CSC 3210 Computer organization and programming

Chapter 4

Chunlan Gao

Georgia State University

Attendance number

- 5894

Before class

Final:

After lecture 9 (included) – end

Materials(slides, assignment, quizzes)

Assignment:

Assignment3 has been posted

I will give assignment 4 soon (lowest assignment will be dropped)

Quick Review

- Last class:**RISC-V Pipeline**

1. **IF (Instruction Fetch):** Fetch the instruction from memory.
2. **ID (Instruction Decode):** Decode the instruction and read registers.
3. **EX (Execute):** Perform ALU operations or compute memory address.
4. **MEM (Memory Access):** Access data memory (for load/store instructions).
5. **WB (Write Back):** Write the result back to the destination register.

Instruction Stages

Instruction Type	IF	ID	EX	MEM	WB	Description
R-Type (e.g., add, sub)	✅	✅	✅	❌	✅	ALU operation, result written to register
Load (e.g., lw)	✅	✅	✅	✅	✅	Compute address, read from memory
Store (e.g., sw)	✅	✅	✅	✅	❌	Compute address, write to memory
Branch (e.g., beq)	✅	✅	✅	❌	❌	Compare values, possibly change PC

Pipeline Speedup

Balanced

$$\text{Time}_{\text{pipelined}} = \frac{\text{Time}_{\text{non-pipelined}}}{\text{Number of stages}}$$

Not Balanced:

- **Latency**: Time taken for a **single instruction** to pass through **all stages** of the pipeline.
- **Throughput:** Number of **instructions completed per unit time**.
- **Speedup** is achieved by **increasing throughput**, more instructions completed per unit time.
- Latency (time for each instruction) does not decrease

Hazard

- **Structural Hazards : Conflict for use of a resource** occurs when two stages need the same hardware at the same time
- **Data Hazards: use the data from previous step**

  Method: Insert bubbles, Forwarding, Code Scheduling
- **Control Hazards:** Branch instructions determine the **flow of control**

Method: Branch Prediction(assume branch is not taken)

Summary

- Pipelining improves performance by increasing instruction throughput.
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

