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

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).

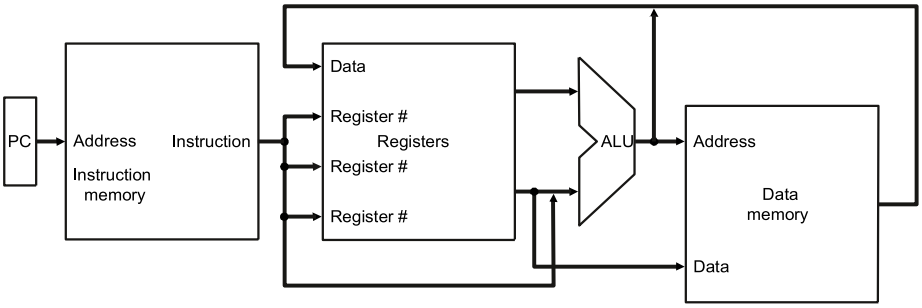

Figure 1: Generic implementation of the RISC-V Data Path.

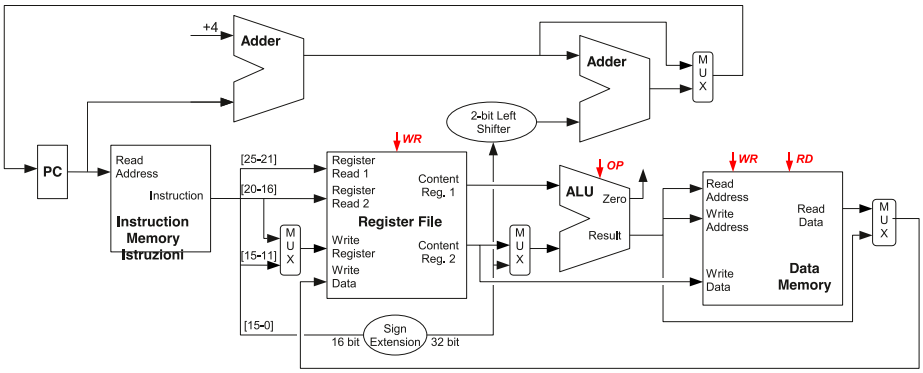

Figure 2: Specific implementation of the RISC-V Data Path.

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`

1.2 RISC-V Pipelining

Pipelining is analogous to an assembly line in a factory. Instead of waiting for one instruction to complete before starting the next, **different instructions are executed simultaneously in different stages**.

If we consider a **non-pipelined execution**:

- Each instruction completes all five stages sequentially before the next instruction starts.
- If each instruction stage (IF stage, ID stage, etc.) takes, say, 2 nanoseconds, executing all stages of an instruction (IF, ID, EX, MEM, WB) takes 5 times 2 nanoseconds, then 10 nanoseconds. If we also want to execute 5 instructions, we need 10 nanoseconds times 5, then 50 nanoseconds!

Now, we consider a **pipelined execution**:

- Once the first instruction moves to the second stage, the next instruction starts in the first stage.
- The **pipeline becomes fully utilized** after the first few cycles, significantly **improving throughput**.

In an **ideal scenario**, a 5-stage pipeline should provide a speedup of 5× reducing execution time to:

$$(5 + 4) \times 2 \text{ ns } = 18 \text{ ns}$$

Where 5 are the steps of the first instruction, 5 are the steps of the last instruction, minus 1 because one step is already counted in the first instruction, so 4. Therefore, 9 is multiplied by 2 nanoseconds, the time taken by each stage. The result, 18 nanoseconds, is the time it takes the pipeline to execute 5 instructions in an ideal scenario.

