Advanced Computer Architecture

Rao
Politecnico di Milano

Originally written in: **2025-02-17**

Last updated at: **2025-03-10**

Contents

1 Fundamental Concepts

1.1 Classes of Parallelism and Parallel Architectures

There are basically two kinds of parallelism in **applications**:

- *Data-level parallelism(DLP)* arises because there are many data items that can be operated on at the same time.
- *Task-level parallelism (TLP)* arises because tasks of work are created that can operate independently and largely in parallel.

Computer hardware in turn can exploit these two kinds of application parallelism in four major ways:

1. *Instruction-level parallelism* exploits data-level parallelism at modest levels with compiler help using ideas like **pipelining** and at medium levels using ideas like **speculative execution**.
2. *Vector architectures, graphic processor units (GPUs)*, and *multimedia instruction sets* exploit **data-level parallelism** by applying a single instruction to a col-lection of data in parallel.
3. *Thread-level parallelism* exploits either *data-level parallelism* or *task-level parallelism* in a tightly coupled hardware model that allows for interaction between parallel threads.
4. *Request-level parallelism* exploits parallelism among largely decoupled tasks specified by the programmer or the operating system.

1.2 Registers in RISC-V

RISC-V defines a set of **registers** that are part of the core ISA. RISC-V base ISA consists of 32 ***general-purpose registers*** `x0-x31` which hold integer values. The register `x0` is **hardwired** to the constant `0`. There is an additional user-visible **program counter** `pc` register which holds the address of the current instruction.

As shown in the Figure 1.1, the width of these registers is defined by the RISC-V base variant used. For RV32, the registers are 32 bits wide; for RV64, they are 64 bits wide; and for RV128, the registers are 128 bits wide.

