IJEAST

INTERNATIONAL JOURNAL
OF ENGINEERING APPLIED SCIENCE
AND TECHNOLOGY

VOLUME : 10 ISSUE : 10 Print / Issue Publication Date: 08-Apr-2026

ISSN : 2455-2143

DOI : 10.33564/IJEAST.2026.v10i10.001

Indexed In

WWW.IJEAST.COM

editor@ijeast.com

International Journal of Engineering Applied Sciences and Technology, 2026
Vol. 10, Issue 10, ISSN No. 2455-2143, Pages 01-06
Published Online February 2026 in IJEAST (http://www.ijeast.com)

PERFORMANCE EVALUATION OF THE UETRV-PCORE USING RISC-V MICROBENCHMARKS: IMPACT OF COMPILER OPTIMIZATIONS ON A FIVE-STAGE RISC-V PIPELINE

Masooma Zia
Department of Electrical Engineering,
University of Engineering and Technology Lahore, Pakistan

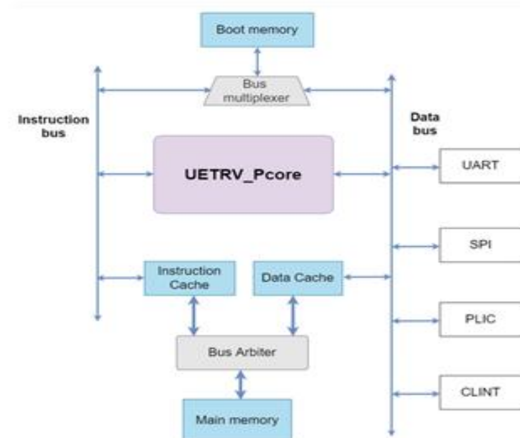
Abstract—**The UETRV-PCore is a custom, five-stage RV32IMAZicsr RISC-V processor featuring an SV32-based virtual memory system and FPGA deployment capabilities. This work presents a comprehensive performance evaluation of the core using standardized microbenchmarks targeting control flow, dependency chains, memory operations, and arithmetic workloads. I analyze the Instructions per Cycle (IPC) across various compiler optimization levels to identify microarchitectural bottlenecks. Experimental results show that while unoptimized code yields a balanced IPC of approximately 0.5, compiler-driven instruction scheduling and loop transformations significantly boost performance to 0.94 IPC for execution-heavy tasks. However, memory-bound benchmarks remain constrained by Read-After-Write (RAW) hazards and the absence of dynamic branch prediction, resulting in persistent stalls. My findings provide critical insights into the efficiency of the UETRV-PCore's pipeline and highlight specific areas for microarchitectural improvement, such as the integration of a branch predictor to mitigate pipeline flushes during control-intensive workloads.**

Keywords: **RISC-V, UETRV-Pcore, Micro-benchmarks, Performance Evaluation, Instructions per Cycle (IPC), Verilator, Compiler Optimization,Pipeline Hazards.**

I. INTRODUCTION

RISC-V is an open-source Instruction Set Architecture (ISA) that has gained significant attention in both academia and industry due to its modular design, extensibility, and ease of adoption. It defines a set of standard base instructions and multiple extensions such as Integer (I), Multiplication and Division (M), Atomic (A), Control and Status Registers (Zicsr), Floating-Point (F/D), and Vector (V), while also allowing designers to incorporate custom instructions tailored to specific applications.[4 5]

The UETRV-PCore is a custom RISC-V processor implementing the RV32IMAZicsr ISA with a five-stage pipeline consisting of Instruction Fetch, Decode, Execute, Load/Store Unit, and Writeback stages. The design supports Control and Status Registers (CSRs) and includes an SV32-based virtual memory system with two-level paging and superpage support. Its memory subsystem integrates a direct-mapped data cache and a 4-way associative instruction cache, while peripheral support includes UART, CLINT, PLIC, and SPI interfaces. The core has been deployed on FPGA platforms and is capable of booting a Linux operating system, demonstrating its practical applicability.[1]



Fig(1): SoC block diagram showing the connectivity of the core with the memory subsystem and different peripherals using the data bus[2]