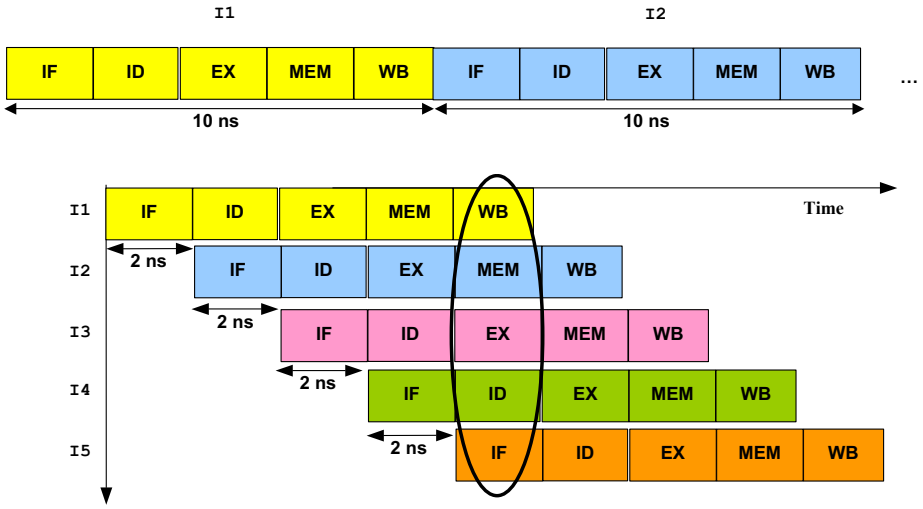

Figure 3: Sequential vs. Pipelining execution.

Pipeline Performance and Speedup

The ideal performance improvement from pipelining is derived from the fact that **once the pipeline is filled**, **a new instruction completes every cycle**. The key performance metrics include:

- **Latency (Execution Time)**: The total time to complete a single instruction does not change (sequential or pipeline).
- **Throughput (Instructions per Unit Time)**: The number of completed instruction per unit time significantly increases.
- **Speedup Calculation**
  - A non-pipelined CPU with 5 execution cycles of 2 ns would take 10 ns per instruction.
  - A pipelined CPU with 5 stages of 2 ns results in 1 instruction completing every 2 ns.
  - This gives a theoretical speedup of 5× (ideal case).

Unfortunately, real-world implementations are subject to **pipeline hazards** that reduce efficiency.

Understanding Pipelining Performance

Pipelining **improves instruction throughput** by allowing multiple instructions to be processed simultaneously in different stages. The **execution of an instruction is divided into 5 pipeline stages**:

1. IF (Instruction Fetch)
2. ID (Instruction Decode)
3. EX (Execution)
4. MEM (Memory Access)
5. WB (Write Back)

Each stage takes 2 ns (a *pipeline cycle*), meaning that an **instruction moves from one stage to the next every 2 ns**. Now, let's analyze the timeline of instruction execution:

Clock Cycle	IF	ID	EX	MEM	WB
1st (0-2 ns)	`I1`				
2nd (2-4 ns)	`I2`	`I1`			
3rd (4-6 ns)	`I3`	`I2`	`I1`		
4th (6-8 ns)	`I4`	`I3`	`I2`	`I1`	
5th (8-10 ns)	`I5`	`I4`	`I3`	`I2`	`I1`
6th (10-12 ns)	`I6`	`I5`	`I4`	`I3`	`I2`
7th (12-14 ns)	`I7`	`I6`	`I5`	`I4`	`I3`
8th (14-16 ns)	`I8`	`I7`	`I6`	`I5`	`I4`
9th (16-18 ns)	`I9`	`I8`	`I7`	`I6`	`I5`

Table 1: Pipelining timeline execution in an ideal case.

- The first instruction `I1` takes 5 (clock) cycles to complete, i.e., 10 ns.
- However, starting from cycle 5, a new instruction finishes every cycle (every 2 ns).
- In a non-pipelined system, each instruction would take 10 ns (5 stages $\times$ 2 ns each).
- In a pipelined system, once the pipeline is full, an instruction completes every cycle (every 2 ns), achieving a 5$\times$ speedup compared to the non-pipelined execution.