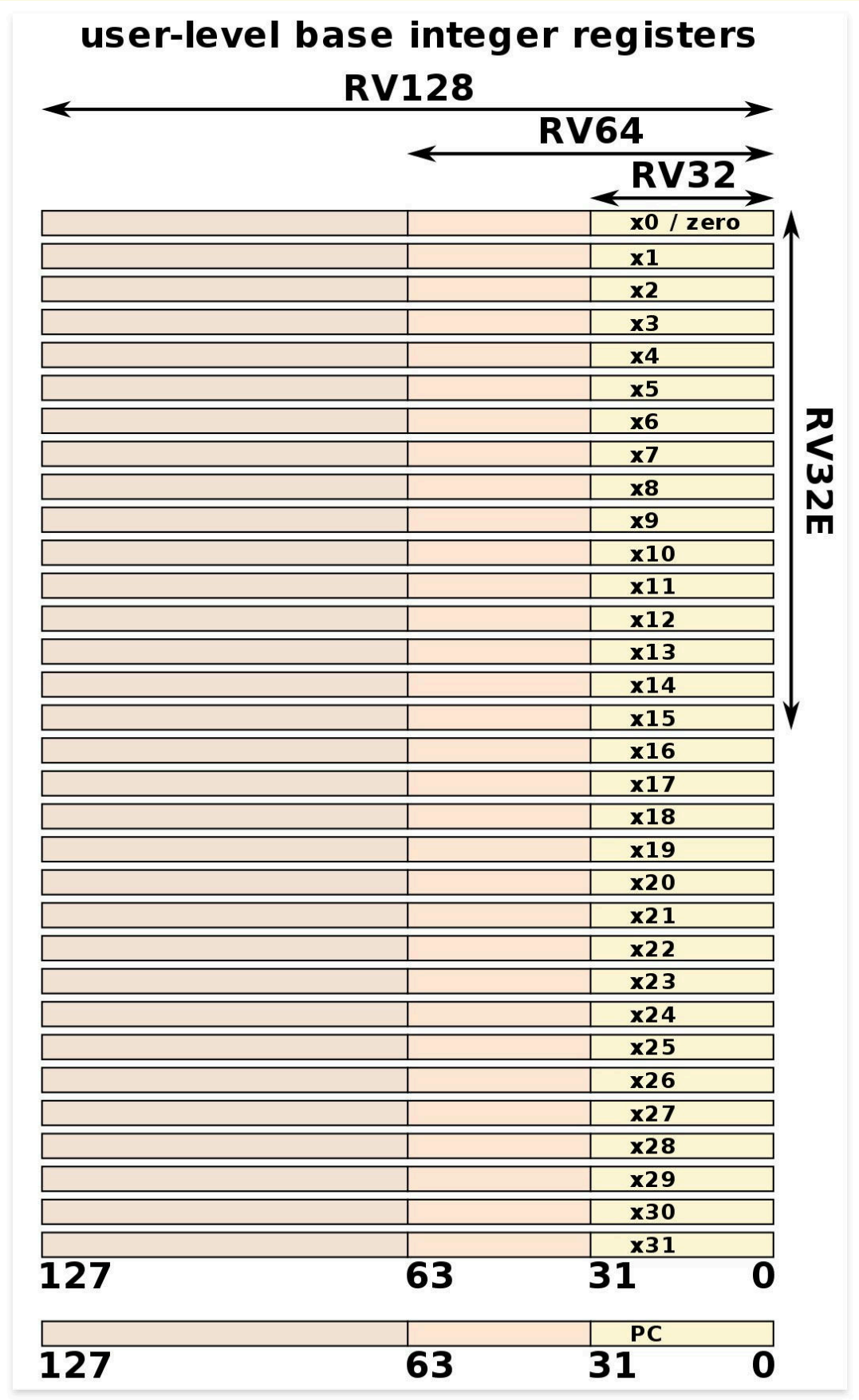

Figure 1.1: RISC-V Registers [1]

1.3 Five classic components of a computer

The five classic components are shown in the Figure 1.2. The five components perform the tasks of **inputting, outputting, processing and storing data**.

