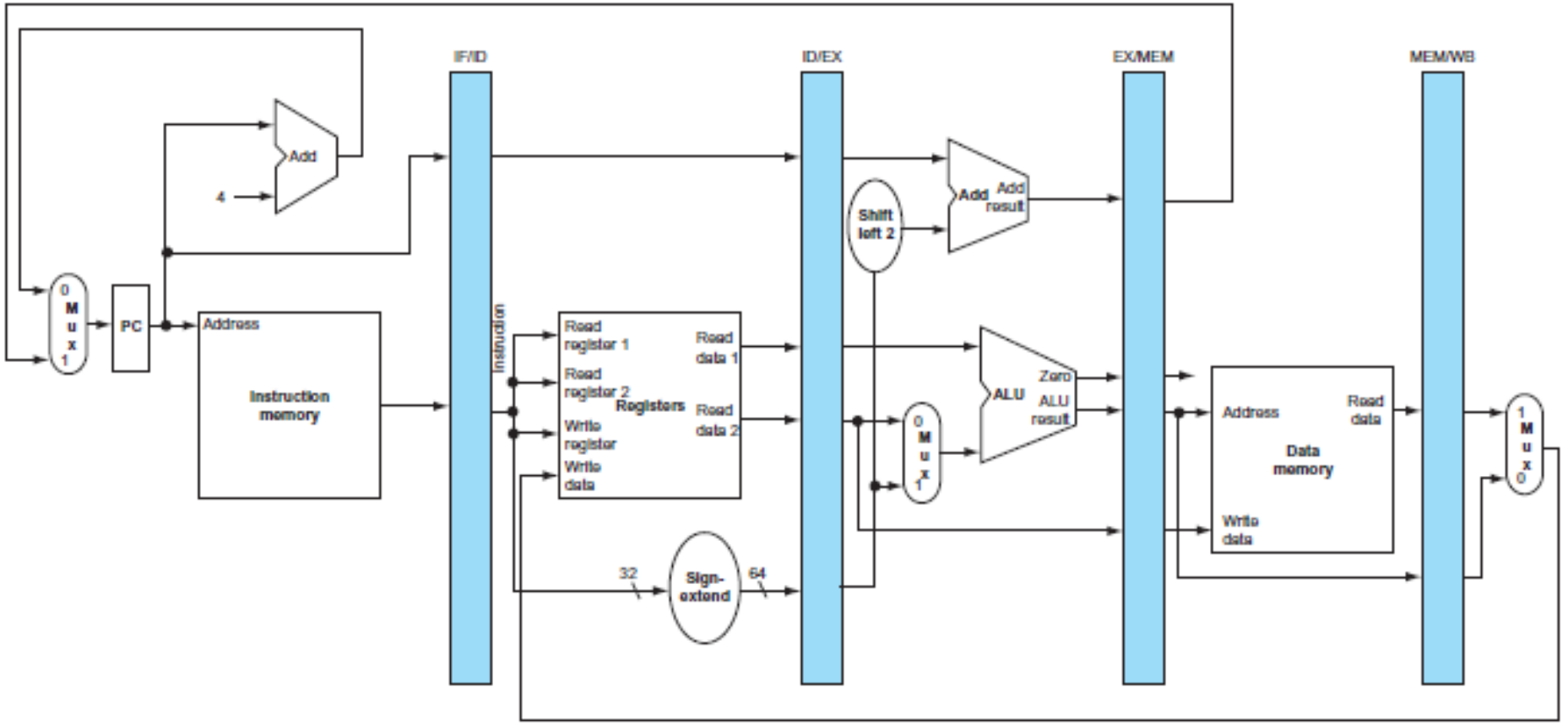
1. IF: Instruction Fetch
2. ID: Instruction Decode (and register file read)
3. EX: Execution (or address calculation
4. MEM: data Memory access
5. WB: Write Back