Thus, after an initial "fill" time (1st, 2nd, 3rd, 4th), **a new instruction completes every 2 ns** (from 5th to 6th, `I1` is finished; from 6th to 7th, `I2` is finished; from 7th to 8th, `I3` is finished), which is the duration of a single pipeline stage.

1.2.1 Pipelined execution of instructions

Each RISC-V instruction follows the five pipeline stages, but their interactions with the pipeline vary depending on the instruction type.

- **ALU Instructions** (e.g., `op $x, $y, $z`)

  These are register-based operations that do not require memory access. Since there is no memory operation, the instruction **bypasses the ME stage**.

Stage	Description
IF	Fetch instruction from memory
ID	Decode instruction, read registers `$y` and `$z`
EX	Perform ALU operation (`$x = $y + $z`)
ME	No memory access (skipped)
WB	Write the ALU result to `$x`

- **Load Instructions** (e.g., `lw $x, offset($y)`)

  These instructions retrieve data from memory and store it in a register. The **memory access stage** (ME) **is crucial** here since the instruction must fetch data from memory.

Stage	Description
IF	Fetch instruction from memory
ID	Decode instruction, read base register `$y`
EX	Compute memory address (`$y + offset`)
ME	Read data from memory
WB	Write data into destination register `$x`

- **Store Instructions** (e.g., `sw $x, offset($y)`)

  These instructions write data from a register into memory. Unlike `lw`, **store instructions do not require the WB stage**, as data is written directly into memory.

Stage	Description
IF	Fetch instruction from memory
ID	Decode instruction, read base register `$y` and source register $x
EX	Compute memory address (`$y + offset`)
ME	Write `$x` into memory at the computed address
WB	No write-back stage (skipped)

- **Conditional Branches** (e.g., `beq $x, $y, offset`)

  Branching introduces control hazards, as the pipeline needs to determine whether the branch is taken or not. Branches can introduce **stalls** due to dependencies on comparison results. This issue is typically mitigated using branch prediction.

Stage	Description
IF	Fetch instruction from memory
ID	Decode instruction, read registers `$x` and `$y`
EX	Compare `$x` and `$y`, compute target address
ME	No memory access (skipped)
WB	Update PC if branch is taken

This section breaks down how **different types of instructions behave in the pipeline**:

- ALU Instructions complete in the EX stage and do not use memory.
- Load Instructions require a memory access in the ME stage.
- Store Instructions write to memory instead of registers.
- Branch Instructions introduce control hazards because they may change the PC.

This means that **not all instructions behave the same** in the pipeline. Some instructions **skip certain stages** (e.g., stores do not have WB, ALU instructions skip ME), and some instructions **introduce potential problems** (e.g., branches can cause delays).

In conclusion, this section sets the stage for understanding pipeline stalls, forwarding, and hazard resolution techniques that are essential for designing high-performance processors.

1.2.2 Pipeline Implementation

The **RISC-V pipeline implementation** is designed to efficiently execute multiple instructions simultaneously, following the classical five-stage pipeline model:

1. IF (Instruction Fetch)
2. ID (Instruction Decode)
3. EX (Execution)
4. MEM (Memory Access)
5. WB (Write Back)

Each clock cycle, a new instruction enters the pipeline while previous instructions move to the next stage, allowing **five different instructions to be in execution at the same time**.