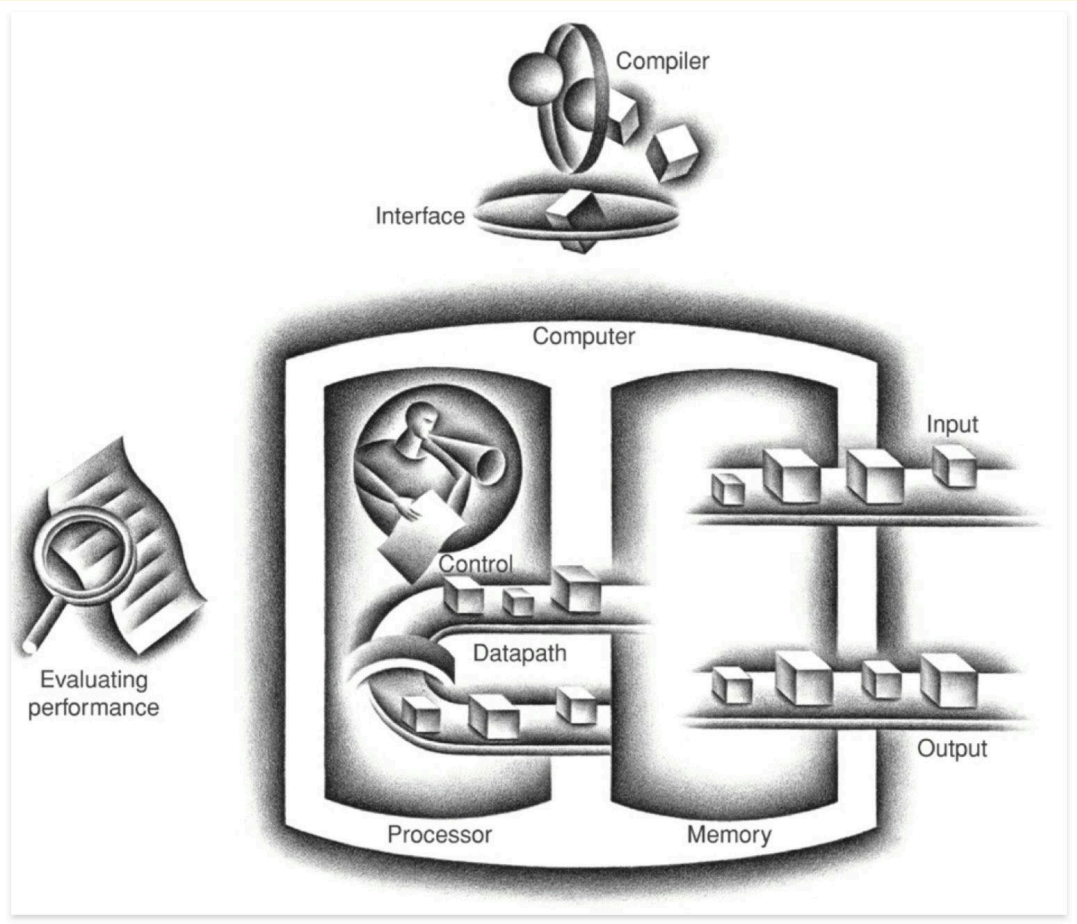

Figure 1.2: Five classic components of a computer

The five classic components of a computer are **input, output, memory, datapath, and control**, with the last two sometimes combined and called the processor.

- The **processor** gets instructions and data from memory.
- **Input** writes data to memory, and **output** reads data from memory.
- **Control** sends the signals that determine the operations of the datapath, memory, input, and output.

1.4 The CPU Performance

Essentially all computers are constructed using a clock running at a constant rate. These discrete time events are called *clock periods, clocks, cycles,or clock cycles*. Computer designers refer to the time of a clock period by its duration (e.g., 1 ns) or by its rate (e.g., 1 GHz). CPU time for a program can then be expressed two ways:

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time} \tag{1.1}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}} \tag{1.2}$$

e.g. Improving Performance **example 1.4.1**

Our favorite program runs in 10 seconds on computer A, which has a 2 GHz clock. We are trying to help a computer designer build a computer, B, which will run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing computer B to require 1.2 times as many clock cycles as computer A for this program.

What clock rate should we tell the designer to target?

Find the number of clock cycles for computer A:

$$\text{CPU time}_A = \frac{\text{CPU clock cycles for a program}_A}{\text{Clock rate}_A}$$

$$10 = \frac{\text{CPU clock cycles for a program}_A}{2 \times 10^9}$$

$$\text{CPU clock cycles}_A = 20 \times 10^9$$

CPU time for B can be found using this equation:

$$\text{CPU time}_B = \frac{\text{CPU clock cycles for a program}_B}{\text{Clock rate}_B}$$

$$6 = 1.2 \times \frac{\text{CPU clock cycles for a program}_A}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 20 \times 10^9}{6} = 4 \text{ Ghz}$$

1.5 Instruction Performance

One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction. Therefore, the number of clock cycles required for a program can be written as **CPI** is computed as:

$$\text{CPU clock cycles} = \text{Instructions fora program} \times \text{Average cycles per instruction} \tag{1.3}$$

The term clock cycles per instruction, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as **CPI**.

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instructions for a program}} \tag{1.4}$$

We can now write this basic performance equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time:

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time} \tag{1.5}$$

or, since the clock rate is the inverse of clock cycle time:

$$\text{CPU time} = \text{Instruction count} \times \frac{\text{CPI}}{\text{Clock rate}} \tag{1.6}$$

2 Pipelining

Pipelining is an implementation technique whereby multiple instructions are **overlapped** in execution; it takes advantage of **parallelism** that exists among the actions needed to execute an instruction.

A pipeline is like an **assembly line**. In a computer pipeline, each step in the pipeline completes a part of an instruction. Like the assembly line, different steps are completing different parts of different instructions in parallel. Each of these steps is called a *pipe stage* or a *pipe segment*.