The pipelined datapath

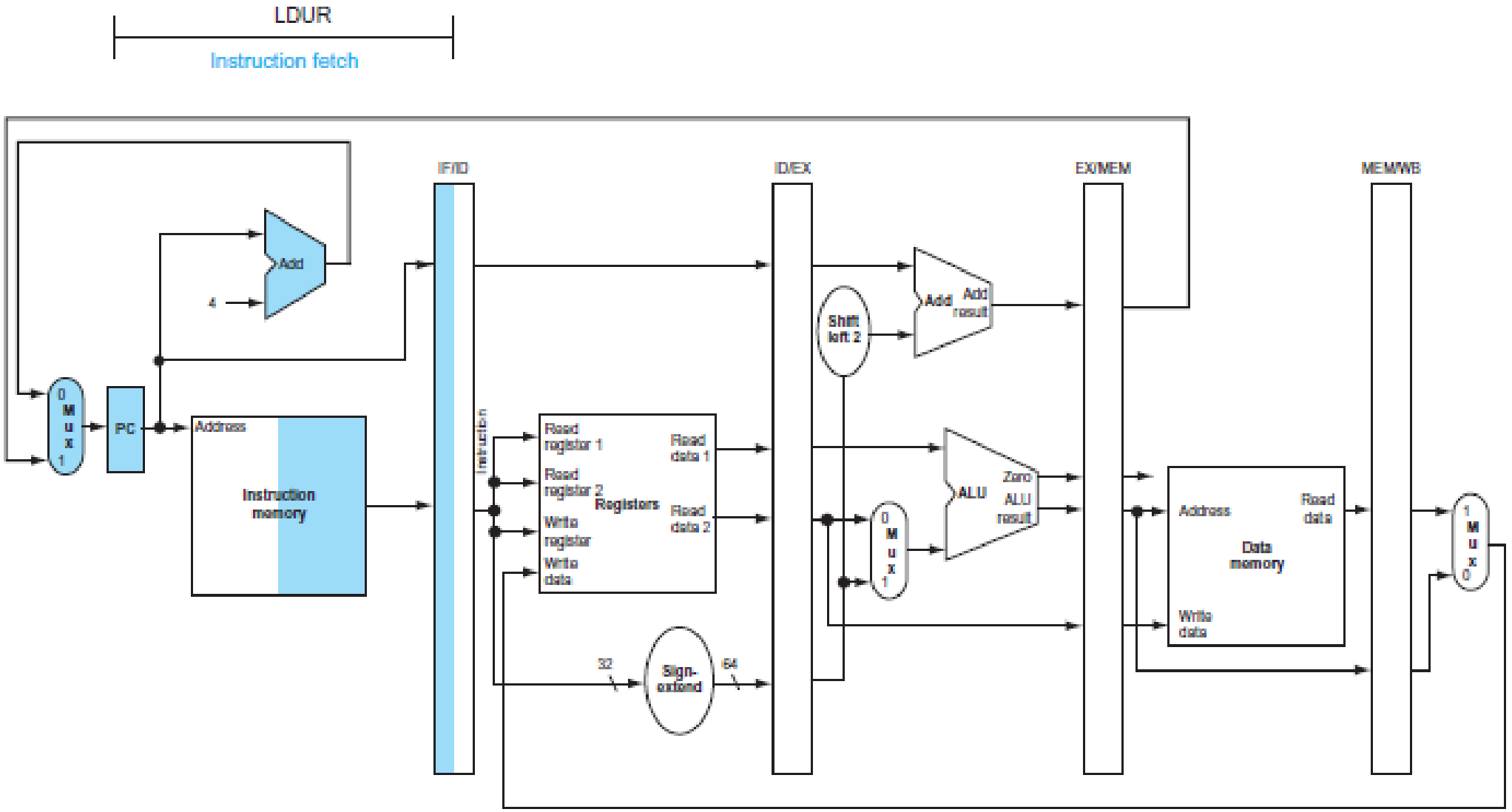


The pipelined datapath

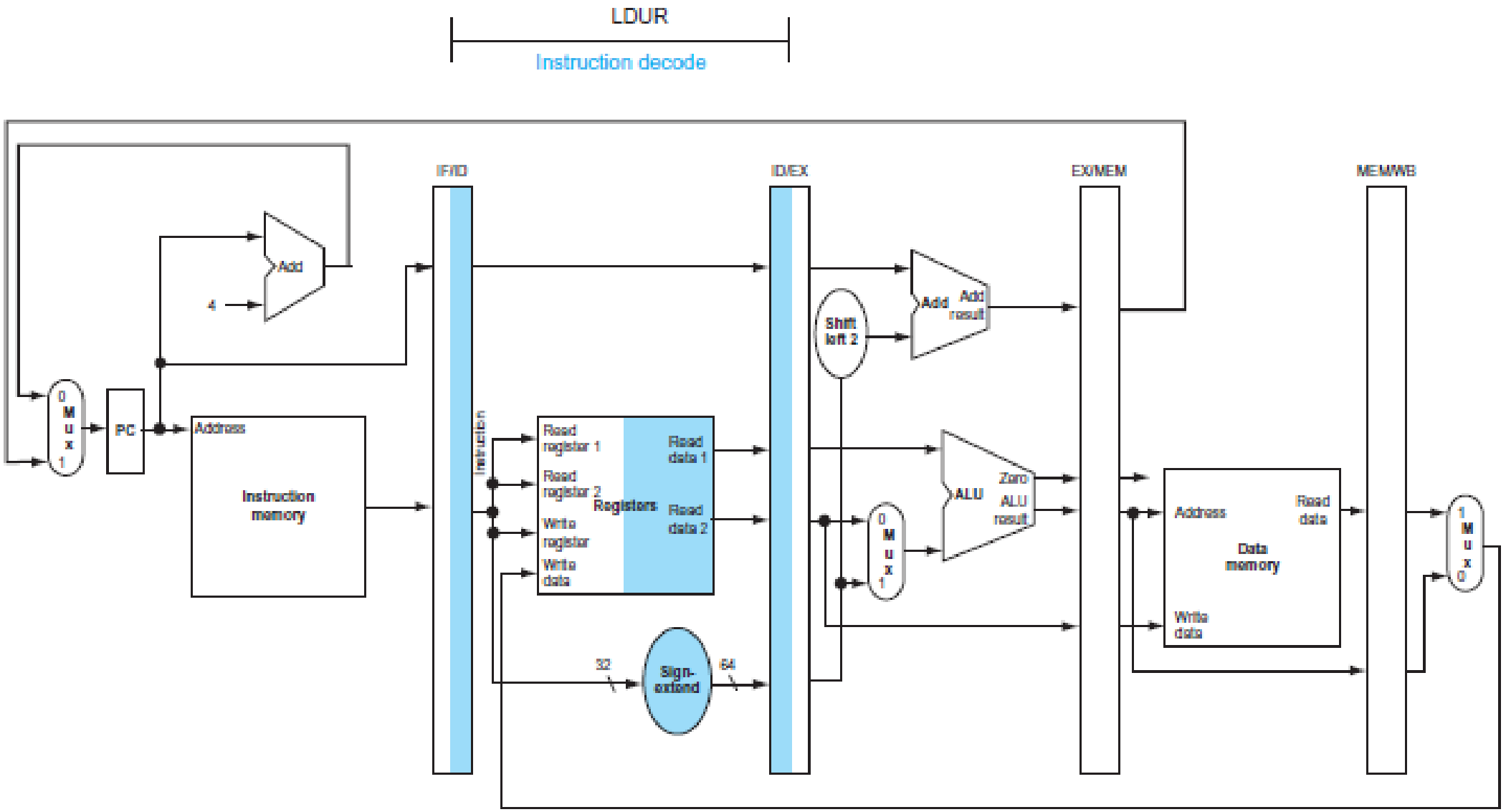
To split up each stage we insert a sequential array of Flip-Flops (A register) in between each pipeline stage.