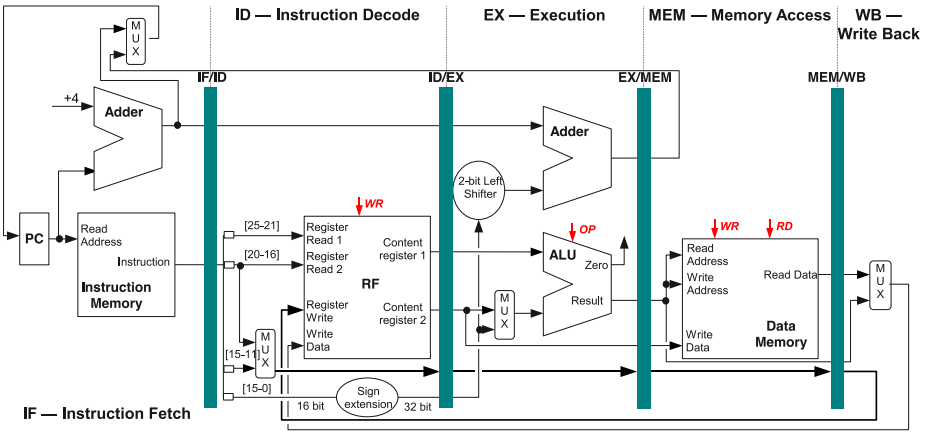

Figure 4: Structure of RISC-V pipeline.

Execution Stages and Pipeline Modules

Each stage of the pipeline corresponds to a specific hardware module in the CPU. The RISC-V pipeline is composed of five primary hardware modules:

- **Instruction Fetch (IF) Module**: Fetches instructions from instruction memory and updates the PC.
- **Instruction Decode (ID) Module**: Decodes the fetched instruction and reads register values.
- **Execution (EX) Module**: Performs arithmetic/logical operations in the ALU or computes memory addresses.
- **Memory Access (MEM) Module**: Reads from or writes data to memory.

- **Write Back (WB) Module**: Writes the computed result back into the register file.

Each module is responsible for a specific **stage of execution**, and together they allow overlapping execution of multiple instructions.

Pipeline Registers

To **maintain separation between stages**, **pipeline registers** are used (see Figure 4, page 14). These registers store intermediate results and ensure proper communication between stages:

- **IF/ID Register**: Holds fetched instruction and updated PC.
- **ID/EX Register**: Stores decoded instruction, read register values, and control signals.
- **EX/MEM Register**: Holds ALU results, destination register, and memory access information.
- **MEM/WB Register**: Stores memory data or ALU result to be written back to registers.

These pipeline registers **eliminate the need for re-fetching or re-decoding instructions** at each cycle, thus maintaining pipeline efficiency.

1.3 Problem of Pipeline Hazards

Assumptions Made

Until now, our discussion on the RISC-V pipeline implementation has relied on several key assumptions to simplify the analysis and focus on fundamental concepts. These **assumptions help in understanding the ideal case of pipelining** before introducing complexities like hazards and optimizations.

1. All instructions are independent, so there are no dependencies between them.
2. No branches or jumps that change execution flow.

This is a theoretical idealization, because in real-world scenarios, **hazards** (structural, data, and control) **interfere with smooth execution**. Also, our second assumption ignores **branch instructions** (`beq`, `bne`, `j`, `jr`), which **cause control hazards** that require branch prediction or pipeline flushing.

What is a Pipeline Hazard?

Now that we have understood the ideal execution of a RISC-V pipeline, we must discuss pipeline hazards, which are obstacles that prevent the pipeline from operating at maximum efficiency.

A **Hazard** (or conflict) is a phenomenon that occurs when the **overlapping execution of instructions in the pipeline changes the expected order of instruction execution**. This can lead to incorrect results or the **need to insert stalls** (*pipeline bubbles*), reducing performance.

In other words, hazards cause the next instruction in the pipeline to be delayed, which reduces the ideal throughput of 1 instruction per cycle. Thus, hazards disrupt the smooth flow of instructions and require techniques to resolve them.

Classes of Pipeline Hazards

- **Structural Hazards**: Attempt to use the same resource from different instructions simultaneously.

  Example: Single memory for both instruction and data access.

- **Data Hazards**: Attempt to use a result before it is ready.

  Example: Instruction depending on a result of a previous instruction still in the pipeline.

- **Control Hazards**: Try to make a decision about the next statement to execute before the condition is evaluated.