The ***throughput*** of an instruction pipeline is determined by how often an instruction exits the pipeline. The time required between moving an instruction one step down the pipeline is a **processor cycle**.

Pipelining yields a reduction in the average execution time per instruction. If the starting point is a processor that takes multiple clock cycles per instruction, then pipelining reduces the CPI. Pipelining is an implementation technique that exploits **parallelism** among the instructions in a sequential instruction stream.

2.1 The Basics of the RISC V Instruction Set

All RISC architectures(**RISC V, MIPS, ARM**) are characterized by a few key properties:

1. **All operations on data apply to data in registers** and typically change the entire register(32 or 64 bits).
2. The only operations that affect memory are **load** and **store** operations that move data from memory to a register or to memory from a register, respectively.
3. The instruction formats are few in number, with all instructions typically being one size. In RISC V, the register specifiers: **rs1**, **rs2**, and **rd** are always in the same place simplifying the control.

e.g. RISC-V Instruction Set **example 2.1.1**

Several common types of instructions in RISC-V:

- ALU instructions:
  - **Sum** between two **registers**:

```
add rd, rs1, rs2     # $rd <- $rs1 + $rs2
```

  - **Sum** between **register** and **constant**:

```
addi rd, rs1, imm    # $rd <- $rs1 + imm
```

- Load/Store instructions:
  - **Load**:

```
ld rd, offset (rs1)  # $rd <- Memory[$rs1 + offset]
```

From the **rs1** register, calculate the index on the memory with the **offset**, take the value and store it in the **rd** register.

  - **Store**:

```
sd rs2, offset (rs1) # Memory[$rs1 + offset] <- $rs2
```

Take the value from the **rs2** register and store it in the memory at the index calculated from the **rs1** register and the **offset**.

- Branch instructions to control the instruction flow:
  - **Conditional branches**:the branch is taken only if the condition is true.

Only if the condition is true (branch on equal):

```
beq rs1, rs2, L1 # go to L1 if (rs1 == rs2)
```

Only if the condition is false (branch on not equal):

```
bne rs1, rs2, L1 # go to L1 if (rs1 != rs2)
```

  - **Unconditional branches**: the branch is always taken.

```
j L1              # go to L1
jr ra             # go to add. contained in ra
```

2.2 A Simple Implementation of a RISC Instruction Set

Every instruction in this RISC subset can be implemented in, at most, **5 clock cycles**. The 5 clock cycles are as follows.

1. *Instruction Fetch(IF)*: Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next **sequential instruction** by adding 4 (because each instruction is 4 bytes) to the PC.
2. *Instruction decode/register fetch cycle (ID)*: Decode the instruction and read the registers corresponding to register source specifiers from the register file. Do the equality test on the registers as they are read, for a possible branch. Sign-extend the offset field of the instruction in case it is needed. Compute the possible branch target address by adding the sign-extended offset to the incremented PC.
3. *Execution/effective address cycle (EX)*: The ALU operates on the operands prepared in the prior cycle, performing one of three functions, depending on the instruction type.
4. *Memory access (MEM)*: If the instruction is a load, the memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.
5. *Write-back cycle (WB)*: Write the result into the **register file**, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction).

2.3 Implementation of RISC-V processor

The **Instruction Memory**(read-only memory) is separated from **Data Memory**. 32 General-Purpose Registers organized in a **Register File(RF)** with 2 read ports and 1 write port.

For every instruction, the first two steps are identical:

1. Send the ***program counter (PC)*** to the memory that contains the code and fetch the instruction from that memory.
2. **Read one or two registers**, using fields of the instruction to select the registers to read. For the `ld` instruction, we need to read only one register, but most other instructions require reading two registers.

Fortunately,for each of the three instruction classes(memory-reference,arithmetic-logical, and branches), the actions are largely the same, independent of the exact instruction:

For example, all instruction classes use the **arithmetic-logical unit** (ALU) after reading the registers.