Microbenchmarks are commonly used to evaluate the performance of a processor core by running small test programs that target specific aspects of execution. These programs are grouped into categories such as control flow,

dependency chains, memory load/store operations, and arithmetic computations. Through these focused tests, it becomes possible to observe how the pipeline behaves, identify performance limitations, and analyze the impact of compiler optimizations. Compared to full benchmark suites, microbenchmarks provide a clearer and more detailed view of the internal efficiency of the core. In this work, a set of RISC-V microbenchmarks from the riscv-validation repository is used for the performance evaluation of the UETRV-PCore, covering control flow, dependency chains, memory operations, and arithmetic computations.[3 9] riscv64-unknown-elf is a GNU cross-compiler toolchain for RISC-V processors that generates programs in the ELF (Executable and Linkable Format) binary format. This toolchain includes standard compiler tools such as gcc, as, ld, and objcopy, but they are built to target bare-metal RISC-V systems (without an operating system). The riscv64-unknown-elf toolchain was used to compile the benchmark programs. [6]

The GNU compiler provides different optimization levels that change how the code is compiled and executed. In this project, three levels were used:[7]

O1 (Basic Optimization):Performs simple optimizations such as removing unnecessary instructions and improving register usage. It gives better performance than unoptimized code with little effect on compilation time.

O2 (Standard Optimization):Includes all -O1 optimizations and adds more techniques such as loop improvements and better instruction scheduling. It aims to give good performance without making the code too large.

O3 (High Optimization):Applies all -O2 optimizations and adds more aggressive changes like function inlining and loop transformations. It usually improves performance further, especially for compute-heavy programs, but can also increase code size.

Loop improvements reduce unnecessary work inside loops, for example by moving constant calculations outside the loop or replacing costly operations with simpler ones. Instruction scheduling arranges instructions in a better order so the processor pipeline spends less time waiting, which helps the program run faster.

Function inlining replaces a function call with the actual code of the function, which removes the overhead of calling and returning. Loop transformations improve the efficiency of loops by techniques such as loop unrolling, fusion, or vectorization, which reduce extra instructions and make execution faster.

II. METHODOLOGY

The performance evaluation was conducted using a cycle-accurate simulation environment to quantify the microarchitectural efficiency of the UETRV-PCore. The core features a five-stage pipeline (Fetch, Decode, Execute, Load/Store Unit, and Writeback) and an SV32 virtual memory system.

A. Benchmarking Suite and Compilation

A selection of microbenchmarks from the **riscv-validation** repository was utilized, covering four distinct computational categories: control flow (branches/switches), dependency chains, memory operations (load/store), and integer arithmetic. To analyze the impact of compiler strategies, the source files were compiled using the **riscv64-unknown-elf-gcc** toolchain. Two optimization profiles were evaluated:

- Unoptimized (-O0): Serving as the baseline for raw microarchitectural throughput.
- Optimized (-O1): Implementing basic instruction scheduling and improved register usage to mitigate pipeline stalls.

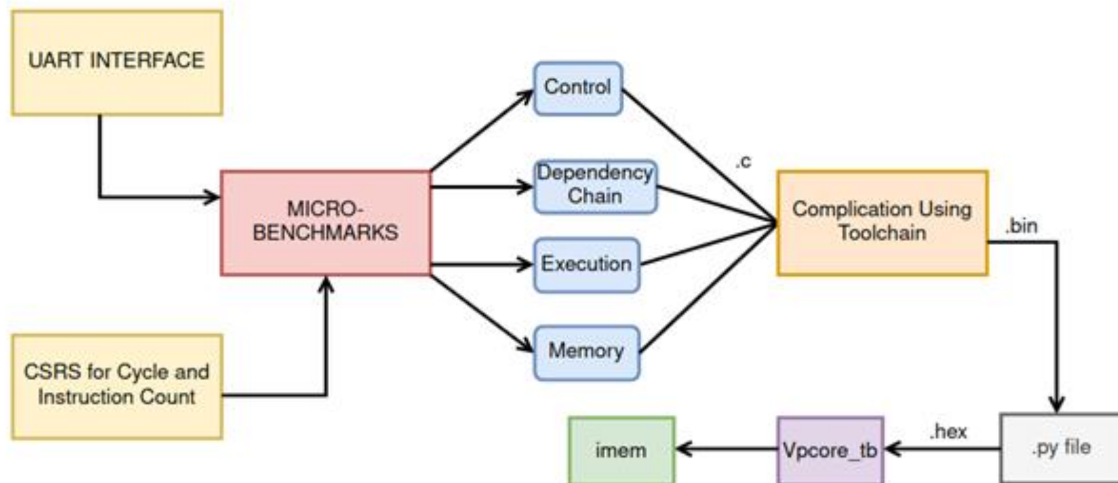
I evaluated -O1, -O2, and -O3 and observed very similar IPC values across all benchmarks; therefore, I report only -O0 and -O1 results.