  Example: Conditional branch execution.

Structural Hazards

A **structural hazard** occurs when **multiple pipeline stages need to use the same hardware resource at the same time**.

Structural Hazard cannot be applied to RISC-V. This is a great thing, because thanks to the Harvard Architecture, **RISC-V uses separate instruction and data memory**, and this adoption avoids structural hazards.

Control Hazards

A **control hazard** occurs when the **pipeline does not know which instruction to fetch next, usually due to a branch or jump instruction**. It is discussed in the following sections.

Data Hazards

A **data hazard** occurs when an **instruction depends on the result of a previous instruction that is still in the pipeline**.

There are several types of data hazards:

- **RAW (Read After Write)**. An instruction tries to read a register before a previous instruction writes to it.

  Example:

```
1 lw x2, 0(x1)
2 add x3, x2, x4
```

  The `add` instruction needs `x2`, but `x2` is still being fetched from memory in the MEM stage. Without hazard resolution, the processor would get the wrong value for `x2`.

- **WAR (Write After Read)**. A later instruction writes to a register before an earlier instruction reads it (rare in RISC).

- **WAW (Write After Write)**. Two instructions try to write to the same register in the wrong order.

1.3.1 RISC-V Optimized Pipeline

The **RISC-V optimized pipeline** introduces refinements that reduce stalls, improve data access, and enhance instruction throughput. The key optimizations include:

✔ **Efficient Register File Access**. In the standard RISC-V pipeline, register accesses **happen in two stages**:

- ID (Instruction Decode) → Reads register values.
- WB (Write Back) → Writes computed values back to registers.

Optimization: Read and Write in the Same Cycle

- In the optimized pipeline:
  - Register **writing** happens in the *first half* of the **clock cycle**;
  - While register **reading** happens in the *second half* of the **clock cycle**.
- This means an instruction can write its result to a register in WB, and the **next instruction can immediately read** that value in ID during the **same cycle**.

This optimization removes unnecessary stalls when an instruction immediately depends on a result written in the previous cycle.

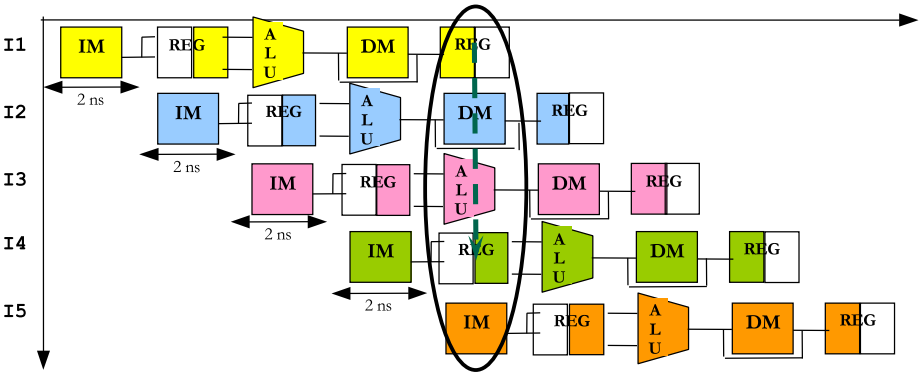

Figure 5: Visual **example** of an optimized pipeline; here the result (WB stage) of `I1` is written in the first half of the clock cycle and the read (ID stage) of `I4` is done in the second half of the clock cycle. So there is no hazards!

✔ **Forwarding (Bypassing) to Reduce Stalls**. **Forwarding** (also called **bypassing**) is a hardware technique that **eliminates stalls by providing ALU results directly to dependent instructions without waiting for the WB stage**. It is a possible solution for Data Hazards.

Forwarding Paths: To support forwarding, the **pipeline includes extra paths** that allow instructions to fetch values from intermediate pipeline registers instead of waiting for WB.