- For the **memory-reference** instructions, the ALU computes the effective address by adding the offset to the base register.

- For the **arithmetic-logical** instructions, the ALU performs the operation specified by the instruction.
- For the **branch** instructions, the ALU compares the two registers to determine if the branch should be taken(**equality test**).

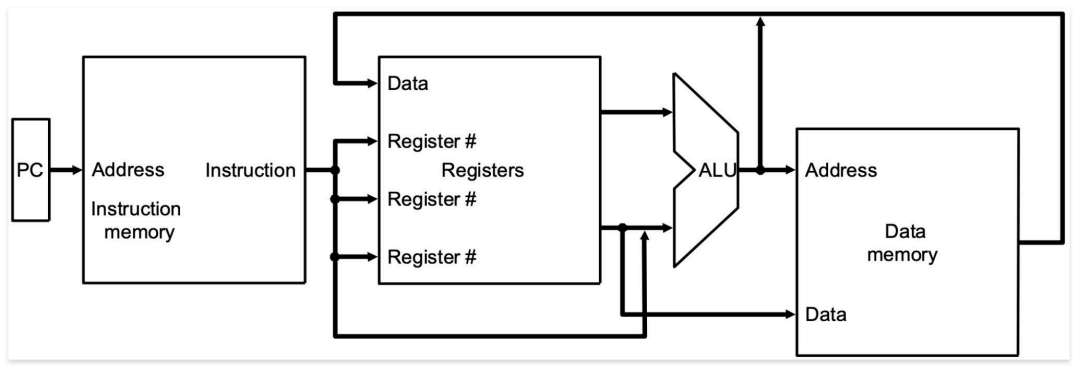

Figure 2.3: Basic Implementation of a RISC Instruction Set

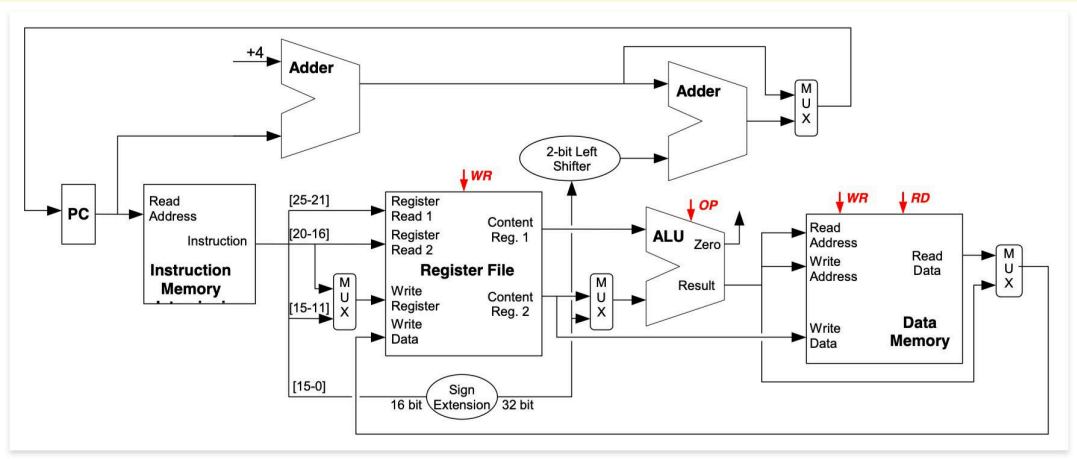

Figure 2.4: A complete implementation of RISC-V data path

2.4 Building a data path

looking at which **datapath elements** each instruction needs, and then work our way down through the levels of abstraction.

Figure 2.5 shows the first element we need: a memory unit to store the instructions of a program and **supply instructions** given an address.