B. Simulation and Data Extraction

The RTL design of the UETRV-PCore was simulated using **Verilator** to create a high-performance C++ executable model(Vpcore_tb). The execution flow followed a structured pipeline:

- Binary Conversion: Compiled .bin files were converted to .hex format via a Python script for memory initialization.
- Instruction Loading: The .hex files were loaded into instruction memory via plusargs during simulation.
- Metric Extraction: To measure the Instructions per Cycle (IPC), the source code was instrumented to read the hardware Performance Monitor Counters (PMCs). Specifically, the mcycle and minstret Control and Status Registers (CSRs) were accessed before and after each benchmark execution.
- Logging: The final cycle counts and retired instruction counts were transmitted via the UART interface and recorded for quantitative analysis.

International Journal of Engineering Applied Sciences and Technology, 2026
Vol. 10, Issue 10, ISSN No. 2455-2143, Pages 01-06
Published Online February 2026 in IJEAST (http://www.ijeast.com)



Fig(2):Flow chart of the overall process

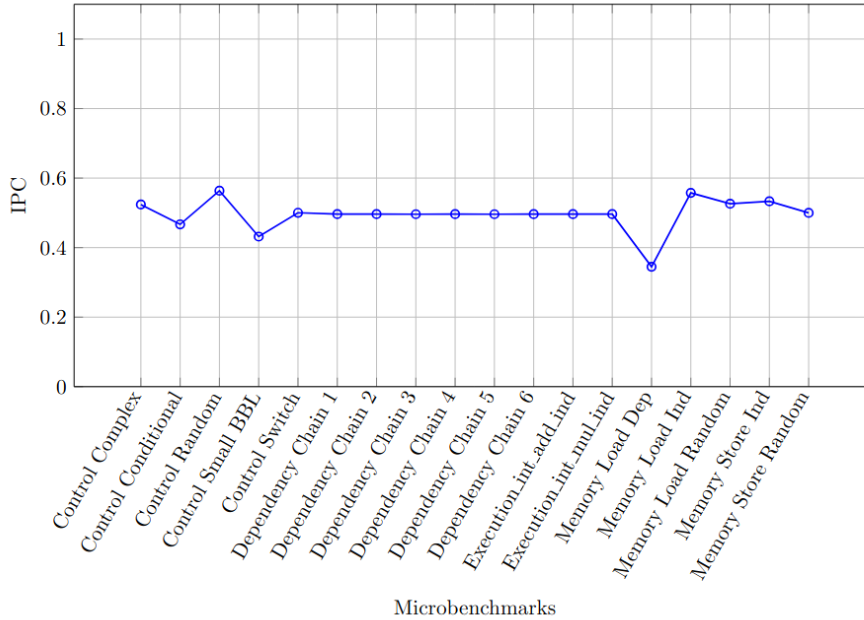
III. RESULTS

[8] TABLE 1. The table summarizes the Instructions Per Cycle (IPC) results obtained for each microbenchmark when compiled using no optimization flag of the riscv64-unknown-elf toolchain.

Microbenchmarks	Cycle Count	Instruction Count	IPC
Control Complex	420121	220172	0.524068066
Control Conditional	300086	140157	0.467056111
Control Random	2200272	1240059	0.563593501
Control Small BBL	143077	61769	0.431718585
Control Switch	460310	230294	0.50030197
Dependency Chain 1	2700110	1340178	0.496342001
Dependency Chain 2	2700079	1340178	0.496347699
Dependency Chain 3	2540142	1260209	0.49611754
Dependency Chain 4	2700091	1340153	0.496336235
Dependency Chain 5	2540118	1260182	0.496111598
Dependency Chain 6	2540091	1260180	0.496116084
Execution_int_add_ind	2700132	1340186	0.49634092
Execution_int_mul_ind	2700144	1340200	0.496343899
Memory Load Dep(Array size=1024, Element size=56bytes)	98834400	34094879	0.344969757
Memory Load Ind(Array size=1024, Element size=56bytes)	168986892	94212615	0.557514337
Memory Load Random(Array size=1024, Element size=56bytes)	193609923	101883076	0.526228586
Memory Store Ind(Array size=1024, Element size=56bytes)	138423230	73790273	0.533077237
	16292263	81440277	0.499870859

Memory Store Random(Array size=1024, Element size=56bytes)			

The results show the cycle count, instruction count, and resulting Instructions per Cycle (IPC) for different control, dependency, execution, and memory-intensive workloads. Control and dependency chain benchmarks achieve an IPC close to 0.5, reflecting balanced instruction throughput. Memory-bound benchmarks, particularly dependent loads, suffer from lower IPC due to latency, while independent and random memory accesses achieve relatively higher IPC.



