Pipelining in Computer Architecture: A Technical Review

Bilal ABDULHADI
Computer Engineering Department
Biruni University
Istanbul, Turkey
251132903@st.biruni.edu.tr

Ali OKATAN
Computer Engineering Department
Biruni University
Istanbul, Turkey
aokatan@biruni.edu.tr

***Abstract*—The continuous demand for higher computational performance has made processor design one of the central topics in computer architecture. Among the techniques used to improve processor throughput, pipelining remains one of the most fundamental and widely adopted. Rather than completing one instruction entirely before starting the next, a pipelined processor divides instruction execution into several stages and allows multiple instructions to occupy different stages at the same time. This organization improves hardware utilization and increases instruction throughput without necessarily requiring a proportional increase in hardware resources or clock frequency.**

This paper presents a technical review of pipelining in computer architecture. It explains the motivation behind pipelined execution, describes the classical five-stage RISC pipeline, and discusses the main types of pipeline hazards: data, control, and structural hazards. It also reviews common techniques used to reduce pipeline stalls, including forwarding, stalling, branch prediction, and instruction scheduling. Finally, the paper highlights the advantages and limitations of pipelining and explains why it continues to be an essential foundation for modern processor design.

***Index Terms*—Computer Architecture, Pipelining, Instruction-Level Parallelism, CPU Performance, Pipeline Hazards, RISC Processors**

I. Introduction

Processor performance has always been a major concern in computer system design. As software systems become more complex, processors must execute larger numbers of instructions efficiently while keeping power, cost, and hardware complexity under control. In early processor designs, instructions were executed sequentially. A processor would complete all steps of one instruction before beginning the next one. Although this model is simple and easy to understand, it does not make efficient use of the internal hardware components of the processor [2].

The main weakness of sequential execution is that different processor units remain idle at different moments. For example, while an instruction is being decoded, the arithmetic logic unit may not be doing useful work. Similarly, while a memory access is taking place, other components may be waiting for the next instruction. This idle time reduces overall processor efficiency and limits instruction throughput [3].

To address this limitation, computer architects introduced instruction-level parallelism (ILP), which allows different instructions or different parts of instructions to be processed at the same time. Pipelining is one of the clearest and most important forms of ILP. It organizes instruction execution into a sequence of stages, allowing several instructions to move through the processor in an overlapping manner. Once the pipeline is filled, one instruction can ideally be completed during each clock cycle [1].

The purpose of this paper is to explain the concept of pipelining, describe its main stages, and analyze the challenges that appear when instructions overlap. The paper also reviews several techniques used to improve pipeline performance and reduce the negative effect of hazards.

II. Background

In a non-pipelined processor, each instruction must pass through all required execution steps before the next instruction can begin. These steps typically include fetching the instruction from memory, decoding it, executing the required operation, accessing memory if necessary, and writing the result back to a register. Since only one instruction is active at a time, many hardware resources are not used continuously [2].

Pipelining improves this situation by dividing instruction execution into smaller stages. Each stage is handled by a different part of the processor, and multiple instructions can be processed at the same time as long as they are in different stages. This does not usually reduce the execution latency of a single instruction. Instead, it increases throughput, which means that more instructions can be completed in a given period of time [1].

A useful way to understand pipelining is to distinguish between latency and throughput. Latency is the time required for one instruction to travel from the beginning to the end of the pipeline. Throughput is the number of instructions completed per unit of time. Pipelining mainly improves throughput by keeping the processor's internal components busy more often.

III. Basic Concept of Pipelining

Pipelining splits instruction execution into a number of ordered stages. Each stage performs one part of the instruction-processing task. After an instruction finishes one stage, it moves to the next stage, while a new instruction enters the

stage that has just become free. When the pipeline is full, several instructions are being processed simultaneously [3].

In an ideal pipeline with no hazards and perfectly balanced stages, the processor can complete one instruction per clock cycle after the initial filling period. However, real processors rarely achieve this ideal behavior all the time. Instruction dependencies, branches, and shared hardware resources can interrupt the smooth flow of instructions and cause stalls [1].

The performance benefit of pipelining depends on several factors, including the number of pipeline stages, the balance between stage delays, the frequency of hazards, and the effectiveness of hazard-handling techniques. A deeper pipeline may allow a higher clock frequency, but it can also increase the cost of stalls and branch mispredictions [2].