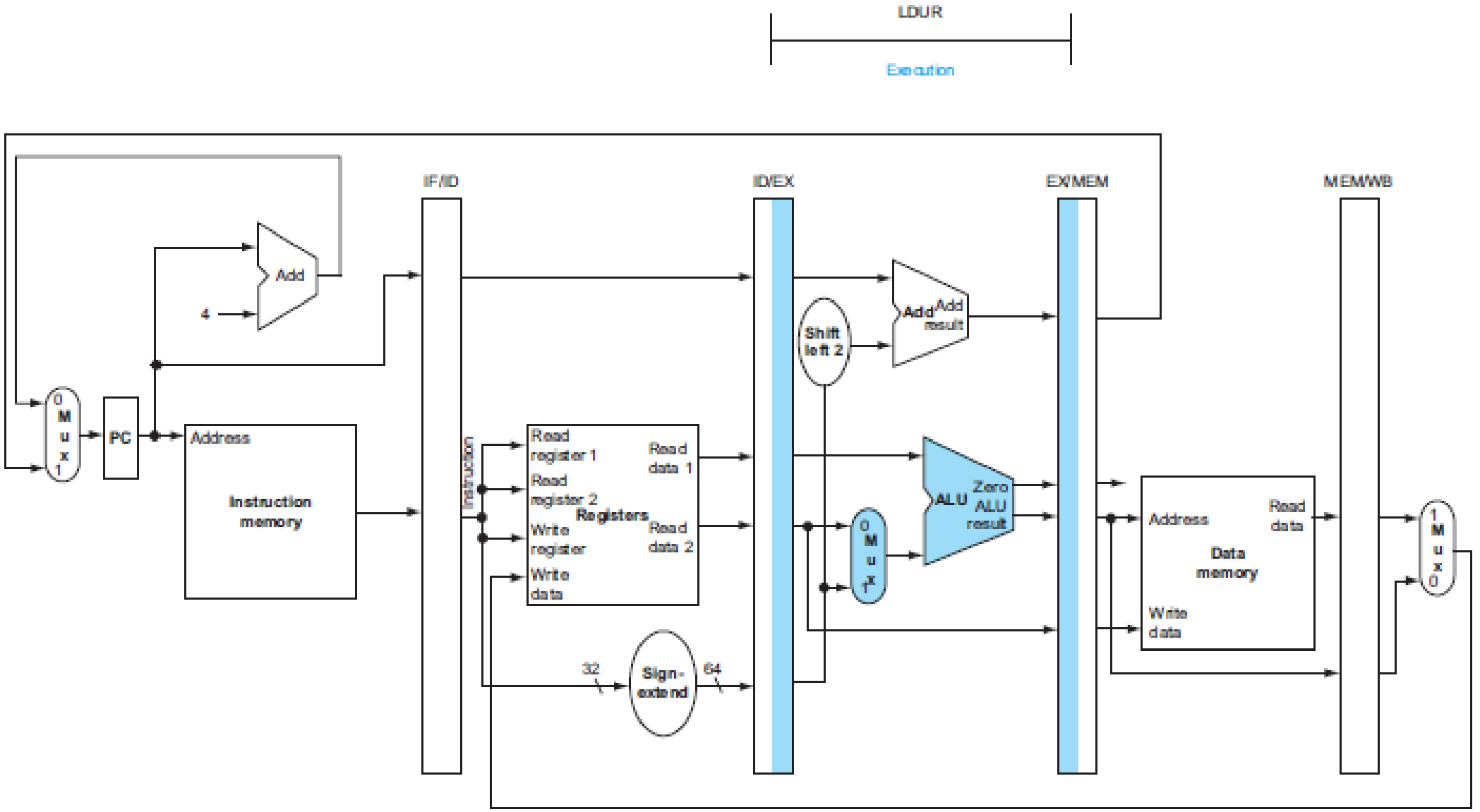
The pipeline for LDUR!