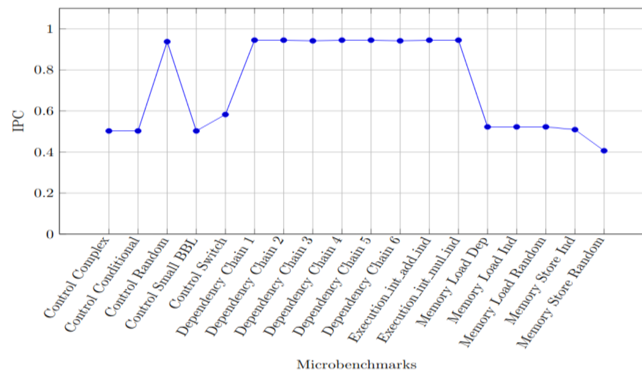
Fig(4)Performance analysis of microbenchmarks without optimization (IPC comparison)

TABLE 2. The table shows IPC results obtained using O1 optimization flag of the riscv64-unknown-elf toolchain.

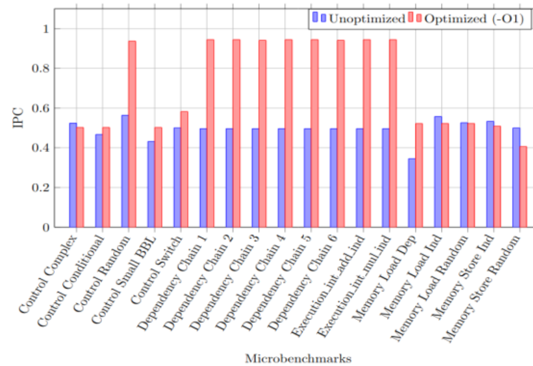
Microbenchmarks	**Cycle Count**	**Instruction Count**	**IPC**
Control Complex	40029	20108	0.502335806
Control Conditional	40035	20113	0.502385412
Control Random	640437	600232	0.937222552
Control Small BBL	40546	20369	0.502367681
Control Switch	129128	75203	0.582391115
Dependency Chain 1	360051	340126	0.944660617
Dependency Chain 2	360052	340127	0.944660771
Dependency Chain 3	340053	320128	0.941406192
Dependency Chain 4	360055	340131	0.944664009
Dependency Chain 5	360055	340131	0.944664009
Dependency Chain 6	340058	320119	0.941365884
Execution_int_add_ind	360067	340131	0.944632526
Execution_int_mul_ind	360062	340123	0.944623425
Memory Load Dep(Array size=1024, Element size=56bytes)	1312783	685229	0.5219666921
Memory Load Ind(Array	1308252	683185	0.522212081

size=1024, Element size=56bytes)			
Memory Load Random(Array size=1024, Element size=56bytes)	1310423	684271	0.522175663
Memory Store Ind(Array size=1024, Element size=56bytes)	22112153	11250108	0.508774880
Memory Store Random(Array size=1024, Element size=56bytes)	37142128	15080118	0.406011147

In the second table (with higher optimization impact), dependency chains and execution benchmarks achieve a much higher IPC of about 0.94, showing the compiler's ability to optimize arithmetic and dependency-heavy code effectively. Control benchmarks also improve, with Control Random reaching 0.93 IPC. However, memory benchmarks remain bottlenecked, with IPC ranging from 0.40 to 0.52, similar to the first table, since memory latency cannot be optimized away.



Fig(5): Performance analysis of microbenchmarks with optimization flag -O1 (IPC comparison)



Fig(6): Performance Comparison of microbenchmarks on pcore with and without optimiztion flag

IV. FINDINGS

Control Benchmarks & Branch Prediction:

In Pcore, there is no branch predictor, so the processor cannot guess whether a branch (e.g., an if or a loop) will be taken or not. Instead, the branch outcome is determined in the execute stage. Until then, the processor continues to fetch and decode instructions next to the branch instruction, assuming sequential flow. If the branch turns out to be taken, those speculatively fetched/decoded instructions are wrong and must be discarded. This is called a pipeline flush. Flushing invalidates work done in the fetch and decode stages, creating "bubbles" (wasted cycles) in the pipeline and reducing effective IPC.