IV. Pipeline Stages

A classical RISC processor is often explained using a five-stage pipeline model. This model is widely used in computer architecture education because it clearly shows how instruction execution can be divided into simple and organized steps [1], [2].

A. Instruction Fetch

In the instruction fetch (IF) stage, the processor fetches the next instruction from memory. The address of the instruction is usually stored in the program counter (PC). After the instruction is fetched, the PC is normally updated to point to the next instruction in sequence.

B. Instruction Decode

In the instruction decode (ID) stage, the processor interprets the instruction and identifies the operation that must be performed. During this stage, the required source registers are also read. The control unit generates the necessary control signals for later stages of the pipeline.

C. Execute

In the execute (EX) stage, the arithmetic logic unit performs the required arithmetic or logical operation. For branch instructions, this stage may also evaluate the branch condition and calculate the target address. The execute stage is therefore important for both computation and control-flow decisions.

D. Memory Access

In the memory access (MEM) stage, the processor reads from or writes to data memory when the instruction requires it. Load instructions read data from memory, while store instructions write data to memory. Instructions that do not need memory access simply pass through this stage.

E. Write Back

In the write back (WB) stage, the result of the instruction is written to the destination register. This result may come from the arithmetic logic unit or from memory, depending on the instruction type. Once write back is complete, the instruction has finished its movement through the pipeline.

TABLE I
Classical Five-Stage Pipeline Model

Stage	Main Function
IF	Fetches the instruction from memory using the program counter.
ID	Decodes the instruction and reads the required registers.
EX	Performs arithmetic, logical, or branch-related operations.
MEM	Accesses data memory when required by load or store instructions.
WB	Writes the final result back to the destination register.

V. Pipeline Hazards

Although pipelining improves throughput, it also introduces situations where the next instruction cannot safely proceed in the expected clock cycle. These situations are known as pipeline hazards. A hazard may cause the processor to stall, forward data, flush instructions, or use prediction techniques [1], [2].

A. Data Hazards

A data hazard occurs when an instruction depends on the result of a previous instruction that has not yet completed. For example, if one instruction calculates a value and the following instruction needs that value immediately, the second instruction may read an old or incorrect value unless the processor handles the dependency correctly [1].

The most common type of data hazard in simple pipelines is the read-after-write (RAW) hazard. This happens when an instruction attempts to read a register before an earlier instruction has written the correct value into it. Without proper handling, data hazards can lead to incorrect program execution.

B. Control Hazards

Control hazards are caused by branch and jump instructions. When a conditional branch appears in the pipeline, the processor may not immediately know which instruction should be fetched next. If the branch is taken, the next instruction comes from the branch target. If it is not taken, execution continues sequentially [2].

Since the pipeline fetches instructions before the branch outcome is fully known, it may fetch the wrong instructions. If this happens, those instructions must be removed from the pipeline, creating a performance penalty.

C. Structural Hazards

A structural hazard occurs when two or more pipeline stages need the same hardware resource at the same time. For example, if instruction memory and data memory share the same physical memory unit, the processor may be unable to fetch a new instruction while another instruction is accessing memory [1].

Structural hazards can often be reduced by duplicating hardware resources or designing the processor so that different pipeline stages use separate units.

VI. Techniques for Improving Pipeline Performance

Several techniques are used to reduce the impact of hazards and keep the pipeline as full as possible. These techniques are important because the performance advantage of pipelining depends on maintaining a continuous flow of instructions [2].

A. Forwarding

Forwarding, also called bypassing, allows intermediate results to be sent directly from one pipeline stage to another without waiting for the result to be written back to the register file. This technique is especially useful for reducing data hazards. By forwarding the needed value early, the processor can avoid unnecessary stalls in many cases [1].

B. Pipeline Stalling

Pipeline stalling is used when a hazard cannot be resolved immediately. During a stall, the processor temporarily delays one or more instructions until the required condition is satisfied. Although stalling reduces performance, it preserves correctness. In practice, stalls are sometimes unavoidable, especially in load-use situations where the required data is not available early enough [2].

C. Branch Prediction

Branch prediction attempts to guess the outcome of a branch before it is fully resolved. If the prediction is correct, the pipeline continues with little or no interruption. If the prediction is incorrect, the incorrectly fetched instructions must be flushed, and the processor must fetch from the correct path.