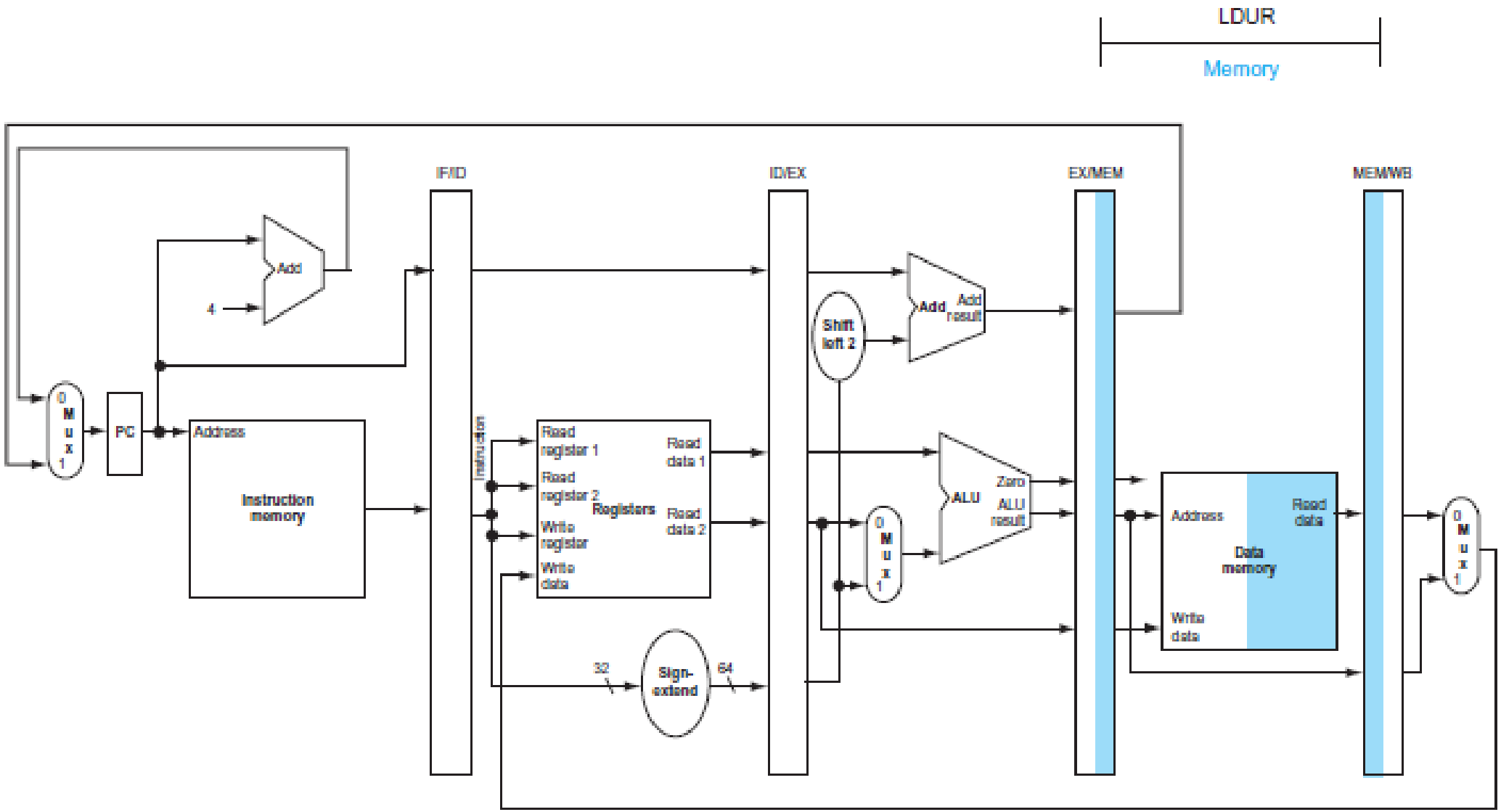
The pipeline for LDUR!



The pipeline for LDUR!



The pipeline for LDUR!



The pipeline for LDUR!