Modern processors often include dynamic branch predictors (two-level, tournament, or even neural predictors) to mitigate this, but your baseline RISC-V core lacks such hardware. That's why control benchmarks, which heavily involve branches, show relatively poor IPC compared to dependency or arithmetic benchmarks.[10]

Load Dependency & RAW Hazards:
In memory benchmarks, the load dependency test stands out as worse performer. This happens due to a Read-After-Write (RAW) hazard: if an instruction depends on the result of a previous load (e.g., lw t0, 0(a0) followed immediately by add t1, t0, t2), the dependent instruction must wait until the load finishes retrieving the value from memory.[11] In Pcore, this dependency creates a pipeline stall of one cycle because the result from the memory stage is not yet available when the dependent instruction reaches the execute stage. Other memory tests (independent loads/stores or random access) don't face as severe a penalty, since their instructions are not directly dependent on one another.

V. CONCLUSION

The rigorous evaluation of the UETRV-PCore confirmed its functional design while highlighting key microarchitectural limitations. The core's IPC, measured at around **0.5**, is primarily constrained by the absence of branch prediction, which leads to significant penalties when branch outcomes are mispredicted. Additionally, read-after-write (RAW) hazards in load dependency tests introduce a one-cycle stall, reducing instruction throughput. While compiler optimizations dramatically improved the performance of dependency and execution-heavy code, boosting IPC to nearly **0.94**, memory-bound benchmarks remained a persistent bottleneck due to unavoidable memory latency.

VI. REFERENCES

[1]. M. Tahir and U. Shahid, "UETRV-Pcore RV32IMA Zicsr Core," GitHub repository, 2024. [Online]. Available: https://github.com/ee-uet/UETRV-PCore

[2]. U. Shahid, Masooma Zia, Abubakar, Aman, Eman, "UETRV-Pcore Design Document," Read the Docs, 2024. [Online]. Available: https://uetrv-pcore-doc.readthedocs.io/en/main/design_document/

[3]. J. Rangi, K. Pai, A. Akram, and J. Lowe-Power, "RISC-V Microbenchmarks," GitHub repository, 2024. [Online]. Available: https://github.com/darchr/riscv-validation

[4]. A. Waterman, Y. Lee, D. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.1," RISC-V Foundation, Tech. Rep., 2016.

[5]. A. Waterman, Y. Lee, R. Avižienis, D. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture Version 1.9," RISC-V Foundation, Tech. Rep., 2016.

[6]. "RISC-V Toolchain Installation," GitHub repository, 2024. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain

[7]. Optimize Options (GNU Compiler Collection (GCC) Internals), Free Software Foundation, 2024. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[8]. M. Zia, "Performance Evaluation Results," GitHub repository, 2024. [Online]. Available: https://github.com/Masooma82/UETRV-PCore/tree/main

[9]. L. Alluri, M. Bhaskar, and H. J. Magadum, "Performance Assessment of RISC-V Architecture," in Proc. ResearchGate, May 2020. [Online]. Available: https://www.researchgate.net/publication/341121058

[10]. H. Miyazaki et al., "RVCoreP: An optimized RISC-V soft processor of five-stage pipelining," in 2020 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART), 2020, pp. 1–6.

[11]. A. P. Shanthi, "Pipeline Hazards," Computer Architecture Online Resources, University of Maryland. [Online]. Available: https://www.cs.umd.edu/~meesh/cmsc411/CourseResources/CA-online/chapter/pipeline-hazards

IJEAST

INTERNATIONAL JOURNAL OF ENGINEERING APPLIED SCIENCE AND TECHNOLOGY

ABOUT IJEAST

International Journal of Engineering Applied Science and Technology (IJEAST) is a peer-reviewed, open access journal that publishes high-quality research papers in the field of Engineering, Applied Science and Technology.

IJEAST aims to provide a platform for researchers, academicians, and professionals to share their innovative ideas, research findings, and practical experiences with the global scientific community.

FOCUS AREAS

- Engineering
- Applied Science
- Technology
- Innovation & Development
- Interdisciplinary Studies

PEER REVIEWED
All submissions are rigorously peer reviewed to ensure quality.

OPEN ACCESS
Free and unrestricted access to research for all.

GLOBAL REACH
Connecting researchers and professionals worldwide.

TIMELY PUBLICATION
We ensure a swift and efficient publication process.

For more information, visit our website
www.ijeast.com

INTERNATIONAL JOURNAL
OF ENGINEERING APPLIED SCIENCE AND TECHNOLOGY

editor@ijeast.com
www.ijeast.com
India

2455-2143