Modern processors often use dynamic branch prediction techniques that learn from the behavior of previous branches. Accurate branch prediction is especially important in deeper pipelines, where a wrong prediction can waste many clock cycles [1].

D. Instruction Scheduling

Instruction scheduling can be performed by a compiler or by hardware. The goal is to reorder instructions in a way that reduces dependencies and keeps useful work in the pipeline. For example, a compiler may place an independent instruction between a load instruction and a dependent instruction to reduce the chance of a stall.

Instruction scheduling improves performance without necessarily adding new hardware, but it must preserve the original meaning of the program [2].

VII. Methodology and Architectural Considerations

This paper follows a literature-review methodology based on established computer architecture textbooks and classical processor design concepts. The discussion focuses on the standard five-stage RISC pipeline, common hazard types, and widely used techniques for improving pipeline behavior [1]–[3].

Several architectural factors must be considered when designing a pipelined processor. One important factor is pipeline depth. A deeper pipeline divides instruction execution into more stages, which may allow a higher clock frequency. However, deeper pipelines also increase the penalty of stalls and branch mispredictions because more instructions may need to be delayed or flushed [1].

Another important factor is the balance between pipeline stages. If one stage takes much longer than the others, it can determine the minimum clock period and reduce the benefit of pipelining. A good pipeline design attempts to divide work so that stage delays are as balanced as possible.

Hazard-handling hardware also affects the design. Forwarding paths, hazard detection units, branch prediction mechanisms, and additional memory ports can improve performance, but they increase complexity. Therefore, pipelining is not only a performance technique but also a design trade-off between speed, cost, complexity, and correctness [2].

VIII. Advantages and Limitations of Pipelining

Pipelining provides several important advantages. First, it improves processor throughput by allowing multiple instructions to be active at the same time. Second, it increases the utilization of processor resources because different units can work on different instructions during the same clock cycle. Third, it forms a foundation for more advanced processor techniques such as superscalar execution and out-of-order execution [1].

However, pipelining also has limitations. It increases design complexity because the processor must detect and handle hazards correctly. It may suffer performance degradation when programs contain many branches or strong instruction dependencies. In addition, deeper pipelines can increase the cost of branch mispredictions and may require more complex control logic [2].

IX. Conclusion

Pipelining is one of the most important techniques in computer architecture because it improves processor throughput by overlapping the execution of multiple instructions. Instead of waiting for one instruction to finish completely before starting the next, a pipelined processor divides execution into stages and keeps several instructions moving through the processor at the same time.

The classical five-stage pipeline provides a clear model for understanding how instruction fetch, decode, execute, memory access, and write back can operate together. However, pipelining also introduces hazards that must be handled carefully.

TABLE II
MAIN ADVANTAGES AND LIMITATIONS OF PIPELINING

Advantages	**Limitations**
Improves instruction throughput.	Increases processor design complexity.
Uses processor resources more efficiently.	Hazards can cause stalls and performance loss.
Supports instruction-level parallelism.	Branch-heavy programs may reduce pipeline efficiency.
Provides a basis for advanced CPU techniques.	Deeper pipelines may increase misprediction penalties.

Data hazards, control hazards, and structural hazards can interrupt the smooth flow of instructions and reduce performance.

Techniques such as forwarding, stalling, branch prediction, and instruction scheduling help reduce these problems and allow pipelined processors to operate more efficiently. Although modern processors use more advanced designs, the basic concept of pipelining remains a foundation for understanding processor performance and instruction-level parallelism [1], [2].

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011. [Online]. Available: https://acs.pub.ro/~cpop/SMPA/Computer%20Architecture%20A%20Quantitative%20Approach%20%285th%20edition%29.pdf

[2] W. Stallings, *Computer Organization and Architecture: Designing for Performance*, 11th ed. Pearson Education, 2019. [Online]. Available: https://os.ecci.ucr.ac.cr/ci0114/material/Stallings/Computer-Organization-Architecture-11th.pdf

[3] M. M. Mano, *Computer System Architecture*, 3rd ed. Prentice Hall, 1993. [Online]. Available: https://www.mbit.edu.in/wp-content/uploads/2020/05/computer-systems-Architecture.pdf