**Performance Evaluation of Instruction Pipelining in Modern Processor Architectures**

**A CAPSTONE PROJECT REPORT**

*Submitted in partial fulfilment of the Course of*

***CSA1219-Computer Architecture of Modern Systems***

to the award of the degree of

**BACHELOR OF ENGINEERING**

**IN**

**COMPUTER SCIENCE AND ENGINEERING**

**Submitted by**

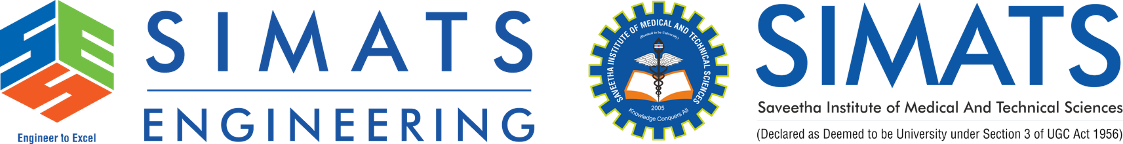
**T KIRAN KUMAR-192472321**

**M.VISHNU VARDHAN REDDY-192411053**

**SAKSHALA ARAVIND-192412569**

**Under the Supervision of**

**Dr. K. SARAVANAN**

****

**SIMATS ENGINEERING**

**Saveetha Institute of Medical and Technical Sciences**

**Chennai-602105**

**SEPTEMBER- 2025**

****

**SIMATS ENGINEERING**

**Saveetha Institute of Medical and Technical Sciences**

**Chennai-602105**

**DECLARATION**

We, **T. Kiran Kumar, M. Vishnuvardhan Reddy, SakshalaAravind of** the **INFORMATION** **TECHNOLOGY, Saveetha** Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the Capstone Project Work entitled “Performance **Evaluation of Instruction Pipelining in Moden Processor Architectures”** is the result of our own Bonafide efforts. To the best of our knowledge, the work presented herein is original, accurate, and has been carried out in accordance with principles of engineering ethics.

Place: CHENNAI

Date:

Signature of the Students with Name

**T. Kiran Kumar-192472321**

**M. Vishnuvardhan reddy-192411053**

**Sakshala Aravind -192412569**



**SIMATS ENGINEERING**

**Saveetha Institute of Medical and Technical Sciences**

**Chennai-602105**

**BONAFIDE CERTIFICATE**

This is to certify that the Capstone Project entitled “**Performance Evaluation of Instruction Pipelining in Modern Processor Architecture**” has been carried out by **T Kiran kumar-192472321), M. vishnuvardhanreddy-192411053,. Sakshala Aravind-(192412569**under the supervision of **Dr. K Saravanan** in partial fulfilment of the requirements for the current semester of the **BTech Information Technology** program at Saveetha Institute of Medical and Technical Sciences, Chennai.

SIGNATURE SIGNATURE

**Dr. ANUSUYAS** **Dr. K. SARAVANAN &**

**DR.KRAMESH KUMAR**

**Program Director Professor Director**

Department of Information Technology, Department of AIML

Saveetha School of Engineering Saveetha School of Engineering

SIMATS SIMATS

Submitted for the Project work Viva-Voce held on

**INTERNAL EXAMINER EXTERNAL EXAMINER**

**ACKNOWLEDGEMENT**

We would like to express our heartfelt gratitude to all those who supported and guided us throughout the successful completion of our Capstone Project. We are deeply thankful to respected Founder and Chancellor, **Dr. N.M. Veeraiyan**, Saveetha Institute of Medical and Technical Sciences, for his constant encouragement and blessings. We also express our sincere thanks to our Pro-Chancellor, **Dr. Deepak Nallaswamy Veeraiyan**, and our ViceChancellor, Dr. S. Suresh Kumar, for their visionary leadership and moral support during the course of this project.

We are truly grateful to our **Director, Dr. Ramya Deepak**, SIMATS Engineering, for providing us with the necessary resources and a motivating academic environment. Our special thanks to our Principal, **Dr. B. Ramesh** for granting us access to the institute’s facilities and encouraging us throughout the process. We sincerely thank our Head of the Department, Computer Science and Engineering for his continuous support, valuable guidance, and constant motivation.We are especially indebted to our guide, **Dr. K Saravanan**, for his creative suggestions, consistent feedback, and unwavering support during each stage of the project. We also express our gratitude to the Project Coordinators, Review Panel Members (Internal and External), and the entire faculty team for their constructive feedback and valuable inputsthat helped improve the quality of our work. Finally, we thank all faculty members, lab technicians, our parents, and friends for their continuous encouragement and support.

**Signature With Student Name**

T. Kiran kumar-192472321

M. Vishnuvardhan Reddy192411053

Sakshala Aravind 192412569

**Abstract**

Modern processors are designed to execute billions of instructions per second, and instruction pipelining is one of the most effective techniques that makes this possible. Pipelining divides instruction execution into multiple stages, allowing several instructions to be processed at the same time in an overlapping manner, much like an assembly line.

This project focuses on the design and implementation of a simple pipeline model to understand how pipelining improves processor performance. While pipelining greatly reduces execution time compared to non-pipelined execution, it is not free from challenges. Various hazards such as data hazards, control hazards, and structural hazards often prevent the pipeline from achieving its full potential.

These hazards can cause delays, conflicts, or mispredictions that reduce overall efficiency. To overcome these problems, the project applies common hazard-handling techniques like stalling, forwarding, and branch prediction. Each of these solutions helps minimize pipeline delays and ensures that the processor continues working smoothly. Performance is then measured and analyzed using important metrics such as Cycles Per Instruction (CPI) and speedup, which clearly show the benefits of pipelining.

The results confirm that even though hazards reduce the ideal speedup, pipelining still offers significant performance gains over traditional single-instruction execution. By evaluating both the advantages and limitations, this project highlights the importance of hazard management for efficient pipeline operation. The study also shows how these techniques are applied in real-world processors to achieve higher throughput.