- **Program Counter (PC)** is a register that holds the address of the current instruction.
- **Adder** is used to increment the PC by 4 to get the address of the next instruction and it is permanently made an adder and cannot perform the other ALU functions.
- The **instruction memory** need only provide read access because the datapath does not write instructions.

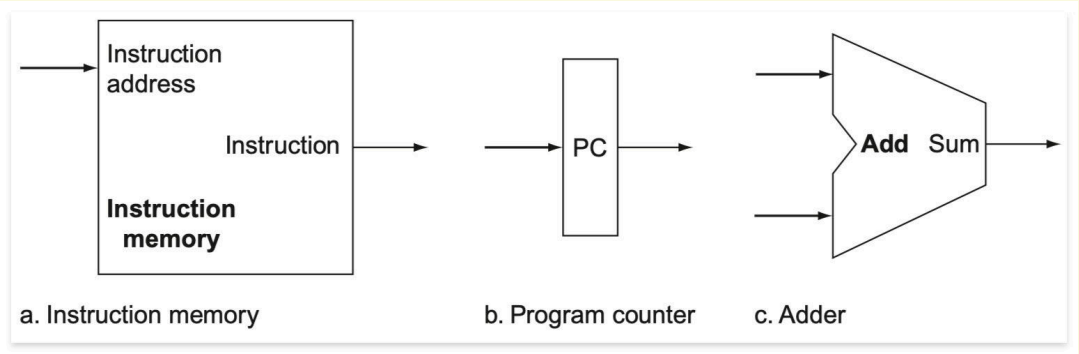

Figure 2.5: First datapath component: Instruction Memory

2.5 RISC-V Pipelining

Pipelining is a performance optimization technique based on the **overlap** of the execution of multiple instructions deriving from a sequential execution flow. Pipelining exploits **instruction parallelism** in a sequential instruction stream.

Sequential is slower than pipeline. The following figure shows the difference (in terms of clock cycles) between sequential and pipeline.

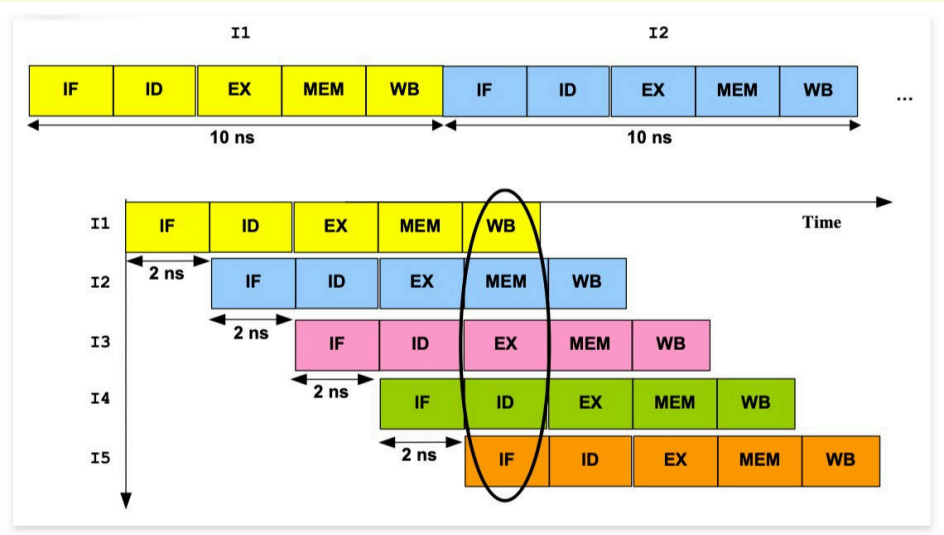

Figure 2.6: Sequential vs Pipeline

The time to advance the instruction of one stage in the pipeline corresponds to a **clock cycle**. The total cost is: 9 clock cycles.

The pipeline stages must be synchronized, the duration of a clock cycle is defined by the time requested by the **slower stage** of the pipeline. The goal is to balance the length of each pipeline stage. If the stages are perfectly balanced, the **ideal speedup** due to pipelining is equal to the number of pipeline stages.

