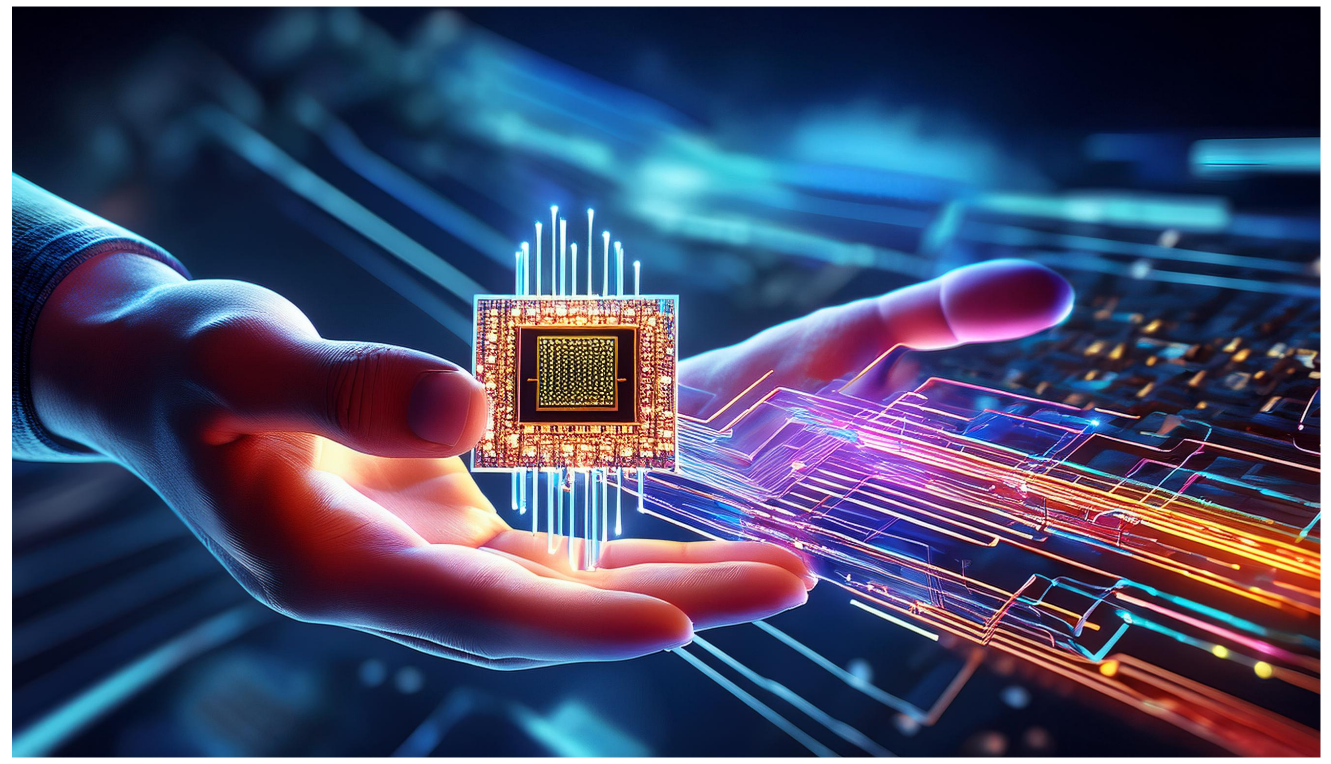
CSCI 250
Introduction to Computer Organisation
Lecture 4: Control Unit and Pipelines III



Jetic Gū

2024 Fall Semester (S3)

Overview

- Architecture: von Neumann
- Textbook: CO: 4.5
- Core Ideas:
  1. Pipelined Computers II: Hazard Control
  2. Pipelined Computers III: Simple Pipelined CPU
  3. Lab 4 Part 2

P0 Review

Properties of CPU Pipeline

- Does pipelining **reduce latency** of a single stage/task?
  - No, but it increases throughput of entire workload
- What could affect pipeline’s efficiency?
  - The slowest stage
  - Total number of stages
  - Unbalanced lengths of stages: some stages significantly slower than others
- When to **fill** pipeline, and when to **drain/flush** it

Review

CPU Pipelines II

Hazard

P1
Hazard

Possible Issues in Implementation

- **Structural hazards**
  Different instructions, at different stages, want to use the same hardware resource
- **Control hazards**
  Succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction already in pipeline
- **Data hazards**
  an instruction in the pipeline requires data to be computed by a previous instruction still in the pipeline

Concept

P1
Hazard

Possible Issues in Implementation

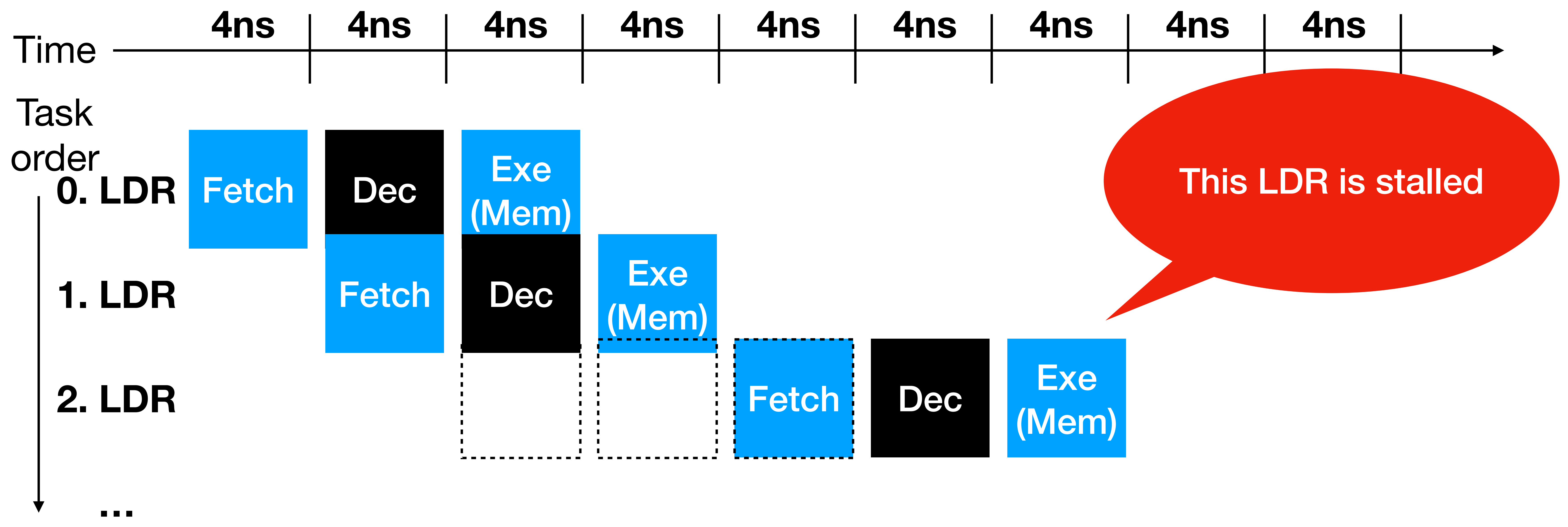
- **Structural hazards**
  Different instructions, at different stages, want to use the same hardware resource
  - Solution: **stalling**
    - e.g. when multiple stages of execution wants to access the main memory, it is served at a first-come-first-server principle
    - the rest of the stages are "**stalled**", and have to wait for their turns

Concept

P1
Pipeline

Possible Issues in Implementation

- **Structural hazards**
  Different instructions, at different stages, want to use the same hardware resource
  - Solution: **stalling**



Example

P1
Hazard

Possible Issues in Implementation

- **Control hazards**
  Succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction already in pipeline

  - Solution #1: **stalling**

    - when a branching instruction is Fetched into the pipeline, subsequent instructions' **fetch** are **stalled**

    - this prevents new instructions from being fetched into the pipeline, effectively **flushes** the entire pipeline

Concept

P1
Hazard

Possible Issues in Implementation

- **Control hazards**
  Succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction already in pipeline
  - Solution #2: **static branch prediction**
    - Proceed with pipeline, keep **fetching.** If outcome from a conditional branch stands (actually goes into branch), then perform **flush**
    - Think: how is this different from **stalling**?
      Is it better or worse?

