Advanced Computer Architectures - Notes - v0.2.0

260236

March 2025

Preface

Every theory section in these notes has been taken from two sources:

- Computer Architecture: A Quantitative Approach. [1]
- Pipelining slides. [2]
- Course slides. [3]

About:

GitHub repository

These notes are an unofficial resource and shouldn't replace the course material or any other book on advanced computer architectures. It is not made for commercial purposes. I've made the following notes to help me improve my knowledge and maybe it can be helpful for everyone.

As I have highlighted, a student should choose the teacher's material or a book on the topic. These notes can only be a helpful material.

Contents

1 Pipelining **4**
1.1 Basic Concepts 4

1 Pipelining

1.1 Basic Concepts

Pipelining is a fundamental **technique** in computer architecture aimed at **improving instruction throughput by overlapping the execution of multiple instructions**. The main idea behind pipelining is to divide the execution of an instruction into distinct stages and process different instructions simultaneously in these stages. This approach significantly increases the efficiency of instruction execution in modern processors.

Understanding the RISC-V instruction set

Before delving into pipelining, it is essential to understand the **basic instruction set** of the RISC-V architecture. The instruction set consists of three major categories:

- **ALU Instructions (Arithmetic and Logic Operations)**
  - Performs **addition between registers**:

```
add rd, rs1, rs2
```

    Performs the addition between the values in registers `rs1` and `rs2` and stores the result in register `rd`.

$$\texttt{rd} \leftarrow \texttt{rs1} + \texttt{rs2}$$

  - Performs an **addition between a constant and a register**:

```
addi rd, rs1, 4
```

    Performs the addition between the value in register `rs1` and the value `4` and stores the result in register `rd`.

$$\texttt{rd} \leftarrow \texttt{rs1} + 4$$

- **Load/Store Instructions (Memory Operations)**
  - **Loads** data from memory:

```
ld rd, offset(rs1)
```

    Load data into register `rd` from an address formed by adding `rs1` to a signed `offset`.

$$\texttt{rd} \leftarrow M\left[\texttt{rs1} + \texttt{offset}\right]$$

  - **Stores** data in memory:

```
sd rs2, offset(rs1)
```

    Store data from register `rs2` to an address formed by adding *rs1* to a signed *offset*.

$$M\left[\texttt{rs1} + \texttt{offset}\right] \leftarrow \texttt{rs2}$$

- **Branching Instructions (Control Flow Management)**
  - **Conditional Branches**
    - Branch on equal:

```
beq rs1, rs2, L1
```

      Branch to the label `L1` if the value in register `rs1` is equal to the value in register `rs2`.

$$\texttt{rs1} = \texttt{rs1} \xrightarrow{\text{go to}} \texttt{L1}$$

    - Branch on not equal:

```
bne rs1, rs2, L1
```

      Branch to the label `L1` if the value in register `rs1` is not equal to the value in register `rs2`.

$$\texttt{rs1} \neq \texttt{rs2} \xrightarrow{\text{go to}} \texttt{L1}$$

  - **Unconditional Jumps**
    - Jump to the label (jump):

```
j L1
```

      Jump directly to the `L1` label.
    - Jump to the address stored in a register (jump register):

```
jr ra
```

      Take the value in register `ra` and use it as the address to jump to. So it is assumed that `ra` contains an address.

These basic instructions will be used throughout the course.

Execution phases in RISC-V

1. **IF (Instruction Fetch)**: The instruction is **fetched** from memory.
2. **ID (Instruction Decode)**: The instruction is **decoded**, and the **required registers are read**.
3. **EX (Execution)**: The instruction is **executed**, typically involving ALU operations.
4. **ME (Memory Access)**: For *load*/*store* instructions, this stage **reads from** or **writes to** memory.
5. **WB (Write Back)**: The **result is written back** to the destination register.

These five stages form the basis of the RISC-V pipeline.

Implementation of the RISC-V Data Path

The **RISC-V Data Path** is a fundamental component of the processor's architecture, responsible for **executing instructions efficiently by coordinating various hardware units**. It defines how instructions flow through different stages of execution, interacting with memory, registers, and the Arithmetic Logic Unit (ALU).