RISC-V Pipelined Datapath

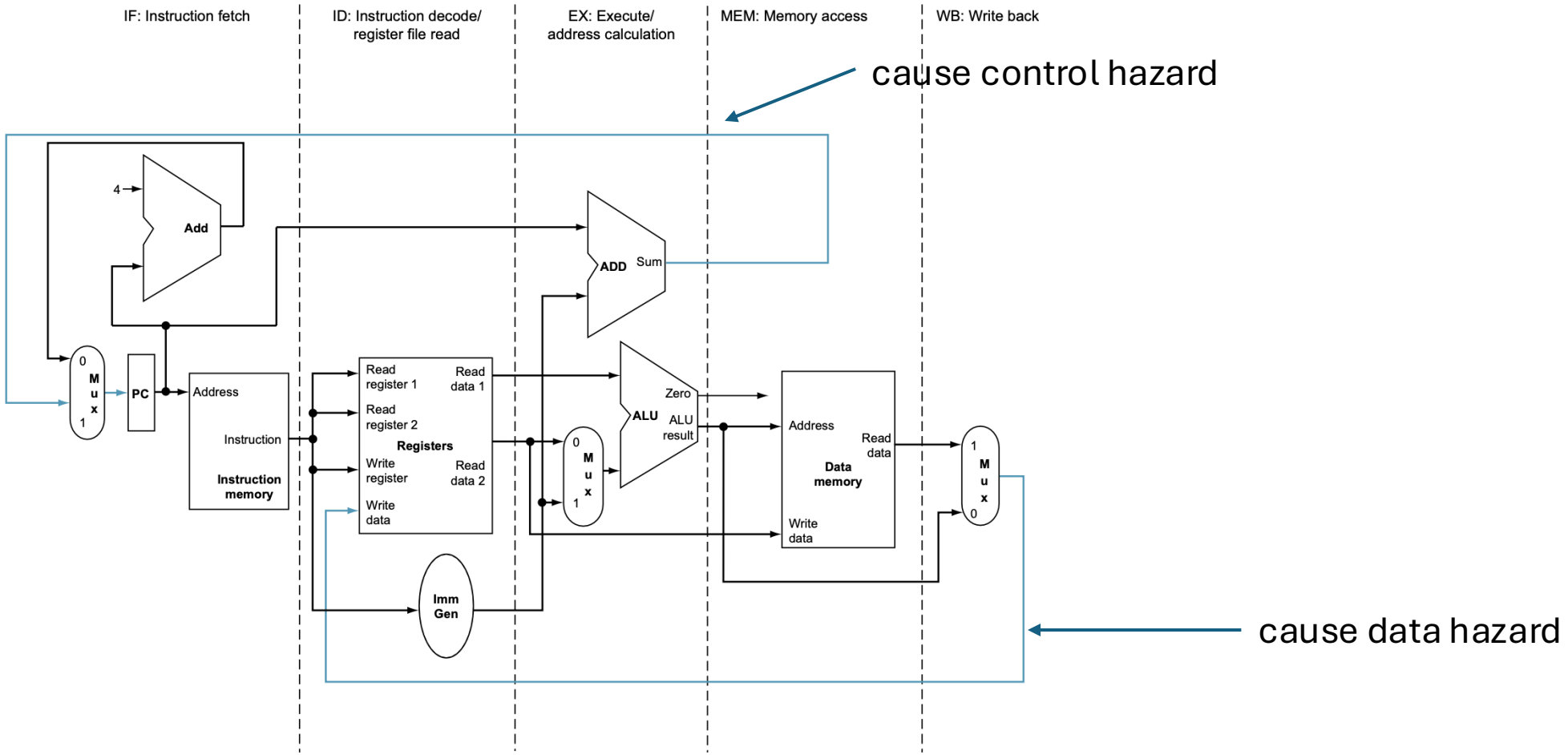


FIGURE 4.35 The single-cycle datapath from Section 4.4 (similar to Figure 4.21). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. (Normally we use color lines for control, but these are data lines.)