Concept

P1
Hazard

Possible Issues in Implementation

- **Control hazards**
  Succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction already in pipeline

  - Solution #3: **delayed branch**

    - Requires the compiler to find branching-independent instructions to put right next to the branching statement, so the pipeline can keep executing even when it encounters branching

    - This relies heavily on compilers, doesn't always work

Concept

P1
Hazard

Possible Issues in Implementation

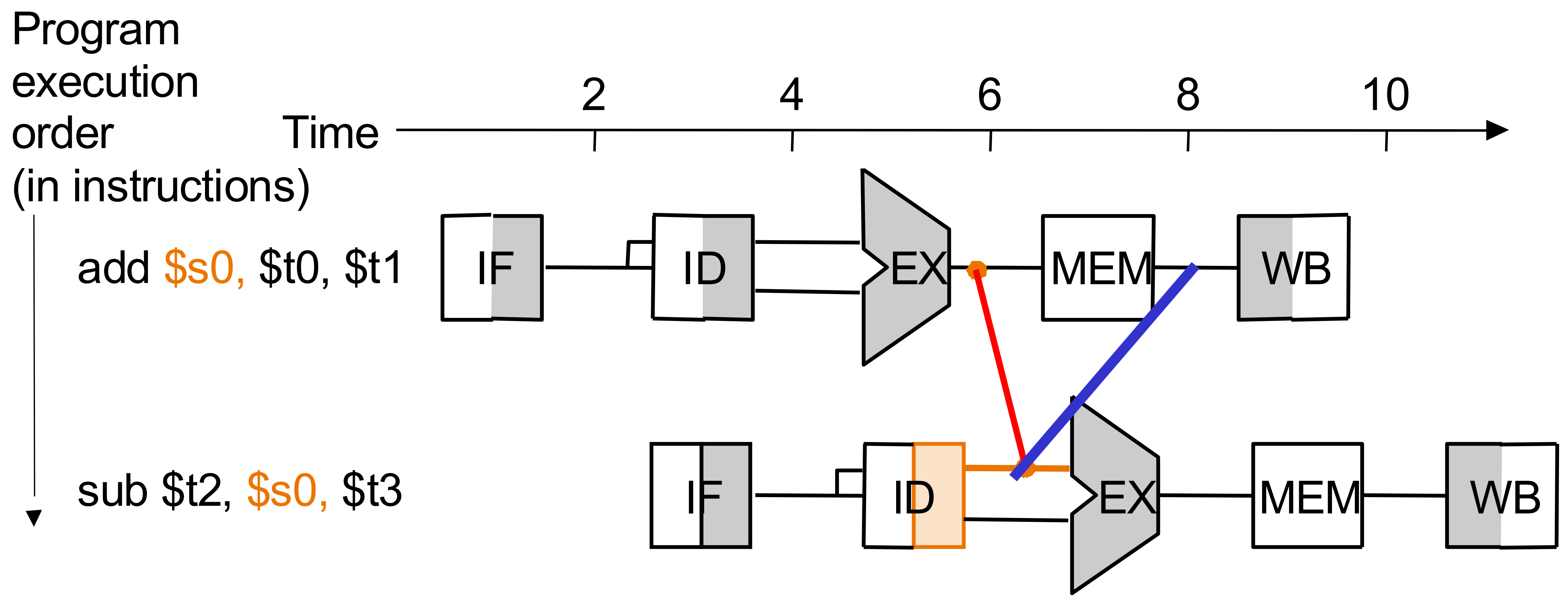
- **Data hazards**
  An instruction in the pipeline, requires data to be computed by a previous instruction still in the pipeline
  - Solution: **forwarding**
    - Create bridges between different stages', so some data can be fast forwarded to the next stage, parallel to e.g. register write operations
    - In the event this isn't enough, **stall**

Concept

P1
Hazard

Possible Issues in Implementation

- **Data hazards**
  An instruction in the pipeline, requires data to be computed by a previous instruction still in the pipeline
  - Solution: **forwarding**



  - **sub** here depends on **add**, so we create a bridge between the Execution stage and Decoding stage

Technical

P1
Hazard

Software Solution

- Is there anything you could do on the software side?
  - Compilers (e.g. gcc, clang, etc.)
    - Code comes in, compiler depends on the CPU architecture, tries to reorder instructions, simplify your code, etc. to prevent issues
    - Compiler flags
      - gcc options: `-O1`, `-O2`, `-O3`
        speed things up for you by aggressively doing reordering among other things. Using `-O3` could cause issues especially if you are managing memory manually, use with caution. `-O2` and `-O3` are also not `gdb`/`lldb` friendly.

Technical

P1
Hazard

Summary

- Hazards
  - Structural Hazard: **Stall**
  - Control Hazard: **Stall** / **Branch prediction** / **Delayed branch**
  - Data Hazard: **Forwarding / Stall**
  - Software: Compiler optimisation, reordering