The sequential and pipelining cases consist of 5 instructions, each of which is divided into 5 low-level instructions of 2 ns each.

- The **latency** (total execution time) of each instruction is not varied, it's always 10 ns.

- The **throughput** (number of low-level instructions completed in the time unit) is improved:
  - Sequential: 1 instruction completed every 10 ns
  - Pipelining: 1 instruction completed every 2 ns

We want to perform the following assembly lines:

```
1 op $x , $y , $z        # assume $x <- $y + $z
2 lw $x , offset ($y)    # $x <- M[$y + offset ]
3 sw $x , offset ($y)    # M[$y + offset ] <- $x
4 beq $x , $y , offset
```

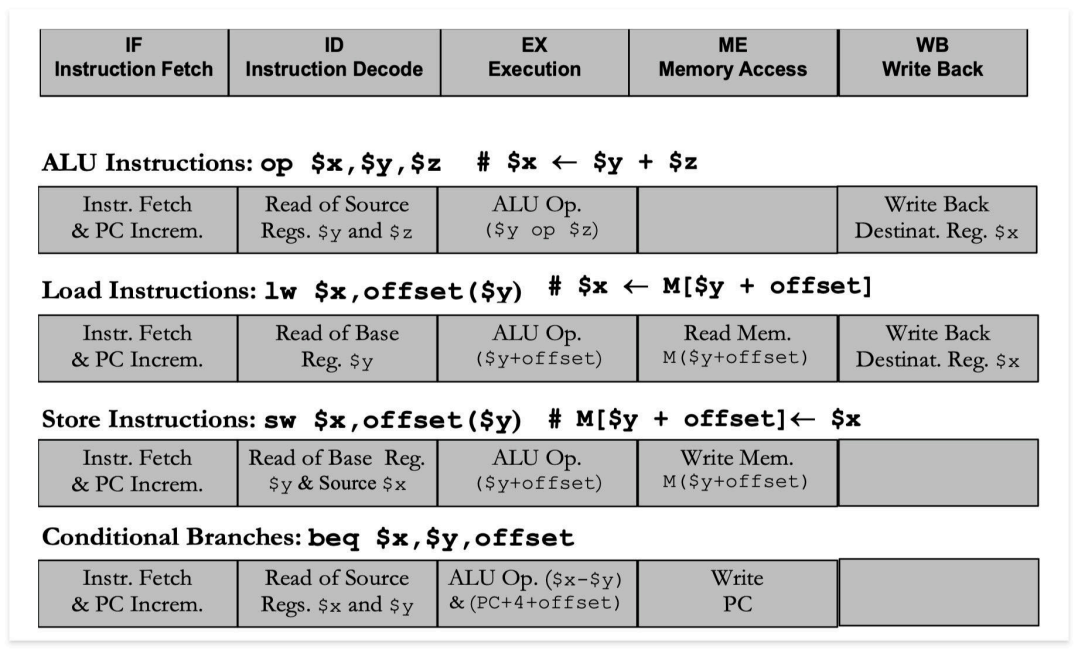

Figure 2.7: Pipeline Execution of RISC-V Instructions

2.6 Resources used during the pipeline execution

IM is Instruction Memory, **REG** is Register File and **DM** is Data Memory.