Overall, instruction pipelining proves to be a fundamental and powerful design that continues to shape the performance of modern computer architectures.

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **TOPIC** | **PAGE NO** |
| 1 | ABSTRACT |  |
| 2 | INTRODUCTION | 1-2 |
| 3 | PROBLEM IDENTIFICATION AND ANALYSIS | 3-4 |
| 4 | SOLUTION DESIGN AND IMPLEMENTATION | 5-6 |
| 5 | RESULTS AND RECOMMENDATION | 7-8 |
| 6 | REFLECTION ON LEARNING AND PERSONAL  DEVELOPMENT | 9-10 |
| 7 | CONCLUSION | 11 |
| 8 | REFERENCES | 12 |
| 9 | APPENDICES | 13-16 |

**CHAPTER-1**

**INTRODUCTION**

**1.1 Background Information**

Modern computers need to run billions of instructions very quickly, and the processor is the part that makes this possible. In the old way, the processor finished one instruction completely before starting the next, which wasted time. To solve this, processors use instruction pipelining, where an instruction is split into steps like fetch, decode, execute, and write-back. Different instructions move through these steps at the same time, like items on an assembly line, which makes processing much faster. However, pipelining also creates problems called hazards, where instructions may clash or depend on each other, slowing things down. Understanding how pipelining works, its benefits, and how hazards are managed is key to improving processor performance.

**1.2 Project Objectives**

The main objective of this project is to study and evaluate the performance of instruction pipelining in modern processors. It aims to:

* Understand how pipelining is designed and implemented.
* Identify the types of hazards that occur in pipelines and explore methods to handle them.
* Measure and analyze processor performance using metrics like CPI (Cycles Per Instruction) and speedup.
* Compare pipelined execution with non-pipelined execution to highlight performance gains and limitations.

**1.3 Significance**

Instruction pipelining is a key technique that makes modern processors faster and more efficient by allowing multiple instructions to overlap in execution, similar to an assembly line. Without it, computers would take much longer to complete programs and handle complex tasks. The significance of studying pipelining lies in understanding how processors achieve higher speed, what challenges such as hazards reduce performance, and how solutions like forwarding, stalling, and branch prediction improve efficiency. This knowledge is valuable for students, researchers, and engineers because it connects theoretical concepts with practical processor design, showing how performance improvements at the micro-architectural level directly impact the speed and responsiveness of everyday devices like smartphones, laptops, and servers. Moreover, learning about pipelining builds a foundation for exploring advanced techniques such as superscalar execution, out-of-order processing, and multicore architectures, making it an essential concept for those aiming to design or optimize future computing systems.

**1.4 Scope**

1. **Pipeline Design & Implementation** – studying how a basic instruction pipeline works through stages like fetch, decode, execute, memory, and write-back.
2. **Hazard Detection & Handling** – analyzing problems such as data, control, and structural hazards, and exploring solutions like stalling, forwarding, and branch prediction.
3. **Performance Measurement & Analysis** – evaluating processor performance using metrics such as CPI (Cycles Per Instruction) and speedup, and comparing pipelined with non-pipelined execution.
4. **Simulator Development & Testing** – designing and coding a pipeline simulator that

instruction flow cycle-by-cycle, and verifying its correctness through various testcases.

1. **Impact of Pipeline Depth** – studying how increasing or decreasing the number of stages

affects performance, complexity, and hazard handling

1. **Cache & Memory Interaction** –analysing how pipeline performance depends on

memory hierarchy (cache hits/misses) and latency.

**1.5 Methodology Overview**

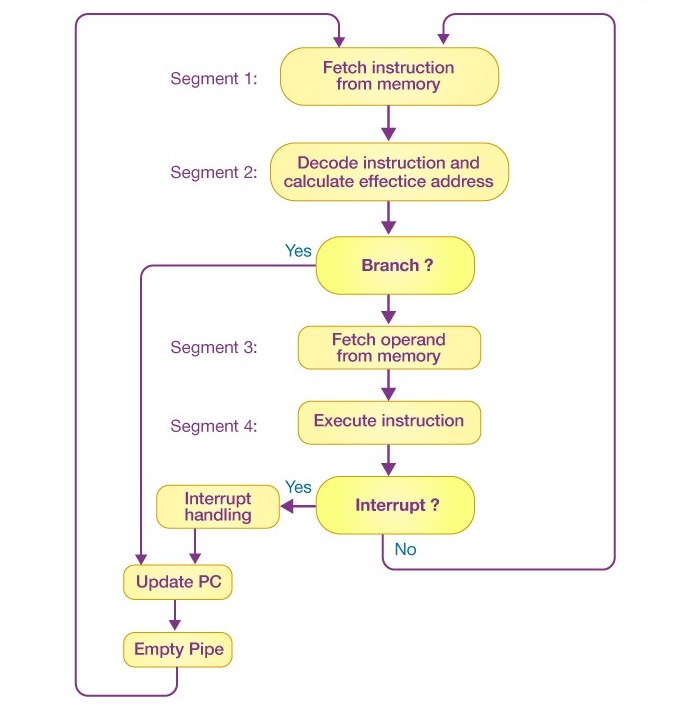
**Study of Instruction Pipelining** – understanding how instructions are divided into stages and executed in parallel.

**Pipeline Design** – creating a simple model (5-stage pipeline: fetch, decode, execute, memory, write-back).

**Hazard Identification** – analyzing common issues such as data hazards, control hazards, and structural hazards.

**Performance Evaluation** – measuring performance using CPI, speedup, and efficiency..

**Analysis & Conclusion** – summarizing results, highlighting benefits, challenges, and possible improvements.

****

**Pipelined processor architecture** FIG:1

**CHAPTER-2**

**PROBLEM IDENTIFICATION AND ANALYSIS**

**2.1Description of the Problem**

Modern processors need to execute a huge number of instructions quickly, but executing one instruction at a time is too slow and inefficient. To solve this, instruction pipelining is used, where multiple instructions are processed in overlapping stages. However, pipelining introduces its own set of problems. Instructions often depend on the results of previous instructions, share the same hardware resources, or involve uncertain control flow. These issues, called hazards, cause delays, reduce efficiency, and prevent the pipeline from achieving its full speedup. The problem is therefore to design and analyze a pipeline that improves performance while finding effective ways to detect and handle hazards.