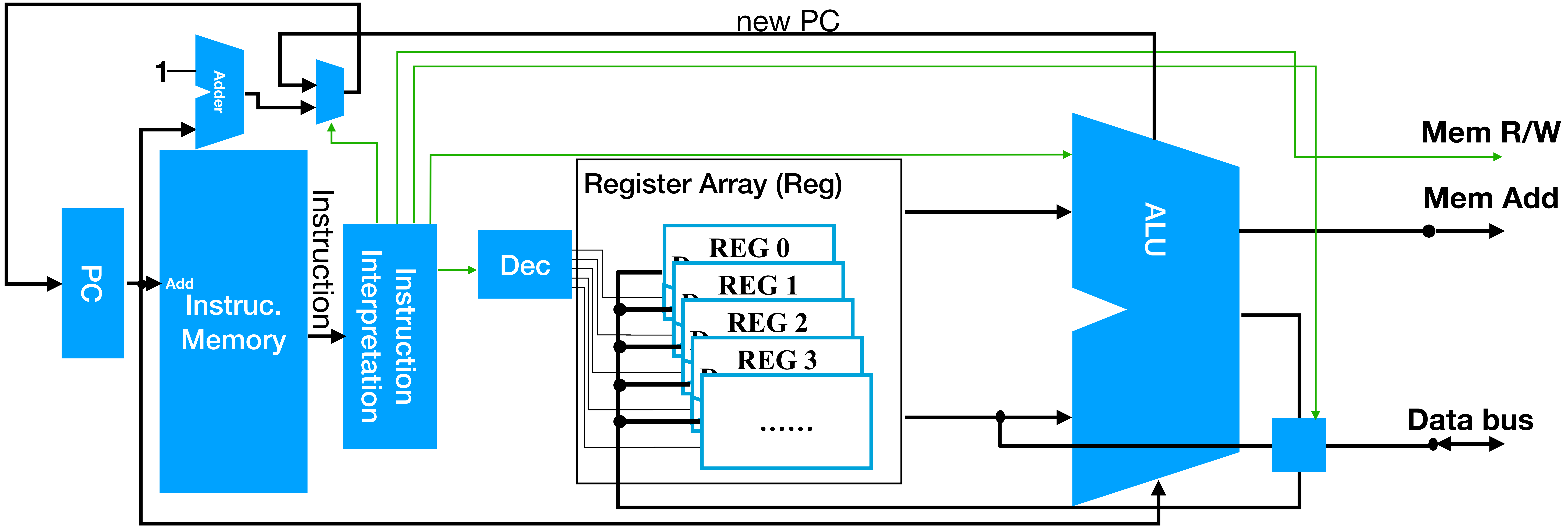
Review

CPU Pipelines III

Simple Pipeline Implementation

P2 Implementation

Our MIPS Example Before



Review

P2
Implementation

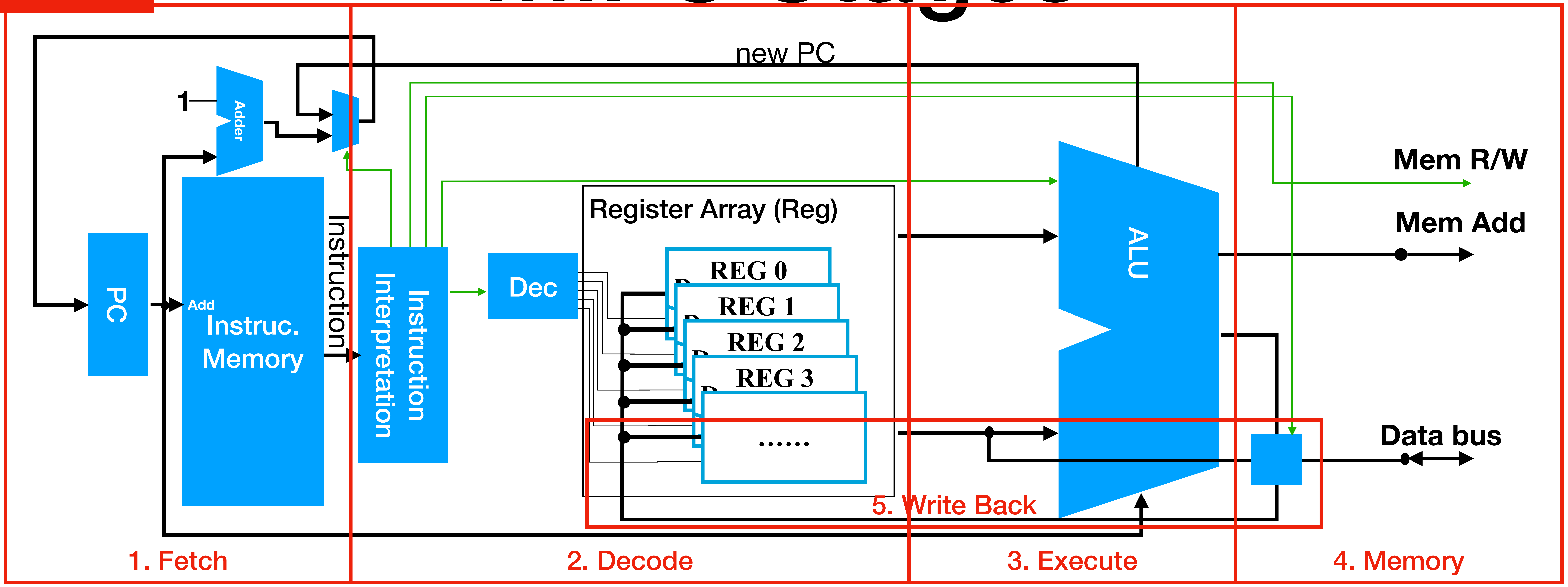
Our MIPS Example Before

- MIPS CPUs commonly uses a 5 stage design
  - Fetch, Decode, Execute
  - Additional stages after Execution: **Memory**, **Write Back**

