ADVANCED COMPUTER ARCHITECTURES

Claudio Tessa

December 25, 2025

Contents

1 Pipelining: basic concepts **2**
1.1 RISC-V instruction set 2
1.2 Phases of execution of RISC-V instructions 2
1.3 RISC-V Pipelining 3
1.4 Pipeline Hazards 3
1.5 Data Hazards 4
1.6 Performance Evaluation Of Pipelines 4

1 Pipelining: basic concepts

Pipelining is an implementation technique where multiple instructions are overlapped in execution. Like an assembly line, every step of the pipeline completes a part of an instruction.

1.1 RISC-V instruction set

RISC architectures are characterized by a few key **properties**:

- All operations on data apply to data in registers and change the entire register.
- The only operations that affect memory are load and store operations.
- The instruction formats are few in number.

These simple properties lead to dramatic simplification in the implementation of the pipelining.

There are three classes of instructions:

1. **ALU instructions** - These instructions take two registers (or a register and a sign-extended immediate), operate on them, and store the result into a third register.
   - `add rd, rs1, rs2` $\Longleftrightarrow$ `rd = rs1 + rs2`
   - `addi rd, rs1, 4` $\Longleftrightarrow$ `rd = rs1 + 4`
2. **Load and store instructions** - These instructions take a register source and an immediate, called *offset*. The sum of the content of the source register and the offset is used as the memory address. Another register is also taken to performing the operation.
   - `ld rd, offset (rs1)` $\Longleftrightarrow$ `rd = Memory[rs1 + offset]`
   - sd rs2, offset (rs1) $\Longleftrightarrow$ `Memory[rs1 + offset] = rs2`
3. **Branches and Jumps** - Branches are conditional transfers of control, while jumps are unconditional.
   - `beq rs1, rs2, L1` $\Longleftrightarrow$ `if (rs1 == rs2) then go to L1`
   - `j L1` $\Longleftrightarrow$ `go to L1`

1.2 Phases of execution of RISC-V instructions

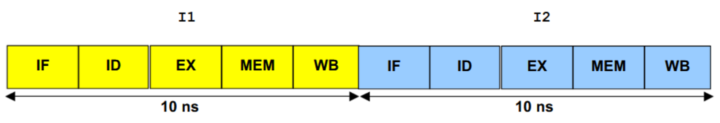
Every instruction takes at most 5 clock cycles. The 5 clock cycles are as follows:

1. **IF (Instruction Fetch)** - Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by, adding 4 as every instruction is 4 bytes long in memory.
2. **ID (Instruction Decode)** - Decode the instruction and read the registers specified in the instruction. Decoding is done in parallel with reading registers.
3. **EX (execution)** - The ALU operates on the operands prepared in the previous cycle.

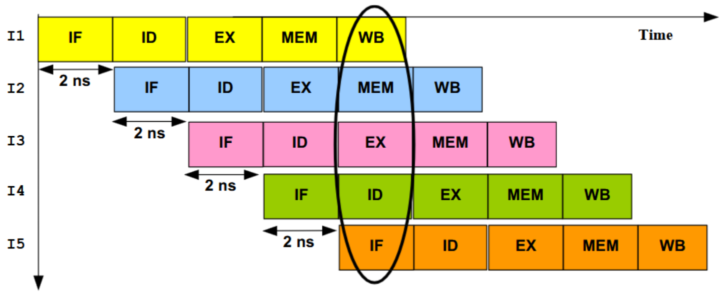
4. **MEM (Memory Access)** - Based on the instruction, the memory writes or reads data using the effective address computed in the previous cycle.
5. **WB (Write Back)** - Register-Register ALU instruction or lead instruction

1.3 RISC-V Pipelining

Instead of executing every instruction sequentially, as follows:



we can pipeline the execution by simply starting a new instruction on each clock cycle:



Each of the clock cycles now becomes a *pipe stage*. Although each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction.

However, we must ensure that the instructions in different stages of the pipeline do not interfere with one another. This separation is done by introducing **pipeline registers** between successive stages of the pipeline, so that at the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next cycle.

1.4 Pipeline Hazards

A **hazard** (conflict) is a situation that prevents the next instruction from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

1. **Structural hazards** - Attempt to use the same resource from different instructions simultaneously.
2. **Data hazards** - Attempt to use a result before it is ready.
3. **Control hazards** - Attempt to make a decision on the next instruction to execute before the condition is evaluated.

Hazards in the pipeline can make it necessary to *stall* the pipeline.

Note that there are *no* structural hazards in the RISC-V architecture, as the instruction memory is separated from the data memory, and the same register can be read and written in the same clock cycle by different instructions.