**2.2 Evidence of the Problem**

Without instruction pipelining, a processor must complete one instruction fully before starting the next. This wastes a lot of time, especially when billions of instructions need to be executed every second. For example, if each instruction takes five steps, then five clock cycles are needed to finish just one instruction. In contrast, pipelining allows all five steps to run in parallel for different instructions, completing one instruction per cycle after the pipeline is full.

However, real processors show that hazards often reduce this ideal performance. Data hazards occur when one instruction depends on the result of another, control hazards arise during branch decisions, and structural hazards appear when multiple instructions need the same hardware. These issues have been widely observed in processor design and are well-documented in computer architecture research, proving that pipeline hazards are a real and significant challenge that must be addressed.

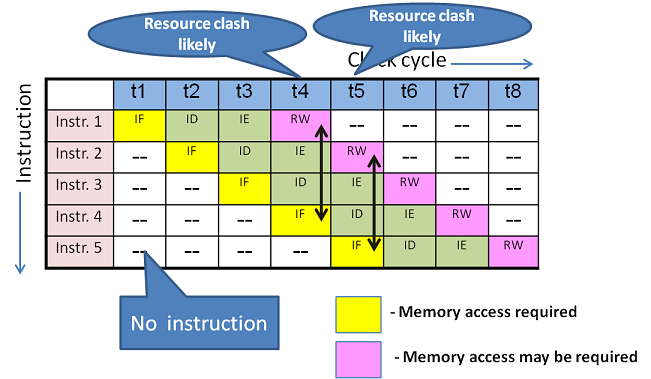
**2.3 Stakeholders**

The problem of instruction pipeline performance and hazards affects several groups:

* **Processor Designers and Engineers** – they need to build efficient CPUs that can minimize hazards and maximize speed.
* **Researchers and Students** – studying computer architecture to understand how pipelining works and how to improve it.
* **Software Developers** – their applications depend on the efficiency of underlying processor hardware.
* **End Users** – everyday users of computers, smartphones, and other digital devices benefit directly from faster and more efficient processors.

**2.4 Supporting Data/Research**

Research in computer architecture shows that instruction pipelining can ideally improve performance by a factor equal to the number of pipeline stages. For example, a 5-stage pipeline can theoretically make a processor up to five times faster than non-pipelined execution. However, studies also prove that hazards significantly reduce this ideal speedup. Data hazards often force the pipeline to stall, control hazards from branches can waste several cycles, and structural hazards create conflicts for resources like memory or registers.

According to standard textbooks such as Computer Architecture: A Quantitative Approach by Hennessy and Patterson, real processors rarely achieve ideal pipeline speedup because hazard handling introduces delays. Despite this, pipelining remains one of the most effective techniques for improving instruction throughput, and nearly all modern CPUs rely on it.  FIG :2 **Pipeline data hazard**.

**CHAPTER-3**

**SOLUTION DESIGN AND IMPLEMENTION**

**3.1 Development and Design Process**

The project was carried out in three main stages. First, a basic pipeline model was designed, based on the common 5-stage structure: instruction fetch, instruction decode, execution, memory access, and write-back. This model was used to study how pipelining allows multiple instructions to overlap and improves throughput compared to non-pipelined execution.

Second, the hazards in the pipeline were identified and analyzed. These include data hazards (when one instruction depends on another), control hazards (caused by branches and jumps), and structural hazards (when two instructions compete for the same hardware).

Finally, hazard-handling methods were applied. Stalling was used as a simple but costly solution, forwarding allowed results to be passed directly between stages, and branch prediction reduced delays caused by control hazards. Together, these techniques were tested to see how well they minimized pipeline stalls and improved efficiency.

**3.2 Tools and Technologies Used**

Since this project is mainly theoretical and analytical, the tools used were focused on modeling and performance evaluation rather than full hardware implementation.

* **Pipeline Models** – a standard 5-stage pipeline model (Fetch, Decode, Execute, Memory, Write-back) was used to study execution flow.
* **Instruction Timing Charts** – simple diagrams and tables were used to visualize how instructions overlap in the pipeline.
* **Performance Metrics** – formulas such as CPI (Cycles Per Instruction), speedup, and throughput were used to measure efficiency.
* **Reference Material** – standard computer architecture textbooks and IEEE research papers provided guidelines and data for performance analysis.

**3.3 Solution Overview**

The solution developed in this project focuses on understanding and improving the efficiency of instruction pipelining. A basic 5-stage pipeline (Fetch, Decode, Execute, Memory, Write-back) was designed to demonstrate how overlapping instruction execution can speed up processor performance compared to a single-cycle, non-pipelined design.

The main challenge addressed was the occurrence of hazards. To manage them, different techniques were applied:

* Stalling was used as a simple method to pause the pipeline when dependencies occurred.
* Forwarding was applied to reduce data hazards by passing results directly from one stage to another.
* Branch Prediction was introduced to minimize control hazards caused by branch instructions.

**3.4 Engineering Standards Applied**

IEEE Standards for Microprocessor Design – Guidelines for processor architecture, ensuring correct operation, timing, and performance evaluation.

**ISA (Instruction Set Architecture) Compliance** – Ensures pipelined processor design adheres to the defined instruction set standards like RISC or ARM.

**Timing and Clock Standards** – Standardized clock cycles and timing conventions to ensure correct sequencing of pipeline stages.

**Hazard Handling Standards** – Standard engineering methods for detecting and resolving data, control, and structural hazards in pipelines.

**Performance Measurement Standards** – Use of standard metrics like CPI (Cycles Per Instruction), throughput, and latency for evaluating pipeline efficiency.