Pipeline Registers

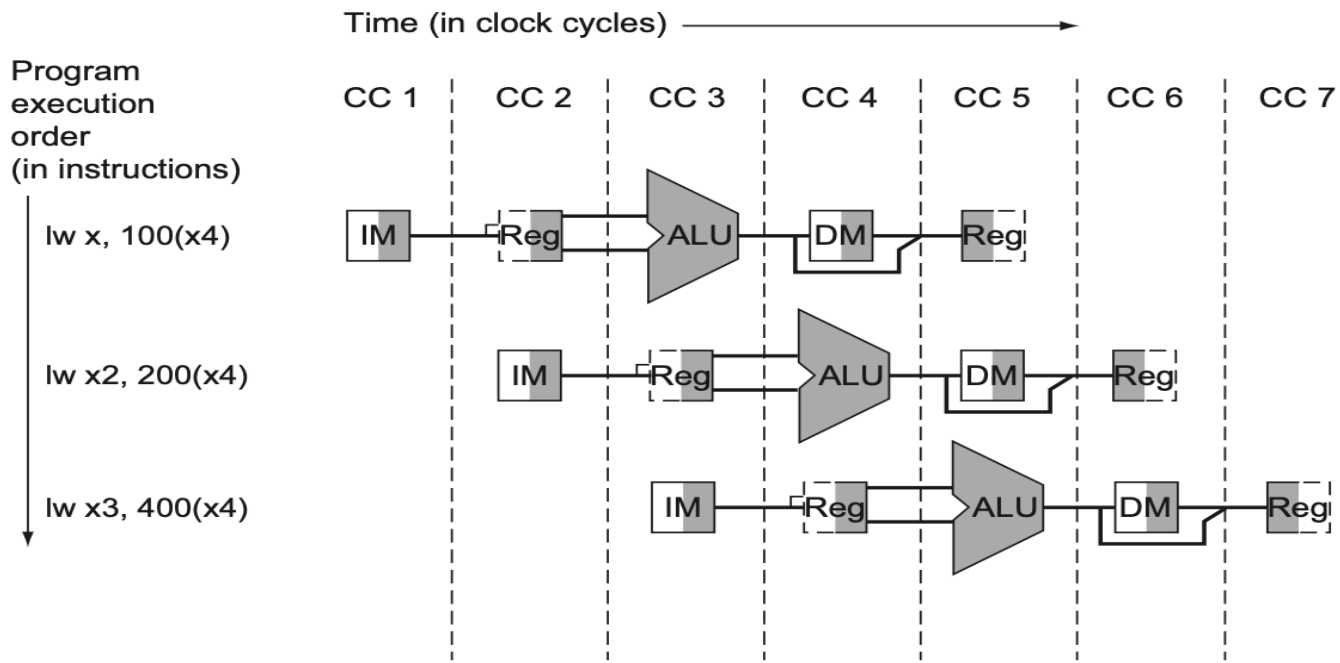


FIGURE 4.36 Instructions being executed using the single-cycle datapath in Figure 4.35, assuming pipelined execution. Similar to Figures 4.30 through 4.32, this figure pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled by the physical resource used in that stage, corresponding to the portions of the datapath in Figure 4.48. *IM* represents the instruction memory and the PC in the instruction fetch stage, *Reg* stands for the register file and sign extender in the instruction decode/register file read stage (ID), and so on. To maintain proper time order, this stylized datapath breaks the register file into two logical parts: registers read during register fetch (ID) and registers written during write back (WB). This dual use is represented by drawing the unshaded left half of the register file using dashed lines in the ID stage, when it is not being written, and the unshaded right half in dashed lines in the WB stage, when it is not being read. As before, we assume the register file is written in the first half of the clock cycle and the register file is read during the second half.