- **EX/EX Path**. Allows ALU results to be forwarded from **EX stage output to the next EX stage input**. Used when an **instruction depends on an arithmetic result of the previous instruction**.

Example 2: EX/EX Forwarding

```
sub x2, x1, x3   # Compute x2 = x1 - x3
and x12, x2, x5  # Use x2 immediately
```

Cycle	`sub x2, x1, x3`	`and x12, x2, x5`
1	IF	
2	ID	IF
3	EX	ID
4	MEM	*Stall*
5	WB	*Stall*
6		EX
7		MEM
8		WB

The `and` instruction **must wait until WB writes `x2` to the register file**. Two stall cycles are introduced and this wastes execution time.
Instead of waiting for WB, we forward the ALU result from the EX stage of `sub` directly to the EX stage of `and`.

Cycle	`sub x2, x1, x3`	`and x12, x2, x5`
1	IF	
2	ID	IF
3	EX	ID
4	MEM	EX (forwarded `x2` from EX)
5	WB	MEM
6		WB

In cycle 4, `and x12, x2, x5` gets the forwarded `x2` from the EX stage of `sub`, **removing stalls**.
This is EX/EX forwarding, taking ALU results from one EX stage directly into the next EX stage.

- **MEM/EX Path**. Forwards the ALU result from **MEM stage to EX stage**. Used when an **instruction depends on an ALU operation two cycles before**.

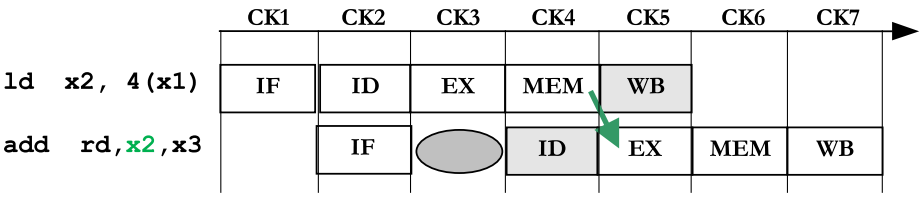

Figure 6: Example of MEM/EX path.

- **MEM/MEM Path**. Forwarding directly **between two memory operations in the MEM stage**. It **removes stalls in Load/Store dependencies**.

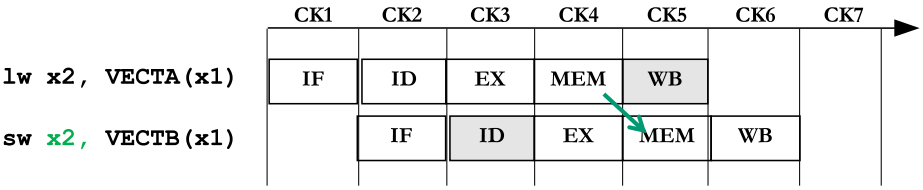

Figure 7: Example of MEM/MEM path.

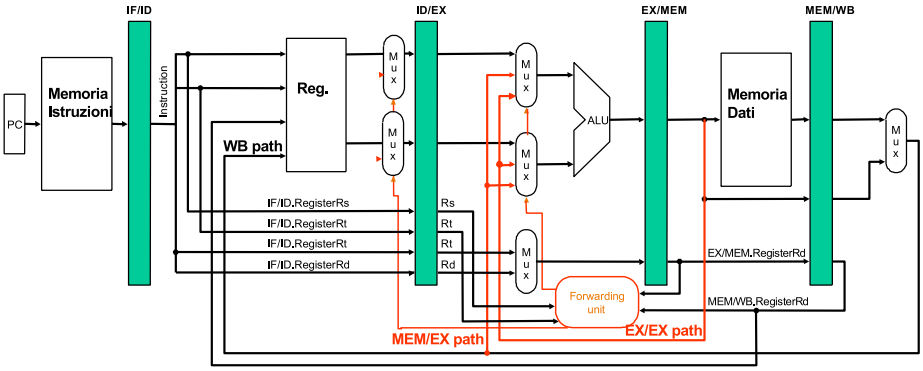

Figure 8: Implementation of RISC-V with Forwarding Unit.