Concept

P2
Implementation

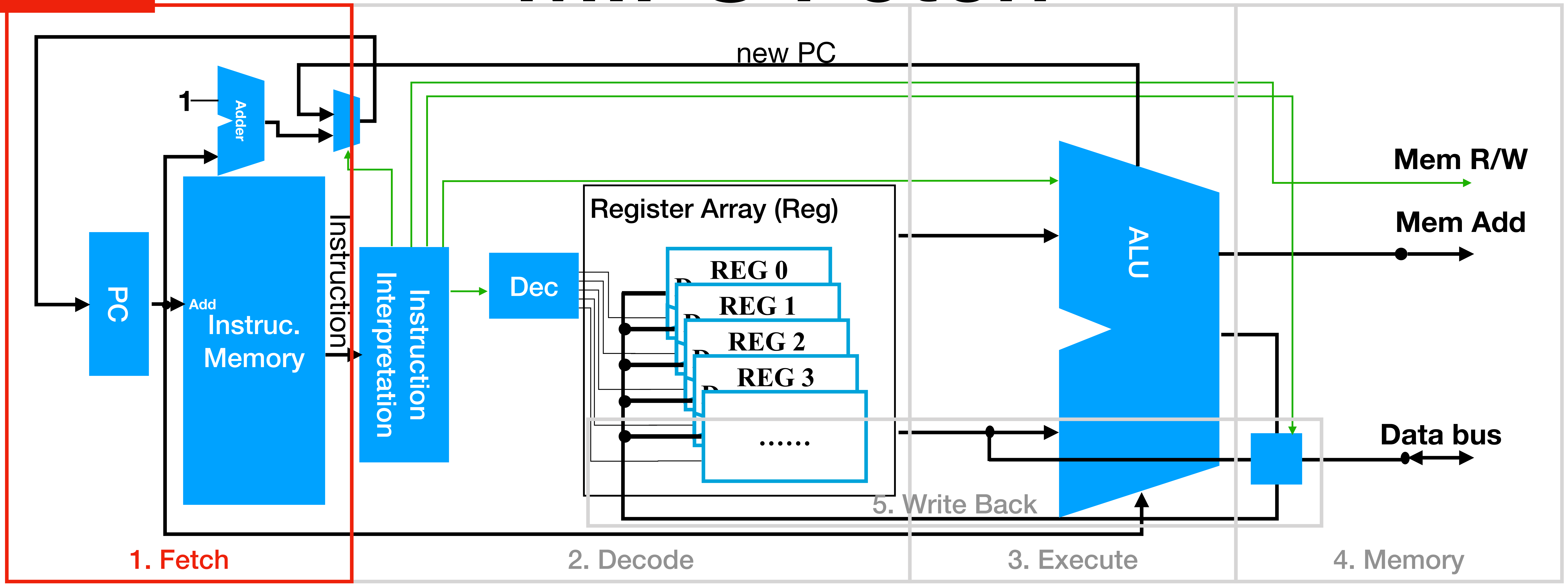
MIPS Stages



Technical

P2
Implementation

MIPS Fetch

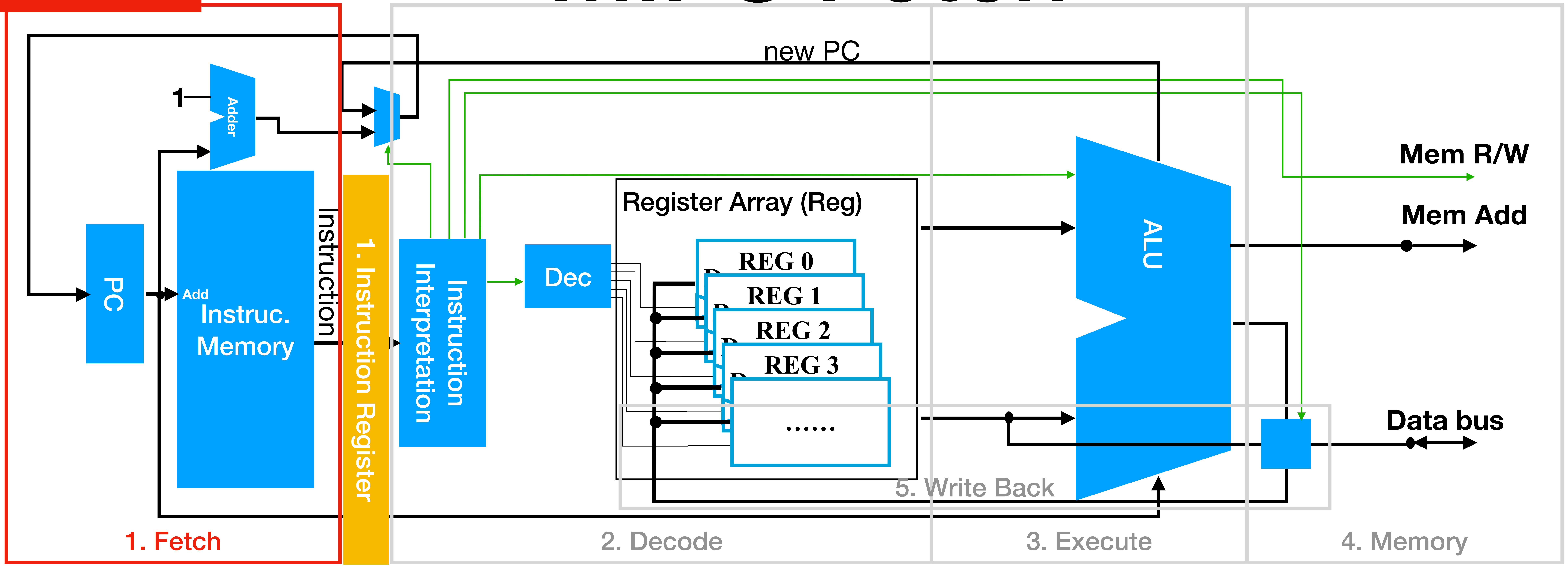


- In the **Fetch** stage, the new instruction is retrieved from the instruction memory, stored in <u>Stage 1 register</u> (<u>S1</u>, or **Instruction Register**). Meanwhile, PC's value increases by 1.

Technical

P2
Implementation

MIPS Fetch

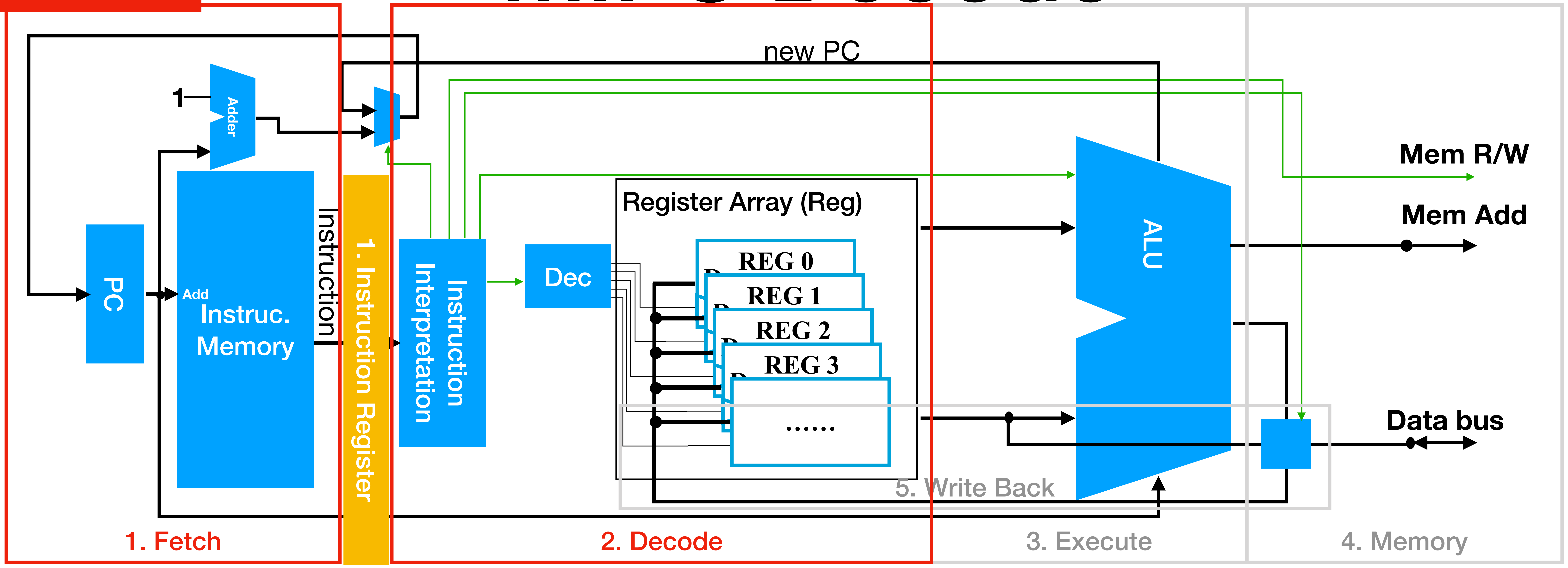


- In the **Fetch** stage, the new instruction is retrieved from the instruction memory, stored in Stage 1 register (S1, or **Instruction Register**). Meanwhile, PC's value increases by 1.

Technical

P2
Implementation

MIPS Decode

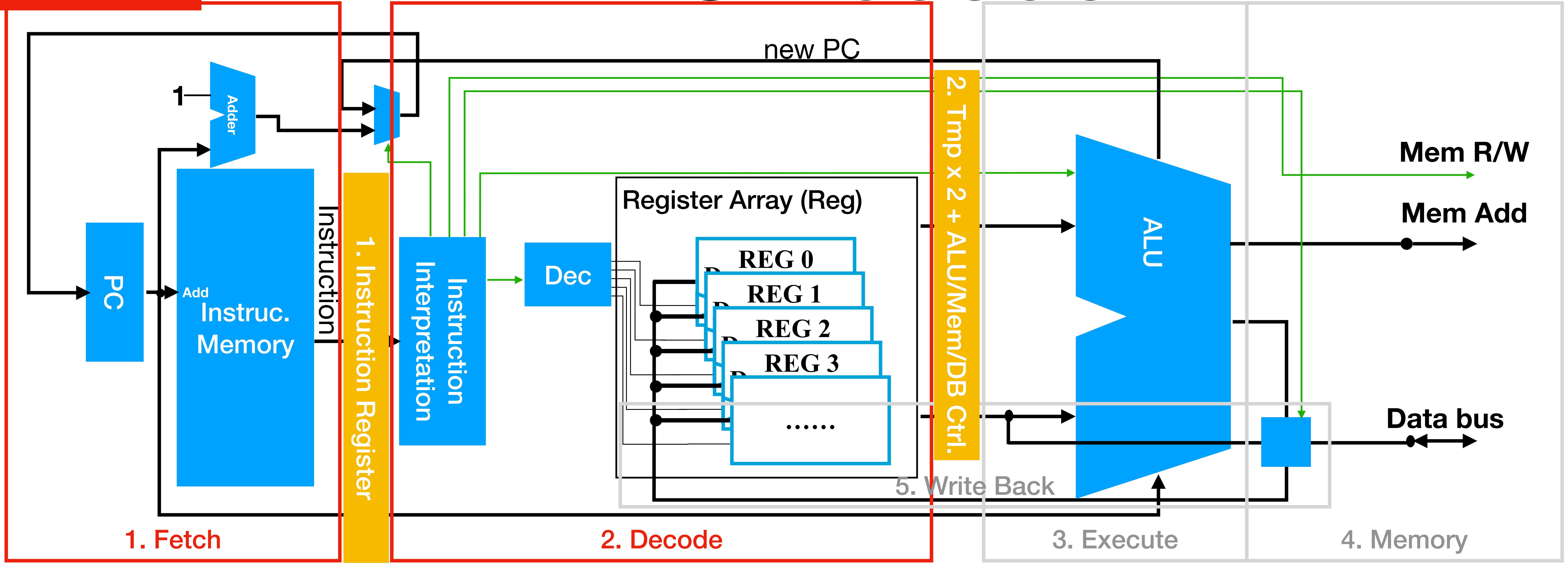


- In the **Decode** stage, information from the Register Array is retrieved, and saved in S2 registers (stage 2 registers) for the ALU to access