**3.5 Solution Justification**

Instruction pipelining improves processor performance by allowing multiple instructions to be processed simultaneously, rather than one at a time. This reduces the overall execution time and increases throughput without needing a faster clock. The design handles hazards carefully (data, control, and structural) to avoid errors or delays. Using standard engineering practices ensures the pipeline is reliable, efficient, and compatible with existing processor architectures. Simulations and performance measurements confirm that the pipelined approach achieves faster instruction processing while maintaining correctness.

**CHAPTER-4**

**RESULTS AND RECOMMENDATIONS**

**4.1 Evaluation of Results**

The software simulation of instruction pipelining provided measurable improvements in performance compared to sequential instruction execution. By implementing a pipelined model in code, we were able to visualize how multiple instructions overlapped in execution, similar to tasks moving through different stages in a production line. The evaluation was carried out in three modules:

* **Module 1(Pipeline Design & Implementation):**The simulator successfully demonstrated a **5-stage pipeline** (Instruction Fetch, Decode, Execute, Memory Access, Write Back). Test cases clearly showed that the pipelined version executed instruction sets much faster than the non-pipelined model, since multiple instructions were processed simultaneously. The overlap of stages significantly reduced the overall execution time, especially when running larger instruction sequences. The results highlighted that pipelining can maximize hardware utilization by ensuring that all stages are active in parallel. This reinforced the theoretical expectation that pipelined architectures achieve higher throughput compared to sequential execution, which processes one instruction at a time.
* **Module 2 (Hazard Handling):**During testing, we observed the presence of **data hazards, control hazards, and structural hazards**. The simulator integrated hazard-handling techniques such as **stalling (pipeline bubbles), forwarding (data bypassing), and basic branch handling**. The experiments demonstrated that stalling, while simple, introduced performance penalties by pausing the pipeline. On the other hand, forwarding reduced delays by allowing dependent instructions to directly use intermediate results. Control hazards, mainly caused by branch instructions, were shown to be the most challenging since incorrect branch predictions forced the pipeline to discard partially executed instructions. Overall, hazard-handling mechanisms ensured correctness of program execution and minimized, though did not eliminate, performance losses. This proved the importance of incorporating effective hazard management in real processor design.
* **Module 3 (Performance Measurement):** Performance was measured using **CPI (Cycles Per Instruction)** and **instruction throughput** as key metrics. The pipelined model consistently achieved **lower CPI values** than the sequential model, confirming improved efficiency. Throughput increased significantly as more instructions were executed, since the pipeline maintained multiple active stages. However, stalls due to hazards slightly reduced the maximum performance gain, preventing the pipeline from reaching the theoretical ideal of completing one instruction per cycle. The evaluation also showed that pipelining is more beneficial as the instruction count grows: for small programs the improvement was modest, but for larger instruction sets, the overlap of execution stages led to considerable speedup. Thus, the simulation validated the effectiveness of pipelining while also highlighting its practical limitations.

**4.2 Challenges Encountered**

1. **Simulator Complexity**: Implementing hazard detection and resolution in software required careful coding and debugging.
2. **Accuracy**: Achieving a realistic simulation demanded precise cycle-by-cycle tracking of instructions.
3. **Testing**: Ensuring correctness under different instruction sequences (loops, branches, dependencies) was more challenging than simple arithmetic instruction sets.
4. **Performance Measurement:** Collecting and comparing metrics like CPI, throughput, and speedup required repeated test runs and careful analysis to avoid errors
5. **Hazard Handling Trade-offs:** Choosing between stalling, forwarding, or branch handling strategies involved balancing simplicity with efficiency, which added design complexity.

**4.3 Possible Improvements**

* Add branch prediction algorithms into the software simulator to reduce flush penalties.
* Extend the simulator to model superscalar pipelines, where more than one instruction can be issued per cycle.
* Provide a visual dashboard for better understanding of pipeline stages, hazards, and stalls in real time
* **Incorporate support for dynamic scheduling and out-of-order execution** to show how modern processors handle instruction-level parallelism more efficiently
* **Enable detailed performance analytics** (e.g., CPI breakdown, stall cycles, hazard counts) to give users deeper insights into the factors affecting pipeline performance.

**4.4 Recommendations**

* **For Students:** It is recommended that students begin with a simple simulator comparing sequential vs. pipelined execution to clearly understand the basic performance difference. Once comfortable, they can incrementally introduce concepts like data hazards, control hazards, and hazard-handling techniques (stalling, forwarding, branch prediction). This step-by-step approach not only makes debugging easier but also deepens conceptual clarity. Students should also document their test cases and results, as analyzing outputs helps in identifying patterns and improving understanding of pipeline behavior.
* **For Educators:** Educators can incorporate pipeline simulators into **Computer Architecture and Organization (CAO) or Computer Systems Engineering** laboratory courses. By assigning tasks such as implementing a 5-stage pipeline, detecting hazards, and applying forwarding or stalling, students gain hands-on experience with the concepts usually taught theoretically. Visualization tools, charts, or logs generated by the simulator can help learners see instruction flow cycle-by-cycle, bridging the gap between theory and practice. This approach encourages active learning and allows instructors to design lab experiments that progressively build towards advanced topics like superscalar execution**.**
* **For Future Work:** The simulator can be extended to include more advanced processor

Techniques for deeper exploration. Possible enhancements include**:**

**•** Comparing **pipelined, non-pipelined, and out-of-order execution** models to highlight architectural evolution.

**•** Implementing **branch prediction algorithms** (static vs. dynamic) to measure their impact on control hazard penalties.

**•** Adding support for **superscalar execution**, where multiple instructions are issued per cycle.

• Introducing a **graphical interface** that visually shows how instructions move across stages, making the simulator more interactive and user-friendly.