1.3.2 Solutions to RAW Hazards

To handle RAW hazards, we can **use both** static (compile-time) and dynamic (hardware-based) **techniques**. These include:

- **Static (compile-time)**:
  - ✔ `nop` **insertion**: compiler adds empty instructions to delay execution.
  - ✔ **Instruction Scheduling**: compiler reorders instructions to avoid conflicts.
- **Dynamic (hardware-based)**:
  - ✔ **Pipeline Stalling (bubbles)**: inserts delay cycles when necessary.
  - ✔ **Forwarding (bypassing)**: uses intermediate values from the pipeline instead of waiting.

✔ *Static (compile-time) solution*: Inserting `nops` (naïve)

One simple way to handle RAW hazards is to **insert `nop` instructions manually between dependent instructions**. This gives the pipeline time to complete the write-back of the needed value.

Key takeaway of inserting `nops`:

- ✘ **Simple**, but inefficient because it wastes clock cycles. It should be the very last solution considered.
- ✘ Instead of using useful instructions, the **processor waits**, reducing performance.

Example 3: `nop` insertion

```
sub x2, x1, x3
nop               # Delay slot (bubble)
and x12, x2, x5   # Now x2 is ready
```

✔ *Static (compile-time) solution*: Instruction Scheduling

A more efficient technique is **instruction reordering**, also known as **compiler scheduling**. The **compiler reorders instructions to avoid data hazards** without inserting `nops`.

Key takeaway of instruction scheduling:

- Instruction reordering is a **compiler optimization**.
- ✔ It works well **if independent instructions are available**.
- ✘ In some cases, no independent instructions exist, so **stalling or forwarding is needed**.

Example 4: Instruction Scheduling

```
sub x2, x1, x3
# Independent instruction
# (can execute while sub is completing)
add x4, x10, x11
and x12, x2, x5   # Now x2 is ready
```

Instead of a `nop`, we insert `add x4, x10, x11`, which does not depend on `x2`. This keeps the pipeline utilized while avoiding RAW hazards.

Dynamic (hardware-based): Pipeline Stalling (Bubble Insertion)

When no independent instructions can be scheduled, the **hardware must stall the pipeline** by inserting a **bubble (stall cycle)**.

Key takeaway of pipeline stalling:

- ✗ Stalling is simple but **reduces performance** (pipeline sits idle).
- ✓ We **prefer forwarding** (next solution) instead of stalling.

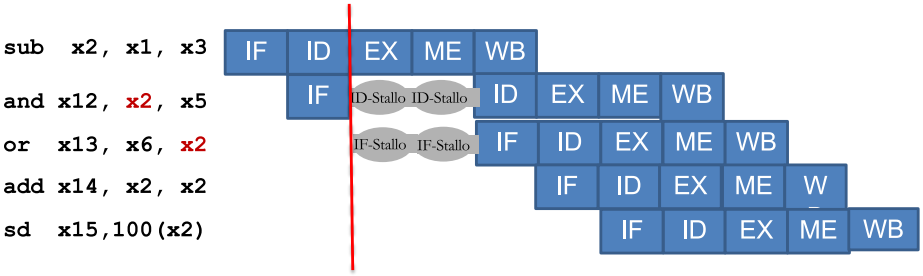

Figure 9: Example of inserting stalls.

Dynamic (hardware-based): Forwarding (Bypassing)

Forwarding is an optimized hardware technique that avoids pipeline stalls by **directly passing results between pipeline registers**. The entire implementation has already been explained on page 18.

Key takeaway of forwarding:

- ✓ **Forwarding is the best solution** because it eliminates stalls and maximizes performance.
- It **requires extra hardware** (MUX and control logic), but it significantly improves throughput.

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.

Index