Technical

P2 Implementation

MIPS Decode

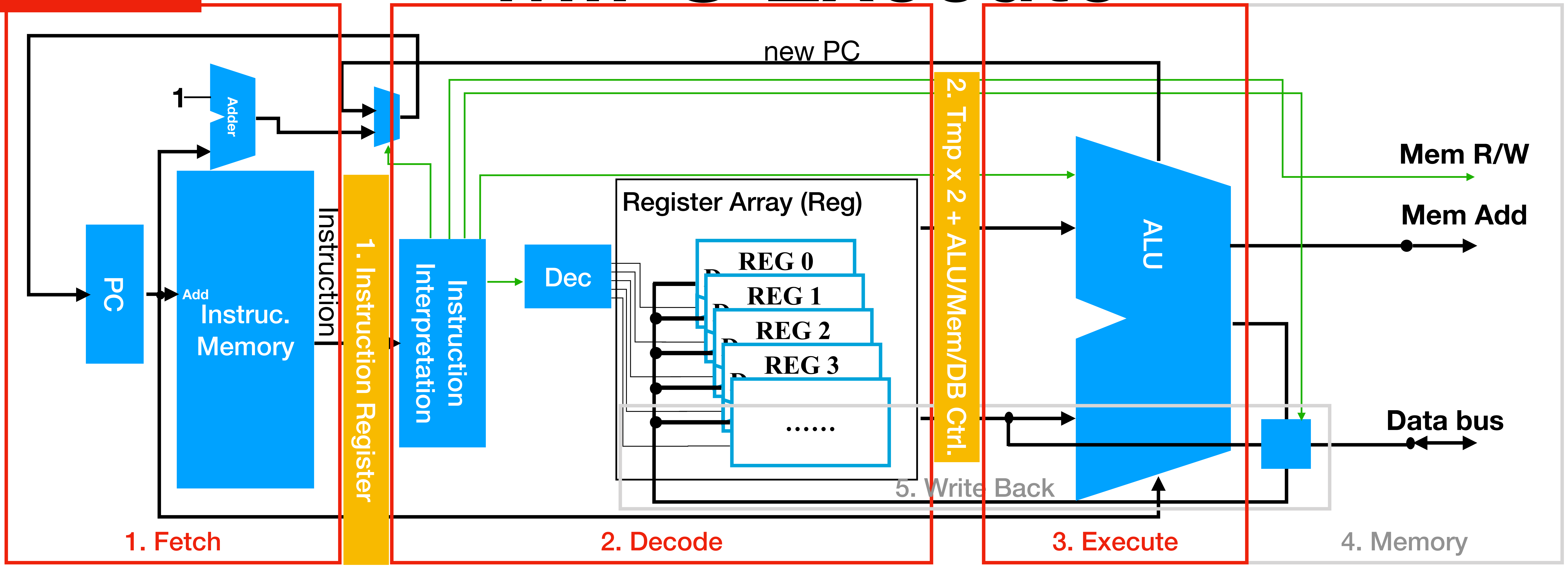


- In the **Decode** stage, information from the Register Array is retrieved, and saved in <u>S2</u> registers (stage 2 registers) for the ALU to access

Technical

P2
Implementation

MIPS Execute

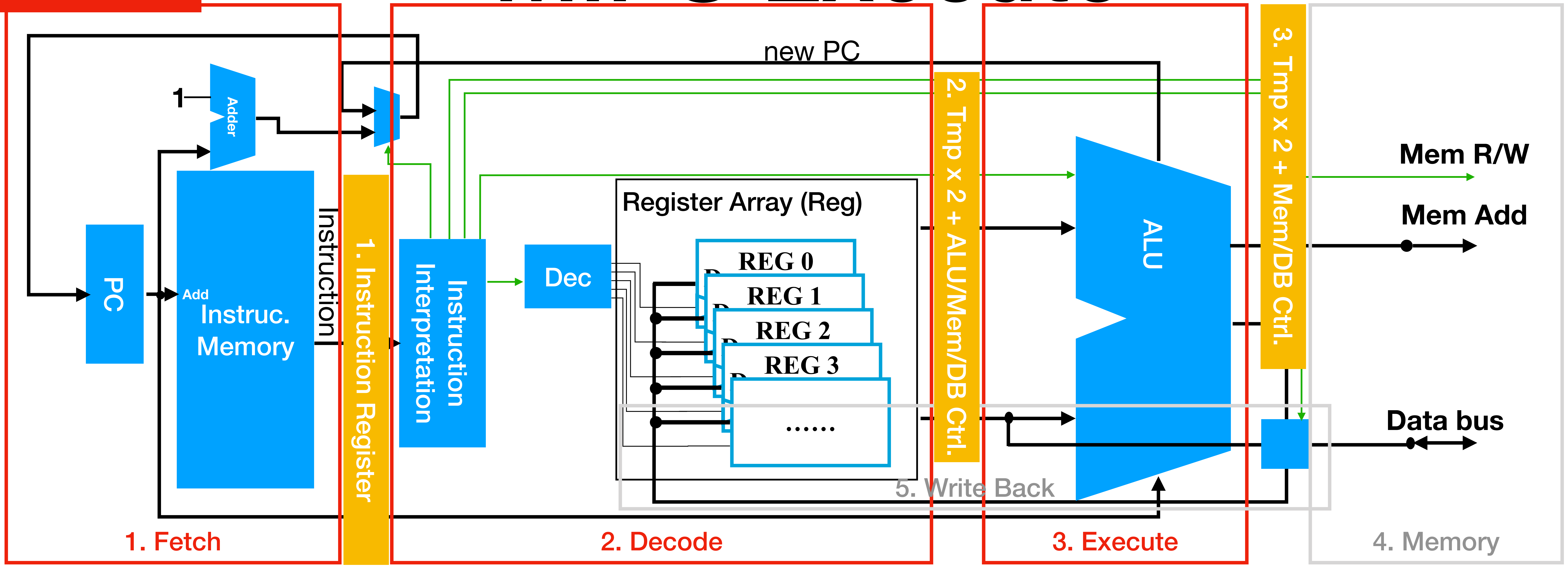


- In the **Execute** stage, ALU performs calculation, and the results are temporarily stored within S3 Registers

Technical

P2 Implementation

MIPS Execute

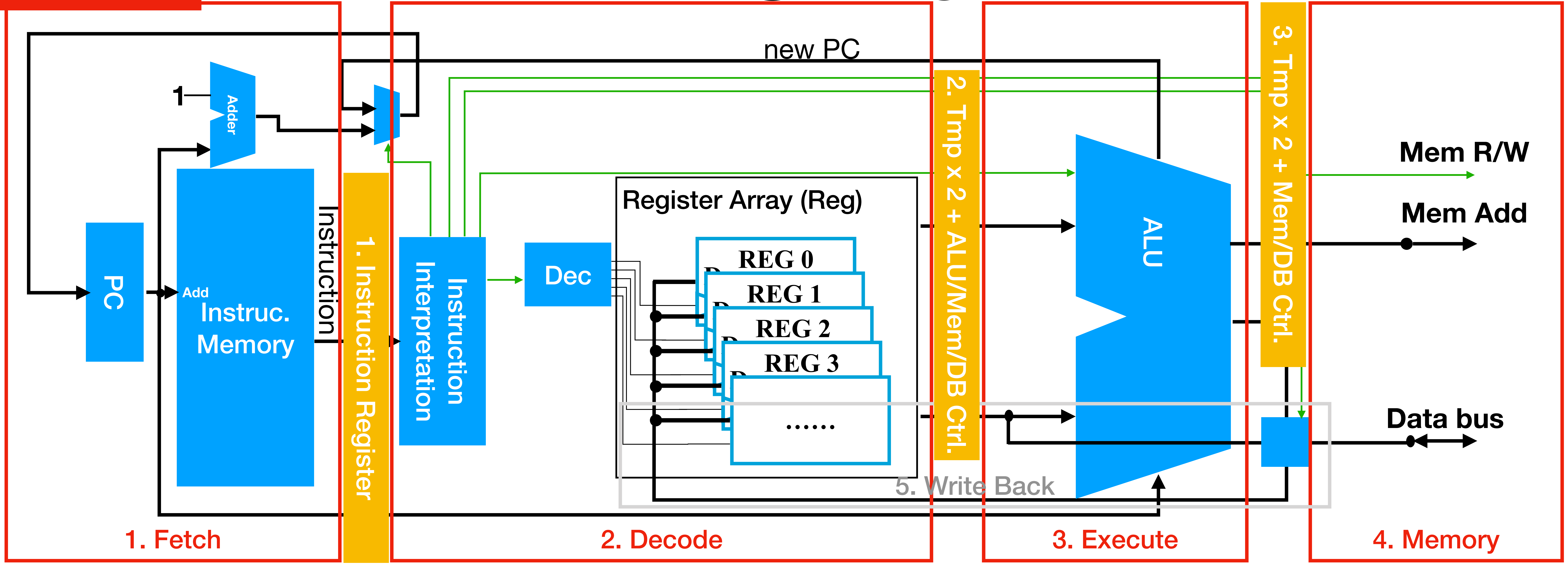


- In the **Execute** stage, ALU performs calculation, and the results are temporarily stored within S3 Registers