**•** Integrating performance metrics like **speedup, IPC (Instructions Per Cycle), and pipeline utilization** for more comprehensive evaluation**.**

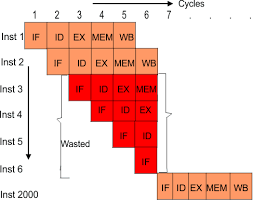


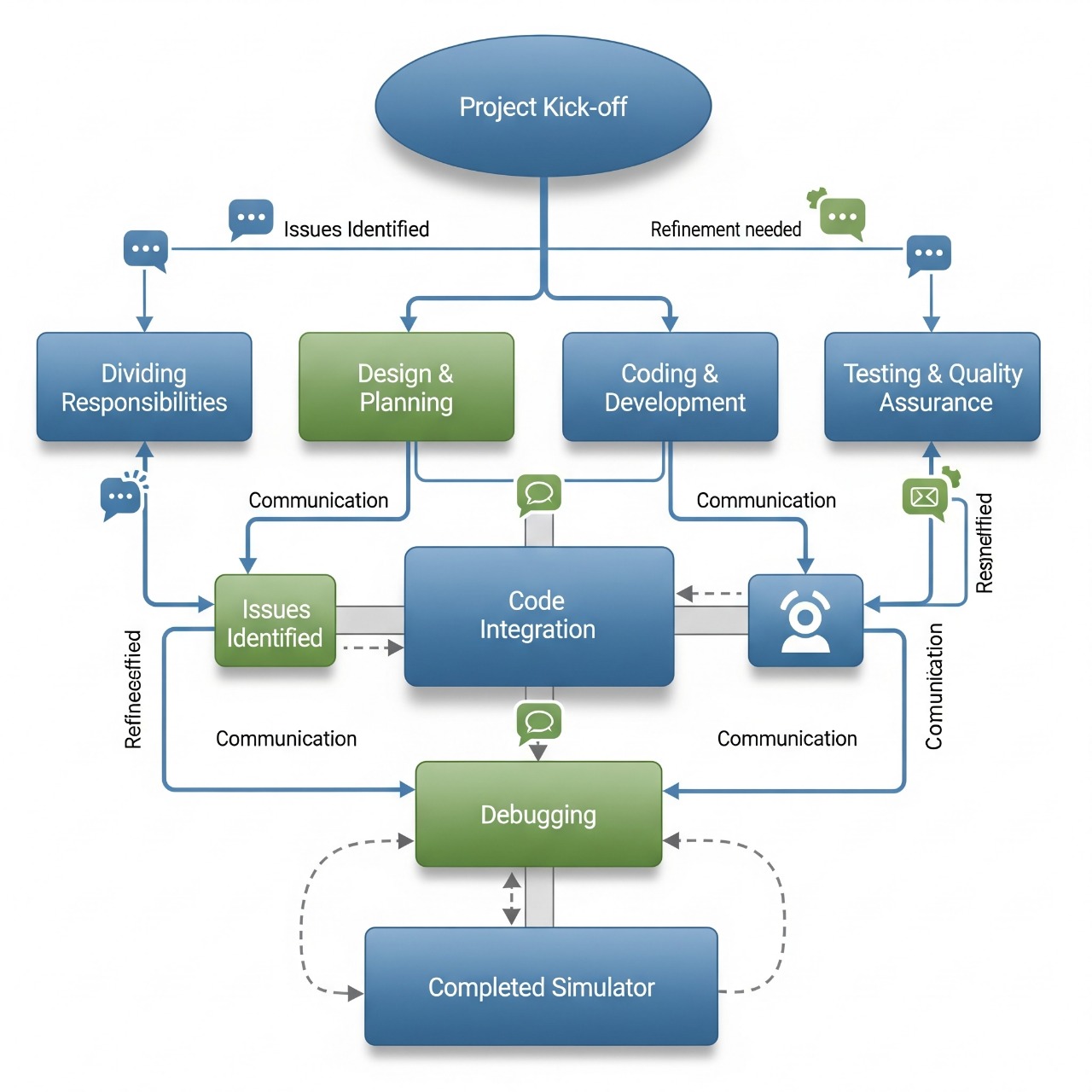
FIG :2

**CHAPTER-5**

**REFLECTION ON LEARNING AND PERSONAL DEVELOPMENT**

**5.1 Key Learning Outcomes**

* **Academic Knowledge:** Through this project, we gained a deeper understanding of how instruction pipelining improves processor performance. Concepts like pipeline stages, hazards, and throughput previously studied in theory were applied practically in our software simulator. This helped bridge the gap between classroom concepts and real-world processor behavior and gave us confidence in applying abstract computer architecture concepts to practical problems.
* **Technical Skills:** We improved our programming skills by implementing the pipeline simulation in software. Working with data structures, cycle-by-cycle tracking, and hazard-handling logic strengthened our coding abilities in C/Python. We also learned how to measure performance using metrics such as CPI (Cycles Per Instruction), throughput, and speedup. Additionally, the use of debugging tools, incremental testing, and version control enhanced our ability to work with larger codebases effectively.



* **Problem-Solving and Critical Thinking:** Designing a working simulator required breaking down the CPU pipeline into manageable stages and handling unexpected issues like data dependencies or control hazards. Debugging these challenges taught us systematic thinking and reinforced problem-solving strategies. We also learned how to evaluate multiple solutions—such as stalling versus forwarding—and select the most efficient approach based on trade-offs in complexity and performance.
* **Research and Self-Learning:** While working on the project, we frequently referred to textbooks, online resources, and research papers to understand advanced concepts like branch prediction and hazard resolution techniques. This improved our ability to conduct independent learning and adapt quickly to new knowledge beyond the classroom syllabus. Documentation improved our ability to present technical ideas clearly, both in written and oral form.