1.5 Data Hazards

Data hazards can arise when instructions that are dependent on each other are *too close* in the pipeline. Data hazards generate **Read After Write (RAW)** hazards in the pipeline. RAW hazards happen when instruction $n + 1$ tries to read a source operand before the previous instruction n has written its value in said register.

dgfds —

Example

![[Pasted image 20250428004558.png — center—500]]

—

We will assume that the RF (register file) reads from registers in the second half of the clock cycle and writes to registers in the first half of the clock cycle. In this way, it is possible to read and write the same register in the same clock cycle without any stall.

Now, to solve the data hazard problem, we can use different techniques:

- **Compilation techniques** (*static-time techniques*) - They must be implemented before running the program, at compile time. - Insertion of ‘nop‘ (no operation) instructions. A ‘nop‘ effectively makes the processor ”wait” (by executing empty instructions) one clock cycle. Enough ‘nop‘ can be inserted until the hazard is no longer there. - Instruction scheduling to avoid that correlating instructions are too close. The compiler tries to insert independent instructions among correlated instructions, otherwise inserts ‘nop‘. - **Hardware techniques** (*runtime techniques*) - Can be implemented directly at runtime when an hazard is detected. - Insertion of *stalls* in the pipeline. Stalls effectively make the pipeline wait a few clock cycles until the current instruction finishes and the hazard is no longer there. They are analogous in performance to ‘nop‘ instructions, but they stop the instructions at any stage, rather than just making the processor wait before starting a new instruction. - Data forwarding or bypassing. It works by using temporary results stored in the pipeline registers instead of waiting for the write back of results in the RF. In other words, the result from the ALU can be fed back in the ALU at the next stage, rather than waiting for it to be stored and then read from a register.

1.6 Performance Evaluation Of Pipelines

To evaluate the performance we use the following metrics: **IC** (Instruction Count), CPI (**Clocks** Per Instructions), and IPC (**Instructions** Per Clock):

- #Clock Cycles = IC + #Stall Cycles + 4 - CPI = $\frac{\text{\#Clock Cycles}}{\text{IC}} = \frac{\text{IC + \#Stall Cycles + 4}}{IC}$ - IPC = $\frac{1}{\text{CPI}}$ - MIPS = $\frac{f_{\text{clock}}}{CPI \times 10^6}$, where - f_{clock} is the frequency of the clock - MIPS stands for Millions of Instructions per Second

Let’s now evaluate the performance of a loop. Consider n iterations of a loop composed of m instructions per iteration, requiring k stalls per iteration:

- $\text{IC}_{\text{per\_iter}} = m$ - # Clock $\text{Cycles}_{\text{per\_iter}} = \text{IC}_{\text{per\_iter}} +$ # Stall $\text{Cycles}_{\text{per\_iter}} + 4$ - $\text{CPI}_{\text{per\_iter}} = \frac{\text{\# Clock Cycles}_{\text{per\_iter}} = \text{IC}_{\text{per\_iter}} + \text{\# Stall Cycles}_{\text{per\_iter}} + 4}{\text{IC}_{\text{per\_iter}}} = \frac{m + k + 4}{m}$ - $\text{MIPS}_{\text{per\_iter}} =$

$\dfrac{f_{\text{clock}}}{\text{CPI}_{\text{per\_iter}} \times 10^6}$

Let's now evaluate the asymptotic performance on the same loop:

- $\text{IC}_{\text{AS}} = m{\cdot}n$ - # Clock Cycles $= \text{IC}_{\text{AS}} + \#$ Stall $\text{Cycles}_{\text{AS}} + 4$ - $\text{CPI}_{\text{AS}} = \lim_{n\to\infty} \dfrac{\text{IC}_{\text{AS}} + \#\ \text{Stall Cycles}_{\text{A}}}{\text{IC}_{\text{AS}}}$

$\lim_{n\to\infty} \dfrac{m \cdot n + k \cdot n + 4}{m \cdot n} = \dfrac{m+k}{m}$ - $\text{MIPS}_{\text{AS}} = \dfrac{f_{\text{clock}}}{\text{CPI}_{\text{AS}} \times 10^6}$

Overall, the **ideal CPI** on a pipelined processor would be 1, but stalls cause the pipeline performance to degrade from the ideal performance, so we have:

$\text{Avg CPI} = \dfrac{\text{Ideal CPI} + \text{Pipe Stall Cycles per Instruction}}{1 + \text{Pipe Stall Cycles per Instruction}}$

Pipeline stall cycles per instructions are due to structural hazards, data hazards, control hazards, and memory stalls.