Technical

P2 Implementation

MIPS Mem

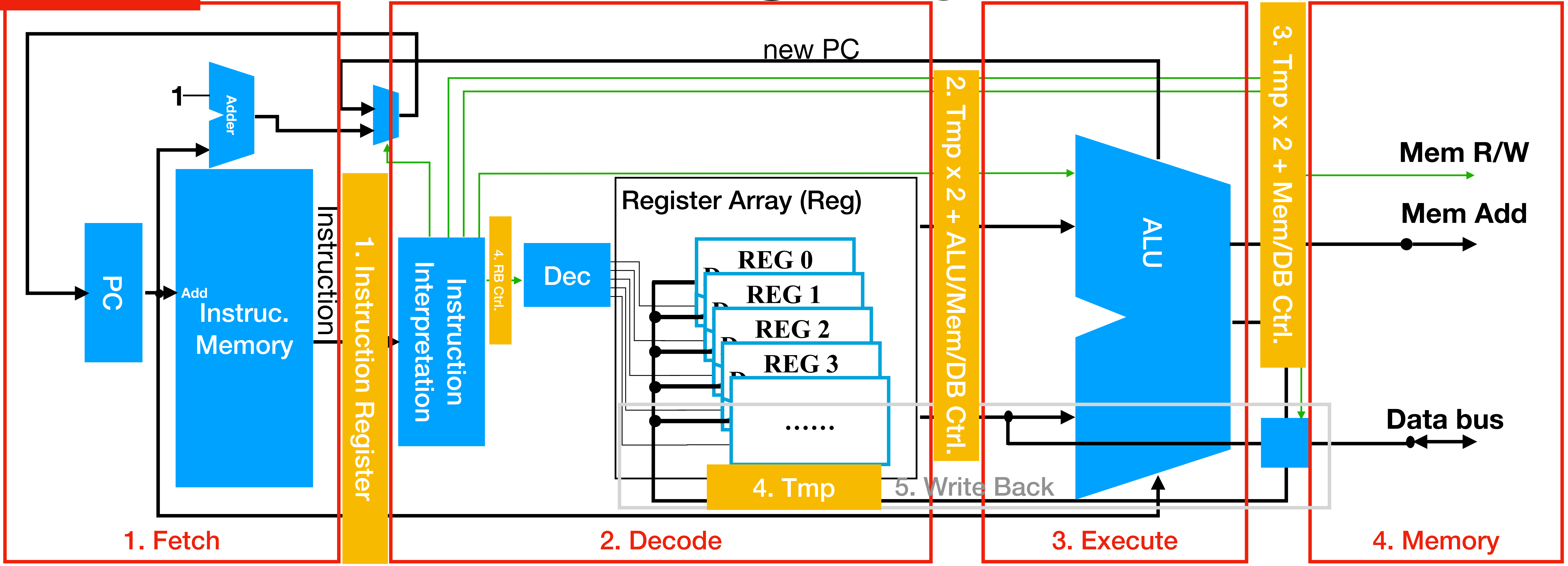


- In the **Mem** stage, data is retrieved to S4 register (from memory (if mem_load) or S3 register), or data is written into memory from S3 register

Technical

P2
Implementation

MIPS Mem

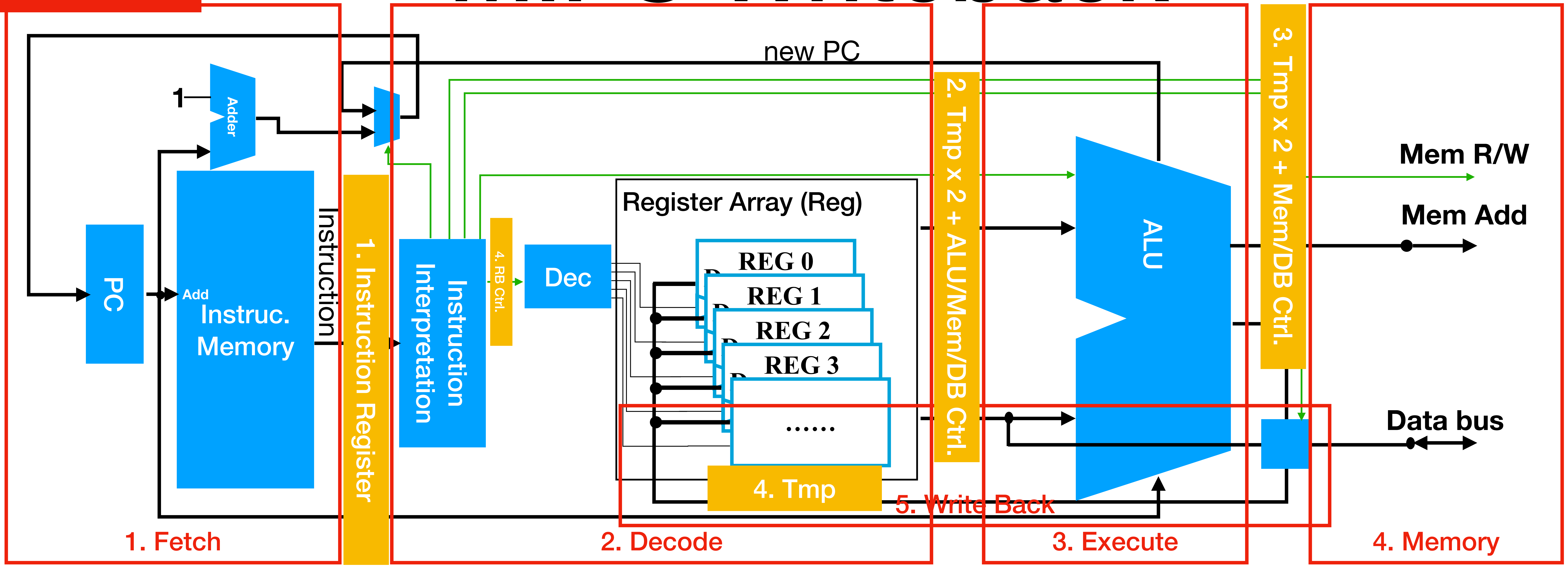


- In the **Mem** stage, data is retrieved to S4 register (from memory (if mem_load) or S3 register), or data is written into memory from S3 register