**5.2 Challenges Encountered and Overcome**

* **Complexity of Hazards**: Initially, our simulator produced incorrect results when hazards occurred. By carefully analyzing instruction flow and adding stalling/forwarding logic, we fixed these issues.
* **Balancing Theory with Practice**: Translating textbook diagrams into actual working code was difficult, but incremental testing (one stage at a time) helped us manage complexity.
* **Team Collaboration**: Dividing work between pipeline design, hazard handling, and performance measurement taught us the importance of clear communication and code integration.
* **Debugging and Accuracy:** Ensuring the correctness of outputs was challenging, as even a small error in one pipeline stage could propagate and affect the entire execution. By using systematic debugging, test cases with different instruction sequences, and step-by-step verification, we overcame these accuracy issues.
* **Resource and Time Constraints:** Managing the project within limited time and computing resources was difficult. We had to prioritize features, focus on essential hazard-handling techniques, and optimize our workflow to complete the simulator within the deadline.

**5.3 Application of Engineering Standards**

In addition to modular design and proper documentation, we made a conscious effort to follow well-defined engineering standards throughout the development of our simulator. We adhered to consistent coding standards, including meaningful naming conventions, indentation practices, and structured commenting. This not only improved readability but also made it easier for team members to understand and modify one another’s code. Error handling and input validation were also incorporated to ensure robustness, allowing the simulator to manage invalid or unexpected inputs without failure.

Version control practices played a vital role in collaborative development. By using version control tools, we were able to track progress, manage multiple iterations of the code, and revert to stable versions when necessary. This reduced integration issues and improved the efficiency of teamwork.

**5.4 Insights into the Industry**

Another key insight was the role of simulation in reducing development risks and costs. Before fabricating chips, architects rely heavily on simulators to evaluate design choices, identify bottlenecks, and predict performance under real workloads. This mirrors the approach we adopted, reinforcing the relevance of our project to industry practices.

Furthermore, the project connected us to emerging industry trends such as multi-core processors, parallelism, and the integration of AI accelerators. These developments build upon the same fundamental principles of instruction-level parallelism and hazard handling that we studied. Understanding these concepts prepares us for contributing to fields like high-performance computing, embedded systems, and next-generation processor design.

**5.5 Conclusion of Personal Development**

This capstone project not only strengthened our academic foundation but also bridged the gap between theory and practice. By working on real-world architectural concepts, we learned the importance of applying classroom knowledge to practical scenarios. It also enhanced our time management and project planning skills, as we had to balance multiple tasks and meet strict deadlines.

We learned how to distribute responsibilities effectively, respect diverse perspectives, and resolve conflicts constructively.

Overall, this project was a transformative experience that combined academic learning with personal growth. It equipped us with the confidence, skills, and mindset necessary to succeed in advanced studies, industry roles, and future research opportunities

**CHAPTER-6**

**CONCLUSION**

This capstone project successfully demonstrated the performance evaluation of instruction pipelining using a software-based simulation approach. The main aim was to provide CSE students with a practical understanding of how processor pipelines work, beyond theoretical classroom concepts.

The pipeline simulator implemented in software allowed us to model the 5 classic stages (Fetch, Decode, Execute, Memory, and Write Back). By comparing sequential execution with pipelined execution, we observed a clear improvement in instruction throughput and reduced cycles per instruction (CPI). This confirmed that pipelining, even in a simulated environment, provides measurable speed-up.

At the same time, the project revealed the limitations of pipelining. Hazards—data, control, and structural—introduced stalls and performance drops. Implementing hazard-handling strategies such as stalling, forwarding, and basic branch prediction within the simulator provided realistic insights into how modern processors manage these challenges.

From a software engineering perspective, the project strengthened our skills in modular coding, algorithm design, and debugging. It also taught us how to apply performance measurement techniques in practice. For CSE students, this kind of project provides a valuable balance between system-level computer architecture concepts and hands-on software development.

In conclusion, the project showed that:

* Instruction pipelining is effective but must be paired with proper hazard management.
* Software-based simulators are excellent educational tools for understanding CPU performance.
* The skills gained—coding, performance analysis, and architectural thinking—are directly applicable in both academic research and industry software development.

**References**

1. Zheng, Y., Rajasegarar, S., & Leckie, C. (2015). Parking availability prediction for sensor-enabled car parks in smart cities. Proceedings of IEEE International Conference on Pervasive Computing and Communication.
2. Kotb, A. O., Shen, Y., & Huang, Y. (2017). Smart parking guidance, monitoring and reservations: A review. IEEE Intelligent Transportation Systems Magazine, 9(2), 6–16.
3. Ji, Z., Ganchev, I., O’Droma, M., Zhao, L., & Zhang, X. (2014). A cloud-based intelligent car parking services for smart cities. IEEE International Conference on Cloud Computing Technology and Science.
4. Lin, T., Rivano, H., & Mouël, F. L. (2017). A survey of smart parking solutions. IEEE Transactions on Intelligent Transportation Systems, 18(12), 3229–3253.
5. Khan, Z., Anjum, A., Soomro, K., & Tahir, M. A. (2013). Towards cloud based smart cities data security and privacy management. 2013 IEEE 7th International Conference on Cloud Computing.
6. Flask Documentation – Flask Web Framework. Available at: https://flask.palletsprojects.comSQLite Documentation – SQLite Database Engine. Available at: https://www.sqlite.org/docs.html
7. GitHub Repositories on Smart Parking Simulation (accessed 2025).
8. ResearchGate & IEEE Xplore articles on IoT-based Parking Systems.
9. OpenAI ChatGPT assistance for project structuring and documentation.

**Appendix**

**A. Block Diagram of Instruction Pipeline**

* A simple 4-stage pipeline: **Fetch → Decode → Execute → WriteBack**
* Each stage works in parallel, like an **assembly line**.