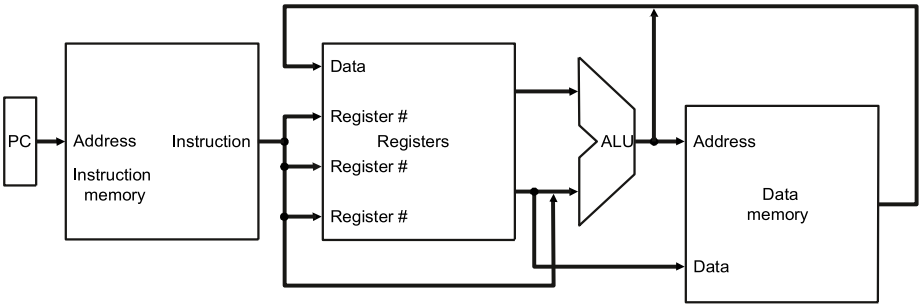

Figure 1: Generic implementation of the RISC-V Data Path.

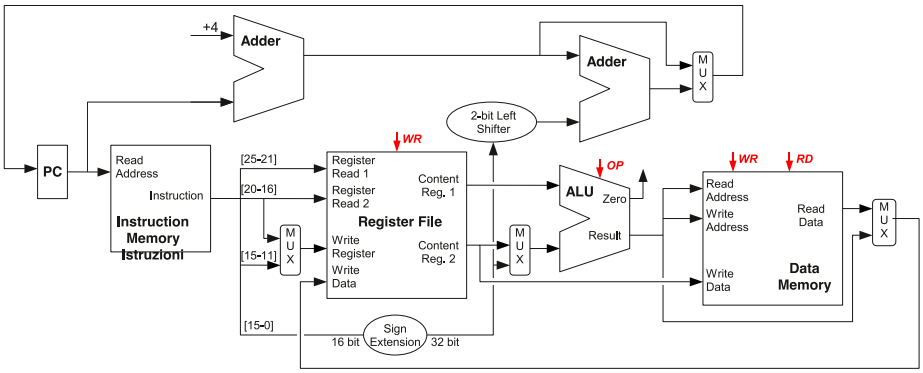

Figure 2: Specific implementation of the RISC-V Data Path.

Its fundamental components include:

- **Instruction Memory and Data Memory Separation**. RISC-V adopts a Harvard Architecture style, where the **Instruction Memory (IM)** and **Data Memory (DM) are separate**. This **prevents structural hazards** where instruction fetch and memory access could conflict in a single-memory design (this topic will be addressed later).
- **General-Purpose Register File (RF)**. It consists of **32 registers**, each **32-bit** wide. The register file has **two read ports** and **one write port** to support simultaneous read and write operations. This setup allows faster register access, which is crucial for pipelined execution.
- **Program Counter (PC)**. It **holds the address of the next instruction to be fetched**. Automatically increments during execution, typically by 4 bytes (for 32-bit instructions).

- **Arithmetic Logic Unit (ALU)**. **Performs arithmetic and logical operations** required by instructions. Inputs to the ALU come from registers or immediate values decoded from the instruction.

Other components that we can see in the general implementation of the RISC-V data path are:

- **Register File**. Stores temporary values used by instructions. Contains read ports (two registers can be read simultaneously for ALU operations) and write port (one register can be updated per clock cycle). The register file ensures high-speed execution of operations by reducing memory accesses.
- **Instruction Fetch (IF)**. The PC (Program Counter) retrieves the next instruction from Instruction Memory. The PC is incremented using an adder (`PC + 4`), ensuring sequential instruction flow.
- **Instruction Decode (ID)**. Extracts opcode (determines the instruction type), source and destination registers, immediate values (if present). It reads values from the Register File based on instruction requirements.
- **Execution (EX)**. The ALU performs arithmetic and logical operations. A multiplexer (MUX) selects the second operand: a register value (for R-type instructions) or an immediate value (for I-type instructions like `addi`). The ALU result is forwarded to the next stage.
- **Memory Access (ME)**. Load (`ld`) and Store (`sd`) instructions interact with data memory. Data is either **loaded from memory into a register** or **stored from a register into memory**.
- **Write Back (WB)**. The result from ALU or memory is **written back to the Register File**.

Example 1: Data Path Execution Example

Let's consider a simple RISC-V **load instruction** (`ld x10, 40(x1)`) passing through the data path:

1. **IF Stage**: Instruction Fetch
   - `PC` → Instruction Memory → `ld x10, 40(x1)` fetched
   - `PC` updated to `PC + 4`
2. **ID Stage**: Instruction Decode
   - Registers read: `x1` (base register for memory access)
   - Immediate value extracted: `40`
3. **EX Stage**: Execution
   - ALU calculates memory address: `x1 + 40`
4. **ME Stage**: Memory Access
   - Data is loaded from `M[x1 + 40]`
5. **WB Stage**: Write Back
   - Data stored in `x10`

References

[1] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. ISSN. Elsevier Science, 2017.

[2] Cristina Silvano. Lesson 1, pipelining. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

[3] Cristina Silvano. Advanced computer architecture. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024-2025.