Technical

P2 Implementation

MIPS Writeback

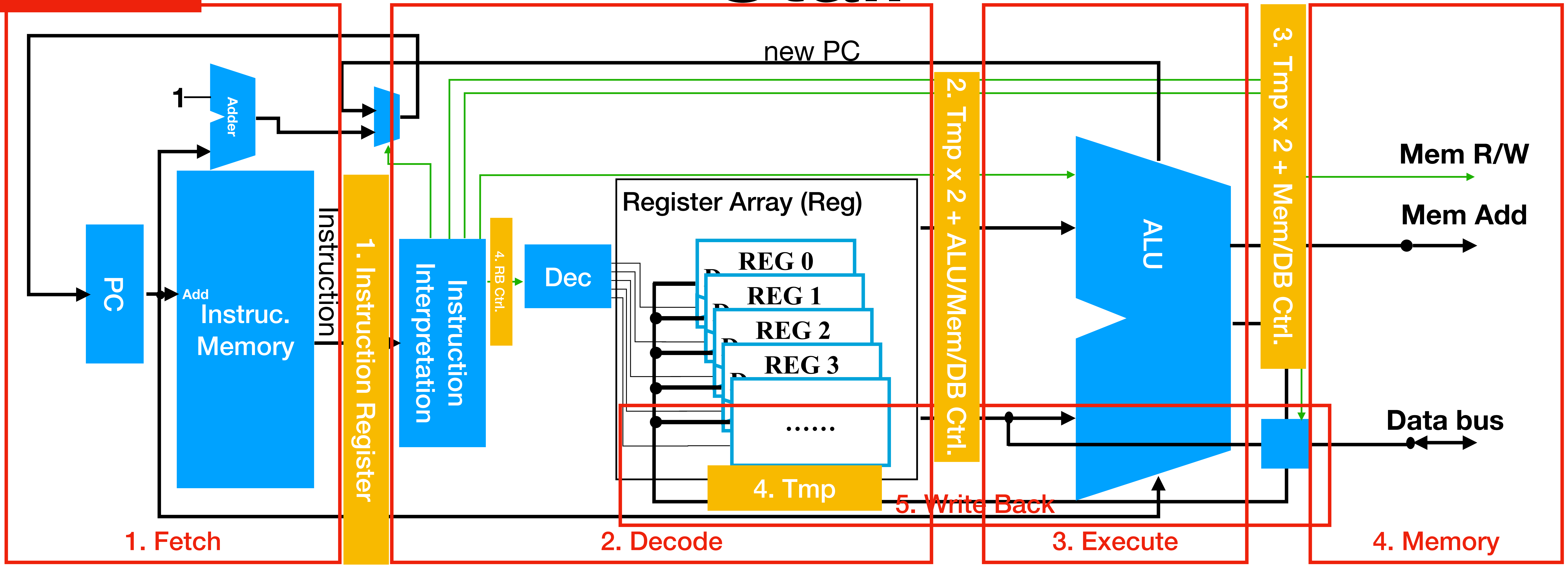


- In the **Writeback** stage, data from S4 register (if necessary), is saved in the register array (Rd). PC's value also receives update if needed.

Technical

P2 Implementation

Stall



- How can we implement stalling/flushing?
- When do we need to stall/flush?

Think

How is ARMv7 different?

- Our ARM16 CPU is based on the specification for ARMv7 thumb instruction set
- In ARMv7, PC is part of the register array
  - We can accomplish that by:
    1. have R7 values (PC) outputted directly from the register array, separate from the `Rm_data, Rn_data`
    2. implement the increment mechanism directly into R7
    (why can't we implement it in the ALU?)
- In ARMv7, there's only the main memory. However, since instructions are commonly cached in the instruction cache, we can still preserve the instruction memory module, as if it's actually the Instruction Cache.

Lab 4 Part 2

Data Bus controller,
Memory Array Modification

P3
Lab

16bit Memory Controller

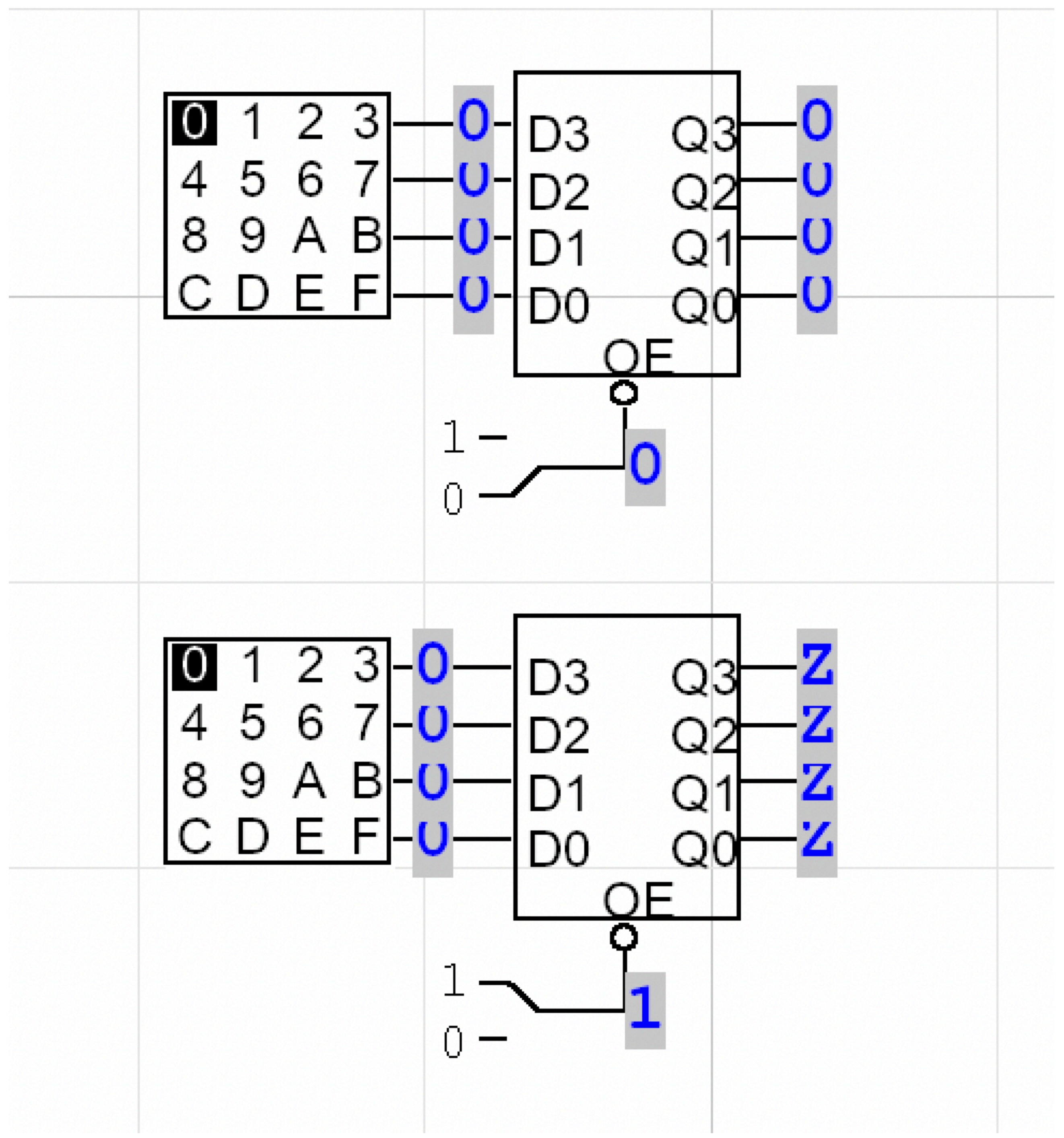
- We need a memory controller that is connected to the data bus, depending on direction react accordingly
  - Bidirectional: 16bit D16, `db`
  - Input: 1bit, `db_dir`; when `db_dir` is 0, data flows into memory, triggering store/mem_write
  - Input: 16bit, `DO` from memory module;
  - Output: 16bit, `DI` to the memory module;
  - Output: 1bit `WE`;
- Always, `WE <= db_dir;`
  When `db_dir` == 0, `DI <= db`;
  When `db_dir` == 1, `db <= DO;`

Lab

P3
Lab

3 State Buffers

- Component name: `Buffer-4 T.S.`
- This is a 4bit 3 state buffer
- You can use this to control whether a line is connected or not
- Remember, `Z` here means Hi-Z (no connection, open circuit)

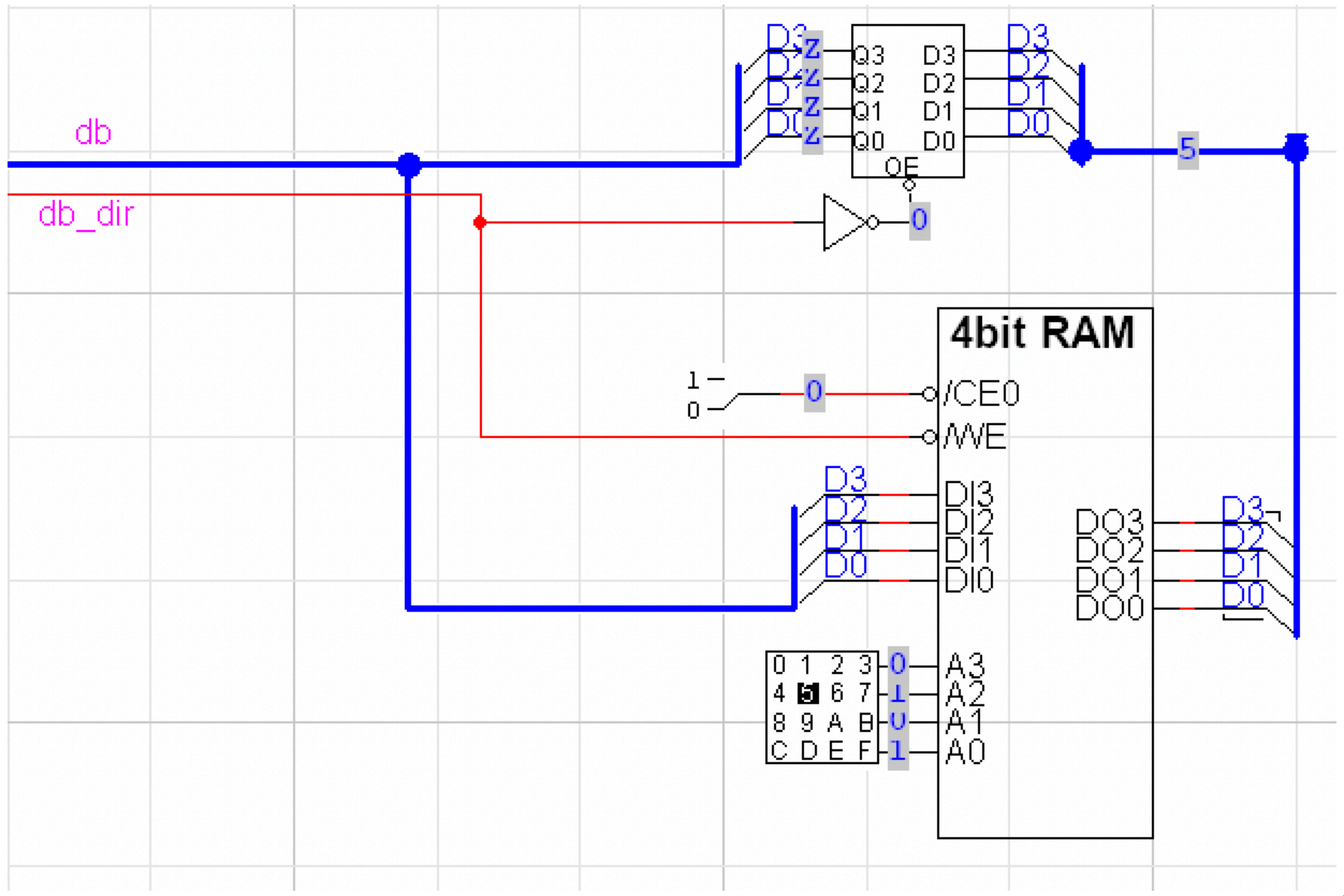


Technical

P3 Lab

4bit Memory Controller

- `db_dir == 0`
  - Memory is in write/store mode
    `WE <= 0`
  - Store value: `DI <= db`
  - `DO` receives `Z`
- `db_dir == 1`
  - Memory is in read/load mode
    `WE <= 1`
  - `DI` receives `db` or `Z`
  - `db <= DO`

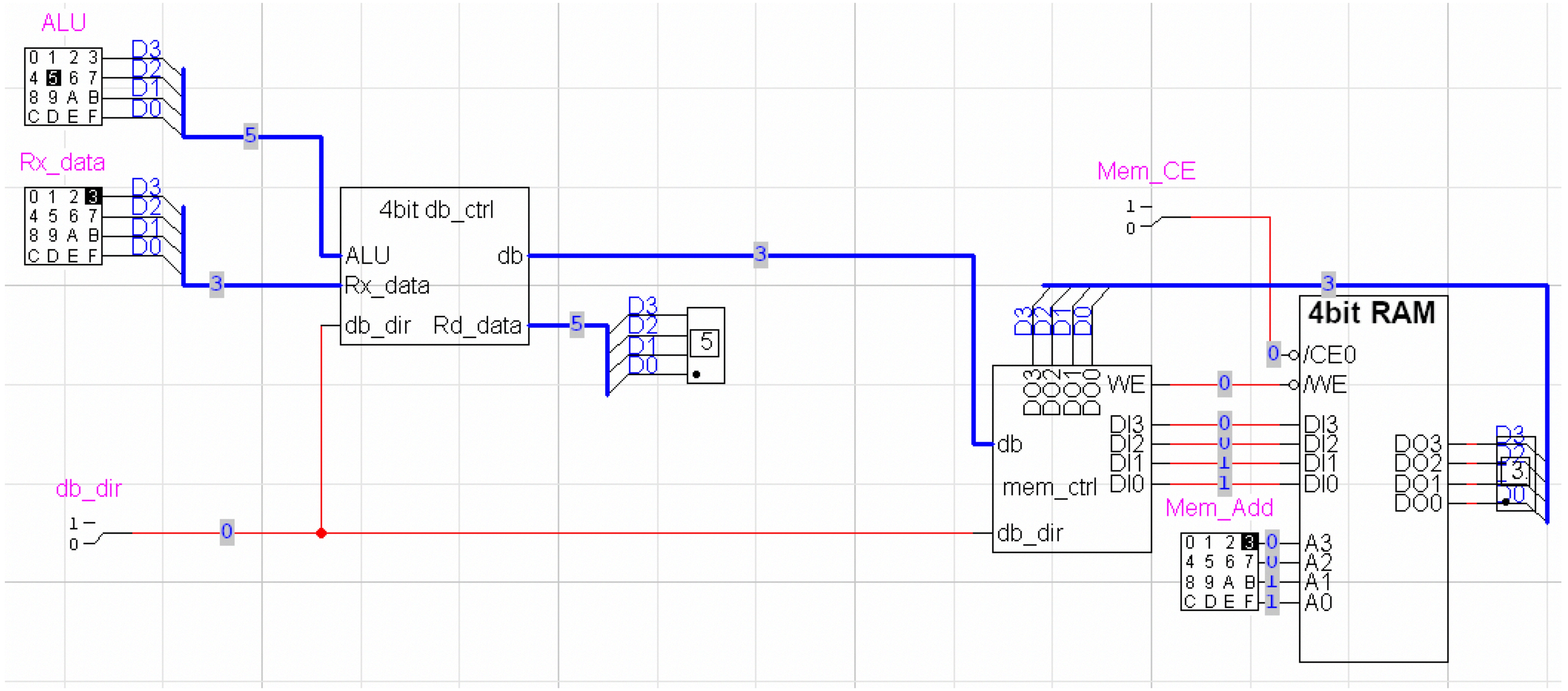


Example

Data Bus Controller

- We need a data bus controller just like the one in our MIPS example
  - Bidirectional: 16bit D16, `db`
  - Input: 16bit D16, `ALU` from ALU; 16bit D16, `Rx_data`, from Register array
  - Input: 1bit `db_dir`
  - Output: 16bit D16 `Rd_data`
- When `db_dir` == 0, `db <= Rx_data; Rd_data <= ALU;`
  When `db_dir` == 1, `Rd_data` <= `db`;

db_ctrl and mem_ctrl tested together



P3 Lab

Register Array Modification

- You need to modify your register array, such that
  - R7's value is updated every CLK, either from `Rt_data`, or to R7 + 1
  - Add an `Rx_data` output bus, selected by `Rx`
  - Add a 16bit `PC` output bus, always showing the value of R7

Lab