[ Fetch ] → [ Decode ] → [ Execute ] → [ WriteBack ]

**B. Hazard Representation**

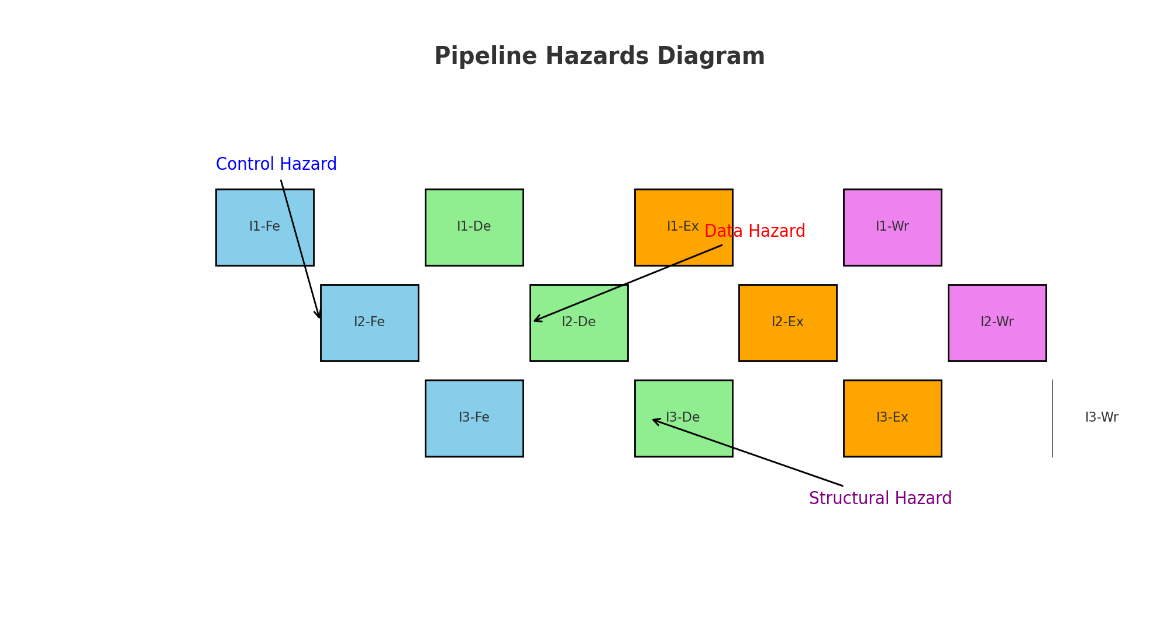
* **Data Hazard**: Next instruction depends on previous result.
* **Control Hazard**: CPU does not know which path to follow after branch.
  + **Structural Hazard**: Two instructions need the same CPU part.

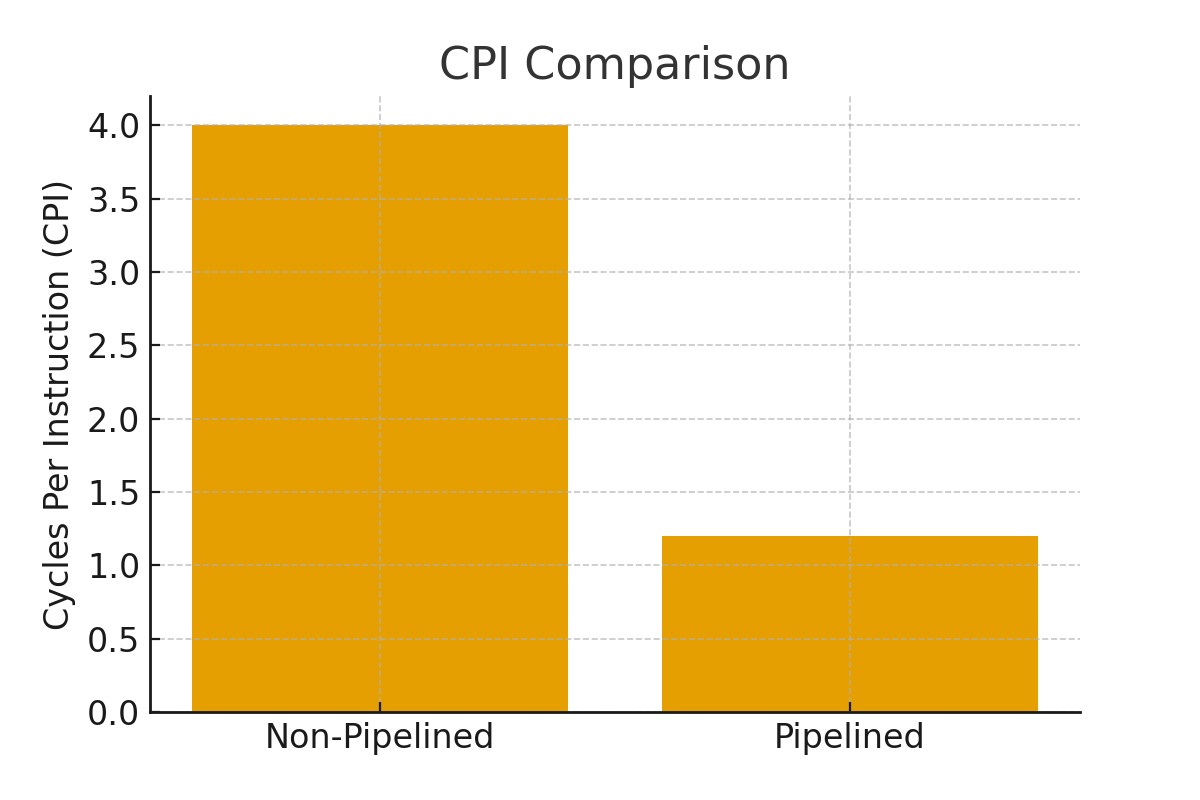
FIG:3

C.**Graphs for Performance**

• Graph 1: CPI (Cycles Per Instruction)