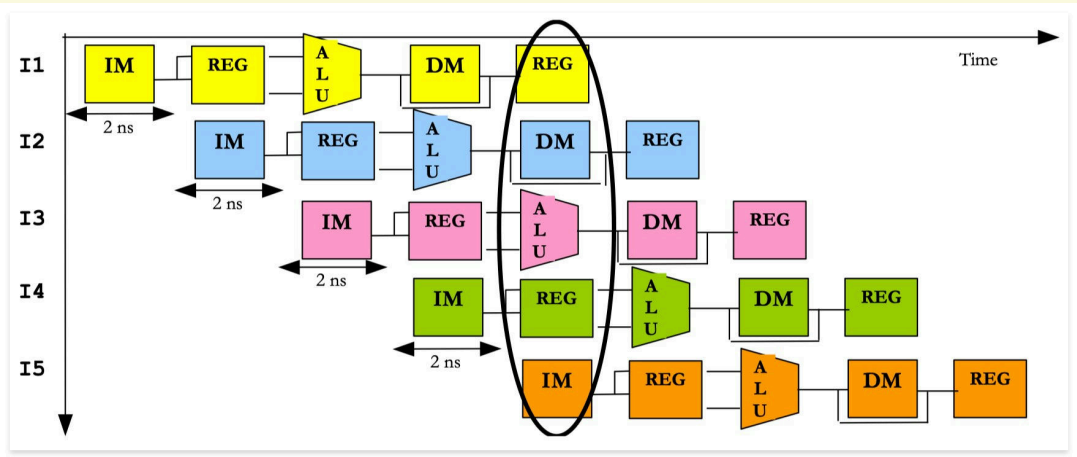

Figure 2.8: Resources used during the pipeline execution

2.7 Performance Metrics

IC = Instruction Count, CPI = Clocks Per Instruction, IPC = Instructions Per Clock Cycle = 1 / CPI, MIPS = $f_{\text{clock}}/(\text{CPI} \times 10^6)$.

Clock Cycles = IC + Stall Cycles + 4. 4 is clocks to conclude the pipeline.

e.g. Speedup of Pipeline **example 2.7.1**

Consider the unpipelined processor in the previous section. Assume that it has a **4 GHz** clock (or a 0.5 ns clock cycle) and that it uses **four cycles** for ALU oper-ations and branches and **five cycles** for memory operations.

Assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively.

Suppose that due to clock skew and setup, **pipelining the processor adds 0.1 ns** of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

The average instruction execution time on the unpipelined processor is:

$$\begin{aligned}\text{Average instruction execution time} &= \text{Clock cycle} \times \text{Average CPI} \\ &= 0.5 \text{ ns} \times [(40\% + 20\%) \times 4 + 40\% \times 5] \\ &= 0.5 \text{ ns} \times 4.4 \\ &= 2.2 \text{ ns}\end{aligned} \tag{2.1}$$

In the pipelined implementation, the clock must run at the speed of the slowest stage plus overhead, which will be $0.5 + 0.1$ or 0.6 ns; this is the average instruction execution time. Thus, the speedup from pipelining is:

$$\begin{aligned}\text{Speedup} &= \frac{\text{Unpipelined execution time}}{\text{Pipelined execution time}} \\ &= \frac{2.2 \text{ ns}}{0.6 \text{ ns}} \\ &= 3.67 \text{ times}\end{aligned} \tag{2.2}$$

2.8 Pipeline Hazards

There are situations, called ***hazards***, that prevent the next instruction in the instruc- tion stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are **three classes of hazards**.

2.8.1 Structural Hazards

Structural hazards occur when we attempt to use the same resource from different instructions simutaneously.

The RISC-V architecture avoids **structural hazards** by two key design decisions:

1. **Separate Instruction and Data Memories (*Harvard-Style Architecture*)**: The **fetch stage (IF)** of the pipeline accesses the **Instruction Memory** to read the next instruction. The **memory stage (MEM)** of the pipeline accesses the **Data Memory** to read/write operands. Since the **IM** and **DM** are separate, fetching an instruction and accessing data can happen in parallel without resource competition.
2. **Multiple-Ported Register File Design**: Read ports allow instructions in the decode stage (ID) to read operands. Write ports allow instructions in the write-back stage (WB) to update registers. Register file read/write operations can occur in the same clock cycle without conflict.

Bibliography

[1] "Registers - RISC-V - WikiChip." [Online]. Available: https://en.wikichip.org/wiki/risc-v/registers