Pipeline Registers

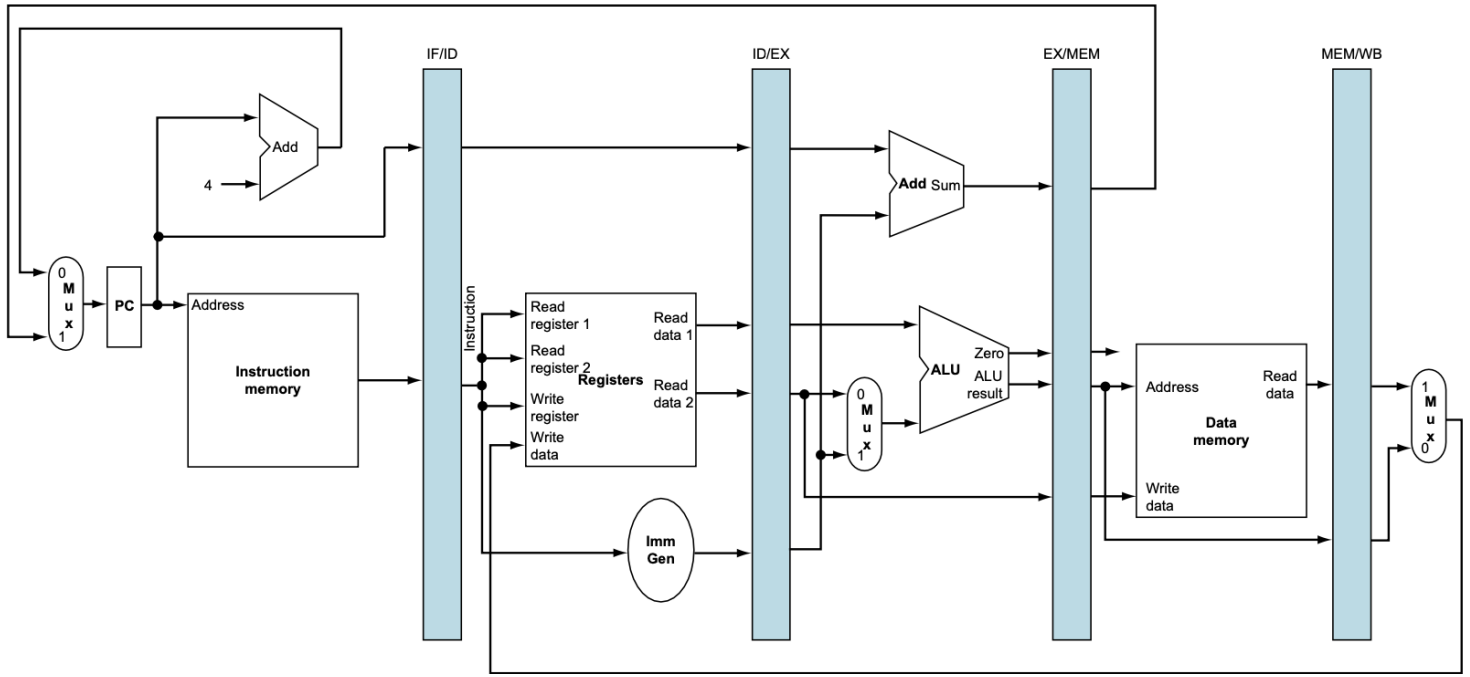


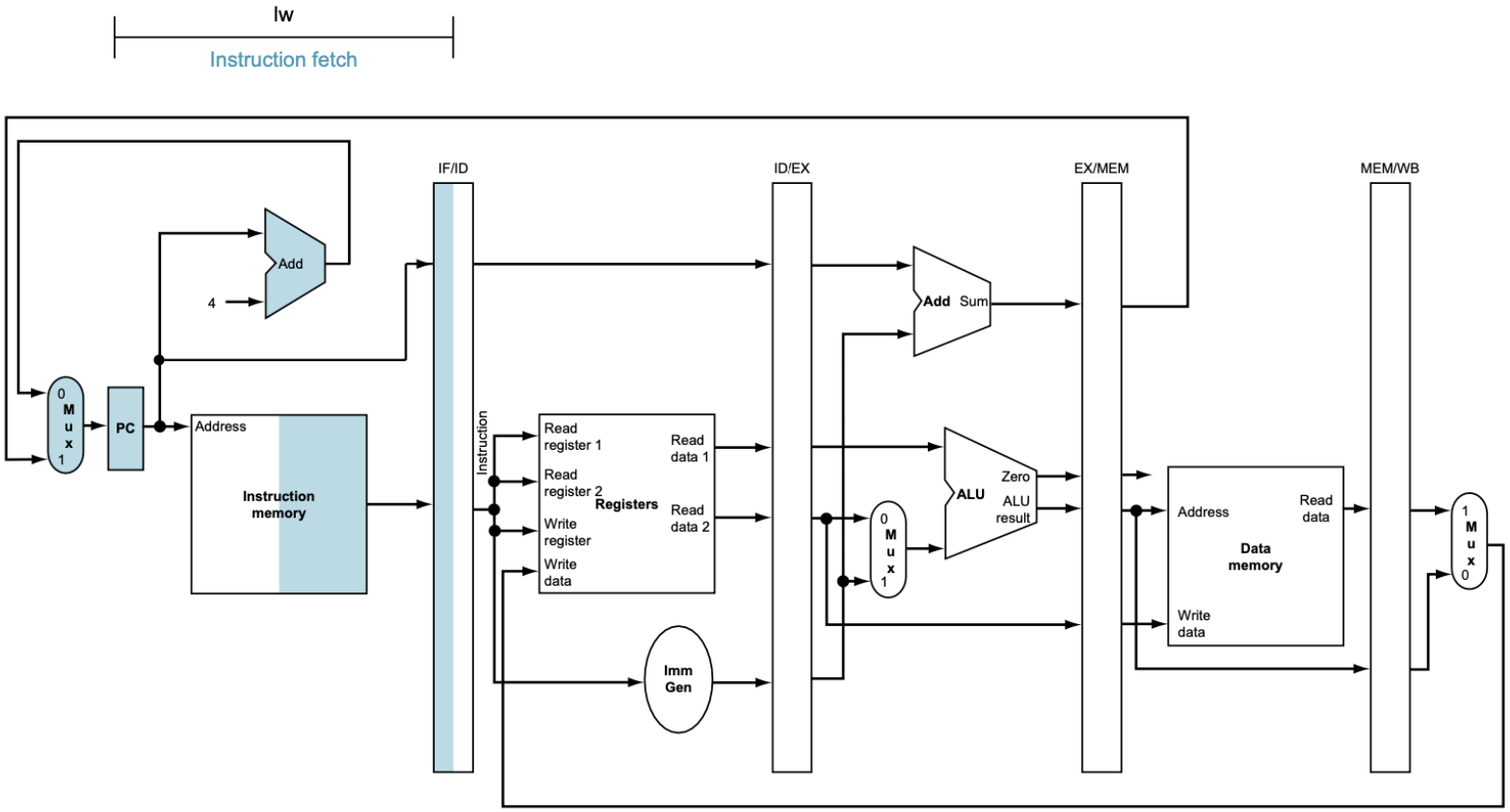
FIGURE 4.37 The pipelined version of the datapath in Figure 4.35. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 96 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 64-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 256, 193, and 128 bits, respectively.

Pipeline Operation

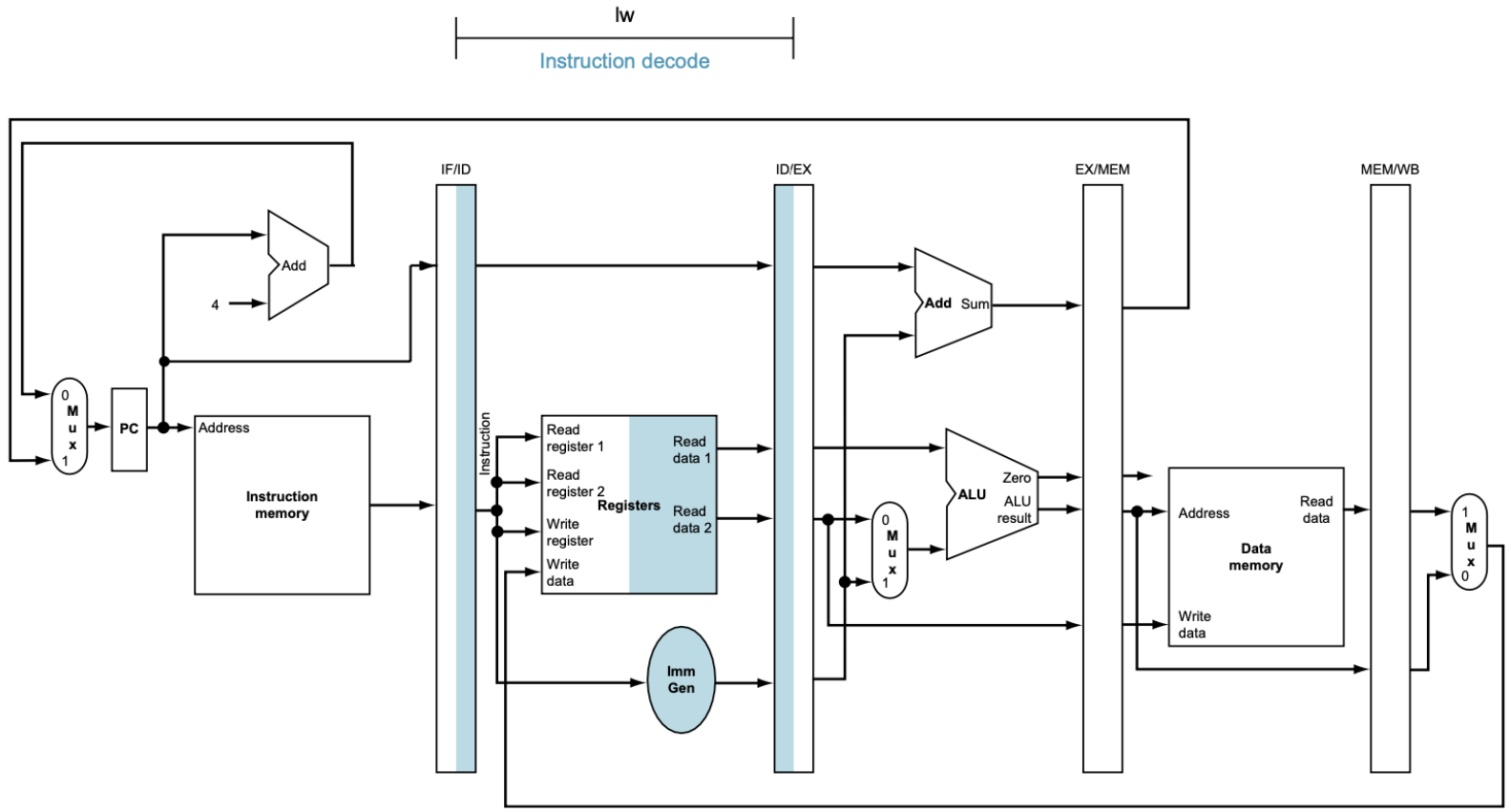
- Cycle-by-cycle flow of instructions through the pipelined datapath
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. “multi-clock-cycle” diagram
    - Graph of operation over time
- We’ll look at “single-clock-cycle” diagrams for load & store

First sequence, show the active portions of the datapath highlighted as a load instruction goes through the five stages of pipelined execution. We show a load first because it is active in all five stages

Instruction fetch(LW)



Instruction decode(LW)



Execute or Address Calculation (LW)

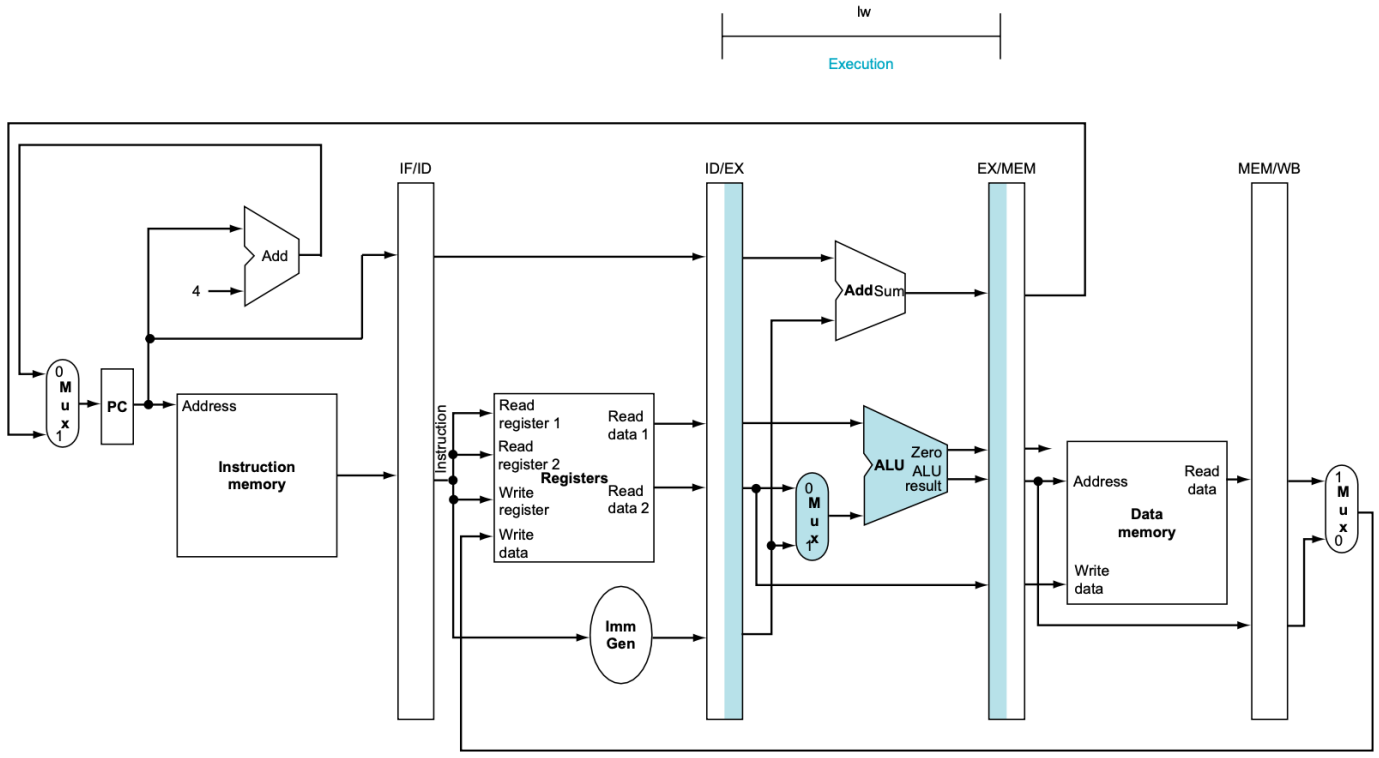
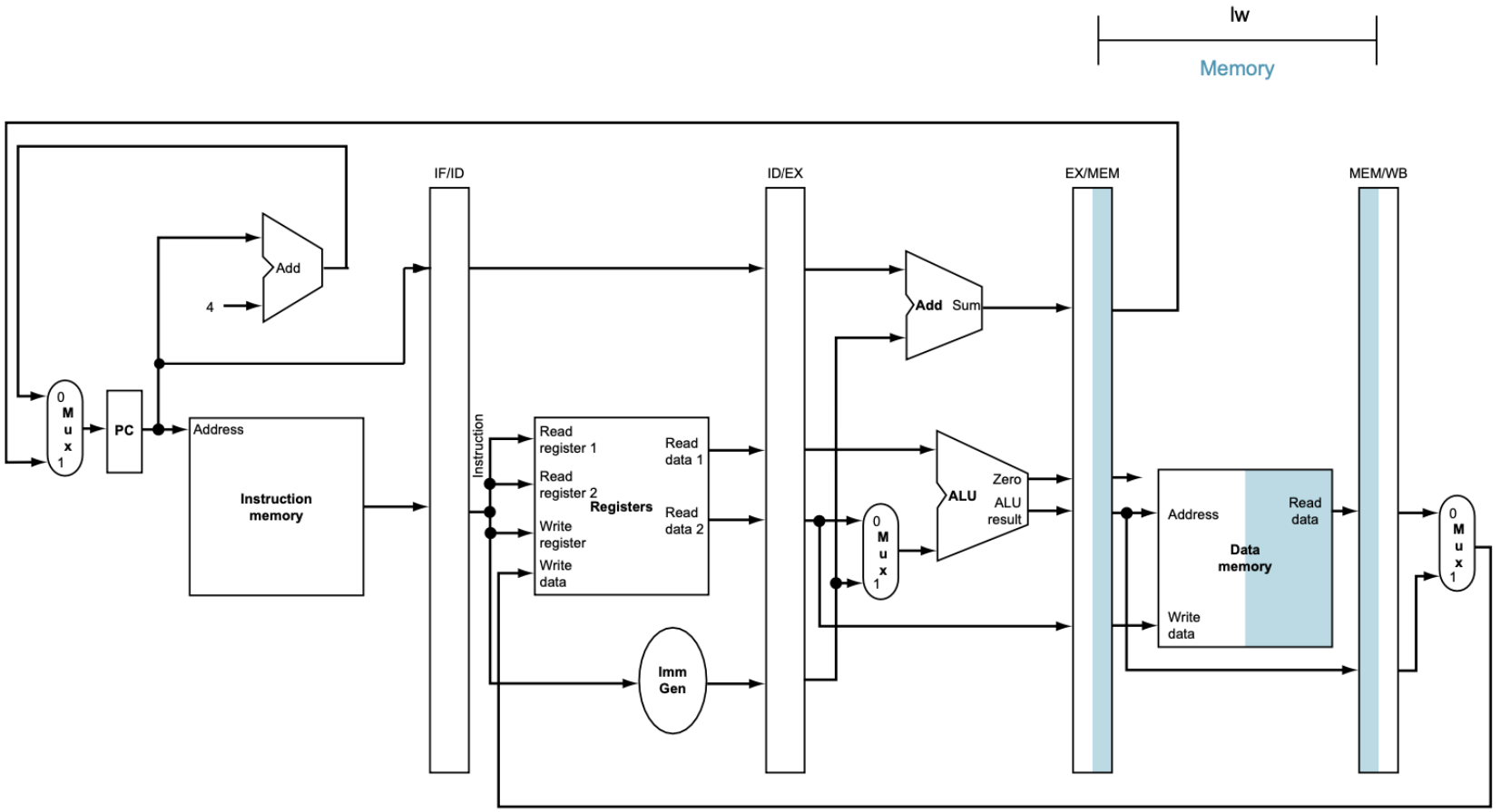


FIGURE 4.39 EX: The third pipe stage of a load instruction, highlighting the portions of the datapath in Figure 4.37 used in this pipe stage. The register is added to the sign-extended immediate, and the sum is placed in the EX/MEM pipeline register.

Memory access (LW)



Write-Back (LW)