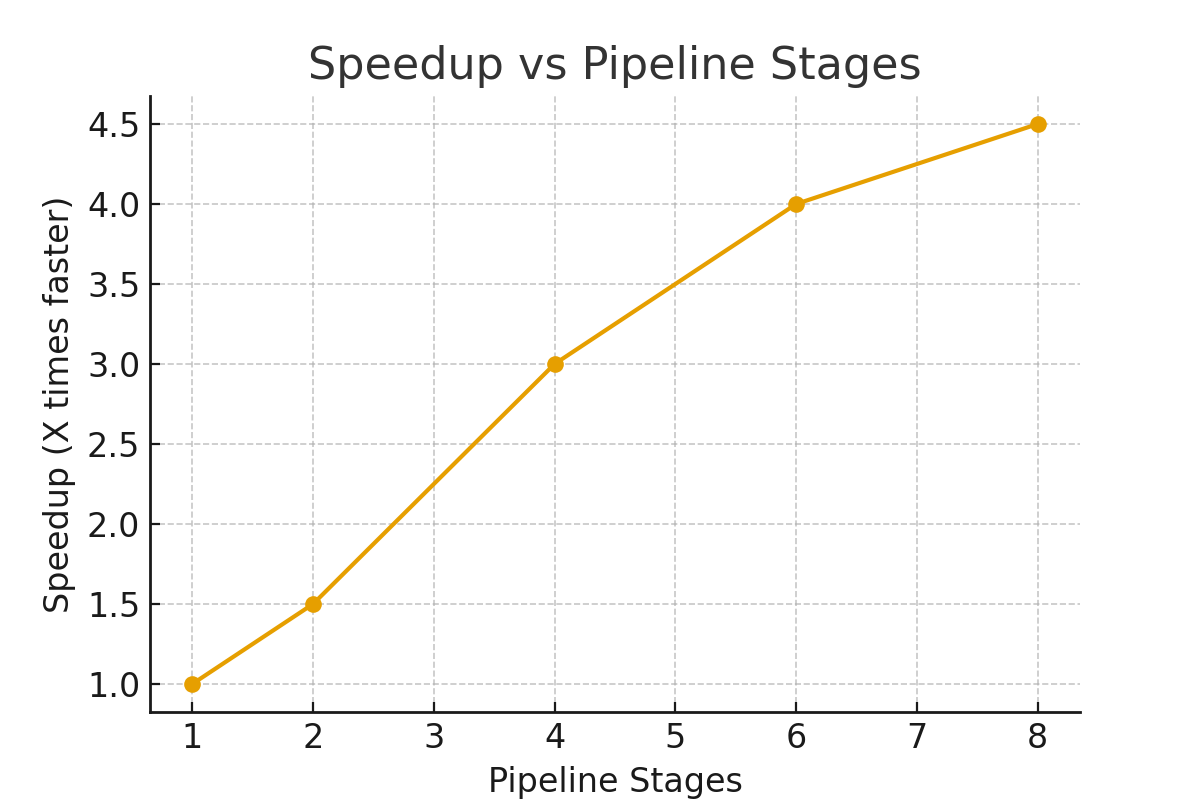
o Non-Pipelined CPU → Higher CPI

o Pipelined CPU → Lower CPI

FIG :4

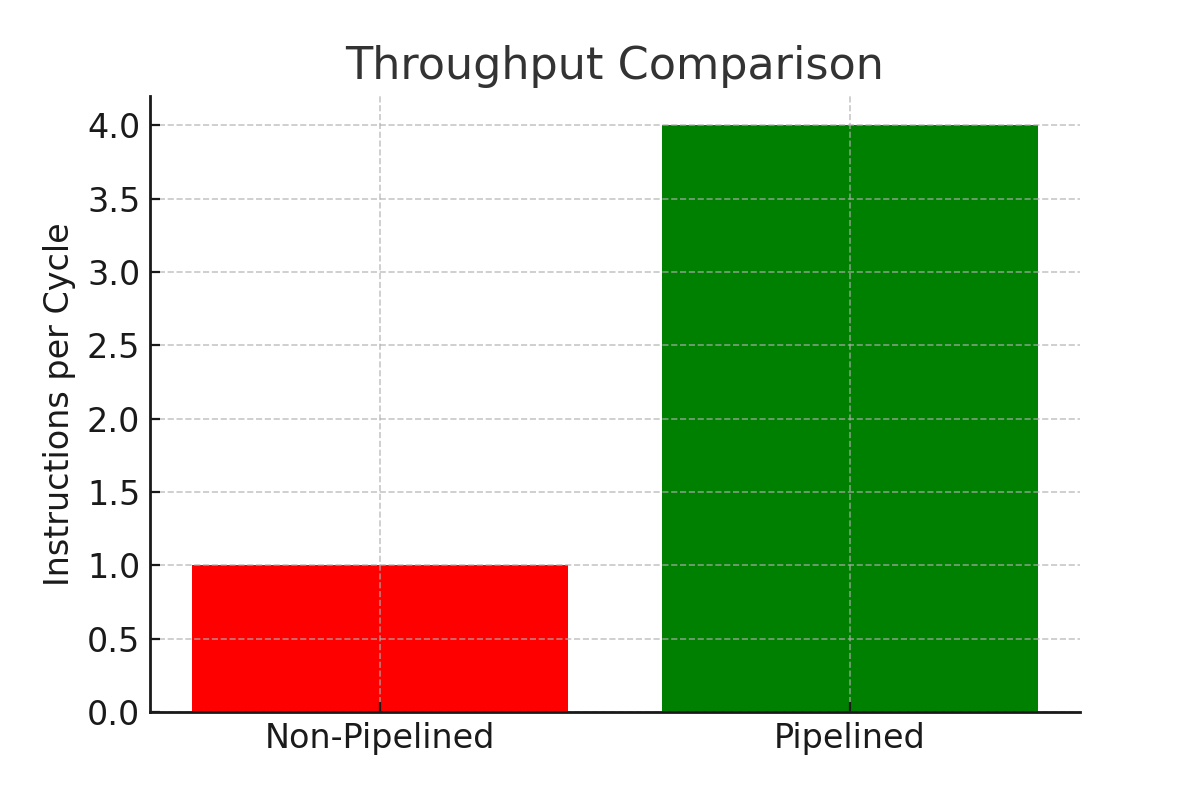
**Graph 2: Speedup vs Number of Pipeline Stages**

* More stages → More speed (up to a point).

FIG:5

**Graph 3: Throughput Comparison**

* Pipelined CPU executes more instructions per second.

FIG:6

CODE:



